

# A Common-Sense Guide to Data Structures and Algorithms Study Guide (Chapters 1 to 18)

## CHAPTER 1: Why Data Structures Matter

When newcomers start coding, they focus mainly on making their code work. The success of the code is determined by whether it functions correctly or not. As they advance, software engineers learn to evaluate their code on more complex levels. They realize that two pieces of code might achieve the same result, but one could be considered superior.

Code quality is assessed by various factors. A significant aspect is code maintainability, including the readability, structure, and modularity of the code. But there's another key quality, code efficiency, which involves how fast the code runs. Two different code snippets might accomplish the same objective, but one could execute faster.

Consider two functions that print even numbers from 2 to 100. The first function checks if each number is even and then prints it, while the second function simply adds 2 to the number and prints it:

```
def print_numbers_version_one():
    number = 2
    while number <= 100:
        # If number is even, print it:
        if number % 2 == 0:
            print(number)
        number += 1
```

```
def print_numbers_version_two():
    number = 2
    while number <= 100:
        print(number)
        # Increase number by 2, which, by definition,
        # is the next even number:
        number += 2
```

The second function runs faster because it loops only 50 times compared to 100 times in the first function, making it more efficient. This illustration highlights the importance of writing efficient code and how understanding data structures can affect the speed of code, essential skills for becoming an advanced software developer.

## Data Structures

Data refers to all kinds of information, including basic numbers and strings. Even complex data can be broken down into these simple components. Data structures describe how data is arranged. The same data can be organized in different ways, and these variations can greatly affect the speed of your code. Consider a simple example where you have three strings that form a message. You can store them as independent variables:

```
x = "Hello! "
```

```
y = "How are you "  
z = "today?"  
print x + y + z
```

Or you can organize them in an array:

```
array = ["Hello! ", "How are you ", "today?"]  
print array[0] + array[1] + array[2]
```

The way you choose to structure your data is not just a matter of tidiness. It can influence how quickly your code runs and even whether it can handle large loads. For instance, if you're building a web app used by many people at once, the right data structures can prevent it from crashing due to overload.

Understanding how data structures affect the performance of your software empowers you to write efficient and sophisticated code, enhancing your skills as a software engineer. This chapter will introduce you to the analysis of two specific data structures: **arrays** and **sets**. You'll learn how to examine their performance implications, even though they might seem quite similar at first glance.

### The Array: The Foundational Data Structure

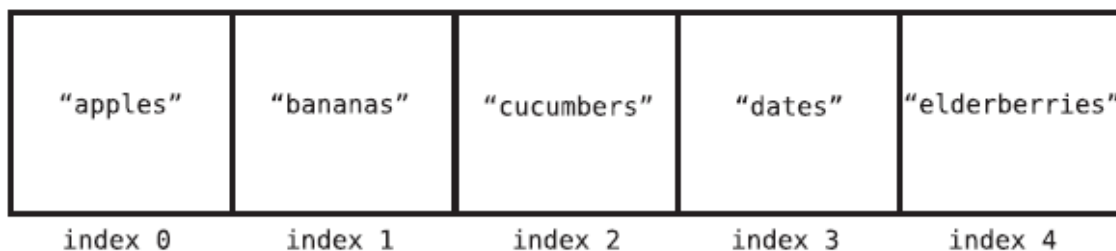
The **array** is a fundamental data structure, essentially a list of data elements. It's a versatile tool used in various situations. For example, if you're writing an application for creating grocery shopping lists, you might have an array like this:

```
array = ["apples", "bananas", "cucumbers", "dates", "elderberries"]
```

This specific array has five strings, representing items to buy at the supermarket. When talking about arrays, some specific terms are often used:

- **Size:** The number of data elements in the array. The above array has a size of 5 because it has five values.
- **Index:** The number that shows the position of a piece of data within the array. In most programming languages, the index starts at 0. So, in the given example, "apples" is at index 0, and "elderberries" is at index 4.

Thus, arrays are a foundational structure, used to store and organize data in an accessible and ordered manner.



### Data Structure Operations

To gauge the performance of a data structure like an array, we need to look at how code interacts with it. This is typically done through four basic operations:

1. **Read:** This operation looks up a specific value within the data structure at a particular spot. In an array, you would find a value at a specific index. For instance, finding which grocery item is at index 2 is a read operation.
2. **Search:** This operation seeks a particular value within the data structure. In the context of an array, it means looking to see if a specific value exists, and if so, at which index. If you were to find the index of "dates" in a grocery list, you would be searching the array.
3. **Insert:** This refers to adding a new value to the data structure. In an array, it means placing a new value in an available slot. Adding "figs" to a shopping list would be an insert operation in the array.
4. **Delete:** This operation involves removing a value from the data structure. In an array, it means taking out one of the values. If "bananas" were removed from a grocery list, that value would be deleted from the array.

### Measuring Speed

Measuring the speed of an operation in programming doesn't mean calculating the time it takes in seconds or minutes. Instead, it refers to the number of computational steps required to complete the operation. Here's why this approach is used:

1. **Inconsistency in Time Measurement:** An operation might take five seconds on one computer but could take more or less time on another machine. Time is an unreliable measure since it varies depending on the hardware used.
2. **Steps as a Universal Measure:** By counting the number of computational steps, you can make a consistent comparison between different operations. If Operation A takes 5 steps and Operation B takes 500 steps, you can always conclude that Operation A will be faster, regardless of the hardware.
3. **Terms Used for Measuring Speed:** Throughout literature and discussions, you may encounter terms like speed, time complexity, efficiency, performance, and runtime. They all mean the same thing in this context: the number of steps an operation takes.

This concept of measuring speed by counting steps is critical for understanding and analyzing the efficiency of different operations, such as those performed on data structures like an array. It provides a standard way to evaluate how quickly a piece of code will run, regardless of where it's executed.

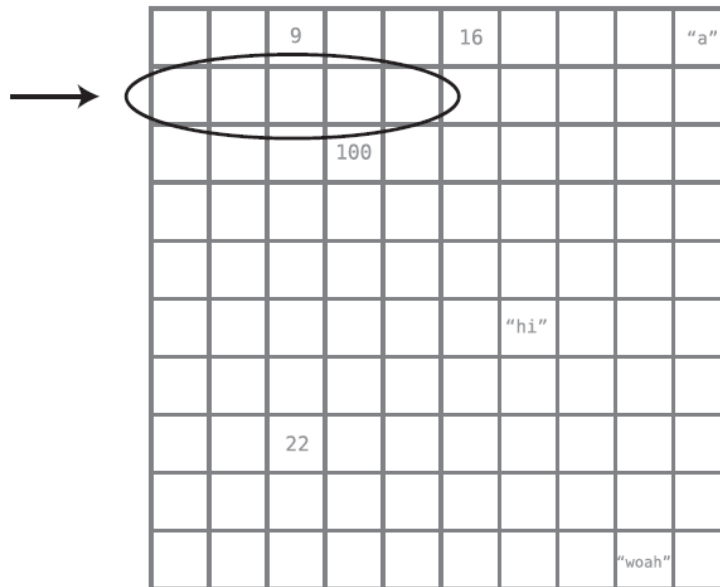
### Reading

Reading from an array is a quick and straightforward process for a computer. An array is like a list, such as ["apples", "bananas", "cucumbers", "dates", "elderberries"]. If you want to find out what's at a certain position, like index 2, the computer can instantly tell you that it's "cucumbers." How does this happen? Let's understand:

1. **Computer Memory:** Think of computer memory as a huge collection of cells or slots. Some are empty, while others contain data bits.

		9		16				"a"
			100					
						"hi"		
		22						
								"woah"

2. **Array Allocation:** When you create an array for five elements, the computer finds five empty slots in a row and reserves them for your array.



		9		16				"a"
			100					
						"hi"		
		22						
								"woah"

3. **Memory Address:** Each cell in the computer's memory has a unique numerical address, like a street address. These addresses are sequential, so each address is one number greater than the previous one.

1000	1001	1002	1003	1004	1005	1006	1007	1008	1009
1010	1011	1012	1013	1014	1015	1016	1017	1018	1019
1020	1021	1022	1023	1024	1025	1026	1027	1028	1029
1030	1031	1032	1033	1034	1035	1036	1037	1038	1039
1040	1041	1042	1043	1044	1045	1046	1047	1048	1049
1050	1051	1052	1053	1054	1055	1056	1057	1058	1059
1060	1061	1062	1063	1064	1065	1066	1067	1068	1069
1070	1071	1072	1073	1074	1075	1076	1077	1078	1079
1080	1081	1082	1083	1084	1085	1086	1087	1088	1089
1090	1091	1092	1093	1094	1095	1096	1097	1098	1099

4. **Reading from an Array:** When you want to read something at a particular index, the computer can jump right to that spot. It knows where the array begins, so it can quickly calculate where the desired index is. For example:

- The array starts at memory address 1010.
- To read the value at index 3, it adds 3 to 1010, resulting in 1013.
- The computer jumps to memory address 1013 and sees that it contains "dates."

"apples"	"bananas"	"cucumbers"	"dates"	"elderberries"
----------	-----------	-------------	---------	----------------

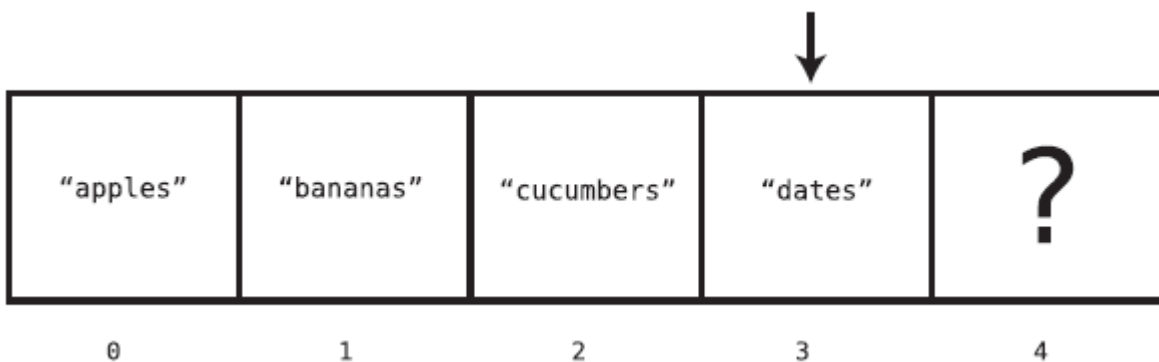
memory address:	1010	1011	1012	1013	1014
index:	0	1	2	3	4

5. **Efficiency:** This operation is fast since the computer can jump to any memory address in a single step. It makes arrays a powerful and efficient data structure for reading values.

A computer's ability to access any memory address in one step and the sequential order of memory addresses allow it to quickly find the value at any index in an array. If you ask about the value at index 3, the computer will instantly know that it's "dates." But if you ask where "dates" can be found in the array, that's a different operation, and we'll look at it next.

## Searching

1. **What is Searching?:** Searching in an array involves finding whether a particular value is present in the array and identifying its index. It's the opposite of reading, where you provide an index and get the value.
2. **Difference from Reading:** While reading is quick, searching can be slow. Reading lets the computer jump to a specific index, while searching requires the computer to examine each value one by one.
3. **Searching Process:** Imagine an array of fruits like ["apples", "bananas", "cucumbers", "dates", "elderberries"]. The computer doesn't know what's inside each cell, so to find "dates", it must:
  - Check index 0, see "apples", and move to the next.
  - Check index 1, see "bananas", and move to the next.
  - Continue this process until it finds "dates" at index 3.



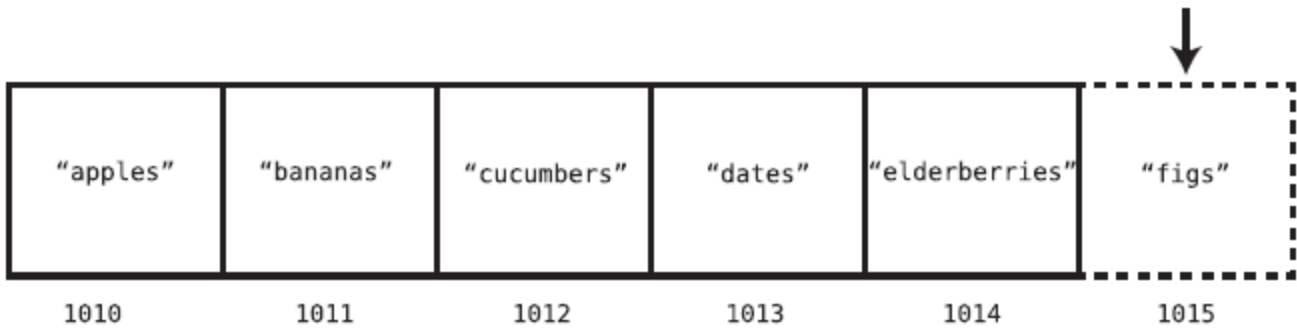
4. **Linear Search:** This method, where the computer checks each cell one by one, is called linear search. It's not always fast, especially if the array is long.
5. **Maximum Number of Steps:** In the worst case, the computer might have to check every cell. For an array of 5 cells, the maximum steps would be 5. For 500 cells, it would be 500. Generally, if there are N cells, the maximum number of steps would be N.
6. **Comparison with Reading:** Searching can take many steps, while reading takes only one, regardless of the array's size. Therefore, searching is less efficient than reading.

While reading from an array is like quickly jumping to the right shelf in a library, searching is like going through every book on each shelf until you find the one you want. If the array is large, searching can be quite time-consuming.

## Insertion

Inserting a new element into an array is a straightforward task, but the efficiency depends on where you want to put the new item. Let's break down how insertion works:

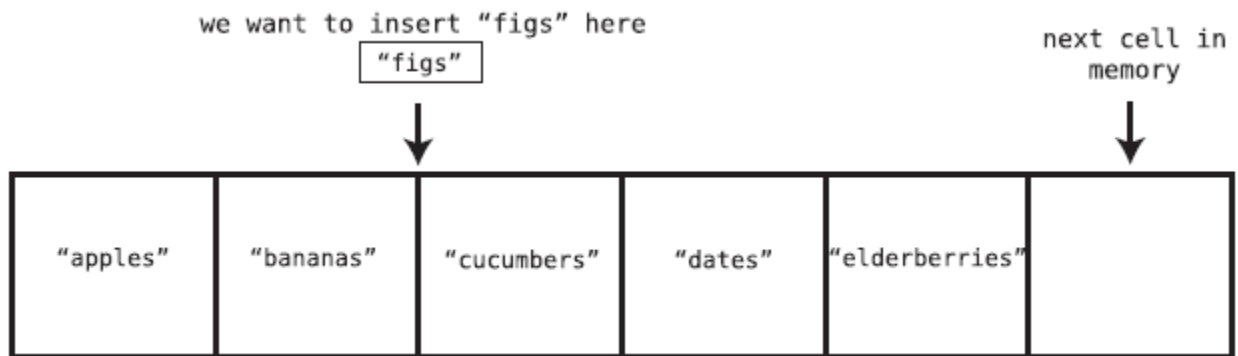
1. **Inserting at the End:** If you want to add an item to the end of an array, it's very simple. For example, adding "figs" to an array that begins at memory address 1010 and has 5 elements would mean placing it at memory address 1015. This is just one step because the computer knows the array's size and starting address.



2. **Inserting in the Middle:** If you want to put the new item in the middle, it's more complicated. For instance, if you want to insert "figs" at index 2, you'll need to shift other elements ("cucumbers", "dates", and "elderberries") to the right to make room. This process requires several steps:

- Move "elderberries" right.
- Move "dates" right.
- Move "cucumbers" right.
- Insert "figs" at index 2.

This example would take four steps in total.



3. **Inserting at the Beginning:** The most time-consuming insertion is at the beginning of the array. You have to move all other values one cell to the right to make room. In the worst case, for an array with  $N$  elements, this would take  $N + 1$  steps ( $N$  shifts and one actual insertion).
4. **Adding Beyond the Allocated Size:** If the array already has its allocated size filled, and you're adding another element, the computer may need to allocate additional cells. How this is done varies by programming language, and it's handled automatically.

So, inserting at the end is a quick one-step process, while inserting in the middle or at the beginning can take multiple steps. Think of it like getting into a packed bus: if there's room at the back, it's easy to get on. But if you need to get to a seat in the middle, you have to wait for others to shuffle around and make space. If you need a seat at the front, it's even more time-consuming!

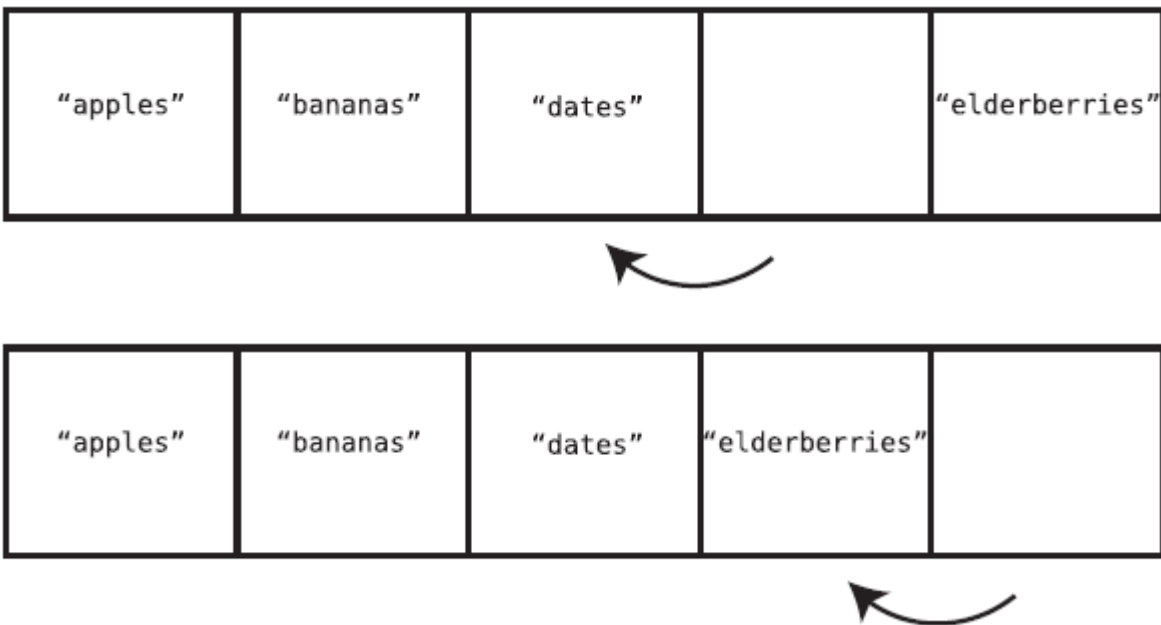
## Deletion

Deletion in an array means removing a value at a specific index, but it's not as simple as just erasing that value. There are a few steps involved, depending on the situation. Here's a breakdown using the example of deleting "cucumbers" at index 2:

1. **Step 1 - Deleting the Value:** The actual deletion of "cucumbers" from the array is straightforward and only takes one step.



2. **The Gap Problem:** After deleting "cucumbers," there's an empty cell in the middle of the array. This gap is a problem because arrays are meant to be continuous without any spaces.
3. **Step 2 & 3 - Shifting the Values:** To fix the gap, the remaining values ("dates" and "elderberries") have to be shifted to the left. Each shift is a separate step, so in this example, there are two additional steps.



4. **Worst-Case Scenario:** Deleting the very first element in an array is the most time-consuming. If the array has  $N$  elements, you'd spend one step deleting the first element and  $N - 1$  steps shifting the rest. So, the maximum steps for deletion would be  $N$ .

Think of deletion like taking a book out of a tightly packed bookshelf. When you remove the book (the deletion step), there's a gap. You then need to slide the other books to fill that space (the shifting steps). If you take a book from the left end, you'll have to shift all the other books, which takes more time.

The analysis of this process helps to understand the time complexity of a data structure. Understanding this is vital because choosing the right data structure affects the efficiency and performance of software. Even though arrays might seem similar to other data structures like sets, their operations and efficiencies can be quite different.



## Sets: How a Single Rule Can Affect Efficiency

A **set** is a data structure that's similar to an array, but with one key difference: it doesn't allow duplicates. This single rule affects how sets operate compared to arrays, especially when it comes to efficiency. Here's a simplified exploration:

### 1. What is a Set?

- **Array-Based Set:** It's like an array, a simple list of values, but it never allows duplicate values.
- **Example:** If you have the set ["a", "b", "c"] and try to add another "b", the computer won't allow it.
- **Usage:** Sets are useful to avoid duplicate data, like in a phone book where you wouldn't want the same number listed twice.

### 2. Operations in an Array-Based Set:

- **Reading:** Same as an array, takes one step.
- **Searching:** Same as an array, up to N steps.
- **Deletion:** Same as an array, up to N steps to delete and shift data.
- **Insertion:** This is where sets differ from arrays.

### 3. Insertion into a Set:

- **Checking Duplicates:** Since sets don't allow duplicates, the computer must first search the set to make sure the value doesn't already exist.
- **Example:** To insert "figs" into the set ["apples", "bananas", "cucumbers", "dates", "elderberries"], the computer must first search all five elements, and then insert if "figs" is not found. Total six steps.
- **Best-Case Scenario:** Inserting at the end takes  $N + 1$  steps (N steps for searching, 1 for insertion).
- **Worst-Case Scenario:** Inserting at the beginning takes  $2N + 1$  steps (N steps for searching, N steps for shifting, 1 for insertion).
- **Contrast with Arrays:** Inserting at the end of an array takes just one step, and at the beginning takes  $N + 1$  steps.

### 4. Should You Use Sets or Arrays?

- **When to Use Sets:** If you need to ensure no duplicates.
- **When to Use Arrays:** If you don't need to worry about duplicates and want more efficient insertions.

Think of an array-based set as a guest list at an exclusive party where no two guests can have the same name. If a new guest tries to join, the host must first check the entire list to ensure that the guest's name is not already there. Only then can the guest be added to the party. In a regular array (or a less exclusive party), anyone can join without the need for a name check.

## Wrapping Up

Understanding how many steps an operation takes is central to grasping the performance of data structures. Your choice between different structures like arrays and sets can have a significant impact on your program's efficiency. Here's a simplified wrap-up:

1. **Importance of Analysis:** By analyzing the number of steps, you can understand how fast a data structure performs tasks. It's like choosing the right vehicle for a journey - a sports car for speed on a smooth road, or an off-road vehicle for rugged terrain.
2. **Array vs. Set:** The decision between using an array or a set depends on your needs. Arrays might be faster for certain operations, but sets prevent duplicates. It's about weighing what's more important for your specific task.
3. **Next Steps:** After learning about these data structures, you can apply the same type of analysis to compare different algorithms. This will help you ensure the fastest and most efficient performance of your code, taking you to the next level of coding mastery.

Consider it like cooking a meal: you have to choose the right ingredients (data structures) and follow the right recipe (algorithm) to create a delicious dish (efficient code). Your newfound understanding of time complexity is the kitchen skill that helps you make those choices wisely. It sets the stage for further exploration and mastery in the world of coding.

## Exercises

1. For an array containing 100 elements, provide the number of steps the following operations would take:
  - a. **Reading:** 1 step, as the computer can directly access the value at a particular index.
  - b. **Searching for a value not contained within the array:** up to 100 steps, as you would have to check each of the 100 elements to confirm that the value isn't there.
  - c. **Insertion at the beginning of the array:** shift all 100 elements to the right, plus the insertion itself, makes it 101 steps.
  - d. **Insertion at the end of the array:** 1 step since no shifting of elements is required.
  - e. **Deletion at the beginning of the array:** shift the remaining 99 elements to the left, plus the deletion step itself, makes it 100 steps.
  - f. **Deletion at the end of the array:** 1 step since no shifting is necessary.
2. For an array-based set containing 100 elements, provide the number of steps the following operations would take:
  - a. **Reading:** one step to look up a value at a particular index.
  - b. **Searching for a value not contained within the array:** 100 steps since all 100 elements would have to be checked.

c. **Insertion of a new value at the beginning of the set:** 201 steps. Search the entire set to ensure no duplicates (100 steps), shift all existing elements to the right to make room (100 steps), and then insert the new value (1 step).

d. **Insertion of a new value at the end of the set:** 101 steps. Search the entire set to ensure no duplicates (100 steps), and then insert the new value (1 step).

e. **Deletion at the beginning of the set:** 100 steps. Remove the value (1 step) and then shifting all remaining elements to the left to fill the gap (99 steps).

f. **Deletion at the end of the set:** 1 step. Since it's at the end of the set, no shifting is required.

3. Normally the search operation in an array looks for the first instance of a given value. But sometimes we may want to look for *every* instance of a given value. For example, **say we want to count how many times the value "apple" is found inside an array. How many steps would it take to find all the "apples"?** Give your answer in terms of  $N$ .

If the array contains  $N$  elements, you would need to take  $N$  steps to examine each element and count all the instances of the given value. Hence, **the total number of steps required to find all the "apples" (or any specific value) in the array would be  $N$ .**

## CHAPTER 2: Why Algorithms Matter

In this chapter, we're delving into the influence that data structure choice and algorithm selection have on the efficiency of our code:

1. **Data Structure and Efficiency:** The data structure, such as an array or set, can greatly impact how quickly our code runs. Even though arrays and sets might look similar, their efficiency can differ a lot.
2. **Algorithm Selection:** An algorithm is like a recipe - a series of steps to achieve a task. Even a daily routine, like making a bowl of cereal, can be considered an algorithm. In coding, choosing the right algorithm for a task can make a big difference in speed and performance.
3. **Two Algorithms, Same Task:** Sometimes, you might find two different algorithms to do the same thing, but one may be much faster than the other. We've seen this in previous examples and will explore it further in this chapter.
4. **A New Data Structure:** To understand these new algorithms, we'll introduce a new data structure.

This chapter will emphasize that both the choice of data structure and the specific algorithm used to operate on that structure can have a major impact on how well our code performs.

### Ordered Arrays

An **ordered array** is similar to a regular array, but with one key difference: the values are always sorted. When you add a value to this array, it must be placed in the right spot to keep everything in order. Take the following array: **[3, 17, 80, 202]**. If you want to add the value 75, you can't just put it at the end like in a classic array. In an ordered array, the 75 must go between 17 and 80 to keep things sorted. Here's how it works:

1. Start with the original array.
2. Check the first value. Since 75 is more than 3, look to the right.
3. Check the next value. Since 75 is more than 17, keep going.
4. Find the 80. Since 75 is less than 80, it must go just before it.
5. Shift the last two values to the right to make space.
6. Insert 75 where it belongs.

This process takes six steps for four elements. In general, for  $N$  elements, the insertion takes  $N + 2$  steps. The steps can vary slightly depending on where the new value goes, but it's still roughly  $N + 2$ . If the new value goes at the end, it's a little quicker at  $N + 1$  steps. This insertion is less efficient than in a classic array. But the ordered array has a benefit when it comes to searching...

### Searching an Ordered Array

Imagine you're looking for the number 22 in a regular array like [17, 3, 75, 202, 80]. You'd have to look at every element since 22 could be anywhere. In an ordered array like [3, 17, 75, 80, 202], however, you can stop searching when you reach a value greater than 22, like 75, because you know 22 can't be after that. Here's a simple way to do a linear search in Ruby:

```
def linear_search(array, search_value)
  array.each_with_index do |element, index|
    return index if element == search_value
    break if element > search_value
  end
  return nil
end
```

This code goes through each element and returns its index if it matches the search value. If it reaches a value greater than the search value, it breaks out of the loop, knowing it won't find the value later in the array.

Using linear search on an ordered array can sometimes be faster than on a classic array. However, in the worst-case scenarios, both types of arrays may need to search through all N elements, taking up to N steps. While this might seem like there's not much difference in efficiency, an ordered array has a big advantage: it allows for another search method called binary search. Binary search is a lot faster than linear search. This makes ordered arrays very powerful for searching tasks.

## Binary Search

Binary search is like a number-guessing game where you try to find a number between a given range by always guessing the middle value. This method keeps eliminating half of the remaining possibilities with each guess, leading to a quick search. Let's see how binary search is applied to an ordered array. Imagine you're searching for the value 7 in an ordered array of nine elements. Here's how it works:

1. **Start with the Middle:** Look at the central cell first, which you can easily find by dividing the array's length by 2. In our example, the middle value is 9, so you know 7 must be to the left. You've just eliminated half the possibilities.



2. **Narrow Down Further:** In the remaining cells to the left of 9, find the middle value(s) again. Say it's 4. Since 4 is less than 7, you know 7 must be to its right, and you can eliminate more cells.



3. **Continue Halving:** Keep repeating the process of looking at the middlemost value and eliminating half of the remaining possibilities. If there are two middle values, you can arbitrarily choose one.



4. **Find or Conclude Absence:** Eventually, you'll find the value you're looking for, or you'll know it's not in the array. In our example, we found 7 in four steps.



While in this specific example binary search took the same number of steps as linear search would have, binary search is typically much faster, especially with large arrays. It keeps dividing the search space in half with each step, so it can quickly zero in on the target value or determine its absence.

Importantly, binary search only works with ordered arrays, where values are sorted. It relies on knowing whether the target value is to the left or right of the current guess, which is only possible if the values are in order. This makes binary search a significant advantage of ordered arrays over classic ones, offering a more efficient search method.

### Code Implementation: Binary Search

Here's a simplified explanation of how the binary search code in Ruby works:

1. **Initialize Bounds:** The search begins by setting a lower and upper bound for where the value might be. Initially, these are set to the first and last index of the array, covering the entire range.
2. **Start Loop:** A loop continues while the lower bound is less than or equal to the upper bound, indicating that there are still elements to search.
3. **Find Midpoint:** Within the loop, the midpoint of the current range is calculated, and the value at this midpoint is examined.
4. **Check Value:**
  - If it's the value you're looking for, you return the midpoint index.
  - If it's less than the search value, it means the target must be to the right, so the lower bound is adjusted to the right of the midpoint.
  - If it's more than the search value, the target must be to the left, so the upper bound is adjusted to the left of the midpoint.
5. **Repeat or Return:** The loop continues, repeatedly narrowing down the range until either the value is found or the bounds reach each other. If the bounds meet and the value hasn't been found, the function returns `nil`, indicating the value is not in the array.

### Binary Search vs. Linear Search

#### Linear Search:

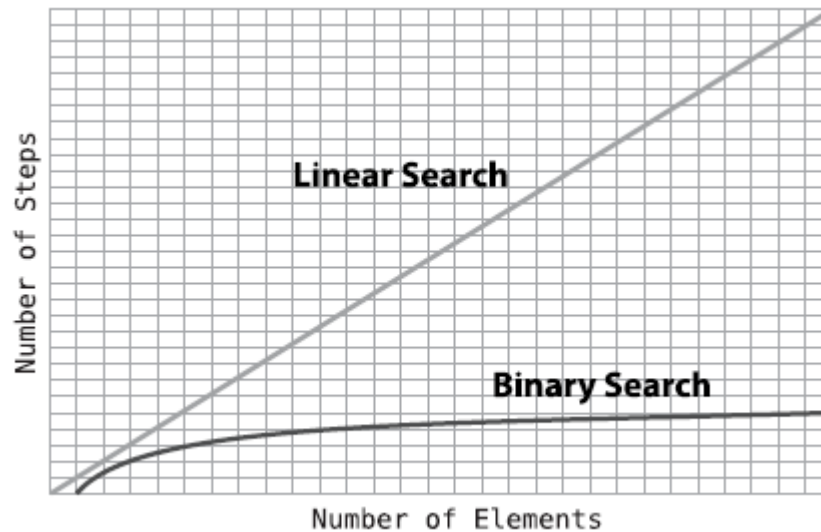
- **Steps Taken:** It inspects each item in the array one by one.
- **Performance:** With 100 values, it takes 100 steps.
- **Growth Pattern:** The number of steps grows linearly with the size of the array. If you double the size of the array, you double the number of steps.

#### Binary Search:

- **Steps Taken:** It cuts the array in half with each step by choosing the middle value.
- **Performance:** With 100 values, it takes just 7 steps.
- **Growth Pattern:** Each time you double the size of the array, you only need to add one more step.

#### Visualization:

- **Linear Search:** Graphing the number of elements versus the number of steps results in a straight diagonal line. Each new element adds a step.
- **Binary Search:** The graph is much flatter, and the number of steps increases very slowly. You must double the size to add one step.



#### Large Array Comparison:

- **10,000 Elements:** Linear takes 10,000 steps; Binary takes 13 steps.
- **1 Million Elements:** Linear takes 1 million steps; Binary takes 20 steps.

#### Considerations:

- **Ordered Arrays:** Binary search only works with ordered arrays, whereas linear search can work with any array.
- **Insertion Speed:** Ordered arrays are slower to insert into, but they allow for faster search.
- **Application Analysis:** You must consider what your application needs more: frequent insertions or frequent searches. If searching is more critical, an ordered array with binary search might be the better choice.

So, while binary search may not offer a huge advantage over linear search in small arrays, as the size grows, its efficiency becomes far superior, making it a popular choice for searching large datasets.

#### Pop Quiz

If binary search takes seven steps to search through an ordered array of 100 elements, how many steps would it take for an array with 200 elements?

8 steps.

While it might seem logical to double the number of steps since the array size is doubled, that's not how binary search works. In binary search, each step eliminates half of the remaining elements. So, when you double the number of elements in the array, you only need one additional step. This is what makes binary search an efficient algorithm, especially for large datasets.

Using binary search can also speed up insertion in an ordered array, as the search part of the insertion process is faster. However, it's worth remembering that insertion in an ordered array is generally still slower than in a standard array, where a search isn't required at all.

## Wrapping Up

When working on a computing task, the algorithm you choose can greatly influence the speed of your code. It's essential to recognize that no single data structure or algorithm will be ideal for every scenario. Take ordered arrays as an example: they enable binary search, which can be very efficient. But that doesn't mean ordered arrays should always be your go-to option. If you're primarily adding data and don't expect to search through it often, standard arrays might be a better choice due to quicker insertion times.

The key to evaluating different algorithms is to look at the number of steps each one requires. Understanding this helps you gauge their efficiency and make informed choices. In the next chapter, there will be a focus on formalizing the way we express the time complexity of various data structures and algorithms. This common language will provide clearer information, allowing for better decision-making regarding algorithm selection.

## Exercises

1. How many steps would it take to perform a linear search for the number 8 in the ordered array, [2, 4, 6, 8, 10, 12, 13]?

4 steps. A linear search would check each element one by one until it finds the target value, so it would need to check 4 elements in this array.

2. How many steps would binary search take for the previous example?

3 steps. A binary search divides the search interval in half with each step.

3. What is the maximum number of steps it would take to perform a binary search on an array of size 100,000?

17 steps.



## CHAPTER 3: O Yes! Big O Notation

The efficiency of an algorithm is often measured by the number of steps it takes, but you can't just label an algorithm by a fixed number of steps since it varies depending on the input size. You can say linear search takes  $N$  steps for  $N$  elements. But that's a bit cumbersome to say repeatedly. Enter Big O Notation, a concept borrowed from mathematics that provides a concise language to describe the efficiency of algorithms. It helps categorize the efficiency of an algorithm in a way that can be easily communicated.

In the context of computer science, Big O Notation simplifies the way we talk about the time complexity of an algorithm. Without getting into the mathematical details, it's a method to analyze algorithms in a consistent manner, like professionals do. The concept will be further refined and explained in subsequent chapters, broken down into simple terms to make it easier to understand.

### Big O: How Many Steps Relative to $N$ Elements?

**Big O Notation** is a way to describe the efficiency of an algorithm by focusing on the number of steps it takes relative to the number of elements,  $N$ , in the data. Here's how it works:

- **Linear Search:** In the worst-case scenario, linear search takes as many steps as there are elements in the array, or  $N$  steps. This is expressed as  $O(N)$ , pronounced "Oh of  $N$ ." It means that if there are  $N$  elements, the algorithm will take  $N$  steps. This is known as linear time.
- **Reading from an Array:** Reading from a standard array always takes one step, regardless of the size of the array. This is expressed as  $O(1)$ , pronounced "Oh of 1." Even if our question involves  $N$ , the answer doesn't depend on  $N$ . Reading from an array always takes one step, making it constant time.

The notation  $O(N)$  and  $O(1)$  answer what we call the "key question": if there are  $N$  data elements, how many steps will the algorithm take? The answer to this key question is found within the parentheses of the Big O expression.

- $O(1)$  is particularly interesting, as it's considered the "fastest" kind of algorithm. No matter how many elements are in the array, the algorithm takes a constant number of steps. It doesn't require additional steps as the data size grows.

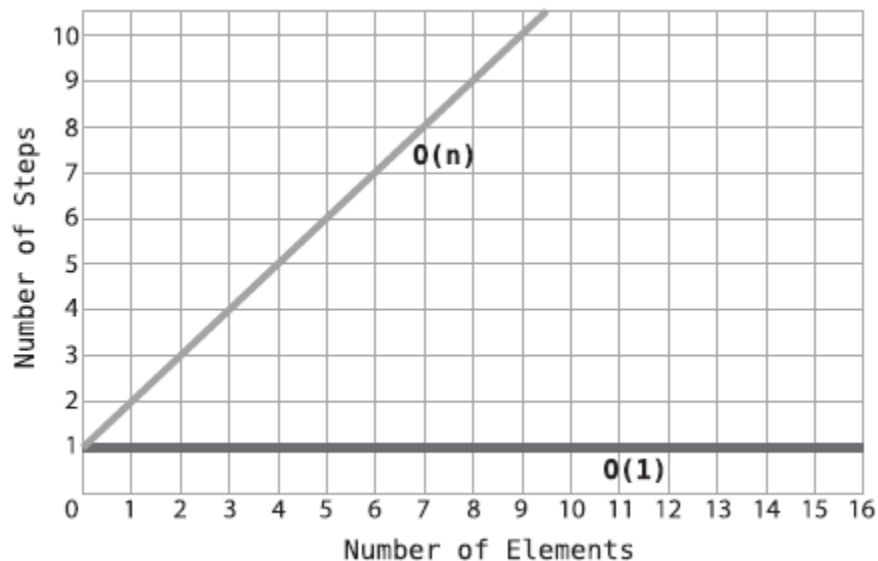
### The Soul of Big O

Big O Notation goes beyond just stating the number of steps an algorithm takes; it captures the relationship between the data size ( $N$ ) and the number of steps the algorithm will take. Here's a deeper look into what's often referred to as the "soul of Big O":

- **Constant Time Algorithms:** Consider an algorithm that takes three steps no matter how many elements there are. While you might think this should be expressed as  $O(3)$ , it's actually  $O(1)$ . This is because Big O is concerned with how an algorithm's performance changes as the data increases. In this case, the number of steps doesn't change, so we say it's  $O(1)$ , or constant time.
- **Linear Time Algorithms:** An algorithm that takes  $N$  steps for  $N$  elements is  $O(N)$ . This tells you that the number of steps increases in direct proportion to the data. It's not just about stating that it takes  $N$  steps; it's about describing how the number of steps increases as the data grows.

Here's how you can visualize these two types of algorithms:

- **O(N) - Linear Time:** If plotted on a graph, O(N) creates a diagonal line. For every additional piece of data, the algorithm takes one more step. The more data, the more steps.
- **O(1) - Constant Time:** If plotted on a graph, O(1) creates a horizontal line. No matter how much data, the number of steps remains the same.



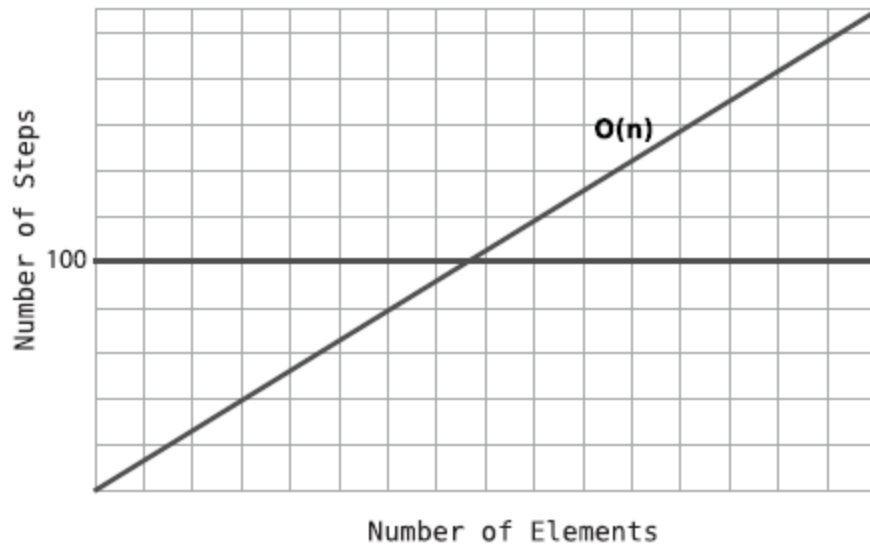
The soul of Big O is thus a story of how the algorithm responds to increasing amounts of data. It doesn't just tell you the steps but explains how those steps might increase or stay constant as the data grows. Whether an algorithm is O(1) or O(3) doesn't matter; they're essentially the same, as the steps remain constant irrespective of the data size. Conversely, O(N) describes a direct relationship between the data and the efficiency of the algorithm.

### Deeper into the Soul of Big O

The "soul of Big O" emphasizes how an algorithm's efficiency changes with the size of the data. Here's a more refined understanding:

- **Constant Time (O(1)):** Imagine an algorithm that always takes 100 steps, regardless of the data size. This is considered O(1).
- **Linear Time (O(N)):** An algorithm that takes as many steps as there are elements, represented by O(N).

Now, consider a graph plotting both of these. For fewer than 100 elements, the O(N) algorithm is more efficient, taking fewer steps. At exactly 100 elements, both algorithms take 100 steps. For anything greater than 100 elements, O(N) takes more steps.



Here's the crucial insight:

- **O(1) is Considered More Efficient:** Even if the O(1) algorithm took a large number of steps (such as 100 or even one million), it's still considered more efficient than O(N) in the grand scheme of things. There will always be a point where O(N) takes more steps and continues to grow less efficient as the data size increases towards infinity.

The "soul of Big O" is about understanding these types of relationships. It's not just about the specific number of steps but about how an algorithm's steps relate to the size of the data, and how that relationship scales as data grows. In this context, constant time algorithms (O(1)) are generally seen as more efficient than linear time algorithms (O(N)), regardless of the absolute number of steps in the constant time algorithm.

### Same Algorithm, Different Scenarios

Big O Notation in algorithms is often used to describe how the algorithm performs in different scenarios, particularly focusing on the worst-case scenario. For the example of linear search:

- **Best-Case Scenario (O(1)):** If the item is found in the first cell of the array, it takes just one step, and the efficiency is described as O(1).
- **Worst-Case Scenario (O(N)):** If the item is in the final cell, it takes as many steps as there are elements (N), and the efficiency is described as O(N).

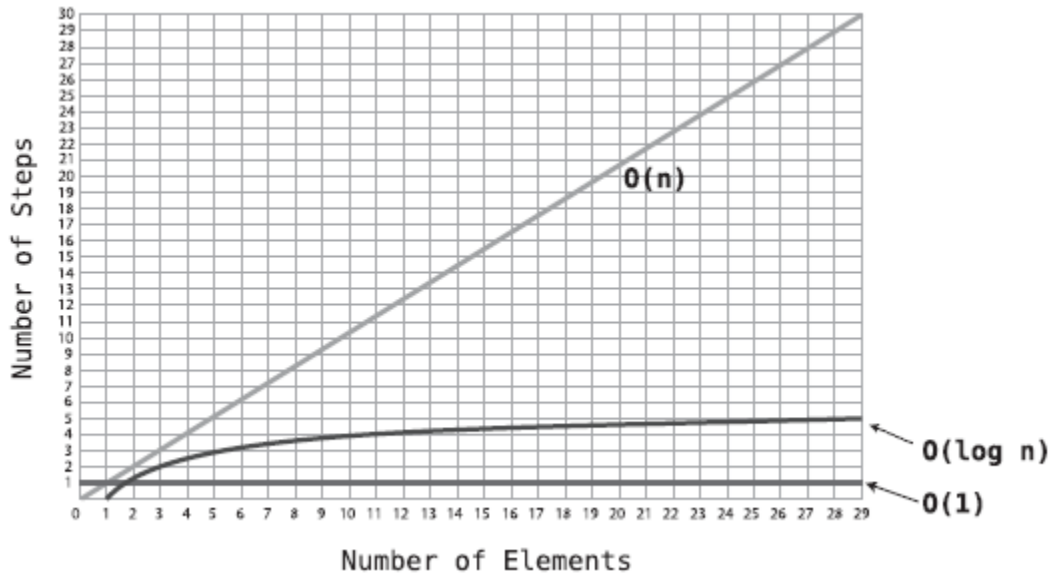
While linear search can be O(1) in the best case, most references will describe it as O(N). This is because:

- **Worst-Case Analysis is More Informative:** By understanding the worst-case scenario, we're prepared for the least efficient situation the algorithm may encounter. This "pessimistic" approach can guide choices in algorithm selection and help to ensure that we know the potential limits of the algorithm's efficiency.

### An Algorithm of the Third Kind

Binary search is an algorithm that is more efficient than linear search but not quite as fast as constant-time operations. Here's how it fits into the context of Big O Notation:

- **O(1)**: Describes algorithms that always take a constant number of steps, regardless of the number of data elements. It's the most efficient category.
- **O(log N)**: This describes binary search. The number of steps increases only slightly as the data size doubles. It's called "log time" because it corresponds to the logarithm of the number of data elements. It's more efficient than O(N) but less than O(1).
- **O(N)**: This describes linear search. The number of steps is directly proportional to the number of data elements. It's the least efficient among the three mentioned here.



The term "O(log N)" refers to the logarithmic relationship between the data size and the number of steps. As the data doubles, only one additional step is needed. On a graph, O(log N) would curve upward slightly, showing that it becomes less efficient as the data size grows, but not nearly as quickly as O(N). Binary search's time complexity of O(log N) means that it performs much better than linear search (O(N)) but isn't quite as efficient as constant time algorithms (O(1)). It represents a middle ground in efficiency, often making it a good choice for searching in sorted arrays.

### Logarithms

**Logarithms**, or "logs," are a mathematical concept that's the inverse of exponents. Here's a simpler explanation of what logs are and why they relate to algorithms like binary search in terms of O(log N):

- Exponents are about multiplying a number by itself a certain number of times. For example,  $2^3 = 2 \times 2 \times 2 = 8$ .
- Logarithms are the opposite. They ask how many times you have to multiply a number by itself to reach a particular value. So,  $\log_2 8 = 3$  because you have to multiply 2 by itself 3 times to get 8.
- Another way to understand logarithms is by thinking about halving. For example,  $\log_2 8 = 3$  can also be understood as the number of times you have to halve 8 to get to 1:  $(8 / 2 / 2 / 2 = 1)$ .

When it comes to Big O Notation, O(log N) describes algorithms that behave in this logarithmic way. Just like how you halve the number repeatedly in the logarithm, an algorithm like binary search keeps dividing the data

set in half with each step. The "log" in  $O(\log N)$  refers to this process of repeatedly halving the size of the problem, making it a very efficient way to search through large sorted lists.

### [O\(log N\) Explained](#)

In the realm of computer science,  $O(\log N)$  is a way to describe the efficiency of an algorithm, and it's a short form for  $O(\log_2 N)$ . Here's what it means in a more straightforward manner:

- $O(\log N)$  refers to the number of steps an algorithm takes based on the logarithm (base 2) of  $N$ , the number of data elements.
- If you have 8 elements, the algorithm would take 3 steps, since  $\log_2 8 = 3$ .
- In practical terms, think of it as how many times you need to keep dividing the data in half until you end up with just 1 element. Binary search is a common example of an algorithm that behaves this way.

Here's a comparison between  $O(N)$  and  $O(\log N)$  to help you see the difference:

- With 8 elements:  $O(N)$  takes 8 steps,  $O(\log N)$  takes 3 steps.
- With 16 elements:  $O(N)$  takes 16 steps,  $O(\log N)$  takes 4 steps.
- As you keep doubling the elements,  $O(N)$  also doubles the steps, but  $O(\log N)$  just adds one more step.

N Elements	O(N)	O(log N)
8	8	3
16	16	4
32	32	5
64	64	6
128	128	7
256	256	8
512	512	9
1024	1024	10

$O(\log N)$  is much more efficient as the number of elements grows, as it takes fewer steps than  $O(N)$  to process the same amount of data. It's an efficient way to describe algorithms that can manage large data sets by repeatedly halving the size of the problem.

### [Practical Examples](#)

#### 1. **Printing Items from a List:**

- Code Example:

```
things = ['apples', 'baboons', 'cribs', 'dulcimers']
for thing in things:
    print("Here's a thing: %s" % thing)
```

- Explanation: This code prints each item in a list using a simple loop. The efficiency of the algorithm is determined by the number of elements in the list. For 4 elements, it takes 4 steps; for 10 elements, it takes 10 steps.
- Big O Notation: Since the number of steps is directly related to the number of elements (N) in the list, the efficiency of this algorithm is  $O(N)$ .

## 2. Checking If a Number is Prime:

- Code Example:

```
def is_prime(number):
    for i in range(2, number):
        if number % i == 0:
            return False
    return True
```

- Explanation: This code checks if a given number is prime. It does this by dividing the number by every integer from 2 up to the given number and checking for a remainder. If there's no remainder for any division, the number is not prime.
- Big O Notation: Since the loop runs as many times as the number passed into the function (N), the efficiency is  $O(N)$ .

In both cases, the efficiency of the algorithm is determined by the number of steps it takes in relation to the input size (N), whether it's a list of elements or a single number. The key question for understanding the Big O Notation in these cases is identifying the primary piece of data (N) that affects the number of steps in the algorithm. Both examples follow a linear pattern, meaning they have a time complexity of  $O(N)$ .

## Wrapping Up

With Big O Notation, we have a consistent system that allows us to compare any two algorithms. With it, we will be able to examine real-life scenarios and choose between competing data structures and algorithms to make our code faster and able to handle heavier loads.

## Exercises

1. Use Big O Notation to describe the time complexity of the following function that determines whether a given year is a leap year:

```
function isLeapYear(year) {
  return (year % 100 === 0) ? (year % 400 === 0) : (year % 4 === 0);
}
```

The time complexity for this function is constant as it doesn't depend on the input size: **O(1)**

2. Use Big O Notation to describe the time complexity of the following function that sums up all the numbers from a given array:

```
function arraySum(array) {
  let sum = 0;
  for(let i = 0; i < array.length; i++) {
    sum += array[i];
  }
  return sum;
}
```

The time complexity depends on the length of the array, as the loop iterates over each element once: **O(N)**

3. The following function is based on the age-old analogy used to describe the power of compounding interest. Imagine you have a chessboard and put a single grain of rice on one square. On the second square, you put 2 grains of rice, since that is double the amount of rice on the previous square. On the third square, you put 4 grains. On the fourth square, you put 8 grains, and on the fifth square, you put 16 grains, and so on.

The following function calculates which square you'll need to place a certain number of rice grains. For example, for 16 grains, the function will return 5, since you will place the 16 grains on the fifth square. Use Big O Notation to describe the time complexity of this function, which is below:

```
function chessboardSpace(numberOfGrains) {
  let chessboardSpaces = 1;
  let placedGrains = 1;
  while (placedGrains < numberOfGrains) {
    placedGrains *= 2;
    chessboardSpaces += 1;
  }
  return chessboardSpaces;
}
```

The time complexity depends on how many times the number of grains can be halved until it reaches or exceeds the given value: **O(log N)**

4. The following function accepts an array of strings and returns a new array that only contains the strings that start with the character "a". Use Big O Notation to describe the time complexity of the function:

```
function selectAStrings(array) {
  let newArray = [];

  for(let i = 0; i < array.length; i++) {
    if (array[i].startsWith("a")) {
      newArray.push(array[i]);
    }
  }

  return newArray;
}
```

The time complexity depends on the length of the array since the loop iterates over each element once: **O(N)**

5. The following function calculates the median from an *ordered* array. Describe its time complexity in terms of Big O Notation:

```
function median(array) {
  const middle = Math.floor(array.length / 2);

  // If array has even amount of numbers:
  if (array.length % 2 === 0) {
    return (array[middle - 1] + array[middle]) / 2;
  } else { // If array has odd amount of numbers:
    return array[middle];
  }
}
```

The time complexity for this function is constant as it doesn't depend on the size of the input: **O(1)**



## CHAPTER 4: Speeding Up Your Code with Big O

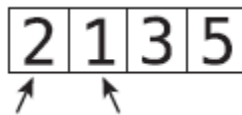
Big O Notation is like a rating system for the efficiency of algorithms. It helps you understand how fast or slow an algorithm is, depending on the size of the input. Consider two search methods: binary search and linear search. Binary search, being  $O(\log N)$ , is much quicker than linear search, which is  $O(N)$ . This comparison tells you that binary search is a better choice if speed matters.

But it doesn't stop there! With Big O, you can evaluate your own algorithm and compare it to known standards. If it turns out your algorithm falls into a "slow" category, you might want to revisit and optimize it. Sometimes, it might not be possible to make it faster, but it's always worth a try. In other words, Big O is not just a theoretical concept but a practical tool. You can use it to write more efficient code and then see if there are ways to make that code even more efficient. It's like a challenge that encourages continuous improvement in coding.

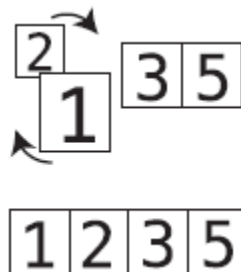
### Bubble Sort

Bubble Sort is a straightforward method for sorting an array of values in ascending order. It's one of the so-called "simple sorts" because of its ease of understanding, but it's not as efficient as some other algorithms. Here's how it works, broken down into simple steps:

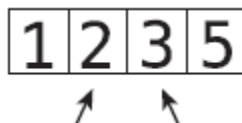
1. **Start at the Beginning:** Look at the first two values in the array.



2. **Compare and Swap if Necessary:** If the first value is greater than the second, swap them. If they're in the correct order, do nothing.



3. **Move Right:** Shift your focus one value to the right, so you're now looking at the second and third values.



4. **Repeat:** Keep comparing and swapping as needed, moving one value to the right each time, until you reach the end of the array or the already sorted portion.
5. **Start Over:** Once you've reached the end, start again from the beginning of the array.
6. **Keep Going Until Done:** Repeat the entire process until you go through the entire array without making a single swap. That means the array is sorted, and you're finished.

This process is called "Bubble Sort" because the larger values gradually "bubble up" to the end of the array as you go through these steps. While this method is relatively simple to understand and implement, it's not always the fastest way to sort an array, especially with a large number of elements.

### Bubble Sort in Action

Bubble Sort is a method used to sort an array into ascending order. Here's a simplified walk-through of how it works using the array [4, 2, 7, 1, 3]:

1. **Start with the array:** [4, 2, 7, 1, 3].
2. **Compare the first two numbers:** 4 and 2 are out of order, so swap them.
3. **Continue through the array:**
  - Compare 4 and 7: correct order.
  - Compare 7 and 1: out of order, so swap.
  - Compare 7 and 3: out of order, so swap.
  - The first pass-through is complete, and 7 has "bubbled" to its correct position.
4. **Repeat the process:**
  - Compare 2 and 4: correct order.
  - Compare 4 and 1: out of order, so swap.
  - Compare 4 and 3: out of order, so swap.
  - The second pass-through is complete, and 4 is in its correct position.
5. **Continue with further pass-throughs:**
  - Compare 2 and 1: out of order, so swap.
  - Compare 2 and 3: correct order.
  - The third pass-through is complete, and 3 is in its correct position.
6. **Final pass-through:** Compare 1 and 2; they are in correct order. No more swaps are needed.
7. **Result:** The array is completely sorted: [1, 2, 3, 4, 7].

### Code Implementation: Bubble Sort

1. **Initialize:** Start by determining the rightmost unsorted index of the array, which is initially the final index in the array. Also, create a boolean variable **sorted** to keep track of whether the array is fully sorted.

```
def bubble_sort(list):  
    unsorted_until_index = len(list) - 1  
    sorted = False
```

2. **Main Loop:** Begin a **while** loop that continues as long as the array is not sorted. Each round of this loop represents a pass-through of the array.

```
while not sorted:
    sorted = True
```

3. **Assume Sorted:** Initially, assume the array is sorted in each pass-through.
4. **Compare and Swap:** Use a **for** loop to compare each pair of adjacent values, and swap them if they're out of order. If a swap occurs, set the **sorted** variable to **False**.

```
for i in range(unsorted_until_index):
    if list[i] > list[i+1]:
        list[i], list[i+1] = list[i+1], list[i]
        sorted = False
```

5. **Decrement Unsorted Index:** Since the value bubbled up to the right is now in its correct position, decrement the **unsorted\_until\_index** by 1, as that index is now sorted.

```
unsorted_until_index -= 1
```

6. **Return Sorted Array:** The **while** loop ends once **sorted** is **True**, meaning the array is completely sorted. Return the sorted array.

```
return list
```

7. **Usage:** Call the function with an unsorted array.

```
print(bubble_sort([65, 55, 45, 35, 25, 15, 10]))
```

This function systematically compares and swaps adjacent elements, ensuring that each pass-through places the next largest unsorted value in its correct position, until the entire array is sorted.

## The Efficiency of Bubble Sort

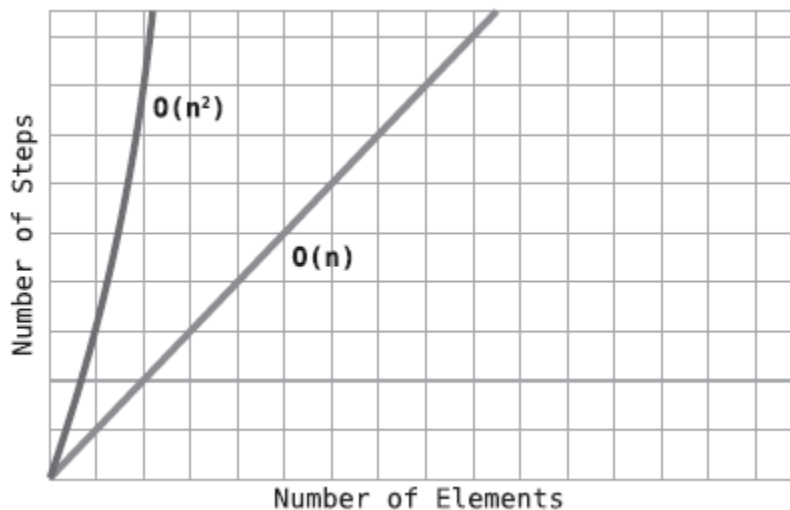
The efficiency of Bubble Sort can be understood by analyzing its behavior in terms of comparisons and swaps.

1. **Comparisons:** Bubble Sort makes comparisons between pairs of numbers to determine their order. For an array with  $N$  elements, the comparisons are made as:
  - First pass-through:  $N - 1$  comparisons
  - Second pass-through:  $N - 2$  comparisons
  - ...
  - Last pass-through: 1 comparison
  - Total:  $(N - 1) + (N - 2) + \dots + 1$  comparisons.
2. **Swaps:** In the worst-case scenario (array sorted in reverse order), Bubble Sort requires a swap for each comparison. So the number of swaps equals the number of comparisons.
3. **Total Steps:** For an array with  $N$  elements, the total number of steps (comparisons + swaps) is given by:

- 5 elements: 20 steps
- 10 elements: 90 steps
- 20 elements: 380 steps
- ...
- The pattern continues as  $N^2$ , growing quadratically.

N Data Elements	# of Bubble Sort Steps	$N^2$
5	20	25
10	90	100
20	380	400
40	1560	1600
80	6320	6400

4. **Time Complexity:** The time complexity of Bubble Sort, therefore, is represented by  $O(N^2)$ , which is considered relatively inefficient, especially for large datasets.
5. **Graphical Representation:** If you visualize this, the  $O(N^2)$  curve sharply rises compared to a linear  $O(N)$  curve. While a linear algorithm grows in direct proportion to the size of the input, a quadratic algorithm grows much faster.



6. **Terminology:**  $O(N^2)$  is often referred to as quadratic time, indicating the nature of its growth.

### A Quadratic Problem

You have an array of ratings (from 1 to 10), and you want to determine if there are any duplicate numbers within the array.

```
function hasDuplicateValue(array) {
  for(let i = 0; i < array.length; i++) {
    for(let j = 0; j < array.length; j++) {
      if(i !== j && array[i] === array[j]) {
        return true;
      }
    }
  }
  return false;
}
```

- Using nested loops, a function iterates through the array comparing every element with every other element.
- If any duplicates are found, it returns true; otherwise, it returns false after checking all possibilities.

#### Efficiency Analysis:

- This nested loop approach takes  $N^2$  steps, where  $N$  is the size of the array.
- You can track the number of steps by adding a counter in the nested loops.
- Testing this approach will confirm that it always results in  $N^2$  comparisons.
- $O(N^2)$  is often a sign of inefficiency, especially when nested loops are used.

#### Concerns:

- $O(N^2)$  algorithms can become slow as the size of the input grows.
- Recognizing this pattern should prompt you to look for more efficient alternatives.

#### A Linear Solution

##### New Function:

```
function hasDuplicateValue(array) {
  let existingNumbers = [];
  for(let i = 0; i < array.length; i++) {
    if(existingNumbers[array[i]] === 1) {
      return true;
    } else {
      existingNumbers[array[i]] = 1;
    }
  }
  return false;
}
```

- An empty array **existingNumbers** is created.
- A loop iterates through the given array.
- For each number in the array, the function places a value of 1 at the corresponding index in **existingNumbers**.

- If a 1 is found at that index before placing it, it means the number is a duplicate, and the function returns true.
- If the loop completes without finding duplicates, the function returns false.

**How It Works (Example):**

- For an array [3, 5, 8], **existingNumbers** would look like [undefined, undefined, undefined, 1, undefined, 1, undefined, undefined, 1].
- The indexes with value 1 correspond to the numbers in the original array, indicating that they've been encountered.

**Efficiency Analysis:**

- This algorithm is  $O(N)$ , where  $N$  is the number of data elements.
- There's only one loop, making  $N$  comparisons for  $N$  elements.
- You can confirm this by tracking steps, and you'll find that the number of steps is equal to the size of the array.

**Advantages:**

- This  $O(N)$  algorithm is much faster than the previous  $O(N^2)$  algorithm, resulting in a significant speed boost.

**Disadvantage:**

- Consumes more memory than the initial approach due to the creation of the **existingNumbers** array.

The new linear approach avoids nested loops, making it a more efficient solution in terms of time complexity, but with the trade-off of increased memory consumption.

[Wrapping Up](#)

It's clear that having a solid understanding of Big O Notation can enable you to identify slow code and select the faster of two competing algorithms. However, there are situations in which Big O Notation will have us believe that two algorithms have the same speed, while one is actually faster. In the next chapter, you're going to learn how to evaluate the efficiencies of various algorithms even when Big O isn't nuanced enough to do so.

[Exercises](#)

1. Replace the question marks in the following table to describe how many steps occur for a given number of data elements across various types of Big O:

N Elements	$O(N)$	$O(\log N)$	$O(N^2)$
100	100	?	?
2000	?	?	?

- For  $O(N)$ , the number of steps is equal to the number of elements.

- For  $O(\log N)$ , the number of steps is the logarithm to the base 2 of the number of elements.
- For  $O(N^2)$ , the number of steps is the square of the number of elements.

N Elements	$O(N)$	$O(\log N)$	$O(N^2)$
100	100	7	10,000
2000	2000	11	4,000,000

2. If we have an  $O(N^2)$  algorithm that processes an array and find that it takes 256 steps, what is the size of the array?

Since the time complexity is  $O(N^2)$ , we can find the size of the array by taking the square root of the number of steps.

- $N = \sqrt{256} = 16$

3. Use Big O Notation to describe the time complexity of the following function. It finds the greatest product of any pair of two numbers within a given array:

```
def greatestProduct(array):
    greatestProductSoFar = array[0] * array[1]

    for i, iVal in enumerate(array):
        for j, jVal in enumerate(array):
            if i != j and iVal * jVal > greatestProductSoFar:
                greatestProductSoFar = iVal * jVal

    return greatestProductSoFar
```

The given code has two nested loops that both iterate through the array. Since there are two nested loops running  $N$  times each, the time complexity is  $O(N^2)$ .

4. The following function finds the greatest single number within an array, but has an efficiency of  $O(N^2)$ . Rewrite the function so that it becomes a speedy  $O(N)$ :

```
def greatestNumber(array):
    for i in array:
        # Assume for now that i is the greatest:
        isIValTheGreatest = True

        for j in array:
            # If we find another value that is greater than i,
            # i is not the greatest:
            if j > i:
                isIValTheGreatest = False

        # If, by the time we checked all the other numbers, i
        # is still the greatest, it means that i is the greatest number:
        if isIValTheGreatest:
            return i
```

```
def greatestNumber(array):
    greatest = array[0]
    for i in array:
        if i > greatest:
            greatest = i
    return greatest
```

This code iterates through the array only once and keeps track of the greatest number found, resulting in a time complexity of  $O(N)$ .



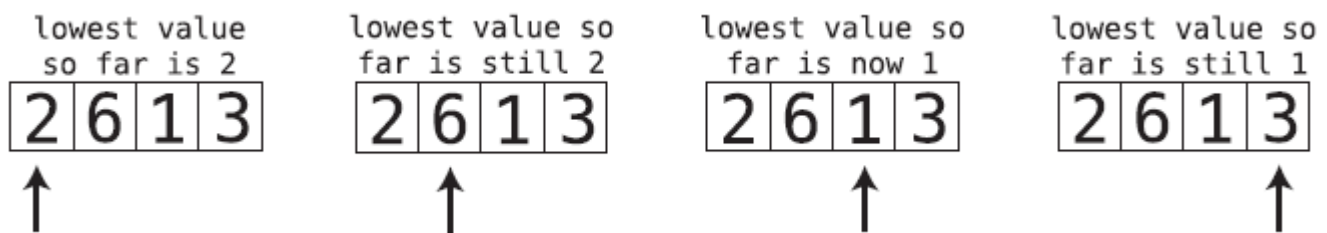
## CHAPTER 5: Optimizing Code with and Without Big O

We've seen that Big O Notation is a great tool for comparing algorithms and determining which algorithm should be used for a given situation. However, it's certainly not the *only* tool. In fact, there may be times when two competing algorithms are described in the same way using Big O, yet one algorithm is faster than the other. In this chapter, you're going to learn how to discern between two algorithms that *seem* to have the same efficiency, and how to select the faster of the two.

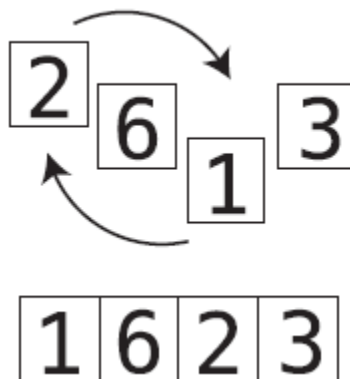
### Selection Sort

In the last chapter, we looked at Bubble Sort, a sorting method with an efficiency of  $O(N^2)$ . Now, we're going to study another sorting technique called Selection Sort and compare it to Bubble Sort. Here's how Selection Sort works:

1. **Find the Lowest Value:** Scan the array from left to right, keeping track of the lowest value and its index in a variable. If a lower value is found, replace the index in the variable with the new one.



2. **Swap with Starting Value:** Once the index with the lowest value is found, swap it with the value at the beginning of the current pass-through. In the first pass-through, this would be index 0, index 1 in the second, and so on.



3. **Repeat Pass-throughs:** Keep repeating steps 1 and 2, moving through the array until you reach the end. By then, the array will be fully sorted.

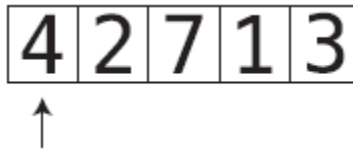
These steps gradually move the smallest elements to the front of the array, sorting it in ascending order.

### Selection Sort in Action

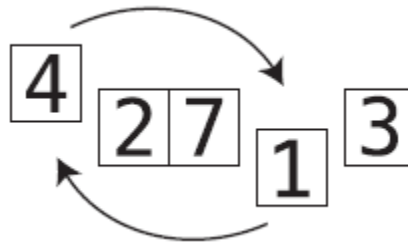
Selection Sort can be understood more clearly through a step-by-step example using the array **[4, 2, 7, 1, 3]**. Here's how the process works:

### First Pass-through:

- Start with index 0, which is 4. This is the lowest value so far.



- Compare 4 with 2, 2 is lower.
- Compare 2 with 7, 2 remains lowest.
- Compare 2 with 1, 1 is lower.
- Compare 1 with 3, 1 remains lowest.
- Swap 1 with the value at index 0.



### Second Pass-through:

- Start with index 1, which is 2. This is the lowest value.



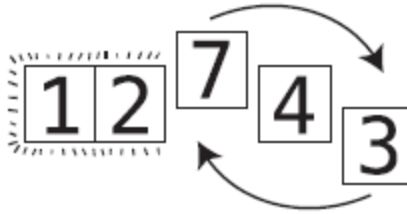
- Compare 2 with 7, 2 remains lowest.
- Compare 2 with 4, 2 remains lowest.
- Compare 2 with 3, 2 remains lowest.
- No swap needed; 2 is already in the correct place.

### Third Pass-through:

- Start with index 2, which is 7. This is the lowest value.



- Compare 7 with 4, 4 is lower.
- Compare 4 with 3, 3 is lower.
- Swap 3 with the value at index 2.



#### Fourth Pass-through:

- Start with index 3, which is 4. This is the lowest value.



- Compare 4 with 7, 4 remains lowest.
- No swap needed; 4 is already in the correct place.

Since every element except the last one is in the correct place, the array is now sorted. The steps taken by Selection Sort are straightforward but iterative. The lowest value in the unsorted section of the array is found and swapped with the first unsorted value. This continues until the array is fully sorted.

#### Code Implementation: Selection Sort

1. **Loop Through the Array:** There's a loop that runs through each element of the array except the last one. The variable `i` represents the current position in the array:

```
for(let i = 0; i < array.length - 1; i++) {
```

2. **Find the Lowest Number's Index:** Inside the loop, we start by assuming the current number is the lowest and keep track of its index:

```
let lowestNumberIndex = i;
```

3. **Compare with Other Numbers:** A nested loop compares the current lowest number with the remaining numbers in the array. If a lower number is found, its index is stored:

```
for(let j = i + 1; j < array.length; j++) {
  if(array[j] < array[lowestNumberIndex]) {
    lowestNumberIndex = j;
  }
}
```

4. **Swap if Necessary:** If the lowest number is not at the current index, they are swapped:

```
if(lowestNumberIndex !== i) {
  let temp = array[i];
  array[i] = array[lowestNumberIndex];
```

```
        array[lowestNumberIndex] = temp;
    }
```

5. **Return Sorted Array:** Finally, the sorted array is returned:

```
    return array;
```

The code defines a function that iterates through an array and, within each iteration, finds the lowest number and swaps it to its proper place. By doing this repeatedly, it eventually sorts the whole array.

### The Efficiency of Selection Sort

Selection Sort's efficiency can be understood by looking at the number of comparisons and swaps it makes:

#### Comparisons:

- For an array with  $N$  elements, the total number of comparisons is the sum of the first  $(N - 1)$  natural numbers, or  $(N - 1) + (N - 2) + (N - 3) \dots + 1$ .
- Using the example array with 5 elements, the comparisons are:
  - 1st pass-through: 4 comparisons
  - 2nd pass-through: 3 comparisons
  - 3rd pass-through: 2 comparisons
  - 4th pass-through: 1 comparison
  - Total:  $4 + 3 + 2 + 1 = 10$  comparisons

#### Swaps:

- Selection Sort makes at most one swap per pass-through.

Comparing Selection Sort to Bubble Sort shows it's generally more efficient:

- For 5 elements, Bubble Sort takes 20 steps, while Selection Sort takes 14 (10 comparisons + 4 swaps).
- For 10 elements, it's 90 vs 54.
- For 20 elements, it's 380 vs 199.
- For 40 elements, it's 1560 vs 819.
- For 80 elements, it's 6320 vs 3239.

N Elements	Max # of Steps in Bubble Sort	Max # of Steps in Selection Sort
5	20	14 (10 comparisons + 4 swaps)
10	90	54 (45 comparisons + 9 swaps)
20	380	199 (180 comparisons + 19 swaps)
40	1560	819 (780 comparisons + 39 swaps)
80	6320	3239 (3160 comparisons + 79 swaps)

These numbers indicate that Selection Sort is about twice as fast as Bubble Sort, requiring roughly half the number of steps. This makes it a more efficient algorithm in comparison.

### Ignoring Constants

Big O Notation is a way to describe the efficiency of an algorithm, but it simplifies the description by ignoring constants and focusing only on the most significant factors that affect growth. Here's what's happening with Selection Sort and Bubble Sort:

1. **Selection Sort:** Though it takes about  $N^2/2$  steps for  $N$  data elements, in Big O Notation, we ignore the  $"/2"$  constant and describe it as  $O(N^2)$ .
2. **Bubble Sort:** It's also described as  $O(N^2)$ .

This might seem counterintuitive because Selection Sort is roughly twice as fast as Bubble Sort, but both are described the same way in Big O. Why does Big O ignore constants?

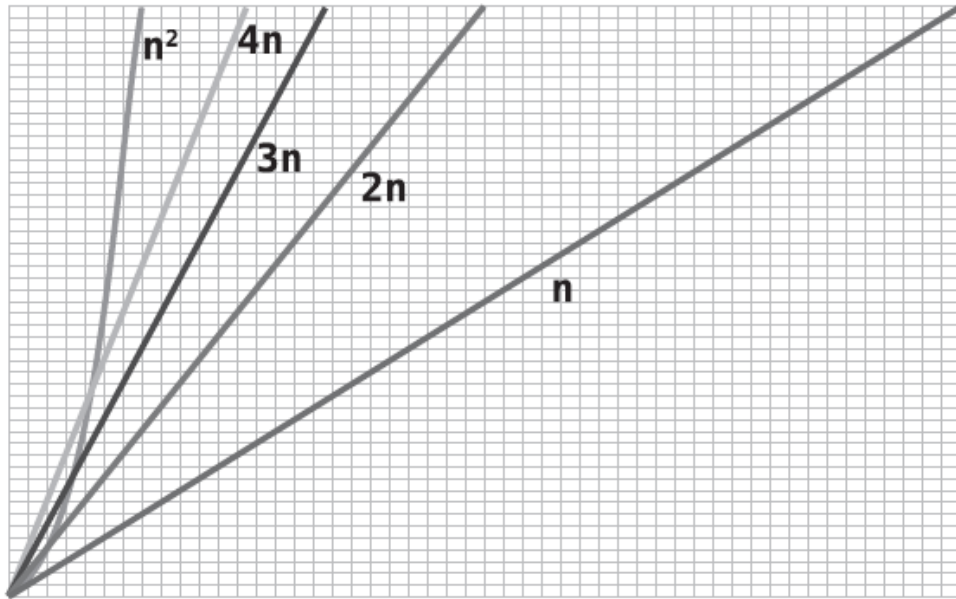
- It focuses on the high-level growth rate, looking at how the algorithm scales as the input size  $N$  grows towards infinity.
- Constants might differ based on hardware or other low-level details, so ignoring them makes the notation more broadly applicable.
- In practice, when comparing two algorithms, other factors might be considered alongside Big O, such as actual run-time performance or ease of implementation.

So, while ignoring constants might seem like a limitation, it actually makes Big O a useful tool for comparing algorithms at a high level, without getting bogged down in specific details that might vary from one context to another. Even though Selection Sort and Bubble Sort are described the same way in Big O, practitioners would consider other aspects to choose between them, such as the actual run-time performance or specific requirements of a given application.

### Big O Categories

Big O Notation categorizes algorithms into general speed classifications, similar to how buildings can be classified into general categories like single-family homes or skyscrapers. Here's a simple breakdown:

1. **General Categories:** Just as we might group buildings into houses or skyscrapers without worrying about specific details like the number of floors, Big O groups algorithms into broad categories based on their growth rate.
2. **Growth Patterns:** Algorithms are categorized by how their steps increase as the data size grows. An  $O(N)$  algorithm grows linearly with  $N$ , while an  $O(N^2)$  algorithm grows exponentially. These represent two entirely different categories.



3. **Ignoring Constants:** Constants like 2 in  $O(2N)$  or 100 in  $O(100N)$  don't change the category. They all fall under  $O(N)$ . Just as the specific number of floors in a house doesn't change the fact that it's a house, these constants don't change the general category of growth.
4. **Comparing Algorithms:** Big O is great for contrasting algorithms in different categories (e.g.,  $O(N)$  vs.  $O(N^2)$ ). It gives you a high-level understanding of how they'll perform as the data size grows. It's like knowing that a skyscraper is inherently different from a house.
5. **Limitations:** However, when two algorithms fall into the same Big O category, it doesn't tell the whole story. For example, both Bubble Sort and Selection Sort are  $O(N^2)$ , but Selection Sort is twice as fast. In cases like this, more detailed analysis is needed, just as you might need to know more details to choose between two houses.
6. **Conclusion:** Think of Big O categories as a way to identify the "type" of building an algorithm is. It gives you a useful, high-level perspective, but sometimes you'll need to look closer to understand the nuances and specific differences between algorithms in the same category.

### A Practical Example

Here are two code functions designed to print all even numbers up to a given limit. Let's break down the examples to understand it more simply:

1. **Version One:** This code iterates from 2 to the upper limit, checking if each number is even before printing it. It increments the number by 1 in each step, taking about  $N$  steps, where  $N$  is the upper limit. The time complexity is  $O(N)$ .

```
def print_numbers_version_one(upperLimit):
    number = 2

    while number <= upperLimit:
        # If number is even, print it:
        if number % 2 == 0:
            print(number)

        number += 1
```

2. **Version Two:** This code also iterates from 2 to the upper limit but directly prints each number. It increments by 2 each time, so it only prints even numbers. It takes  $N/2$  steps, but in Big O, we ignore the constant factor of  $1/2$ , so the time complexity is also  $O(N)$ .

```
def print_numbers_version_two(upperLimit):
    number = 2

    while number <= upperLimit:
        print(number)

        # Increase number by 2, which, by definition,
        # is the next even number:
        number += 2
```

3. **Comparison:** Both algorithms fall into the  $O(N)$  category, so Big O doesn't show the difference between them. However, version two is actually twice as fast, since it takes half as many steps.
4. **Big O Limitation:** This example illustrates that while Big O gives a broad understanding of the algorithm's efficiency, it might not capture all nuances. Two algorithms might be in the same Big O category but still have different speeds. In such cases, you need to look beyond Big O to understand the differences.
5. **Choice:** If you had to choose between these two algorithms, the second one would be better because it's more efficient, even though both are classified as  $O(N)$  in Big O notation.

Both functions do the same job, but the second one does it in half the time, even though they're both categorized the same way in Big O terms. It's like having two routes to a destination that are about the same length, but one takes half as long because it's a faster road.

### Significant Steps

In analyzing algorithms, it's not just about counting how many times a loop runs but what happens within each loop. This example emphasizes that each part of the loop has its contribution to the total steps but demonstrates how Big O notation abstracts these details. Let's look at this more simply:

1. **Loop Components:** In the `print_numbers_version_one` function, the loop has three main steps:
  - **Comparison:** Checks if a number is even (`number % 2 == 0`). This happens  $N$  times.
  - **Printing:** Prints the even numbers. This happens  $N/2$  times, since only even numbers are printed.

- **Incrementing:** Adds 1 to the number in each loop. This also happens  $N$  times.
2. **Total Steps:** If you add up all these steps, you have  $N$  comparisons,  $N$  incrementings, and  $N/2$  printings, adding up to  $2.5N$  steps in total.
  3. **Big O Simplification:** When we apply Big O notation, we ignore the constant factor (in this case, 2.5), leading to a time complexity of  $O(N)$ .
  4. **Significance:** All the steps are significant in understanding how the algorithm works, but in Big O terms, they are all summed up into a single expression. The focus is more on the growth rate, rather than the specific steps.

## Wrapping Up

We now have some powerful analysis tools at our disposal. We can use Big O to broadly determine the efficiency of an algorithm, and we can also compare two algorithms that fall within one classification of Big O.

However, another important factor must be taken into account when comparing the efficiencies of two algorithms. Until now, we've focused on how fast an algorithm is in a worst-case scenario. Now, worst-case scenarios, by definition, don't happen all the time. On average, most scenarios that occur are...well...average-case scenarios. In the next chapter, you'll learn how to take all scenarios into account.

## Exercises

The following exercises provide you with the opportunity to practice analyzing algorithms.

1. Use Big O Notation to describe the time complexity of an algorithm that takes  $4N + 16$  steps.

Since Big O Notation ignores constants, the time complexity is  $O(N)$ .

2. Use Big O Notation to describe the time complexity of an algorithm that takes  $2N^2$ .

Again, the constant is ignored, so the time complexity is  $O(N^2)$ .

3. Use Big O Notation to describe the time complexity of the following function, which returns the sum of all numbers of an array after the numbers have been doubled:



```

def double_then_sum(array)
  doubled_array = []

  array.each do |number|
    doubled_array << number * 2
  end

  sum = 0

  doubled_array.each do |number|
    sum += number
  end

  return sum
end

```

This function iterates over the array twice. In the first iteration, it doubles each number, and in the second iteration, it sums them up. Since the constants are ignored, and both operations are linear, the time complexity is  $O(N)$ .

4. Use Big O Notation to describe the time complexity of the following function, which accepts an array of strings and prints each string in multiple cases:

```

def multiple_cases(array)
  array.each do |string|
    puts string.upcase
    puts string.downcase
    puts string.capitalize
  end
end

```

This function iterates through the array of strings and performs three print operations for each string. Since the constants are ignored and the operations are done linearly with respect to the number of strings, the time complexity is  $O(N)$ .

5. The next function iterates over an array of numbers, and for each number whose index is even, it prints the sum of that number plus every number in the array. What is this function's efficiency in terms of Big O Notation?

```
def every_other(array)
  array.each_with_index do |number, index|
    if index.even?
      array.each do |other_number|
        puts number + other_number
      end
    end
  end
end
```

This function contains a nested loop. For every even-indexed element in the array, it iterates over all the elements in the array again. This means that for  $N/2$  elements (the even indices), it performs  $N$  operations, leading to a time complexity of  $O(N^2)$ .

## CHAPTER 6: Optimizing for Optimistic Scenarios

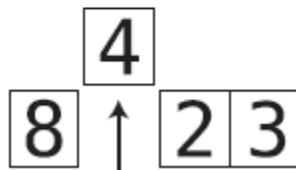
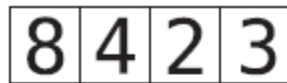
Until this point, we've focused primarily on how many steps an algorithm would take in a worst-case scenario. The rationale behind this is simple: if you're prepared for the worst, things will turn out okay. However, you'll discover in this chapter that the worst-case scenario isn't the only situation worth considering. Being able to consider all scenarios is an important skill that can help you choose the appropriate algorithm for every situation.

### Insertion Sort

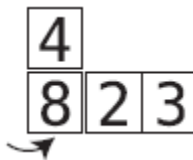
**Insertion Sort** is a simple sorting algorithm that builds the final sorted array (or list) one item at a time. Here's a simplified explanation of the process:

#### 1. First Pass-Through:

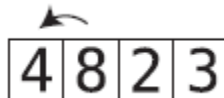
- Remove the second item in the array and keep it in a temporary variable.



- Compare this temporary value to the one to the left of the gap.
- If the left value is greater, shift it to the right.



- Repeat until a lower value is encountered or the left end of the array is reached.
- Insert the temporary value into the gap.

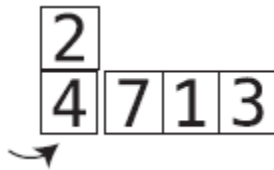
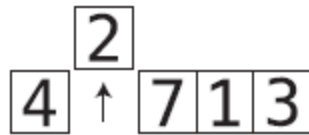


#### 2. Subsequent Pass-Throughs:

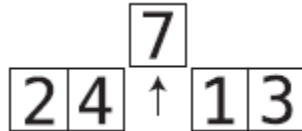
- Repeat these steps for each subsequent value in the array.

The given example with the array **[4, 2, 7, 1, 3]** illustrates the process:

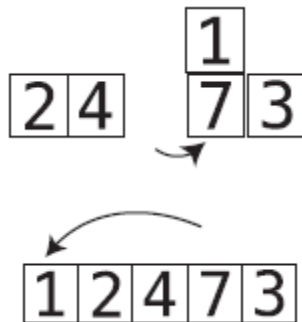
- First, 2 is removed, and values are shifted to make way for it at the front of the array.



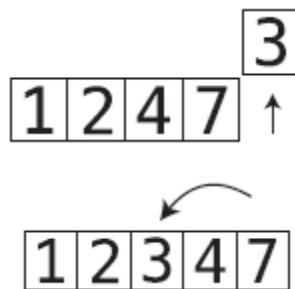
- Then, 7 is handled but doesn't need to be shifted.



- Next, 1 is removed, and several shifts occur to position it at the start.



- Finally, 3 is removed, and shifts are performed to insert it in its proper place.



The array is now sorted as **[1, 2, 3, 4, 7]**.

Here's why Insertion Sort can be efficient in some cases:

- **Simplicity:** It's straightforward to understand and implement.
- **Adaptive:** It becomes more efficient when dealing with a partially sorted array.
- **In-Place:** It requires only a constant amount of additional memory space.

While the worst-case efficiency is still  $O(N^2)$ , in practice, it can outperform other  $O(N^2)$  sorting algorithms like Bubble Sort and Selection Sort. It does so by reducing the number of shifts and comparisons required, especially if many elements are already in their sorted positions. This algorithm shines in scenarios where you

are sorting small lists or adding items to an already sorted list, and simplicity is more valuable than the absolute fastest performance.

### Code Implementation: Insertion Sort

Let's break down the Python code for Insertion Sort:

```
def insertion_sort(array):
    for index in range(1, len(array)):
        temp_value = array[index]
        position = index - 1

        while position >= 0:
            if array[position] > temp_value:
                array[position + 1] = array[position]
                position = position - 1
            else:
                break

        array[position + 1] = temp_value

    return array
```

1. **Loop Through Array:** Start from the second element (index 1), and loop through the entire array. Each loop represents a pass-through.
2. **Save Current Value:** Inside each pass-through, save the current value in a variable called **temp\_value**.
3. **Set Position to Compare:** Create a variable called **position**, starting immediately to the left of the current index. This will be used to compare values to the left of **temp\_value**.
4. **Inner While Loop:** Continue as long as **position** is greater or equal to 0.
  - **Compare Values:** If the value at **position** is greater than **temp\_value**, shift it to the right by one position.
  - **Move Left:** Decrement **position** by 1, to compare the next left value against **temp\_value**.
  - **End Pass-through:** If you reach a value not greater than **temp\_value**, it's time to place **temp\_value** in the gap, so break the loop.
5. **Insert Temp Value:** Place the **temp\_value** into the current gap (position + 1).
6. **Return Sorted Array:** After all pass-throughs are done, return the sorted array.

### The Efficiency of Insertion Sort

Here's a breakdown of the efficiency of Insertion Sort:

1. **Four Key Actions:** The operations that Insertion Sort performs include removals, comparisons, shifts, and insertions.

## 2. Comparisons and Shifts:

- **Worst-case Scenario:** When the array is sorted in reverse order, every number has to be compared to every number on its left. This is done in each pass-through.
- **Total Comparisons:** It's calculated by adding up each pass-through's comparisons ( $1 + 2 + 3 + \dots + (N - 1)$ ). For  $N$  elements, it's roughly  $N^2 / 2$  comparisons.
- **Total Shifts:** There are as many shifts as comparisons.

3. **Removals and Insertions:** Happens once per pass-through, equaling  $N - 1$  removals and insertions.

4. **Total Steps:** The total amount of work is the sum of all these parts, simplified to  $N^2 + 2N - 2$  steps.

## 5. Big O Notation and Time Complexity:

- **Ignoring Constants:** Big O ignores constant values, simplifying to  $O(N^2 + N)$ .
- **Only Highest Order Matters:** Since we only consider the most significant order of  $N$ , this reduces further to  $O(N^2)$ .
- **Comparison with Other Algorithms:** This places Insertion Sort in the same category as Bubble Sort and Selection Sort. They all have  $O(N^2)$  complexity.

## 6. Additional Insights:

- **Relative Speed:** Despite the same time complexity, these algorithms may not all be equally fast. It's not as simple as just looking at the Big O notation; more factors might affect their relative speeds.

Insertion Sort has a worst-case time complexity of  $O(N^2)$ , which means that its efficiency decreases as the size of the sorted list increases. Though it falls into the same complexity category as other basic sorting algorithms, different practical factors might affect its actual speed relative to others.

## The Average Case

Here's a breakdown of Insertion Sort's efficiency in different scenarios:

1. **Worst-Case Scenario:** Insertion Sort is slower in the worst-case scenario (when the array is sorted in descending order), requiring  $N^2$  steps, which is the same for both comparisons and shifts.
2. **Best-Case Scenario:** When the data is already sorted in ascending order, it requires only one comparison per pass-through and no shifts at all. This leads to  $N$  steps, where  $N$  is the number of elements in the array.
3. **Average-Case Scenario:**
  - Real-world data is more likely to be randomly sorted, not in perfect ascending or descending order.
  - In an average scenario, the number of comparisons and shifts can vary greatly. Sometimes all the data is compared and shifted, sometimes some, sometimes none.
  - Overall, the average case can be considered to take about  $N^2 / 2$  steps.

#### 4. Comparison with Selection Sort:

- Selection Sort always takes  $N^2 / 2$  steps, whether it's the best, worst, or average case. It doesn't have a mechanism to end a pass-through early.
- Insertion Sort's efficiency varies:  $N$  steps in the best case,  $N^2 / 2$  in the average case, and  $N^2$  in the worst case.

#### 5. Which is Better?:

- **Similar in Average Case:** Both algorithms perform similarly if the array is randomly sorted.
- **Insertion Sort for Mostly Sorted Data:** If the data is likely to be mostly sorted, Insertion Sort is a better choice.
- **Selection Sort for Mostly Reverse Sorted Data:** If the data is likely to be mostly sorted in reverse order, Selection Sort is faster.
- **No Clear Idea about Data:** If you're uncertain about the data's order, both algorithms will perform equally in an average case scenario.

#### A Practical Example

You're writing a JavaScript application, and you need to find the intersection between two arrays, meaning the common values that occur in both arrays. For example, if you have **[3, 1, 4, 2]** and **[4, 5, 3, 6]**, the intersection is **[3, 4]**. The initial implementation uses two nested loops, iterating over each value in both arrays and comparing them to find matching values:

```
function intersection(firstArray, secondArray){
  let result = [];
  for (let i = 0; i < firstArray.length; i++) {
    for (let j = 0; j < secondArray.length; j++) {
      if (firstArray[i] == secondArray[j]) {
        result.push(firstArray[i]);
      }
    }
  }
  return result;
}
```

#### Efficiency:

- **Comparisons:** If both arrays are of equal size  $N$ , the comparisons performed are  $N^2$ .
- **Insertions:** At most,  $N$  steps if both arrays are identical.
- **Overall Efficiency:**  $O(N^2)$ , or  $O(N * M)$  if the arrays are of different sizes ( $N$  and  $M$ ).

The initial implementation has a drawback: it always makes  $N^2$  comparisons, even when a common value is found between the two arrays. A better approach is to cut the inner loop short as soon as a common value is found. Adding a **"break"** keyword makes this optimization:

```
function intersection(firstArray, secondArray){
  let result = [];
  for (let i = 0; i < firstArray.length; i++) {
    for (let j = 0; j < secondArray.length; j++) {
      if (firstArray[i] == secondArray[j]) {
        result.push(firstArray[i]);
        break;
      }
    }
  }
  return result;
}
```

#### Updated Efficiency:

- **Worst-Case Scenario:** No shared values,  $N^2$  comparisons.
- **Best-Case Scenario:** Identical arrays,  $N$  comparisons.
- **Average Case:** Different but shared values, between  $N$  and  $N^2$  comparisons.

By adding a simple "break" statement, the algorithm's efficiency improves significantly in scenarios where common values are found between the arrays. It avoids unnecessary comparisons, resulting in a more efficient and optimized function.

#### Wrapping Up

Being able to discern between best-, average-, and worst-case scenarios is a key skill in choosing the best algorithm for your needs, as well as taking existing algorithms and optimizing them further to make them significantly faster. Remember, while it's good to be prepared for the worst case, average cases are what happen most of the time. Now that we've covered the important concepts related to Big O Notation, let's apply our knowledge to practical algorithms. In the next chapter, we're going to take a look at various everyday algorithms that may appear in real codebases, and identify the time complexity of each one in terms of Big O.

#### Exercises

The following exercises provide you with the opportunity to practice with optimizing for best- and worst-case scenarios.



1. Use Big O Notation to describe the efficiency of an algorithm that takes  $3N^2 + 2N + 1$  steps.

In Big O Notation, you generally focus on the highest-order term, since it will dominate as  $N$  grows larger. In this case, the highest-order term is  $N^2$ , so the efficiency of the algorithm can be described as  $O(N^2)$ .

2. Use Big O Notation to describe the efficiency of an algorithm that takes  $N + \log N$  steps.

$\log N$  is a lower order than  $N$ , so it's simply reduced to  $O(N)$ .

3. The following function checks whether an array of numbers contains a pair of two numbers that add up to 10.

```
function twoSum(array) {
  for (let i = 0; i < array.length; i++) {
    for (let j = 0; j < array.length; j++) {
      if (i !== j && array[i] + array[j] === 10) {
        return true;
      }
    }
  }
  return false;
}
```

What are the best-, average-, and worst-case scenarios? Then, express the worst-case scenario in terms of Big O Notation.

- **Best-case scenario:** The best case occurs if the first two numbers add up to 10, in which case the function returns immediately. In this scenario, the efficiency is  $O(1)$ .
- **Average-case scenario:** On average, the function would have to look through a substantial portion of the array. This scenario would be  $O(N^2)$ .
- **Worst-case scenario:** The worst case occurs if no two numbers add up to 10, requiring the function to check every possible pair of numbers. The efficiency in this scenario is also  $O(N^2)$ .

4. The following function returns whether or not a capital "X" is present within a string.

```
function containsX(string) {
  foundX = false;
  for(let i = 0; i < string.length; i++) {
    if (string[i] === "X") {
      foundX = true;
    }
  }
  return foundX;
}
```

What is this function's time complexity in terms of Big O Notation? Then, modify the code to improve the algorithm's efficiency for best- and average-case scenarios.

- **Time complexity:** The efficiency of this function is  $O(N)$ , where  $N$  is the length of the string.
- **Improved code:** To optimize the function for the best- and average-case scenarios, you can add a **break** statement after finding "X". This stops the loop as soon as "X" is found, saving unnecessary iterations.

```
function containsX(string) {
  foundX = false;
  for(let i = 0; i < string.length; i++) {
    if (string[i] === "X") {
      foundX = true;
      break; // Exit loop once "X" is found
    }
  }
  return foundX;
}
```

## CHAPTER 7: Big O in Everyday Code

Understanding Big O notation is more complex than it might seem at first glance, but it's incredibly useful in practical situations. Here's why:

1. **Identifying Speed:** Before trying to make code run faster, we need to know how quickly it runs in the first place. Big O notation helps us identify this.
2. **Deciding When to Optimize:** By categorizing the code's efficiency with Big O notation, we can decide if it needs to be made faster. For instance, code with a time complexity of  $O(N^2)$  is generally seen as slow, and that might prompt us to see if we can improve it.
3. **Recognizing Limits:** Sometimes,  $O(N^2)$  might be the fastest we can make a particular algorithm. However, recognizing it as slow will encourage us to look for other possible solutions or accept that this is the best we can do for that specific task.

In the upcoming lessons, you'll be learning techniques to make code run faster. The foundation of that work is understanding the current speed of the code, and that's what this chapter aims to teach. Essentially, before you can speed up the code, you have to know how fast it's going now.

### Mean Average of Even Numbers

We have a Ruby method that takes an array of numbers and calculates the mean average of all the even numbers in that array. The key question is: how can we express the efficiency of this method using Big O notation?

```
def average_of_even_numbers(array)

  # The mean average of even numbers will be defined as the sum of
  # the even numbers divided by the count of even numbers, so we
  # keep track of both the sum and the count:

  sum = 0.0
  count_of_even_numbers = 0

  # We iterate through each number in the array, and if we encounter
  # an even number, we modify the sum and the count:

  array.each do |number|
    if number.even?
      sum += number
      count_of_even_numbers += 1
    end
  end

  # We return the mean average:

  return sum / count_of_even_numbers
end
```

1. **Understanding N:** In this context, "N" refers to the size of the array, or the number of data elements we are working with.
2. **Counting the Steps:** We loop through the array, and for each number, we perform certain steps to check if it's even and modify variables if it is. In the worst case (when all numbers are even), we perform three steps for each number, leading to a total of  $3N$  steps.
3. **Additional Steps:** Apart from the loop, there are a few other steps such as initializing variables and division, adding up to three extra steps.
4. **Big O Notation:** When we use Big O notation, we focus on the most significant factor in the total steps and ignore constants. So even though the actual count is  $3N + 3$ , we represent the efficiency of this algorithm simply as  $O(N)$ .

The overall efficiency of the method to calculate the mean average of even numbers in an array is  $O(N)$ , meaning that the number of steps the algorithm takes grows linearly with the size of the array.

### Word Builder

The **wordBuilder** function is a way to build combinations of characters from an array. Depending on the number of nested loops, its efficiency varies:

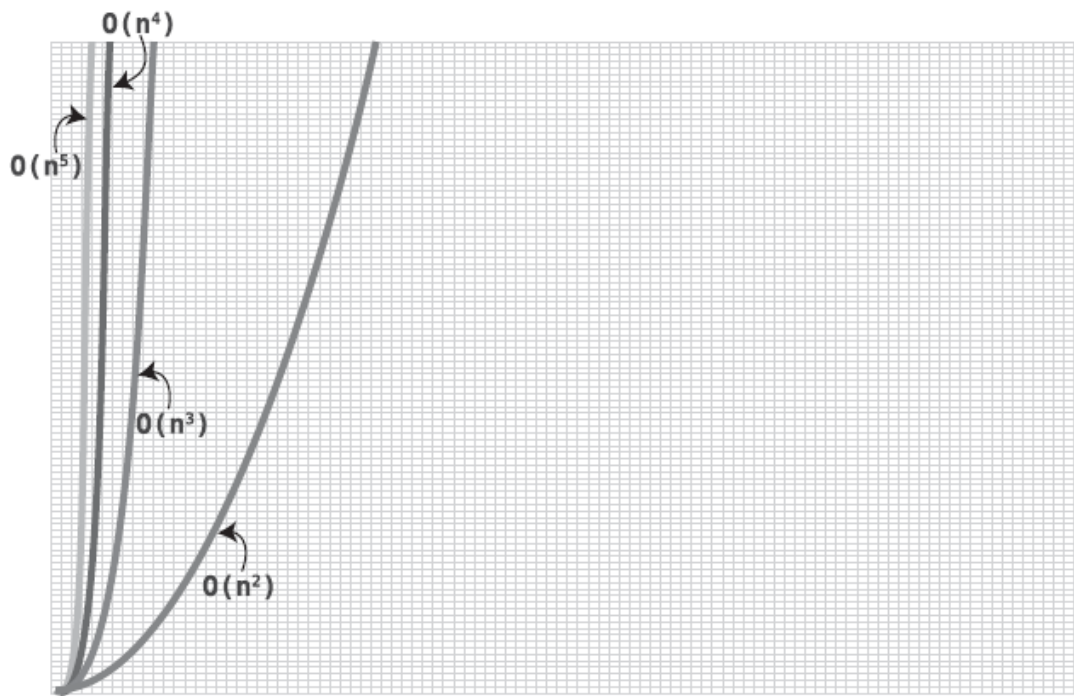
#### 1. Two-Character Combinations (Two Nested Loops):

- Given an array of characters, the function will create all possible combinations of two different characters.
  - For example, given the array: ["a", "b", "c", "d"], we'd return a new array containing the following string combinations: [ 'ab', 'ac', 'ad', 'ba', 'bc', 'bd', 'ca', 'cb', 'cd', 'da', 'db', 'dc' ]
- It does this by running a loop within a loop, iterating over the characters and pairing them up, but skipping when the indices are the same.
- Since it iterates over all N elements for each of the N elements, it has  $N * N$  steps.
- Therefore, its time complexity is  $O(N^2)$ .

#### 2. Three-Character Combinations (Three Nested Loops):

- By adding a third nested loop, the function creates all possible combinations of three different characters.
  - That is, for our example array of ["a", "b", "c", "d"], our function would return the array: [ 'abc', 'abd', 'acb', 'acd', 'adb', 'adc', 'bac', 'bad', 'bca', 'bcd', 'bda', 'bdc', 'cab', 'cad', 'cba', 'cbd', 'cda', 'cdb', 'dab', 'dac', 'dba', 'dbc', 'dca', 'dcb' ]
- This means iterating over all N elements for each of the N elements for each of the N elements again, resulting in  $N * N * N$  steps.
- Therefore, its time complexity is  $O(N^3)$ .

In general, each additional nested loop adds another power of N to the time complexity. The larger the power, the slower the code, so optimizing from  $O(N^3)$  to  $O(N^2)$  would significantly speed up the algorithm.



### Array Sample

The following **sample** function in Python takes an array and returns the first, middle, and last values from it.

```
def sample(array):  
    first = array[0]  
    middle = array[int(len(array) / 2)]  
    last = array[-1]  
  
    return [first, middle, last]
```

No matter the size of the array, the function always does the following:

1. Fetches the first value of the array.
2. Calculates the middle index and fetches the value at that index.
3. Fetches the last value of the array.

These steps remain the same and don't depend on how large the array is. In other words, it takes a constant number of steps to perform this operation. Hence, the efficiency of this function in terms of Big O notation is  $O(1)$ , meaning the time it takes is constant and doesn't change with the size of the input array.

### Average Celsius Reading

In the following **average\_celsius** function written in Ruby, we're calculating the mean average of temperatures converted from Fahrenheit to Celsius.

```

def average_celsius(fahrenheit_readings)
  # Collect Celsius numbers here:
  celsius_numbers = []

  # Convert each reading to Celsius and add to array:
  fahrenheit_readings.each do |fahrenheit_reading|
    celsius_conversion = (fahrenheit_reading - 32) / 1.8
    celsius_numbers.push(celsius_conversion)
  end

  # Get sum of all Celsius numbers:
  sum = 0.0

  celsius_numbers.each do |celsius_number|
    sum += celsius_number
  end

  # Return mean average:
  return sum / celsius_numbers.length
end

```

1. **Conversion Loop:** We loop through each Fahrenheit reading and convert it to Celsius. This is done for all N readings, so it takes N steps.
2. **Summation Loop:** After that, we loop through the converted Celsius numbers to calculate the sum. This also takes N steps.

Since both loops are not nested inside one another but rather follow each other, the total steps taken by the algorithm is  $N$  (for conversion) +  $N$  (for summation) =  $2N$ . However, in Big O notation, we ignore constant multiples. So the overall efficiency of this function, in terms of Big O notation, is  $O(N)$ . It describes an algorithm whose performance will grow linearly and in direct proportion to the size of the input dataset,  $N$ .

### Clothing Labels

Imagine you're working for a clothing manufacturer, and you have an array of clothing items. You want to create labels for each item with sizes ranging from 1 to 5.

```

["Purple Shirt Size: 1", "Purple Shirt Size: 2", "Purple Shirt Size: 3", "Purple Shirt Size: 4", "Purple Shirt Size: 5",
"Green Shirt Size: 1", "Green Shirt Size: 2", "Green Shirt Size: 3", "Green Shirt Size: 4", "Green Shirt Size: 5"]

```

```

def mark_inventory(clothing_items):
    clothing_options = []

    for item in clothing_items:
        # For sizes 1 through 5 (Python ranges go UP TO second
        # number, but do not include it):
        for size in range(1, 6):
            clothing_options.append(item + " Size: " + str(size))

    return clothing_options

```

Here's how you can understand the efficiency of the provided Python code:

1. **Outer Loop:** This loop goes through each clothing item in the array, so it runs  $N$  times, where  $N$  is the number of clothing items.
2. **Inner Loop:** For each clothing item, this loop iterates 5 times to create labels for sizes 1 through 5. It doesn't change with the size of  $N$ , it's always 5.

So, in total, the code runs  $N$  times for the outer loop and 5 times for each inner loop. The total efficiency becomes  $5 * N$ . But when we look at this in terms of Big O notation, which is about the growth of the algorithm and not the exact step count, we ignore the constant factor (in this case, 5). Therefore, the efficiency of the code, in terms of Big O, is simply  $O(N)$ . This means the time it takes for the code to run will grow linearly with the number of clothing items,  $N$ .

### Count the Ones

We have a function that takes an array of arrays, with the inner arrays containing 1's and 0's. The goal is to count how many 1's there are in total.

```

[
    [0, 1, 1, 1, 0],
    [0, 1, 0, 1, 0, 1],
    [1, 0]
]

def count_ones(outer_array):
    count = 0

    for inner_array in outer_array:
        for number in inner_array:
            if number == 1:
                count += 1

    return count

```

1. **Outer Loop:** This loop goes through each inner array, but doesn't concern itself with the size of each inner array. It's concerned with the number of inner arrays.

2. **Inner Loop:** This loop goes through each number in the inner arrays. It will run once for every number, no matter how those numbers are distributed among the inner arrays.

So, instead of thinking of the algorithm as dealing with two levels of nested structure (and thus jumping to  $O(N^2)$ ), we can think of  $N$  as representing the total number of numbers in all the inner arrays combined. The code looks at each number exactly once, so the total number of steps is equal to the total number of numbers, which is  $N$ . Thus, the time complexity of this algorithm is  $O(N)$ . It grows linearly with the number of numbers in the array of arrays, rather than being affected by the particular structure of those arrays.

### Palindrome Checker

The following function checks if a string is a palindrome, meaning that it reads the same forwards and backwards. It does this by comparing the characters from the start and end of the string, moving inward.

```
function isPalindrome(string) {  
    // Start the leftIndex at index 0:  
    let leftIndex = 0;  
    // Start rightIndex at last index of array:  
    let rightIndex = string.length - 1;  
  
    // Iterate until leftIndex reaches the middle of the array:  
    while (leftIndex < string.length / 2) {  
        // If the character on the left doesn't equal the character  
        // on the right, the string is not a palindrome:  
        if (string[leftIndex] !== string[rightIndex]) {  
            return false;  
        }  
  
        // Move leftIndex one to the right:  
        leftIndex++;  
        // Move rightIndex one to the left:  
        rightIndex--;  
    }  
  
    // If we got through the entire loop without finding any  
    // mismatches, the string must be a palindrome:  
    return true;  
}
```

1. **Starting Point:** Two indexes are created, one starting from the beginning and the other from the end of the string.
2. **Comparison:** The characters at these indexes are compared. If they're different, the function immediately returns false. If they're the same, the indexes move toward the middle of the string.
3. **Loop Continuation:** This continues until the indexes meet in the middle.

Now, here's the critical point for understanding the Big O complexity:



- Although the loop runs only until the midpoint of the string (so, effectively, only half the length of the string), in Big O notation, we ignore constant factors like this 2.
- This means that even though the function is doing something a bit clever to check only half the string, its efficiency is still considered linear with respect to the size of the string.

So the time complexity of the algorithm is  $O(N)$ , where  $N$  is the length of the string. It grows directly with the size of the input.

### Get All the Products

We have an algorithm that takes an array of numbers and returns the product of every combination of two numbers. For example, if you provide [1, 2, 3, 4, 5], it will return products like [2, 3, 4, 5, 6, 8, 10, 12, 15, 20].

```
function twoNumberProducts(array) {
  let products = [];

  // Outer array:
  for(let i = 0; i < array.length - 1; i++) {

    // Inner array, in which j always begins one index
    // to the right of i:
    for(let j = i + 1; j < array.length; j++) {
      products.push(array[i] * array[j]);
    }
  }

  return products;
}
```

Here's how the algorithm works:

1. It uses two loops: an outer loop and an inner loop.
2. The outer loop runs for each number in the array except the last one.
3. The inner loop runs for the remaining numbers to the right of the current number in the outer loop.
4. The inner loop runs fewer times each time it is launched by the outer loop.

To figure out how efficient this algorithm is, we need to look at how many times the loops run. The outer loop runs  $N$  times, where  $N$  is the number of items in the array. The inner loop runs roughly  $N + (N - 1) + (N - 2) + (N - 3) \dots + 1$  times.

When we calculate this pattern, it turns out to be about  $N^2 / 2$ . But in Big O notation, we ignore constants, so we can say that the time complexity of this algorithm is  $O(N^2)$ . In other words, the algorithm's efficiency is determined by the square of the number of items in the array, so it may become slow if you pass in a very large array.

## Dealing with Multiple Datasets

Suppose you have two different arrays, and you want to find the product of every number from the first array with every number of the second array. Here's an example:

- First array: [1, 2, 3]
- Second array: [10, 100, 1000]
- Products: [10, 100, 1000, 20, 200, 2000, 30, 300, 3000]

The code to do this has two loops, one for each array. The complexity isn't straightforward, though, because the arrays can be different sizes.

```
function twoNumberProducts(array1, array2) {
  let products = [];

  for(let i = 0; i < array1.length; i++) {
    for(let j = 0; j < array2.length; j++) {
      products.push(array1[i] * array2[j]);
    }
  }

  return products;
}
```

Here's the dilemma:

- If both arrays have the same size, the complexity would be like multiplying the size of one array by itself ( $O(N^2)$ ).
- If one array is much larger than the other, the complexity would be more like multiplying the size of the larger array by 1 ( $O(N)$ ).

Because of these different scenarios, we can't pin down the complexity as a simple  $O(N)$  or  $O(N^2)$ . Instead, we use  $O(N * M)$ , where  $N$  is the size of one array, and  $M$  is the size of the other. This means that the efficiency of the algorithm depends on the sizes of both arrays. If they're the same size, it's closer to  $O(N^2)$ , and if one is much larger, it's closer to  $O(N)$ . While this isn't a perfectly clear way to describe efficiency (like comparing apples to oranges), it's the best way to understand the relationship between the two different datasets in this particular scenario.

## Password Cracker

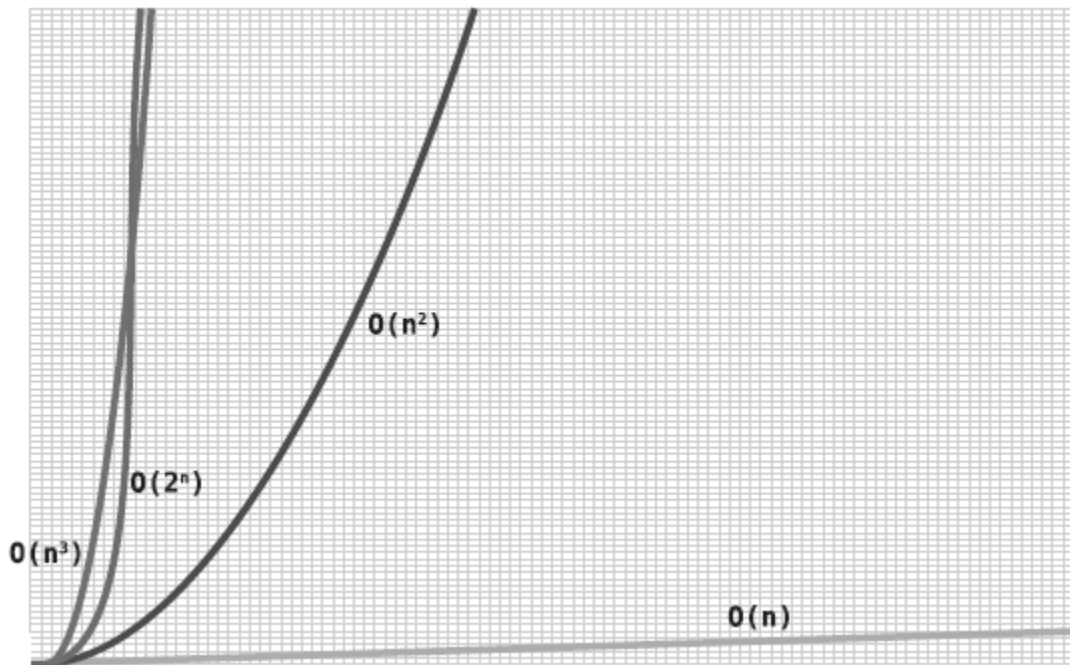
Imagine you're trying to guess someone's password (for ethical reasons) and you decide to try every possible combination of letters. You write code to create every string of a certain length, using the letters "a" to "z".

```
def every_password(n)
  (("a" * n)..("z" * n)).each do |str|
    puts str
  end
end
```

- If the length is 1, you try 26 combinations (just every letter).
- If the length is 2, you try 26 \* 26 combinations (every two-letter combination).
- If the length is 3, you try 26 \* 26 \* 26 combinations, and so on.

The pattern here is that for each additional character in the length, you multiply the number of steps by 26. So if the length of the password (N) is 3, you have 26<sup>3</sup> combinations to try. In terms of Big O Notation, this is expressed as O(26<sup>N</sup>). Why is this a problem?

- It's extremely slow. Even a length of 5 takes an incredibly long time.
- The graph of this algorithm's speed compared to others shows that it gets slower even than O(N<sup>3</sup>) at a point.



- It's the opposite of an efficient algorithm like O(log N) where adding data only slightly increases the steps. Here, adding just one character multiplies the steps by 26.

In short, using a brute-force approach like this to crack a password takes a huge amount of time, making it an inefficient method. It's a vivid example of why understanding an algorithm's efficiency is vital in computer science.

## Wrapping Up

Congratulations! You're now a Big O pro. You can analyze all sorts of algorithms and categorize their time complexities. Armed with this knowledge, you'll be able to methodically optimize your code for speed. Speaking of which, in the next chapter, we'll discover a new data structure that is the one of the most useful and common tools for speeding up algorithms. And I'm talking about some serious speed.

## Exercises

The following exercises provide you with the opportunity to practice with algorithms in practical situations.

1. Use Big O Notation to describe the time complexity of the following function. The function returns true if the array is a “100-Sum Array,” and false if it is not. A “100-Sum Array” meets the following criteria:

- Its first and last numbers add up to 100.
- Its second and second-to-last numbers add up to 100.
- Its third and third-to-last numbers add up to 100, and so on.

Here is the function:

```
def one_hundred_sum?(array)
  left_index = 0
  right_index = array.length - 1

  while left_index < array.length / 2
    if array[left_index] + array[right_index] != 100
      return false
    end

    left_index += 1
    right_index -= 1
  end

  return true
end
```

This function iterates over half the length of the array to check whether the sum of the corresponding elements from the beginning and end of the array is 100. Since it checks half the elements of the array, the time complexity is  $O(N/2)$ , but we can simplify that to  **$O(N)$** , where N is the length of the array.

2. Use Big O Notation to describe the time complexity of the following function. It merges two sorted arrays together to create a new sorted array containing all the values from both arrays:

```

def merge(array_1, array_2)
  new_array = []
  array_1_pointer = 0
  array_2_pointer = 0

  # Run the loop until we've reached end of both arrays:
  while array_1_pointer < array_1.length ||
        array_2_pointer < array_2.length

    # If we already reached the end of the first array,
    # add item from second array:
    if !array_1[array_1_pointer]
      new_array << array_2[array_2_pointer]
      array_2_pointer += 1

    # If we already reached the end of the second array,
    # add item from first array:
    elsif !array_2[array_2_pointer]
      new_array << array_1[array_1_pointer]
      array_1_pointer += 1

    # If the current number in first array is less than current
    # number in second array, add from first array:
    elsif array_1[array_1_pointer] < array_2[array_2_pointer]
      new_array << array_1[array_1_pointer]
      array_1_pointer += 1

    # If the current number in second array is less than or equal
    # to current number in first array, add from second array:
    else
      new_array << array_2[array_2_pointer]
      array_2_pointer += 1
    end
  end

  return new_array
end

```

This function iterates through both arrays until the end of both is reached. In the worst case, it will go through both arrays completely, so the time complexity is  $O(N + M)$ , where  $N$  is the length of the first array and  $M$  is the length of the second array.

3. Use Big O Notation to describe the time complexity of the following function. This function solves a famous problem known as “finding a needle in the haystack.”

Both the needle and haystack are strings. For example, if the needle is "def" and the haystack is "abcdefghi", the needle is contained somewhere in the haystack, as "def" is a substring of "abcdefghi". However, if the needle is "dd", it cannot be found in the haystack of "abcdefghi".

This function returns true or false, depending on whether the needle can be found in the haystack:

```
def find_needle(needle, haystack)
  needle_index = 0
  haystack_index = 0

  while haystack_index < haystack.length
    if needle[needle_index] == haystack[haystack_index]
      found_needle = true

      while needle_index < needle.length
        if needle[needle_index] != haystack[haystack_index + needle_index]
          found_needle = false
          break
        end
        needle_index += 1
      end

      return true if found_needle
      needle_index = 0
    end

    haystack_index += 1
  end

  return false
end
```

The outer loop goes through the entire haystack, and the inner loop goes through the entire needle. In the worst case, the time complexity will be  $O(N * M)$ , where N is the length of the haystack and M is the length of the needle.

4. Use Big O Notation to describe the time complexity of the following function. This function finds the greatest product of three numbers from a given array:

```

def largest_product(array)
  largest_product_so_far = array[0] * array[1] * array[2]
  i = 0

  while i < array.length
    j = i + 1
    while j < array.length
      k = j + 1
      while k < array.length
        if array[i] * array[j] * array[k] > largest_product_so_far
          largest_product_so_far = array[i] * array[j] * array[k]
        end
        k += 1
      end
      j += 1
    end
    i += 1
  end

  return largest_product_so_far
end

```

This function contains three nested loops, each iterating through the length of the array. The time complexity is  $O(N^3)$ , where  $N$  is the length of the array.

5. I once saw a joke aimed at HR people: “Want to immediately eliminate the unluckiest people from your hiring process? Just take half of the resumes on your desk and throw them in the trash.”

If we were to write software that kept reducing a pile of resumes until we had one left, it might take the approach of alternating between throwing out the top half and the bottom half. That is, it will first eliminate the top half of the pile, and then proceed to eliminate the bottom half of what remains. It keeps alternating between eliminating the top and bottom until one lucky resume remains, and that’s who we’ll hire!

Describe the efficiency of this function in terms of Big O:

```
def pick_resume(resumes)
  eliminate = "top"

  while resumes.length > 1
    if eliminate == "top"
      resumes = resumes[resumes.length / 2, resumes.length - 1]
      eliminate = "bottom"
    elsif eliminate == "bottom"
      resumes = resumes[0, resumes.length / 2]
      eliminate = "top"
    end
  end

  return resumes[0]
end
```

This function repeatedly halves the number of resumes until only one is left. Since it divides the pile in half at each step, the time complexity is  **$O(\log N)$** , where  $N$  is the initial number of resumes.



## CHAPTER 8: Blazing Fast Lookup with Hash Tables

Imagine you are running a fast-food restaurant and want to create a digital menu for customers to order food. You could store the menu items and their prices in a list, something like this:

```
menu = [ ["french fries", 0.75], ["hamburger", 2.5], ["hot dog", 1.5], ["soda", 0.6] ]
```

If you store this information in an array, finding the price of a specific item could take some time. If the array is unordered, your computer would have to look through each item one by one, taking  $O(N)$  time, where  $N$  is the number of items on the menu. If the array is ordered, you could use a method called binary search, which would be faster and take  $O(\log N)$  time.

However, there's an even better way to do this. By using a structure called a hash table, you can find the price of an item almost instantly, in  $O(1)$  time. A hash table is like a super-efficient index that directly points to the location of the item you're looking for. It can make the process of looking up prices on your menu extremely quick, allowing for a smoother and faster ordering experience for your customers.

### Hash Tables

A hash table is like a dictionary. Imagine if you want to find out the price of "french fries" in a menu. In a hash table, "french fries" is the word you're looking for (called the "key"), and \$0.75 is the definition or the associated value. Here's how it works:

1. **Key and Value:** In a hash table, you have pairs of keys and values. In this case, the food item's name is the key, and its price is the value.
2. **Hash Function:** The computer uses something called a hash function to turn the key into an address in memory. Think of this like turning a word into a page number in a dictionary.
3. **One-Step Lookup:** Once the computer has that address, it can go straight to it and find the value. This is why it's so fast - it takes just one step on average.

Using Ruby, you can find the price of "french fries" by simply asking for `menu["french fries"]`, and the computer will instantly tell you the price is \$0.75. This  $O(1)$  efficiency (one step, on average) makes hash tables a powerful tool in programming. They allow for really quick lookups, and you can use them whenever you need to associate keys with values, like menu items with prices or names with phone numbers.

### Hashing with Hash Functions

**Hashing** is like creating a secret code that turns letters into numbers. A **hash function** is the rule that explains how to turn the letters into numbers. It must always turn the same letters into the same numbers.

- Say we have a simple rule that turns each letter into its place in the alphabet ( $A = 1$ ,  $B = 2$ , etc.).
- Now we want to convert the word "BAD" into a number using a multiplication hash function.
- First, we turn each letter into its number ( $B = 2$ ,  $A = 1$ ,  $D = 4$ ), so BAD becomes 214.
- Second, we multiply those numbers together:  $2 * 1 * 4 = 8$ .
- So, according to this hash function, "BAD" becomes 8.

We use hash functions to put things like words into a hash table (a type of storage in a computer). This lets the computer find things really quickly.

### What's Important in a Hash Function?

1. It must be consistent. It always turns the same input into the same output.
2. It should be different for different inputs, but sometimes different inputs might give the same output (like "BAD" and "DAB" both becoming 8). This can cause problems.

**Why Not Use Random Numbers or Current Time?** If a hash function used random numbers or the current time, it would give different numbers for the same input at different times. That's like changing the secret code all the time. It wouldn't work because the computer needs to know that "BAD" will always become 8 (in our example), not something different each time. Think of it like a librarian who has a special way of turning book titles into shelf numbers. If she always does it the same way, she can find any book very quickly. If she did it differently each time, she'd never find anything!

### Building a Thesaurus for Fun and Profit, but Mainly Profit

You're working on a unique thesaurus app called Quickasaurus that provides only one synonym for each word, unlike other traditional thesaurus apps. This makes it quick and different.

**Using a Hash Table:** You choose to use a hash table to store the word pairs (word and its synonym). A hash table is like a row of storage boxes (cells), and each box has a number.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

### Here's How It Works:

#### 1. Add the word "bad":

- First, turn "bad" into a number using the multiplication hash function:  $BAD = 2 * 1 * 4 = 8$ .
- Then, put the synonym "evil" into box number 8.
- Now, your hash table looks like: {"bad" => "evil"}.

							"evil"								
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

#### 2. Add the word "cab":

- Turn "cab" into a number:  $CAB = 3 * 1 * 2 = 6$ .
- Put the synonym "taxi" into box number 6.
- Hash table now looks like: {"bad" => "evil", "cab" => "taxi"}.

					"taxi"		"evil"								
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

### 3. Add the word "ace":

- Turn "ace" into a number:  $ACE = 1 * 3 * 5 = 15$ .
- Put the synonym "star" into box number 15.
- Hash table now looks like: {"bad" => "evil", "cab" => "taxi", "ace" => "star"}.

					"taxi"		"evil"							"star"	
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

**Why Is This Useful?** By turning words into numbers and storing them in specific boxes, the computer can quickly find the synonym for any word. This makes your thesaurus app super fast.

**Picture It Like a Shoe Rack:** Imagine each word is a type of shoe, and each synonym is a color. The hash function tells you which shelf to put each colored shoe on. When you need to find a color for a particular shoe, you know exactly which shelf to look at. Your Quickasaurus app does something similar, and using a hash table to store words and their synonyms helps make it quick and efficient!

### Hash Table Lookups

**Imagine a Hash Table as a Magical Bookshelf:** You've stored words and their meanings (like "bad" means "evil") in specific places on a bookshelf. Each word has a number, and you place its meaning at that number on the shelf.

### How to Find a Word's Meaning Quickly:

1. **You want to find the meaning of "bad":** In your code, you'd say: `thesaurus["bad"]`.
2. **Find the number for "bad":** You have a special way (hashing) to find the number:  $BAD = 2 * 1 * 4 = 8$ .
3. **Go to that number on the shelf:** Look at shelf number 8, and there is the meaning "evil."

### Why Is This Awesome?

- **Speedy:** Finding a meaning is super quick. You just turn the word into a number and go to that spot on the shelf. No need to search every shelf one by one.
- **Efficient:** The way you store words and their meanings is smart. The word itself tells you where its meaning is stored.

### Compare with Other Bookshelves (Arrays):

- **Unordered Bookshelf:** You'd have to search every shelf until you find the word. This takes time.

- **Ordered Bookshelf:** You might search faster, but still, have to check different places.

### Hash Table (Magical Bookshelf) is Better:

- **Instant Lookup:** You directly go to the shelf you want. It's a one-step process ( $O(1)$ ).
- **Perfect for Menus:** Imagine a restaurant menu where you want to find prices fast. You could use this magical bookshelf method to look up prices instantly.

**Think of It Like a Treasure Map:** If you have a treasure map where 'X' marks the spot, you can go straight to the treasure. You don't have to dig everywhere. That's what a hash table does with data. The word (or key) is the 'X', and its meaning (or value) is the treasure. You can find it instantly!

### One-Directional Lookups

#### Hash Table Lookups: One Way Street

Imagine a big wall with lots of small lockers. You have a bunch of keys and corresponding items to store inside these lockers. Here's how it works:

1. **Key to Value:** You use a special method (hashing) to decide which locker (cell) to put each item in. You know that "Key A" puts the item in "Locker 8," "Key B" in "Locker 3," and so on.
2. **Easy to Find Items:** If you have "Key A," you can directly open "Locker 8" and get the item. It's super fast because you know exactly where to look!
3. **But, What if You Only Have the Item?:** Now, if you have an item and want to find its key, you have a problem. You'd have to check every locker, one by one, to find the key that fits it. It's slow and takes a long time.

#### Key Points

- **One Directional:** You can easily use keys to find items (values), but not the other way around.
- **Unique Keys, Repeated Values:** Every key is like a fingerprint; it's unique. But you can have the same item in different lockers (like having the same price for different foods on a menu).
- **What About Collisions?:** Sometimes, two keys may point to the same locker. That's a problem called a collision, and it's a bit more advanced (we'll talk about it later).
- **Overwriting Values:** If you try to put something in a locker that already has an item, the old item gets replaced.

#### Imagine Your House Keys

It's like having a key that opens your front door (key to value) but not being able to look at your house and know which key opens it (value to key). You could try all your keys one by one, but it would take a while!

- **Quick to Find Values with Keys:** Like having a map to a treasure.
- **Slow to Find Keys with Values:** Like having a treasure but no map to find where it came from.
- **Can't Have Duplicate Keys:** Like how each person has a unique phone number.

- **Can Have Duplicate Values:** Like how many people can have the same birthday.

This "one-way street" nature is what makes hash tables so useful for certain tasks but also explains why they aren't suitable for reversing the lookup process.

### Dealing with Collisions

What happens if two keys try to direct you to the same locker? This is known as a collision, and it can be a problem.

#### Collisions: Two Keys, One Locker

Imagine two keys (e.g., "bad" and "dab") both trying to open the same locker number 8. The locker is already filled with the value "evil" for the key "bad." Now, you want to add the value "pat" for the key "dab" in the same locker. Uh-oh, there's a collision!

#### Handling Collisions: Separate Chaining

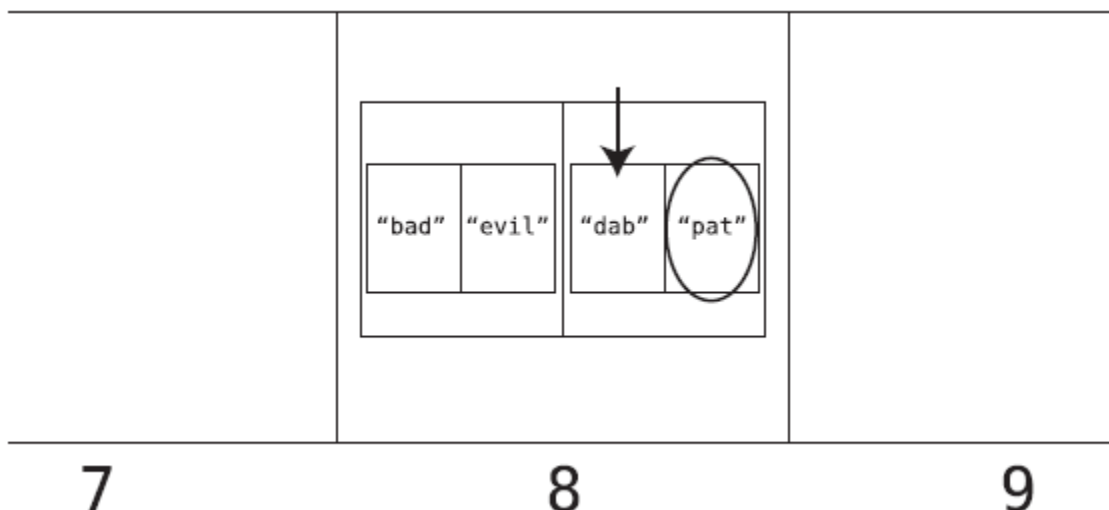
To solve this, we can use a method called separate chaining. Instead of putting just one value in the locker, we add a small box (array) that can hold multiple key-value pairs. Here's how it works with our example:

1. **Hash the Key:** The computer calculates that "dab" equals 8 (locker number 8).
2. **Check the Locker:** It sees that locker 8 is already filled with "evil."
3. **Add a Small Box:** Inside locker 8, it adds a small box that can hold multiple key-value pairs.
4. **Add Both Values:** It puts both "bad" and "dab" inside the box, each with its corresponding value.

#### Looking Up a Value with Separate Chaining

Now, if you want to find the value for "dab," the computer follows these steps:

1. **Find the Right Locker:** It calculates that "dab" leads to locker 8.
2. **Open the Locker:** It finds the small box inside with multiple key-value pairs.
3. **Look Through the Box:** It checks each pair until it finds "dab" and returns its value "pat."



## Why This Matters: Efficiency

- **Best Case:** If there are no or few collisions, finding a value in a hash table is super fast ( $O(1)$  time).
- **Worst Case:** If many keys collide into the same locker, it's like having to look through a whole bunch of keys and values in one locker, which slows things down to  $O(N)$  time.

## Summary:

- **Collisions are When Two Keys Point to the Same Locker:** It's like two people having the same locker combination at the gym.
- **Separate Chaining Fixes This:** It's like adding a small organizer inside the locker for multiple people to use.
- **Most Programming Languages Handle This for You:** So you don't usually have to worry about it, but knowing how it works helps you understand why hash tables are typically fast but can sometimes be slow.

## Making an Efficient Hash Table

Creating an efficient hash table involves finding the right balance among three crucial elements:

- 1. Amount of Data:** This refers to the total number of items (key-value pairs) you want to store in the hash table.
- 2. Number of Cells (Lockers):** These are the spaces available to store your items. Imagine a set of lockers; each one can store a key-value pair.
- 3. Hash Function:** This is like a formula that takes a key and turns it into a locker number. A good hash function spreads keys evenly among the available lockers.

## Why Are These Factors Important?

- **Too Much Data and Too Few Cells:** If you try to store too many items in too few lockers, you'll end up with collisions, where two keys try to go into the same locker. This slows things down.
- **Bad Hash Function:** If your hash function keeps pointing to the same few lockers, you'll also get collisions. For instance, if your function only uses numbers 1 to 9, then no matter how many lockers you have, all the items will end up in those first nine lockers, leaving the rest empty.

## Example: A Flawed Hash Function

Imagine you're using a hash function that turns letters into numbers and keeps adding those numbers until you get a single digit. Like this:

- $PUT = 16 + 21 + 20 = 57$
- Break it down:  $5 + 7 = 12$
- Break it down again:  $1 + 2 = 3$

So, the word "PUT" ends up in locker number 3. The problem is that this method will always result in numbers 1 to 9, so you'll always use only those first nine lockers, regardless of how many others you have.

## So, What Makes a Good Hash Table?

- **Enough Cells:** Have enough lockers to accommodate your data without cramming too much into each one.
- **A Good Hash Function:** Pick or create a hash function that makes good use of all available lockers. It shouldn't just keep pointing to the same few over and over.
- **Balanced Data and Cells:** A balance between the amount of data and the number of available cells ensures that the hash table can work efficiently without too many collisions.

You want your hash table to use its space wisely and spread out the data evenly. It's like organizing a big closet; if everything is crammed into one corner, it's hard to find what you need. But if everything has its place, you can find items quickly and easily. That's what makes hash tables such a powerful and efficient tool when done right!

### The Great Balancing Act

A hash table needs to balance two factors to be efficient: minimizing collisions and avoiding unnecessary use of memory.

### Why Not Just Use a Massive Number of Cells?

Imagine having a closet with 1,000 hangers, but you only have five coats. Sure, there's no chance your coats will overlap, but you're wasting a lot of space! Similarly, a hash table with 1,000 cells for only five data items avoids collisions but uses memory poorly.

### Striking the Right Balance

Computer scientists recommend a balance called the load factor, which is a ratio of data elements to cells. The ideal load factor is 0.7. That means for every 7 data elements, there should be 10 cells in the hash table. If you plan to store 14 elements, you'll want 20 cells, and so on.

### What Happens If You Add More Data?

If you initially set up a hash table with 10 cells for 7 data pieces and then add more data, the computer will automatically expand the hash table, adding more cells. It will also adjust the hash function to distribute the new data evenly across all cells.

### Who Manages All of This?

Most of this work happens behind the scenes, managed by the programming language you're using. It figures out how big the hash table should be, what hash function to use, and when to expand the hash table. You can usually trust that your language has implemented these details efficiently.

### Why Does All of This Matter?

Hash tables offer superior lookup efficiency, with an average time complexity of  $O(1)$ , meaning you can retrieve any value in constant time. Understanding how they work allows you to use them effectively in various scenarios and optimize your code for speed.

### In Brief:

- Hash tables must avoid collisions but also not waste memory.
- The ideal balance (load factor) is 0.7, or 7 elements for every 10 cells.
- As you add more data, the computer will automatically manage the hash table's size and function.
- Hash tables offer a fast way to organize and retrieve data, making them a valuable tool in programming.

## Hash Tables for Organization

1. **Natural Pairs.** Hash tables excel in storing naturally paired data:

- **Menus:** Food items with prices.
- **Thesauri:** Words with synonyms.
- **Tallies:** Political candidates with votes, or inventory items with quantities.

In Python, hash tables are even called dictionaries, emphasizing their use in paired data representation.

2. **Simplifying Logic.** Hash tables can simplify code by replacing conditional logic. Consider the function for HTTP status codes:

- Instead of multiple conditional statements for each code, we can define a hash table with codes and their meanings.
- This turns a series of **if** and **elif** statements into a single lookup in the hash table.

```
STATUS_CODES = {200: "OK", 301: "Moved Permanently", 401: "Unauthorized", 404: "Not Found", 500: "Internal Server Error"}
```

```
def status_code_meaning(number):
    return STATUS_CODES[number]
```

3. **Representing Objects:**

Hash tables can represent objects with various attributes:

- A dog might be represented as: **{"Name": "Fido", "Breed": "Pug", "Age": 3, "Gender": "Male"}**.
- A list of dogs can be an array of hash tables, each containing the attributes of a dog.

**What Makes Hash Tables So Useful?**

- **Versatility:** They can be used to represent a wide variety of paired data.
- **Simplicity:** They can reduce complex logic into simple key-value lookups.
- **Structure:** They provide a clean and organized way to represent objects and their attributes.



## Hash Tables for Speed

Hash tables have exciting applications in speeding up your code. Let's explore how hash tables can transform a search operation from linear time to constant time:

### Understanding Through an Example:

- **An Unordered Array:** Suppose you have an array like `[61, 30, 91, 11, 54, 38, 72]`, and you want to find if a specific number exists in it. Because the array is unordered, you would perform a linear search, taking up to  $N$  steps, where  $N$  is the size of the array.
- **Converting to Hash Table:** Now, imagine converting the array into a hash table:

```
hash_table = {61 => true, 30 => true, 91 => true, 11 => true, 54 => true, 38  
=> true, 72 => true}
```

Here, each number is stored as a key with the value **true**.

- **One-Step Lookup:** Searching in the hash table becomes a one-step process. By simply doing `hash_table[72]`, you can check if 72 is present in the original array. If the key is in the hash table, you'd get back `true`, otherwise, you'd get a `nil` value (or equivalent, depending on the language).
- **The Magic:** This conversion from array to hash table turns an  $O(N)$  search into an  $O(1)$  search, which is a significant performance boost!
- **What's the Trick?** Even if the data isn't paired, you can utilize a hash table by placing each number as a key. The value doesn't matter; it could be **true** or any "truthy" value.
- **Using as an Index:** Think of this hash table as an index, just like at the back of a book. It instantly tells you whether a specific item is contained within the original array without having to search through it.

### Why This is Powerful:

- **Speed Boost:** By leveraging the efficiency of hash table lookups, you can significantly reduce the time complexity of search operations.
- **Versatility:** It shows that hash tables can be applied creatively beyond just storing paired data.
- **Practical Applications:** This method can be used to optimize real-world algorithms, making it not just a theoretical concept but something you can apply in practice.

## Array Subset

Next, let's determine whether one array is a subset of another. That means all the elements of the second array should be present in the first array.

### Initial Approach: Nested Loops

1. **Comparing Arrays:** Loop through the elements of the smaller array, and for each, loop through the larger array to check if the element is contained.
2. **JavaScript Function:** Using nested loops, a function `isSubset` is implemented to return `true` if the smaller array is a subset of the larger one, otherwise `false`.

3. **Time Complexity:** This approach has a time complexity of  $O(N * M)$ , where N and M are the sizes of the arrays.

### Improved Approach: Using Hash Table

1. **Determine Larger and Smaller Arrays:** Check which array is larger and which is smaller.
2. **Create Hash Table Index:** Loop through the larger array and store each value as a key in a hash table with the value **true**.
  - Example: ["a", "b", "c", "d", "e", "f"] becomes {"a": true, "b": true, "c": true, "d": true, "e": true, "f": true}
3. **Check Smaller Array Against Hash Table:** Loop through the smaller array and check if each value exists as a key in the hash table.
4. **JavaScript Function:** Implement the **isSubset** function using the hash table approach.
5. **Time Complexity:** The new approach has a time complexity of  $O(N)$ , where N is the total number of items in both arrays combined. This is a significant improvement over the nested loops method.

### Benefits of the Hash Table Approach:

- **Efficiency:** The hash table reduces the time complexity, providing a faster solution.
- **Using as an Index:** The hash table acts as an "index" to enable  $O(1)$  lookups.
- **Versatility:** This method can be applied to various algorithms that require multiple searches within an array.

### Wrapping Up

Hash tables are indispensable when it comes to building efficient software. With their  $O(1)$  reads and insertions, it's a difficult data structure to beat. Until now, our analysis of various data structures revolved around their efficiency and speed. But did you know that some data structures provide advantages other than speed? In the next lesson, we're going to explore two data structures that can help improve code elegance and maintainability.

### Exercises

The following exercises provide you with the opportunity to practice with hash tables.

1. Write a function that returns the intersection of two arrays. The intersection is a third array that contains all values contained within the first two arrays. For example, the intersection of [1, 2, 3, 4, 5] and [0, 2, 4, 6, 8] is [2, 4]. Your function should have a complexity of  $O(N)$ . (If your programming language has a built-in way of doing this, don't use it. The idea is to build the algorithm yourself.)

```
function intersection(array1, array2) {  
  let hashTable = {};  
  let result = [];
```

```

    for (const value of array1) {
        hashTable[value] = true;
    }

    for (const value of array2) {
        if (hashTable[value]) {
            result.push(value);
        }
    }

    return result;
}

```

2. Write a function that accepts an array of strings and returns the first duplicate value it finds. For example, if the array is ["a", "b", "c", "d", "c", "e", "f"], the function should return "c", since it's duplicated within the array. (You can assume that there's one pair of duplicates within the array.) Make sure the function has an efficiency of  $O(N)$ .

```

function findFirstDuplicate(array) {
    let seen = {};

    for (const value of array) {
        if (seen[value]) {
            return value;
        }
        seen[value] = true;
    }

    return null; // No duplicates found
}

```

3. Write a function that accepts a string that contains all the letters of the alphabet except one and returns the missing letter. For example, the string, "the quick brown box jumps over a lazy dog" contains all the letters of the alphabet except the letter, "f". The function should have a time complexity of  $O(N)$ .

```

function findMissingLetter(str) {
    let alphabet = 'abcdefghijklmnopqrstuvwxyz';

```

```

let hashTable = {};

for (const letter of alphabet) {
  hashTable[letter] = true;
}

for (const char of str) {
  if (hashTable[char.toLowerCase()]) {
    delete hashTable[char.toLowerCase()];
  }
}

return Object.keys(hashTable)[0];
}

```

4. Write a function that returns the first non-duplicated character in a string. For example, the string, "minimum" has two characters that only exist once—the "n" and the "u", so your function should return the "n", since it occurs first. The function should have an efficiency of  $O(N)$ .

```

function findFirstNonDuplicate(str) {
  let charCount = {};

  for (const char of str) {
    charCount[char] = (charCount[char] || 0) + 1;
  }

  for (const char of str) {
    if (charCount[char] === 1) {
      return char;
    }
  }

  return null; // No non-duplicated characters found
}

```

## CHAPTER 9: Crafting Elegant Code with Stacks and Queues

In programming, managing data efficiently is key. Two data structures that help in dealing with temporary data are stacks and queues. Both are like arrays but with some restrictions that make them quite useful.

### Stacks

Imagine a stack of plates. You can only add a plate to the top (push) and remove the top plate (pop). This "last in, first out" (LIFO) method is what characterizes a stack. You use stacks when you need to keep track of temporary data in the order in which they were added, like the "undo" feature in a text editor.

### Queues

A queue is like a line at a bank. People join the line at the end (enqueue) and leave from the front (dequeue). This "first in, first out" (FIFO) method is what defines a queue. Queues are used in scenarios like printing documents, where the first document sent to the printer is the first one to be printed.

Both stacks and queues can be used to handle temporary data efficiently, focusing on the specific order needed for processing. Once the data is handled, it's discarded, as it no longer holds significance. These structures simplify code and make it easier to read, and are essential in various applications, from operating systems to everyday programs.

### Stacks

A stack is a data structure that resembles a pile of items, like a stack of dishes. It's easy to visualize and follow, and here's how it works:

#### 1. Inserting (Pushing):

- You can only add an item to the top of the stack.
- Imagine placing a dish on top of the pile.

#### 2. Deleting (Popping):

- You can only remove the item at the top of the stack.
- It's like taking the top dish off the pile.

#### 3. Reading:

- You can only look at the item on the top.
- It's like checking the dish on the top without moving any others.

#### Example:

Let's build a stack of numbers:

- Start with an empty stack.
- Push 5 onto the stack; it now contains [5].
- Push 3 onto the stack; it now contains [5, 3].
- Push 0 onto the stack; it now contains [5, 3, 0].

- Pop from the stack; it now contains [5, 3].
- Pop again from the stack; it now contains [5].

## Why Use a Stack?

Though these rules might seem limiting, they are beneficial for specific tasks. The "Last In, First Out" (LIFO) principle means the last item you added (pushed) will always be the first one you remove (pop). It's useful for things like undo functionality, where you want to reverse the last action taken.

## Abstract Data Types

Abstract Data Types (ADTs) are a high-level concept in programming that doesn't specifically depend on how they are implemented. They describe how data structures should behave, but not how they should be built. Here's a simplified explanation:

### Stack as an Abstract Data Type

A stack, which follows the "Last In, First Out" (LIFO) principle, is a typical example of an abstract data type. Though most programming languages don't offer a built-in stack class, you can easily create one using existing data structures, like an array. Example Implementation in Ruby:

```
class Stack
  def initialize
    @data = []
  end

  def push(element)
    @data << element
  end

  def pop
    @data.pop
  end

  def read
    @data.last
  end
end
```

In this Ruby example, a stack is built using an array. The methods **push**, **pop**, and **read** ensure that you can only interact with the array following the stack's rules.

### What Makes It Abstract?

The abstract part means that the stack doesn't care how it's implemented underneath. You could use an array or another data structure; the stack only cares about acting in a LIFO way.

### Other Examples:

Another example of an abstract data type is a set, which is a collection of unique elements. Different programming languages may implement sets using arrays, hash tables, or other structures.

### Why Are ADTs Important?

Abstract data types allow for flexibility in implementation while ensuring that specific rules or behaviors are followed. They help programmers understand what a data structure is supposed to do without getting tied down to how it should do it. Whether built-in or custom-made, abstract data types are theoretical concepts that can be implemented using various underlying data structures, depending on the needs and constraints of the specific situation.

## Stacks in Action

Stacks can be used to develop algorithms for specific problems like checking the syntax of code for matching opening and closing braces. Here's a simplified explanation of how you could create a JavaScript linter for this purpose using a stack:

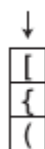
We want to identify three types of syntax errors in JavaScript code related to braces (parentheses, square brackets, and curly braces):

1. An opening brace without a matching closing brace.
2. A closing brace without a matching opening brace.
3. A closing brace that doesn't match the type of the immediately preceding opening brace.

We can use a stack to implement a linting algorithm that checks for these errors:

1. **Prepare an Empty Stack:** We'll use this to track the opening braces we encounter.
2. **Read Characters from Left to Right:**
  - **Ignore Non-Brace Characters:** Skip any character that isn't a brace.
  - **Push Opening Braces:** If we find an opening brace, push it onto the stack.
  - **Check Closing Braces:** If we find a closing brace:
    - **Pop the Top Element:** This should be the corresponding opening brace.
    - **Check for Errors:**
      - If the popped item doesn't match the current closing brace, it's Error Type #3.
      - If the stack was empty (nothing to pop), it's Error Type #2.
      - If it's a match, continue parsing.
3. **Check for Remaining Opening Braces:** If anything is left on the stack at the end, it's Error Type #1.

```
(var x = {y: [1, 2, 3]})
```



Given the line of code with various types of braces, we follow the above steps:

- Push every opening brace we encounter onto the stack.
- When we find a closing brace, we pop the top element from the stack and ensure that it's a match.
- By the time we reach the end of the line, if there are no errors, and our stack is empty, the syntax related to braces is correct.

The use of a stack in this scenario provides an elegant solution for checking brace-related syntax in code. By maintaining the order of the opening braces and comparing them with the corresponding closing braces, a stack enables us to create a simple and effective linting algorithm.

### Code Implementation: Stack-Based Code Linter

The **Linter** class uses a stack to verify that a line of code follows proper syntax for matching braces.

#### Main Components:

- **Initialize Method:** Initializes a stack using a Ruby array.
- **Lint Method:** Accepts a text string and inspects it for syntax errors related to braces.
- **Helper Methods:** Determine if a character is an opening or closing brace, or if a pair of braces do not match.

#### Key Steps:

1. **Iterate through Each Character:** The text is inspected character by character.
2. **Handle Opening Braces:** If an opening brace is found, it's pushed onto the stack.
3. **Handle Closing Braces:** If a closing brace is found:
  - Pop the top of the stack to get the corresponding opening brace.
  - If the stack was empty (i.e., nil was popped), return an error message for a missing opening brace.
  - If the popped opening brace doesn't match the current closing brace, return an error message for mismatched braces.
4. **Check for Leftover Opening Braces:** If there are any opening braces left on the stack at the end, return an error message for missing closing braces.
5. **Return True for No Errors:** If no errors are found, the method returns true.

The **Linter** class can be used to inspect a line of code like `linter.lint(" var x = { y: [1, 2, 3] } )"`, and it will return true if the syntax is correct, or an error message if it is not.

#### Why Use a Stack Instead of an Array?

Using a stack here brings clarity and encapsulation to the algorithm. Even though a stack might be implemented using an array under the hood, the abstraction of a "stack" aligns well with the nature of the problem. The principles of "pushing" and "popping" in a last-in, first-out (LIFO) manner perfectly mirror the logic needed to match opening and closing braces. While it would be possible to solve this using a plain array,



using a stack abstracts the details and allows for a more intuitive understanding of the code. It also enhances the reusability and maintainability of the code by following a well-known data structure and its associated operations.

### The Importance of Constrained Data Structures

Constrained data structures like stacks provide certain advantages over general-purpose structures like arrays, even though a stack could technically be considered a specific type of array. Here's why using a stack can be beneficial:

1. **Avoiding Bugs:** By using a stack, you're limiting the ways in which data can be accessed. In a stack, you can only add or remove items from the top. This constraint prevents accidental misuse, like removing items from the middle, which could lead to bugs, as demonstrated in the linting algorithm example.
2. **Providing a Mental Model:** The concept of a stack, with its Last-In-First-Out (LIFO) nature, offers a specific way of thinking about problems. This mindset helps in designing algorithms, making it easier to conceptualize and solve problems.
3. **Enhancing Code Readability:** When other developers see a stack being used, they immediately understand that the code is using a LIFO process. This common understanding promotes more readable and maintainable code.
4. **Encapsulation and Abstraction:** A stack hides the underlying complexity and enforces specific rules on how data can be accessed. This abstraction ensures that the data is handled in a consistent way, which can make the code more robust.

### Stack Wrap-Up

Stacks are ideal for processing any data that should be handled last in, first out. The “undo” function in a word processor, for example, is a great use case for a stack. As the user types, we keep track of each keystroke by pushing the keystroke onto the stack. Then, when the user hits the “undo” key, we pop the most recent keystroke from the stack and eliminate it from the document. At this point, their next-to-most recent keystroke is now sitting at the top of the stack, ready to be undone if need be.

### Queues

A queue is a data structure that processes elements in a First-In, First-Out (FIFO) order. It's similar to a stack but operates in a different way. Here's how a queue works:

- **Visualize a Queue:** Imagine a line of people at a movie theater. The person who's first in line is the first one to enter the theater, and new people join the line at the end.
- **Adding Elements:** When you add items to a queue (called "enqueueing"), they go to the end of the line. If you add the numbers 5, 9, and 100, they will be in the queue in that order.
- **Removing Elements:** When you remove items from a queue (called "dequeueing"), you take them from the front of the line. If you start with the queue of 5, 9, and 100, you would remove the 5 first, then the 9.

- **Three Key Restrictions:**
  - You can only insert data at the end.
  - You can only delete data from the front.
  - You can only read the element at the front.
- **Difference from Stacks:** While both queues and stacks are restricted forms of arrays, they function in opposite ways when it comes to removing data. Stacks follow a Last-In, First-Out (LIFO) pattern, while queues use the FIFO pattern.
- **Useful in Various Scenarios:** Queues are helpful in scenarios where you need to maintain the order of processed items, such as handling requests on a web server or managing print jobs in a printer queue.

### Queue Implementation

The queue is a specific data structure, and it may not be provided in some programming languages. However, you can create your own implementation of a queue using an array. Here's a simplified explanation of a Ruby implementation:

1. **Initialization:** A new queue is created with an empty array to hold the data.
2. **Enqueue Method (Insert):** The **enqueue** method allows you to add an element to the end of the queue. In Ruby, you can use the **<<** operator to append the element to the array.
3. **Dequeue Method (Remove):** The **dequeue** method is used to remove and return the first element in the queue. In Ruby, the **shift** method does this by removing the first element of the array.
4. **Read Method (Peek):** The **read** method lets you look at the first element in the queue without removing it. In Ruby, you can use the **first** method to achieve this.

### Queues in Action

Queues play a critical role in various applications, including handling printing jobs and processing asynchronous requests. Here's how a queue might be used to manage a printing system using Ruby:

1. **Creating a Print Manager Class:** A class named **PrintManager** is created, which includes a queue to manage the print jobs.
2. **Queuing Print Jobs:** The method **queue\_print\_job** allows you to add a document (represented as a string) to the print queue.
3. **Running the Print Jobs:** The **run** method reads the front document from the queue, prints it, and then removes it from the queue (dequeues). This continues until the queue is empty.
4. **Example Usage:**

```
print_manager = PrintManager.new
print_manager.queue_print_job("First Document")
print_manager.queue_print_job("Second Document")
print_manager.queue_print_job("Third Document")
print_manager.run
```

This code adds three documents to the queue and prints them in the same order they were received.

5. **Real-world Applications:** The concept of a queue isn't just theoretical; it has practical applications in managing various sequential processes. This includes ensuring that asynchronous requests are handled in order, modeling real-world scenarios like airplanes waiting for takeoff, and patients waiting for medical attention.

## Wrapping Up

As you've seen, stacks and queues are programmers' tools for elegantly handling all sorts of practical algorithms. Now that you've learned about stacks and queues, you've unlocked a new achievement: you can learn about recursion, which depends upon a stack. Recursion also serves as the foundation for many of the more advanced and super-efficient algorithms that I'll cover in the rest of this book.

## Exercises

The following exercises provide you with the opportunity to practice with stacks and queues.

1. If you were writing software for a call center that places callers on hold and then assigns them to "the next available representative," would you use a stack or a queue?

Queue

2. If you pushed numbers onto a stack in the following order: 1, 2, 3, 4, 5, 6, and then popped two items, which number would you be able to read from the stack?

4

3. If you inserted numbers into a queue in the following order: 1, 2, 3, 4, 5, 6, and then dequeued two items, which number would you be able to read from the queue?

3

4. Write a function that uses a stack to reverse a string. (For example, "abcde" would become "edcba".) You can work with our earlier implementation of the Stack class.

```
class Stack
  def initialize
    @data = []
  end

  def push(element)
    @data << element
  end
end
```

```
end

def pop
  @data.pop
end
end

def reverse_string(str)
  stack = Stack.new
  str.each_char { |char| stack.push(char) }
  reversed_str = ''
  str.length.times { reversed_str << stack.pop }
  reversed_str
end

original_str = "abcde"
reversed_str = reverse_string(original_str)

puts reversed_str # Output: "edcba"
```

## CHAPTER 10: Recursively Recurse with Recursion

Recursion is when a function calls itself, like a repeating loop in computer programming. It's an essential idea for complex algorithms. If used well, recursion can make tough problems much easier to solve. It may even feel like magic sometimes. But beware of infinite recursion, where a function calls itself without end. The code snippet provided shows an example of this:

```
function blah() {  
  blah();  
}
```

If you call the **blah()** function, it will call itself forever, which is not useful. However, when used with proper controls and boundaries, recursion becomes a potent tool in problem-solving.

### Recurse Instead of Loop

Recursion can be used as an alternative to loops. For example, you might use a loop to write a countdown function that prints numbers from 10 to 0. Here's what that might look like using a loop:

```
function countdown(number) {  
  for(let i = number; i >= 0; i--) {  
    console.log(i);  
  }  
}  
  
countdown(10);
```

This works fine, but you can also accomplish the same task using recursion:

```
function countdown(number) {  
  console.log(number);  
  countdown(number - 1);  
}
```

The recursive approach works by calling the countdown function within itself, decreasing the value of number by 1 each time until it reaches -1. At that point, the recursion stops, and the numbers have all been printed to the console.

Using recursion can make some code more elegant, but it's not always better or more efficient than a loop. It's another tool in a programmer's toolkit, and understanding when and how to use it can be valuable. In this specific example, the loop and recursive methods achieve the same goal, and neither one is necessarily superior to the other. However, there may be cases where recursion provides a more elegant solution to a complex problem.

### The Base Case

In the countdown function that uses recursion, a problem arises where it keeps printing negative numbers forever. This happens because there's no instruction to tell the function when to stop. To fix this, we can add a condition (or "base case") to stop the recursion when the number reaches 0. Here's how the revised code looks:

```
function countdown(number) {
  console.log(number);
  if(number === 0) {
    return;
  } else {
    countdown(number - 1);
  }
}
```

Now, when the number reaches 0, the function will stop calling itself, and the countdown will end. This base case ensures that the recursion doesn't continue indefinitely.

### Reading Recursive Code

Understanding recursion can be tricky, but it gets easier with practice. Here's a guide to reading and understanding a recursive code to calculate factorials. Factorials are a sequence of multiplying numbers together. The factorial of 3 is  $3 * 2 * 1 = 6$ , and the factorial of 5 is  $5 * 4 * 3 * 2 * 1 = 120$ . The given code calculates the factorial using recursion in Ruby:

```
def factorial(number)
  if number == 1
    return 1
  else
    return number * factorial(number - 1)
  end
end
```

Here's a step-by-step way to understand the code:

1. **Identify the Base Case:** This is the simplest case where the function stops calling itself. Here, it's when **number** is 1; the function returns 1.
2. **Understand the Base Case:** If you call **factorial(1)**, it will return 1.
3. **Understand the Next-to-Last Case:** What happens when the function is close to reaching the base case? For **factorial(2)**, the code will return  $2 * \text{factorial}(1)$ , or 2.
4. **Build Up from There:** Continue to analyze the next cases in the same way. For **factorial(3)**, it will be  $3 * \text{factorial}(2)$ , or 6.
5. **Repeat the Process:** Keep going, building up from the simplest case to the more complex ones, using the previous answers to help you.

By working through the function from the base case and building up, you can better understand how the recursive code works and see how it calculates factorials. It's like putting together the pieces of a puzzle, starting with the simplest piece and adding on from there.

### Recursion in the Eyes of the Computer

When a computer runs a recursive function, like calculating the factorial, it follows a sequence of steps, and it has to remember where it is in the process at every step.

1. **Calling the Function:** Say you call **factorial(3)**. Since 3 is not the base case (1), the computer reaches the line to calculate **number \* factorial(number - 1)**, which leads to **factorial(2)**.
2. **Pausing and Continuing:** Here's where it gets tricky. The computer doesn't finish **factorial(3)**; it pauses that process to start running **factorial(2)**.
3. **Going Deeper:** Now, the computer starts running **factorial(2)**, but again it needs to pause to start running **factorial(1)**, the base case.
4. **Reaching the Base Case:** Once it reaches **factorial(1)**, it has the answer for that part, which is 1.
5. **Working Backwards:** Now, the computer works its way back through the calls. It returns to **factorial(2)**, multiplies 2 by the result of **factorial(1)**, which gives 2.
6. **Continuing Upwards:** Then, it goes back to **factorial(3)**, multiplies 3 by the result of **factorial(2)**, which gives 6.
7. **Completing the Calculation:** Finally, it has the answer for **factorial(3)**, which is 6.

You can imagine this process like a stack of trays. Each time the computer makes a new call to the function, it adds a tray to the stack. When it reaches the base case, it starts to take trays off the stack, working its way back up to the first call, and completing the calculations as it goes.

The computer uses a data structure called a call stack to keep track of where it is in the process. It keeps all the information about each call to the function so it knows what to do next when it comes back to that point. It's like a to-do list that keeps getting longer as it digs deeper into the recursion, and then shorter again as it works its way back up to the original call.

By understanding this, you can see how recursion can be powerful but also how it requires careful handling. If you don't have a clear base case, or if you make too many recursive calls, it can lead to problems like an infinite loop or a stack overflow error.

## The Call Stack

1. **What is the Call Stack?:** The call stack is like a stack of notes the computer uses to remember what it was doing. It keeps track of the functions the computer is in the middle of calling.
2. **Starting the Calculation:** When calling **factorial(3)**, the computer puts a note on the stack to remember it's in the middle of this calculation.
3. **Going Deeper into Recursion:** Before finishing **factorial(3)**, the computer calls **factorial(2)** and adds another note on top of the stack to remember this.
4. **Reaching the Base Case:** When the computer calls **factorial(1)**, it adds one more note. Since 1 is the base case, it completes **factorial(1)** without calling the function again.
5. **Working Backwards - Using the Stack:** Now, the computer looks at the stack and sees that it needs to complete **factorial(2)**. It takes the result of **factorial(1)**, multiplies it by 2, and finishes **factorial(2)**, removing the top note from the stack.
6. **Finishing Up:** Next, the computer sees the note for **factorial(3)**, takes the result of **factorial(2)**, multiplies it by 3, and completes **factorial(3)**, clearing the last note from the stack.

7. **The Process is Complete:** With no more notes left on the stack, the computer knows it's done. The final result of `factorial(3)` is 6.
8. **Understanding the Flow:** Essentially, the calculation goes down to the base case (`factorial(1)`) and then works its way back up, passing values from one level to the next. The call stack ensures that each step along the way is done in the right order.

By understanding the call stack, you can see how it helps the computer navigate the complex process of recursion, keeping track of what it's doing at each step and ensuring everything is done in the correct sequence. It's like having a to-do list where you keep adding new tasks on top, then work your way down through the tasks, completing them in reverse order from how they were added.

## Stack Overflow

1. **Infinite Recursion:** Imagine a function that keeps calling itself forever, without stopping. It's like a never-ending loop in the code.
2. **What Happens to the Call Stack:** Every time a function calls itself, the computer puts a reminder on the call stack. If the function keeps calling itself forever, the call stack keeps getting more and more reminders, piling up without end.
3. **Running Out of Memory:** The computer only has so much short-term memory to keep track of these reminders. If the call stack grows too big, it fills up this memory.
4. **Stack Overflow Error:** When there's no more room for reminders, the computer hits an error called a "stack overflow." It's like trying to put more books on a bookshelf than it can hold. The computer says, "I can't do this anymore, I'm out of space!" and stops the recursion.
5. **Why It's a Problem:** Infinite recursion and stack overflow are problems because they can crash a program or cause it to behave unpredictably.
6. **Solution:** To avoid this, you need to make sure that recursive functions have a clear stopping point, called a base case, so they don't go on forever.

## Filesystem Traversal

Let's discuss the concept of filesystem traversal using recursion.

1. **What Is Filesystem Traversal?:** Imagine a folder on your computer that contains other folders (subdirectories) and files. Traversing the filesystem means exploring all these folders and their contents.
2. **Why Is It Complex?:** Sometimes, folders contain other folders, which might contain even more folders, and so on. This creates multiple layers or "depths" that you need to explore.
3. **Initial Approach Without Recursion:** Initially, you might create a script that looks inside a folder, finds all the subfolders, and prints their names. But if you want to go deeper into subfolders of those subfolders, things get complicated. You'd have to create nested loops for each new level you want to explore. This quickly becomes unmanageable.



4. **The Problem:** What if you want to explore all the subdirectories, regardless of how many levels deep they go? You might not even know how many levels there are.
5. **The Solution with Recursion:** Instead of manually writing nested loops, you can use recursion. You create a function that explores a folder. If it finds a subfolder, it calls itself on that subfolder, again and again, as deep as it needs to go.
6. **Recursive Code:**

```
def find_directories(directory)
  Dir.foreach(directory) do |filename|
    if File.directory?("#{directory}/#{filename}") &&
      filename != "." && filename != ".."
      puts "#{directory}/#{filename}"

      # Recursively call this function on the subdirectory:
      find_directories("#{directory}/#{filename}")
    end
  end
end
```

7. **Why This Works:** This code keeps calling itself for each new subdirectory it finds, going deeper and deeper. It doesn't matter how many levels there are; the function will explore them all.
8. **Visualizing:** Think of this like a robot exploring a maze of rooms. When it finds a door to a new room, it goes in and starts exploring that room, finding more doors and rooms within. It keeps going deeper until there are no more new rooms to explore.
9. **Applications:** This approach can be used in various computer algorithms, like Depth-First Search, which you'll encounter later.

Through recursion, you can create a neat, simple script that can explore an entire directory structure, no matter how complex, without needing to know how many layers deep it goes. It's a clear example of the power and elegance of recursive programming.

## Wrapping Up

As you've seen with the filesystem example, recursion is often a great choice for an algorithm in which the algorithm needs to dig into an arbitrary number of levels deep into something. You've now seen how recursion works and how incredibly useful it can be. You've also learned how to walk through and read recursive code. However, most people have a difficult time writing their own recursive functions when they're first starting out. In the next chapter, we'll explore techniques to help you learn to write recursively. Along the way, you'll also discover other important use cases where recursion can be an incredible tool.

## Exercises

1. The following function prints every other number from a low number to a high number. For example, if low is 0 and high is 10, it would print:

0

2

4  
6  
8  
10

Identify the base case in the function:

```
def print_every_other(low, high)
  return if low > high
  puts low
  print_every_other(low + 2, high)
end
```

The base case in the given function is return **if low > high**. This is what stops the recursion when the low number becomes greater than the high number.

2. My kid was playing with my computer and changed my factorial function so that it computes factorial based on  $(n - 2)$  instead of  $(n - 1)$ . Predict what will happen when we run `factorial(10)` using this function:

```
def factorial(n)
  return 1 if n == 1
  return n * factorial(n - 2)
end
```

The modified factorial function is computing the product by decrementing  $n$  by 2 instead of 1. So when we run `factorial(10)`, the function will calculate  $10 * 8 * 6 * 4 * 2$ . Since there is a base case for  $n == 1$ , the function will not reach that case and result in infinite recursion. To fix it, we should add another base case:

```
return 1 if n <= 1
```

3. Following is a function in which we pass in two numbers called `low` and `high`. The function returns the sum of all the numbers from `low` to `high`. For example, if `low` is 1, and `high` is 10, the function will return the sum of all numbers from 1 to 10, which is 55. However, our code is missing the base case, and will run indefinitely! Fix the code by adding the correct base case:

```
def sum(low, high)
  return high + sum(low, high - 1)
end
```

The missing base case in the sum function should return the value of `low` when `low` is equal to `high`. Here's the corrected code:

```
def sum(low, high)
  return low if low == high
  return high + sum(low, high - 1)
end
```

4. Here is an array containing both numbers as well as other arrays, which in turn contain numbers and arrays:

```
array = [ 1,
          2,
          3,
          [4, 5, 6],
          7,
          [8,
            [9, 10, 11,
              [12, 13, 14]
            ]
          ],
          [15, 16, 17, 18, 19,
            [20, 21, 22,
              [23, 24, 25,
                [26, 27, 29]
              ], 30, 31
            ], 32
          ], 33
        ]
```

Write a recursive function that prints all the numbers (and *just* numbers).

```
def print_numbers(array)
  array.each do |element|
    if element.is_a?(Array)
      print_numbers(element) # If the element is an array, recursively call the
                             # function
    else
      puts element # If the element is not an array, print the number
    end
  end
end
```

```
array = [1, 2, 3, [4, 5, 6], 7, [8, [9, 10, 11, [12, 13, 14]]], [15, 16, 17, 18, 19,
[20, 21, 22, [23, 24, 25, [26, 27, 29]], 30, 31]], 32], 33]
```

```
print_numbers(array)
```

The function **print\_numbers** checks each element of the array. If the element is an array itself, the function calls itself on that sub-array. If it's a number, the function prints it. This will recursively explore all nested arrays and print every number it finds, no matter how deeply nested.

## CHAPTER 11: Learning to Write in Recursive

In the previous chapter, you were introduced to recursion and its fundamentals. Despite understanding the concept, writing recursive functions may still be challenging. By practicing and recognizing patterns, I uncovered methods to more easily write recursive functions, and I'll share those techniques with you. This chapter will illuminate more areas where recursion is effective, but we won't be discussing its efficiency or time complexity. That topic will be covered in the next chapter. Our current focus is solely on nurturing a recursive mindset.

### Recursive Category: Repeatedly Execute

While solving different recursive problems, I noticed they can be grouped into categories, with each category having its own solving technique. The simplest category I found involves algorithms that repeatedly execute a task. A prime example is the NASA spacecraft countdown algorithm, where the code prints numbers like 10, 9, 8, down to 0. The essence of the code is that it's repeatedly printing a number, as shown in this JavaScript function:

```
function countdown(number) {
  console.log(number);

  if(number === 0) { // number being 0 is the base case
    return;
  } else {
    countdown(number - 1);
  }
}
```

In this category of problems, the final line usually consists of a single call to the recursive function, such as **countdown(number - 1)**, to make the next recursive call. Another example is a directory-printing algorithm. It continually performs the task of printing directory names. The Ruby code for this task looks like:

```
def find_directories(directory)
  Dir.foreach(directory) do |filename|
    if File.directory?("#{directory}/#{filename}") &&
      filename != "." && filename != ".."
      puts "#{directory}/#{filename}"

      # Recursively call this function on the subdirectory:
      find_directories("#{directory}/#{filename}")
    end
  end
end
```

Here too, the last line is a simple call to the recursive function, **find\_directories("#{directory}/#{filename}")**, which triggers the function again.

### Recursive Trick: Passing Extra Parameters

Let's explore a recursive problem to double each number in an array in place. It falls under the category of problems that "repeatedly execute" a task. Generally, there are two approaches to manipulating data. The first is to create a new array with modified data, leaving the original array unchanged. The second is called in-place modification, where the original array itself is changed.

We want to write a recursive function in Python to double the numbers in an array without creating a new array. An iterative approach using a loop might look like this:

```
def double_array(array):
    index = 0
    while (index < len(array)):
        array[index] *= 2
        index += 1
```

However, we want to use recursion, so we need a way to keep track of and increment an index. The trick is to pass extra parameters. We modify the function to accept two arguments, the array itself, and an index:

```
def double_array(array, index):
    array[index] *= 2
    double_array(array, index + 1)
```

We increment the index in each recursive call, allowing us to keep track of it, as we would in a loop. Using default parameters, we can call the function without passing in the index and still use it in successive calls. This recursive function doubles the numbers in an array in place, by passing in an extra index parameter. It demonstrates a common and handy technique in writing recursive functions, using extra function parameters to maintain state across recursive calls.

### Recursive Category: Calculations

In recursion, besides the first category where tasks are repeatedly executed, there is another category where a calculation is performed based on a subproblem. A subproblem is a smaller instance of the same problem. It can be used to solve the larger problem. Consider the factorial of 6 (6!):

- Using a loop, you would multiply numbers from 1 to 6, building up the result.
- Using recursion, you can define the factorial of 6 (6!) as 6 multiplied by the factorial of 5 (5!).

Here's a Ruby code snippet for a recursive factorial calculation:

```
def factorial(number)
  if number == 1
    return 1
  else
    return number * factorial(number - 1)
  end
end
```

The line **return number \* factorial(number - 1)** utilizes the subproblem by multiplying the current number with the factorial of the previous number.

### Two Approaches to Calculations

**Bottom Up:** You build the solution from the smallest part to the final answer. You can achieve this using a loop or recursion by passing extra parameters. Example in Ruby:

```
def factorial(n, i=1, product=1)
  return product if i > n
  return factorial(n, i + 1, product * i)
end
```

This approach uses recursion to achieve a bottom-up strategy by incrementing an index and multiplying the product as it goes.

**Top Down:** You divide the problem into smaller subproblems, then combine the answers to solve the original problem. This can only be achieved with recursion. The example of calculating the factorial using recursion is a top-down approach, as it computes the result based on the subproblem (factorial of the previous number).

### Why Recursion?

- The bottom-up approach can be done using a loop or recursion, but there's no specific advantage to using recursion.
- The top-down approach needs recursion, and it's one of the key factors that makes recursion a valuable tool in programming.

### Top-Down Recursion: A New Way of Thinking

What is Top-Down Recursion?

- **Mentally “Kicking the Problem Down the Road”:** In top-down recursion, you don't need to think about every detailed step. Instead, you can focus on the larger problem and leave the details to be solved by the subproblems.
- **Example:** The key line in the top-down factorial implementation

```
(return number * factorial(number - 1))
```

shows how the calculation is based on the result of the subproblem

```
(factorial(number - 1)).
```

What Makes It Special?

1. **Freedom from Details:** You don't have to know how the subproblem is solved. Even though you're calling the very function you're in, you only need to understand that it will return the correct value.
2. **Efficient Mental Strategy:** It allows you to tackle a problem in a completely new way. You can concentrate on the main problem, trusting that the subproblem will handle the smaller details.
3. **Relaxed Approach:** Top-down recursion lets you think more broadly without getting bogged down in the complexities. You don't have to worry about how the calculation works; you can rely on the subproblem to manage the specifics.

### The Top-Down Thought Process

If you're new to top-down recursion, it may seem unfamiliar. But with some time and practice, it becomes an intuitive way to solve problems. Here's how you can approach it:

1. **Pretend the Function Exists Already:** Imagine that the function you're going to write is already created by someone else. You don't need to know how it works inside, just what it does.
2. **Find the Subproblem:** Identify the smaller part of the problem that you're trying to solve. This subproblem is a smaller instance of the main problem.
3. **Work with the Subproblem:** See what results you get when you call the function on the subproblem. Focus on how the subproblem connects to the overall problem rather than getting lost in the details.

Though these steps may seem unclear initially, they start to make sense as you apply them in practice.

## Array Sum

Say we have to write a function called **sum** that sums up all the numbers in a given array. For example, if we pass the array, [1, 2, 3, 4, 5] into the function, it'll return 15, which is the sum of those numbers.

1. **Pretend the Function Exists:** Imagine that the **sum** function you're about to write is already there, and it works perfectly. This may feel strange, but it's part of the process.
2. **Identify the Subproblem:** Break down the problem into smaller parts. In this case, the subproblem is the array minus the first element. So if the initial array is [1, 2, 3, 4, 5], the subproblem is [2, 3, 4, 5].
3. **Apply the Function to the Subproblem:** Think about what happens when you call **sum([2, 3, 4, 5])**. Since the function "works," it will return 14.
4. **Add the First Number:** To get the sum of the original array, add the first number to the sum of the subproblem. In this case,  $1 + 14 = 15$ .
5. **Write the Code:** In Ruby, you can implement this logic with the following code:

```
def sum(array)
  return array[0] + sum(array[1, array.length - 1])
end
```

6. **Add the Base Case:** Make sure to handle the case when there's only one element in the array. This prevents infinite recursion and ensures that the function will eventually return a result.

```
def sum(array)
  # Base case: only one element in the array:
  return array[0] if array.length == 1
  return array[0] + sum(array[1, array.length - 1])
end
```

The beauty of this approach is that it allows you to solve the problem without thinking about the complex details of adding all the numbers together. By breaking down the problem into smaller parts and assuming that the function works on those smaller parts, you simplify the overall process. The approach relies on recursive thinking, where the problem is repeatedly reduced to a smaller version of itself. By trusting that the smaller version works, you eventually build up to the complete solution. It's an elegant way to solve problems and can be quite powerful once you get used to thinking in this way.

## String Reversal

Let's try another example. We're going to write a reverse function that reverses a string. So, if the function accepts the argument "abcde", it'll return "edcba".

1. **Identify the Subproblem:** Consider the next-to-smallest version of the problem. If you want to reverse "abcde," the subproblem is the original string minus the first character, i.e., "bcde."
2. **Pretend the Function Exists:** Imagine that the **reverse** function is already implemented and works as expected.
3. **Apply the Function to the Subproblem:** If you call **reverse("bcde")**, it would return "edcb." Since you've pretended that the function already works, you can rely on this result.
4. **Add the First Character to the End:** The original string's first character is "a," so add it to the end of the reversed subproblem's result. This gives you "edcba."
5. **Write the Code:** The Ruby code for this logic would look like this:

```
def reverse(string)
  return reverse(string[1, string.length - 1]) + string[0]
end
```

6. **Add the Base Case:** As with the sum example, you need to handle the base case. In this case, the base case is when the string has only one character.

```
def reverse(string)
  # Base case: string with just one character
  return string[0] if string.length == 1
  return reverse(string[1, string.length - 1]) + string[0]
end
```

Just like with summing an array, this process allows you to avoid thinking about the nitty-gritty details of reversing a string. You break the problem down into a subproblem and trust that the reverse function will handle it. By adding the original string's first character to the reversed subproblem's result, you achieve the complete solution.

## Counting X

Let's write a function called `count_x` that returns the number of "x's" in a given string.

1. **Identify the Subproblem:** Focus on the original string without its first character. For the string "axbxcxd," the subproblem is "xbxcxd."
2. **Assume the Function is Implemented:** Pretend that `count_x` is already working. If you call `count_x("xbxcxd")`, you would get 3, the number of "x" in the subproblem.
3. **Add 1 if the First Character is "x":** Check if the first character is "x." If it is, add 1 to the count from the subproblem. If it's not, just take the count from the subproblem.
4. **Write the Code:** Here's how you could write this logic in Ruby:



```

def count_x(string)
  if string[0] == "x"
    return 1 + count_x(string[1, string.length - 1])
  else
    return count_x(string[1, string.length - 1])
  end
end

```

5. **Handle the Base Case:** The base case here is when the string is empty. The function returns 0 because there will never be an "x" in an empty string.

```

def count_x(string)
  # Base case:
  if string.length == 1
    if string[0] == "x"
      return 1
    else
      return 0
    end
  end

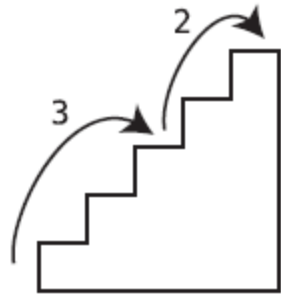
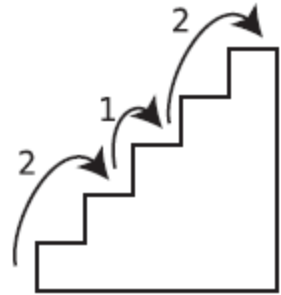
  if string[0] == "x"
    return 1 + count_x(string[1, string.length - 1])
  else
    return count_x(string[1, string.length - 1])
  end
end

```

This process leverages the strength of recursive thinking. You focus on one part of the problem, trusting that the rest of it can be solved with the function itself. The base case ensures that you don't get caught in an infinite loop. The trick used in the base case, where calling **string[1, 0]** returns an empty string, could be utilized in many other recursive examples as well, including in dealing with arrays. It makes handling the base case more straightforward and elegant.

### The Staircase Problem

The Staircase Problem is a classic computational problem. Here's how it works and the solution using a recursive approach: You have a staircase of N steps, and you can climb one, two, or three steps at a time. How many different ways can you reach the top?



**1. Start with Simpler Cases:**

- For 1 step, there's 1 path.
- For 2 steps, there are 2 paths.

1, 1  
2

- For 3 steps, there are 4 paths.

1, 1, 1  
1, 2  
2, 1  
3

- For 4 steps, there are 7 paths.

1, 1, 1, 1  
1, 1, 2  
1, 2, 1  
1, 3  
2, 1, 1  
2, 2  
3, 1

**2. Think Recursively:**

- For N steps, you can find the total paths by considering the paths for (N - 1), (N - 2), and (N - 3) steps.

3. **Write the Code:** The recursive function follows this logic:

```
def number_of_paths(n)
  return 0 if n < 0
  return 1 if n == 1 || n == 0
  return number_of_paths(n - 1) + number_of_paths(n - 2) +
    number_of_paths(n - 3)
end
```

#### Explanation:

- **Subproblems:** The first step is to break the problem into smaller parts. The paths for N steps can be found from the paths of (N - 1), (N - 2), and (N - 3) steps.
- **Base Cases:** These are the simplest cases that don't need to be broken down further.
- **Combining Subproblems:** You add the number of paths from the three previous steps to find the total paths for N steps.

This problem might look complicated with loops, especially as the number of steps grows. The recursive approach simplifies the problem, allowing you to think about it in terms of smaller, similar problems. It's a top-down approach, starting with the overall problem and breaking it into parts.

**Be Mindful of Efficiency:** While the code above is correct, calling the function with a large number of steps might slow down because of redundant calculations. This can be optimized using techniques like memoization to store previously calculated results.

#### Staircase Problem Base Case

The Staircase Problem's base case is an essential part of solving the problem recursively, and there are different ways to define it. A straightforward way is to explicitly define the base cases for when the number of steps is less than or equal to 3:

```
def number_of_paths(n)
  return 0 if n <= 0
  return 1 if n == 1
  return 2 if n == 2
  return 4 if n == 3
  return number_of_paths(n - 1) + number_of_paths(n - 2) +
    number_of_paths(n - 3)
end
```

This approach is intuitive and clearly defines the base cases. Another way to define the base cases is to leverage the computation itself. By setting up the base cases for n less than or equal to 1, you can "rig" the system to calculate the right numbers:

```

def number_of_paths(n)
  return 0 if n < 0
  return 1 if n == 1 || n == 0
  return number_of_paths(n - 1) + number_of_paths(n - 2) +
    number_of_paths(n - 3)
end

```

This version is less intuitive but more concise, using just two lines for the base cases.

- In the second approach, **number\_of\_paths(0)** and **number\_of\_paths(1)** return 1, and **number\_of\_paths(-1)** returns 0.
- These base cases are designed so that the sums for **number\_of\_paths(2)** and **number\_of\_paths(3)** automatically yield the correct results (2 and 4 respectively).

The base case is essential to the recursive solution for the Staircase Problem, and there's flexibility in how it can be defined. The first approach is more intuitive but verbose, while the second approach is more succinct but may be less transparent. Both methods leverage the top-down recursive mindset, making the problem more manageable.

## Anagram Generation

Anagram Generation is a complex recursive problem. An anagram rearranges the letters of a word to form new words. Let's write a function to return all possible anagrams of a given string:

1. **Base Case:** If the string has only one character, return an array containing that character itself. This represents the simplest form of the problem.
2. **Recursive Case:** For strings with more than one character, the approach can be as follows:
  - Find the anagrams of the substring that starts from the second character of the string.
  - Take each anagram of that substring and insert the first character of the original string at every possible position within it.
  - Add each of these new strings to a collection.

## Ruby Code:

```

def anagrams_of(string)
  return [string[0]] if string.length == 1
  collection = []
  substring_anagrams = anagrams_of(string[1..-1])
  substring_anagrams.each do |substring_anagram|
    (0..substring_anagram.length).each do |index|
      copy = substring_anagram.dup
      collection << copy.insert(index, string[0])
    end
  end
end

```

```
    return collection
end
```

### Explanation:

- The base case handles strings of length one, returning the character itself as an anagram.
- The recursive case finds anagrams of a substring (without the first character), and then places the first character of the original string at every position in those anagrams.
- This process builds up anagrams for progressively larger substrings, eventually yielding the anagrams for the entire string.

### Example:

For the string "abc", the function will return:

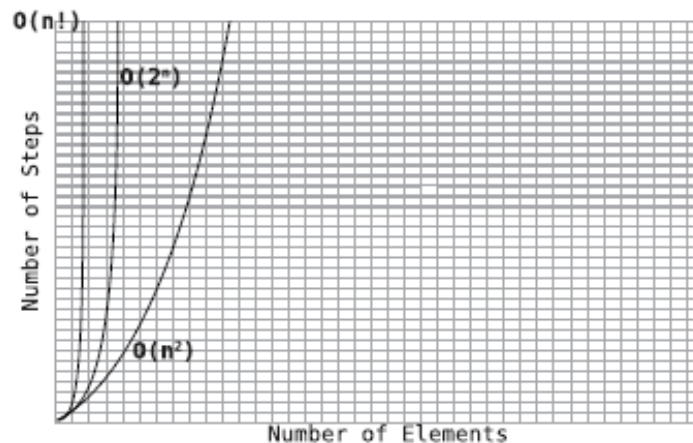
```
["abc", "acb", "bac", "bca", "cab", "cba"]
```

### The Efficiency of Anagram Generation

The efficiency of the anagram generation algorithm leads us to an interesting pattern in terms of time complexity, one that we can categorize using Big O notation in a way we may not have seen before. When we analyze the number of anagrams we generate for various string lengths, we notice the pattern follows a factorial:

- For 3 characters:  $3 * 2 * 1 = 6$  anagrams
- For 4 characters:  $4 * 3 * 2 * 1 = 24$  anagrams
- For 5 characters:  $5 * 4 * 3 * 2 * 1 = 120$  anagrams
- For 6 characters:  $6 * 5 * 4 * 3 * 2 * 1 = 720$  anagrams

This pattern is a factorial, represented by the symbol "!", where the factorial of a number is the product of all positive integers less than or equal to that number. In terms of Big O notation, the time complexity for this algorithm is  $O(N!)$ , known as factorial time complexity.  $O(N!)$  is an extremely slow category of Big O, especially compared to other "slow" Big O categories like  $O(N^2)$ ,  $O(N^3)$ , etc. It grows rapidly with the increase in N (the length of the string).



The factorial time complexity isn't something we can optimize further for this particular problem. Since we are tasked with generating all possible anagrams of an N-character word, and there simply are N! anagrams, the algorithm must consider all these possibilities.

## Wrapping Up

Learning to write functions that use recursion certainly takes practice. But you're now armed with tricks and techniques that will make the learning process easier for you. We're not done with our journey through recursion just yet, though. While recursion is a great tool for solving a variety of problems, it can actually slow your code down *a lot* if you're not careful. In the next chapter, you'll learn how to wield recursion while still keeping your code nice and speedy.

## Exercises

The following exercises provide you with the opportunity to practice with recursion.

1. Use recursion to write a function that accepts an array of strings and returns the total number of characters across all the strings. For example, if the input array is ["ab", "c", "def", "ghij"], the output should be 10 since there are 10 characters in total.

```
def total_characters(arr):
    if len(arr) == 0:
        return 0
    return len(arr[0]) + total_characters(arr[1:])

strings = ["ab", "c", "def", "ghij"]
print(total_characters(strings)) # Output: 10
```

2. Use recursion to write a function that accepts an array of numbers and returns a new array containing just the even numbers.

```
def even_numbers(arr):
    if len(arr) == 0:
        return []
    if arr[0] % 2 == 0:
        return [arr[0]] + even_numbers(arr[1:])
    else:
        return even_numbers(arr[1:])

numbers = [1, 2, 3, 4, 5, 6]
print(even_numbers(numbers)) # Output: [2, 4, 6]
```

3. There is a numerical sequence known as “Triangular Numbers.” The pattern begins as 1, 3, 6, 10, 15, 21, and continues onward with the Nth number in the pattern, which is N plus the previous number. For example, the 7th number in the sequence is 28, since it’s 7 (which is N) plus 21 (the previous number in the sequence). Write a function that accepts a number for N and returns the correct number from the series. That is, if the function was passed the number 7, the function would return 28.

```
def triangular_number(N):  
    if N == 1:  
        return 1  
    return N + triangular_number(N - 1)
```

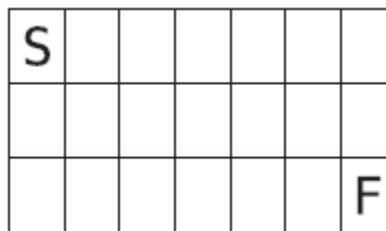
```
N = 7  
print(triangular_number(N)) # Output: 28
```

4. Use recursion to write a function that accepts a string and returns the first index that contains the character “x.” For example, the string, "abcdefghijklmnopqrstuvwxy<sup>z</sup>" has an “x” at index 23. To keep things simple, assume the string *definitely* has at least one “x.”

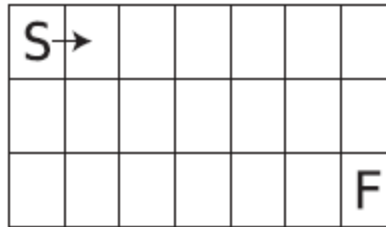
```
def find_x(s, index=0):  
    if s[index] == 'x':  
        return index  
    return find_x(s, index + 1)
```

```
string = "abcdefghijklmnopqrstuvwxyz"  
print(find_x(string)) # Output: 23
```

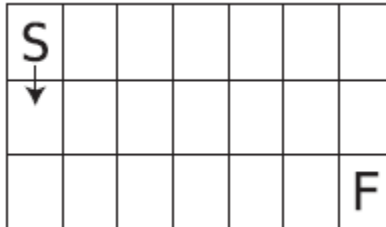
5. This problem is known as the “Unique Paths” problem: Let’s say you have a grid of rows and columns. Write a function that accepts a number of rows and a number of columns, and calculates the number of possible “shortest” paths from the upper-leftmost square to the lower-rightmost square. For example, here’s what the grid looks like with three rows and seven columns. You want to get from the “S” (Start) to the “F” (Finish).



By “shortest” path, I mean that at every step, you’re moving either one step to the right:



or one step downward:



Again, your function should calculate the number of shortest paths.

```
def unique_paths(rows, columns):
    if rows == 1 or columns == 1:
        return 1
    return unique_paths(rows - 1, columns) + unique_paths(rows, columns - 1)
```

```
rows, columns = 3, 7
print(unique_paths(rows, columns)) # Output: 28
```



## CHAPTER 12: Dynamic Programming

You learned previously how to use recursion to tackle complex problems. While recursion is a powerful tool, it can cause issues if misused, leading to slow performance in certain cases like  $O(2^N)$ . This chapter introduces simple and effective methods to identify and rectify common speed traps in recursive code, transforming potential pitfalls into manageable solutions. The focus is on how to express algorithms in terms of Big O and applying techniques to fix performance issues. These methods are surprisingly simple and can turn recursive challenges into more efficient solutions.

### Unnecessary Recursive Calls

Here's a recursive function that finds the greatest number from an array:

```
def max(array)
  # Base case - if the array has only one element, it is
  # by definition the greatest number:
  return array[0] if array.length == 1

  # Compare the first element with the greatest element
  # from the remainder of the array. If the first element
  # is greater, return it as the greatest number:
  if array[0] > max(array[1, array.length - 1])
    return array[0]

  # Otherwise, return the greatest number from the remainder of the array:
  else
    return max(array[1, array.length - 1])
  end
end
```

The function `max` recursively finds the largest number in an array by comparing the first element with the maximum number from the rest of the array. If the first element is greater, it's returned; otherwise, the function calls itself again with the rest of the array. The code does work, but there's an inefficiency. The expression `max(array[1, array.length - 1])` is repeated twice in the conditional statement, triggering multiple unnecessary recursive calls.

To illustrate the issue, consider the array `[1, 2, 3, 4]`. The function will compare 1 with the max of `[2, 3, 4]`, then 2 with the max of `[3, 4]`, and so on until the base case of `[4]` is reached. The repeated expression in the code causes extra recursive calls that make the function less efficient.

### Max Recursive Walk-Through

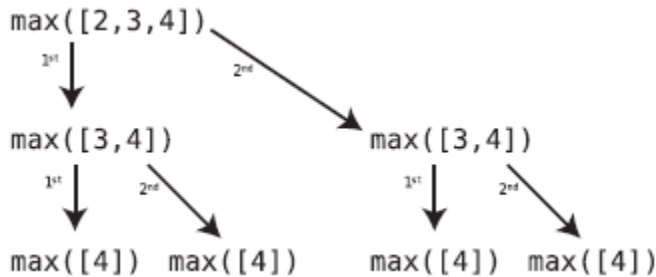
The function `max` is defined as follows:

```
def max_element(arr):
  if len(arr) == 1:
    return arr[0]
  else:
    rest_max = max_element(arr[1:])
```

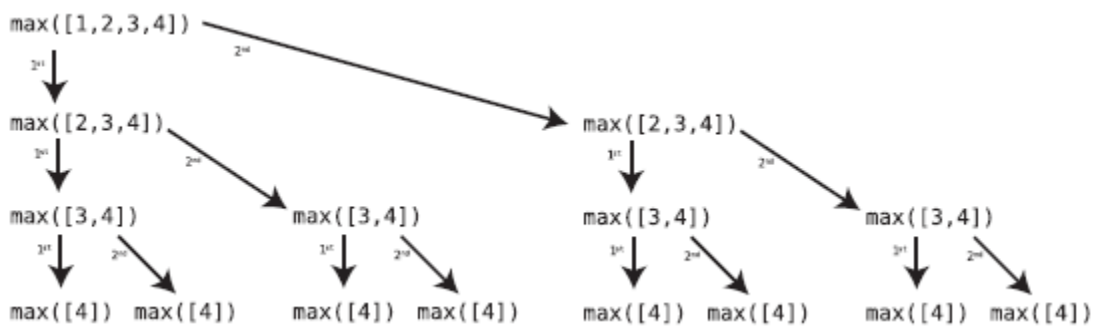
```
return arr[0] if arr[0] > rest_max else rest_max
```

When calling `max([4])`, it returns 4 due to the base case. When moving up and calling `max([3, 4])`, the function triggers two calls to `max([4])`, once in each half of the conditional statement. The problem amplifies as you move up the call chain:

- In `max([2, 3, 4])`, it ends up calling `max([3, 4])` twice.



- In `max([1, 2, 3, 4])`, the max function is triggered 15 times.



You can see this by adding a print statement in the function, printing "RECURSION" 15 times when calling `max([1, 2, 3, 4])`. The problem lies in the repeated function calls that have already been computed, such as calling `max([4])` eight times when once would suffice. This redundancy leads to unnecessary computations and inefficiency in the code.

### The Little Fix for Big O

The original function was inefficient because it repeatedly called the same recursive function. The solution is to call the max function only once for the remainder of the array and store the result in a variable. Here's the revised code:

```

def max(array)
  return array[0] if array.length == 1
  # Calculate the max of the remainder of the array
  # and store it inside a variable:
  max_of_remainder = max(array[1, array.length - 1])
  # Comparison of first number against this variable:
  if array[0] > max_of_remainder
    return array[0]
  else
    return max_of_remainder
  end
end

```

By saving the result of the recursive call in a variable, you avoid redundant computations. The function is now called just four times instead of fifteen, greatly improving efficiency. It's a simple change, but it makes a significant difference in performance.

### The Efficiency of Recursion

The first version of the function calls itself twice for each element in the array, leading to an efficiency of  $O(2^N)$ , where  $N$  is the number of elements in the array. This is an extremely slow algorithm because the number of calls roughly doubles when you add one more element to the data. The table below shows how many times the function gets called for various array sizes, and this pattern of doubling is evident.

N Elements	Number of Calls
1	1
2	3
3	7
4	15
5	31

The second version is an improved version and is more efficient. It only calls itself as many times as there are elements in the array, resulting in an efficiency of  $O(N)$ . Even if there are multiple steps in the function, such as five, the time complexity can be reduced to  $O(N)$ . This improvement demonstrates that avoiding extra recursive calls is essential for making recursion fast. A small change to the code, such as storing a computation in a variable, can significantly change the speed of the function from  $O(2^N)$  to  $O(N)$ .

### Overlapping Subproblems

Let's discuss the concept of "overlapping subproblems" through the example of calculating numbers in the Fibonacci sequence.

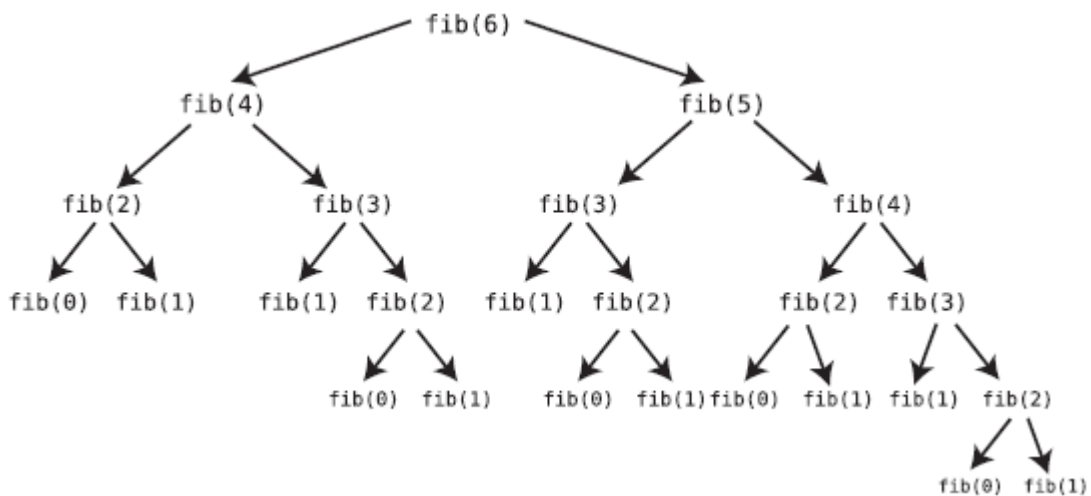
1. **Fibonacci Sequence:** This sequence starts with 0 and 1, and each subsequent number is the sum of the previous two. For instance, the sequence begins as 0, 1, 1, 2, 3, 5, 8, 13, 21, and so on.

2. **Python Function:** The Python function `fib(n)` below returns the Nth number in the Fibonacci sequence using recursion. If `n` is 0 or 1, the function returns `n`. Otherwise, it calls itself twice to get the previous two numbers and returns their sum.

```
def fib(n):
    # The base cases are the first two numbers in the series:
    if n == 0 or n == 1:
        return n

    # Return the sum of the previous two Fibonacci numbers:
    return fib(n - 2) + fib(n - 1)
```

3. **Problem with the Function:** While the function looks neat, it is inefficient due to calling itself twice. This leads to a time complexity of  $O(2^N)$ , which is slow.
4. **Example:** When computing the sixth number (`fib(6)`), it calls both `fib(4)` and `fib(5)`, leading to duplicated calls like `fib(3)` and others.



5. **Overlapping Subproblems:** This is a situation where solving the problem involves solving smaller versions of the same problem (subproblems), and these subproblems overlap, meaning they are calculated multiple times. In the Fibonacci example, `fib(n - 2)` and `fib(n - 1)` end up making many of the same calculations.
6. **Challenge:** Optimizing this recursive Fibonacci sequence isn't straightforward because there's not just one piece of data that can be saved to make it more efficient. The overlapping calls seem to put the algorithm at a dead end, causing it to operate at a slow pace.

### Dynamic Programming through Memoization

Dynamic programming is a method used to optimize recursive problems that have overlapping subproblems, and one of the techniques to achieve this optimization is memoization. **Memoization** is a technique (not to be confused with the word "memorization") that helps reduce repetitive recursive calls in cases where the same calculations are performed multiple times. This is done by "remembering" previously computed results.

Whenever a calculation is performed, the result is stored in a hash table. Next time the same calculation is needed, the function first checks the hash table to see if the result is already there. If it is, the result is used directly; if not, the calculation is performed and then added to the hash table. In the Fibonacci example, the

function stores results for each call in a hash table. For example, after computing fib(3), fib(4), fib(5), and fib(6), the hash table will contain:

```
{
  3: 2,
  4: 3,
  5: 5,
  6: 8
}
```

Then, if fib(4) needs to be computed again, it first checks the hash table and finds the result there, eliminating the need for additional recursive calls.

One challenge is how each recursive function can access the same hash table. This is handled by passing the hash table as a second parameter to the function. Since the hash table is a specific object in memory, it can be passed from one recursive call to the next, even as it's being modified along the way. Memoization targets the core issue with overlapping subproblems, where the same calculations are repeatedly performed. By storing each new calculation in a hash table for future use, the function only performs a calculation if it has never been made before. This significantly optimizes the performance of algorithms with overlapping subproblems.

### Implementing Memoization

Memoization is a powerful technique to optimize the calculation of the Fibonacci sequence, reducing the complexity from  $O(2^N)$  to  $O(N)$ . Here's how it's implemented in Python:

1. **Function with Two Arguments:** The Fibonacci function is modified to accept two arguments: **n** and a hash table called **memo**. This hash table will be used to store previously computed results.

```
def fib(n, memo):
```

2. **Calling the Function:** When calling the function for the first time, we pass in the desired number **n** and an empty hash table.

```
fib(6, {})
```

3. **Checking the Hash Table:** Before computing a Fibonacci number, we check whether it has already been calculated by looking it up in the hash table. If it's found, we simply return the stored result.
4. **Storing in Hash Table:** If the number hasn't been computed before, we calculate it using recursion and then store the result in the hash table **memo[n] = fib(n - 2, memo) + fib(n - 1, memo)**.
5. **Passing the Hash Table:** Each time the function calls itself, it passes along the **memo** hash table, ensuring that all calls can access the same stored results.
6. **Optional Default Value:** The hash table can also be set to a default value, so you don't have to explicitly pass an empty hash table when first calling the function: **def fib(n, memo={})**.
7. **Result:** By using memoization, the function avoids redundant calculations, and the number of recursive calls becomes  $(2N) - 1$ . In Big O notation, this is simplified to  $O(N)$ , a substantial improvement over the non-memoized version with  $O(2^N)$  complexity.

Here's the code:

```

def fib(n, memo):
    if n == 0 or n == 1:
        return n

    # Check the hash table (called memo) to see whether fib(n)
    # was already computed or not:
    if not memo.get(n):
        # If n is NOT in memo, compute fib(n) with recursion
        # and then store the result in the hash table:
        memo[n] = fib(n - 2, memo) + fib(n - 1, memo)

    # By now, fib(n)'s result is certainly in memo. (Perhaps
    # it was there before, or perhaps we just stored it there
    # in the previous line of code. But it's certainly there now.)
    # So let's return it:
    return memo[n]

```

To get the 6th Fibonacci number, you would call **fib(6)**.

The memoization technique involves checking and storing results in a hash table to avoid repeated calculations. It turns the calculation of the Fibonacci sequence from a time-consuming process into an efficient one. It's a smart way to tackle problems with overlapping subproblems by reducing the number of recursive calls.

### Dynamic Programming through Going Bottom-Up

The second technique for dynamic programming is known as "going bottom-up." Unlike memoization, which optimizes recursion, going bottom-up is about solving the problem without recursion at all.

1. **What It Means:** Going bottom-up means taking a problem that can be solved recursively and finding a solution using iteration (loops) instead.
2. **Why Use It:** Since recursion can lead to duplicate calls for overlapping subproblems, switching to iteration ensures that each subproblem is solved only once.
3. **Less Fancy:** This technique might seem less elegant or intuitive compared to recursion, especially for problems where a recursive approach seems natural, like generating Fibonacci numbers.
4. **Example - Fibonacci Sequence:** Instead of computing Fibonacci numbers through recursive calls, one can use a simple loop, calculating the numbers in order from the smallest to the largest (from bottom to up). Here's how you might do it:

```

def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    fib_values = [0, 1]
    for i in range(2, n + 1):

```

```
        fib_values.append(fib_values[i - 1] + fib_values[i - 2])
    return fib_values[n]
```

This code sets up a list to store the Fibonacci numbers and then iteratively builds up the sequence from the bottom (starting with the first two numbers), working upwards to the desired number.

5. **Other Problems:** For problems that are naturally suited to recursion, like the Staircase Problem, an iterative approach might seem more complicated and less intuitive.
6. **Part of Dynamic Programming:** While it may not seem as sophisticated as memoization, going bottom-up is still considered a dynamic programming technique because it optimizes the solving of overlapping subproblems, just like memoization does.
7. **Comparison with Memoization:** Both memoization and going bottom-up achieve the same goal of reducing duplicate calculations, but they do so in different ways. Memoization optimizes a recursive approach, while going bottom-up replaces recursion with iteration.

## Bottom-Up Fibonacci

The bottom-up approach to computing Fibonacci numbers uses simple iteration instead of recursion.

```
def fib(n):
    if n == 0:
        return 0

    # a and b start with the first two numbers in the
    # series, respectively:
    a = 0
    b = 1

    # Loop from 1 until n:
    for i in range(1, n):

        # a and b each move up to the next numbers in the series.
        # Namely, b becomes the sum of b + a, and a becomes what b used to be.
        # We utilize a temporary variable to make these changes:
        temp = a
        a = b
        b = temp + a

    return b
```

1. **Initializing:** If the input number **n** is 0, the function returns 0 immediately. Otherwise, it starts with the first two numbers of the Fibonacci sequence, 0 and 1, represented by variables **a** and **b**.
2. **Iterating:** The code then loops from 1 to **n-1**, calculating each subsequent Fibonacci number.
3. **Calculating Next Number:** Inside the loop, the code uses a temporary variable **temp** to help update **a** and **b**:
  - **temp** takes the value of **a**.
  - **a** takes the value of **b**.
  - **b** becomes the sum of **temp** (the old value of **a**) and **a** (the old value of **b**).

These steps effectively move **a** and **b** up to the next numbers in the Fibonacci sequence, with **b** always holding the most recent value.

4. **Returning Result:** After the loop finishes running, **b** holds the value of the **n**-th Fibonacci number, and the function returns **b**.
5. **Efficiency:** Since the code only loops from 1 to **n**, performing a fixed amount of work on each iteration, the overall time complexity is  $O(N)$ , just like the memoization approach.

The bottom-up technique for Fibonacci is an example of dynamic programming where simple iteration is used to build up the sequence from the start, avoiding the overlap and redundancy of recursive calls. It's a clear and efficient way to calculate the desired Fibonacci number.

### Memoization vs. Bottom-Up

Memoization and the bottom-up approach are the two primary techniques of dynamic programming, each with their own benefits and drawbacks.

#### Memoization:

- **Intuitive:** Often used when recursion provides an elegant solution to a problem.
- **Reduces Redundancy:** Eliminates repeated calculations in recursive calls by storing previously computed results in a hash table.
- **Memory Overhead:** Requires additional memory for the call stack and the hash table to store previously computed values.

#### Bottom-Up:

- **Efficient:** Typically more memory-efficient as it doesn't involve the call stack, and doesn't need to store previous results in a hash table.
- **May Be Less Intuitive:** In some problems, it can be harder to formulate the problem in an iterative manner compared to recursion.
- **Preferred when Suitable:** Often the better choice if an iterative solution is equally intuitive or if memory efficiency is a concern.

#### Decision Making:

- **Choose Memoization:** If the recursive solution is more intuitive and elegant for the problem at hand, and if the overhead of additional memory isn't a concern.
- **Choose Bottom-Up:** If an iterative solution is just as clear, or if you want to minimize memory usage.

### Wrapping Up

Now that you're able to write efficient recursive code, you've also unlocked a superpower. You're about to encounter some really efficient—yet advanced—algorithms, and many of them rely on the principles of recursion.



## Exercises

The following exercises provide you with the opportunity to practice with dynamic programming.

1. The following function accepts an array of numbers and returns the sum, as long as a particular number doesn't bring the sum above 100. If adding a particular number will make the sum higher than 100, that number is ignored. However, this function makes unnecessary recursive calls. Fix the code to eliminate the unnecessary recursion:

```
def add_until_100(array)
  return 0 if array.length == 0
  if array[0] + add_until_100(array[1, array.length - 1]) > 100
    return add_until_100(array[1, array.length - 1])
  else
    return array[0] + add_until_100(array[1, array.length - 1])
  end
end
```

We can eliminate the redundant recursive call by storing the result of the recursive function in a variable and then using that variable in the subsequent conditions.

```
def add_until_100(array):
  if array.length == 0:
    return 0
  next_sum = add_until_100(array[1:])
  if array[0] + next_sum > 100:
    return next_sum
  else:
    return array[0] + next_sum
```

2. The following function uses recursion to calculate the Nth number from a mathematical sequence known as the "Golomb sequence." It's terribly inefficient, though! Use memoization to optimize it. (You don't have to actually understand how the Golomb sequence works to do this exercise.)

```
def golomb(n)
  return 1 if n == 1
  return 1 + golomb(n - golomb(golomb(n - 1)));
end
```

By adding a memoization structure to the recursive function, we can avoid redundant calculations.

```
def golomb(n, memo={1: 1}):
  if n not in memo:
```

```
        memo[n] = 1 + golomb(n - golomb(golomb(n - 1, memo), memo),
memo)
return memo[n]
```

3. Here is a solution to the “Unique Paths” problem from an exercise in the previous chapter. Use memoization to improve its efficiency:

```
def unique_paths(rows, columns)
  return 1 if rows == 1 || columns == 1
  return unique_paths(rows - 1, columns) + unique_paths(rows, columns - 1)
end
```

By using a memoization technique, we can store previously computed unique paths to reduce redundant calculations.

```
def unique_paths(rows, columns, memo=None):
  if memo is None:
    memo = {}
  if (rows, columns) in memo:
    return memo[(rows, columns)]
  if rows == 1 or columns == 1:
    return 1
  memo[(rows, columns)] = unique_paths(rows - 1, columns, memo) +
    unique_paths(rows, columns - 1, memo)
  return memo[(rows, columns)]
```

## CHAPTER 13: Recursive Algorithms for Speed

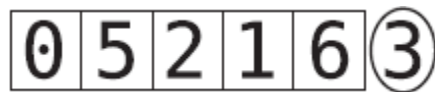
Recursion is a method that allows us to make code run much faster and is key to unlocking new algorithms like file system traversal or making anagrams. While previous chapters covered sorting algorithms like Bubble Sort, Selection Sort, and Insertion Sort, these aren't used in real life for sorting arrays. Instead, most computer languages use built-in sorting functions, often relying on Quicksort.

We'll look into Quicksort, even though it's already in many languages, because understanding how it works will teach us to use recursion to speed up various real-world algorithms. Quicksort is incredibly fast, especially in average scenarios, and although it can perform similarly to other sorts in the worst cases, it is typically much faster. The concept of partitioning is a significant part of Quicksort, and we will explore that first.

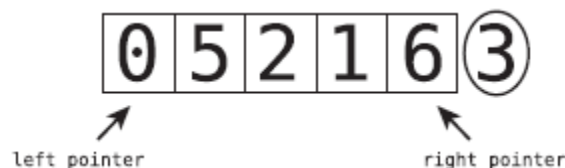
### Partitioning

**Partitioning** is a method used to organize an array around a chosen value called the pivot. Here's how it works:

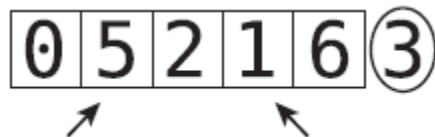
1. **Choosing the Pivot:** Select a value from the array as the pivot. Often, the rightmost value is chosen. In our example, the pivot is 3.



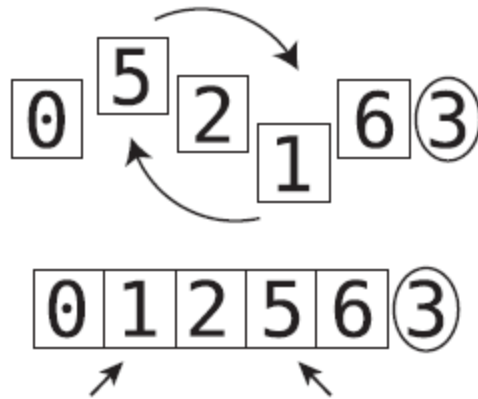
2. **Assigning Pointers:** Assign two "pointers" – one at the left-most value of the array, and the other at the rightmost value, excluding the pivot.



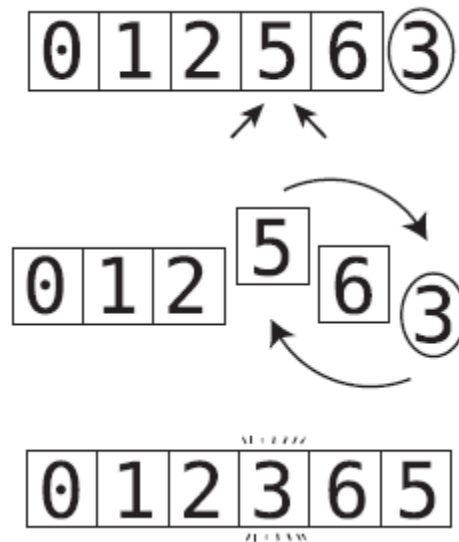
3. **Moving the Left Pointer:** Move the left pointer to the right until it reaches a value greater than or equal to the pivot, then stop.
4. **Moving the Right Pointer:** Move the right pointer to the left until it reaches a value less than or equal to the pivot, then stop. If it reaches the beginning of the array, stop.



5. **Swapping Values:** If the left pointer hasn't reached or gone beyond the right pointer, swap the values that they're pointing to, and repeat steps 3 and 4.



6. **Swapping the Pivot:** If the left pointer has reached or gone beyond the right pointer, swap the pivot with the value the left pointer is currently pointing to.



After completing these steps, the pivot is in its correct place, with all values to its left being less than the pivot and all values to its right being greater.

### Code Implementation: Partitioning

Here's a code implementation of partitioning for a **SortableArray** class in Ruby:

1. **Initialization:** The **SortableArray** class is initialized with an array.
2. **Partition Method:** The **partition!** method is defined to take two pointers, left and right, as parameters.
3. **Selecting the Pivot:** The pivot is chosen as the right-most element. The right pointer is then moved one step to the left of the pivot.
4. **Main Loop:**
  - **Left Pointer Movement:** Move the left pointer to the right as long as it points to a value that is less than the pivot.
  - **Right Pointer Movement:** Move the right pointer to the left as long as it points to a value greater than the pivot.

- **Swapping Values:** If the left pointer is still to the left of the right pointer, swap the values they point to.
- **Exiting Condition:** If the left pointer has reached or gone beyond the right pointer, exit the loop.

5. **Swapping Pivot:** Swap the value at the left pointer with the pivot.

6. **Return Value:** The method returns the `left_pointer`, which will be used later in the Quicksort algorithm.

The code does exactly what was explained earlier: it chooses a pivot and rearranges the values in the array so that values less than the pivot are on its left, and values greater than the pivot are on its right. Here's a quick breakdown of the key elements:

- **Initialization and Pivot Selection:**

```
def initialize(array)
  @array = array
end

def partition!(left_pointer, right_pointer)
  pivot_index = right_pointer
  pivot = @array[pivot_index]
  right_pointer -= 1
```

- **Main Loop with Pointers Movement:**

```
while true
  while @array[left_pointer] < pivot do left_pointer += 1 end
  while @array[right_pointer] > pivot do right_pointer -= 1 end
  if left_pointer >= right_pointer; break else
    @array[left_pointer], @array[right_pointer] = @array[right_pointer],
    @array[left_pointer]
    left_pointer += 1
  end
end
```

- **Final Pivot Swap:**

```
@array[left_pointer], @array[pivot_index] = @array[pivot_index],
@array[left_pointer]

return left_pointer
```

## Quicksort

The Quicksort algorithm is an efficient sorting method that uses a divide-and-conquer approach. Here's how it works:

1. **Choose a Pivot and Partition the Array:** Select a pivot element from the array and rearrange the other elements into two subarrays, one with elements less than the pivot and one with elements greater than the pivot. The pivot is now in its final sorted position.
2. **Recursively Apply to Subarrays:** Perform steps 1 and 2 on the two subarrays on either side of the pivot. Continue dividing and sorting the subarrays in this way.
3. **Base Case:** When a subarray has zero or one element, it is already sorted, and no further action is needed.

Here's an example breakdown of the process using the array [0, 5, 2, 1, 6, 3]:

1. Choose 3 as the pivot and partition to get [0, 1, 2, 3, 6, 5].
2. Focus on the subarray to the left of 3, [0, 1, 2], and repeat steps 1 and 2.
3. Focus on [0, 1], partition to get [0, 1]. Since the subarray to the left of 1 has only one element, it's a base case.
4. Return to the subarray to the right of 3, [6, 5], and partition to get [5, 6]. The subarray to the right of 5 is a base case.

The array is now sorted as [0, 1, 2, 3, 5, 6]. The Quicksort algorithm repeatedly divides the array into smaller parts, sorts them, and ultimately combines them into the sorted whole.

#### Code Implementation: Quicksort

1. **Function Definition:** A function `quicksort!` is defined to sort a part of the array from `left_index` to `right_index`.
2. **Base Case:** If the range (from `left_index` to `right_index`) contains 0 or 1 elements, the function just returns since no sorting is required.
3. **Partitioning:** A partitioning method (`partition!`) is called on the range of elements, and the index of the pivot is stored in `pivot_index`.
4. **Recursive Sorting on Left Side:** The `quicksort!` method is called recursively on the subarray to the left of the pivot (from `left_index` to `pivot_index - 1`).
5. **Recursive Sorting on Right Side:** The `quicksort!` method is called recursively on the subarray to the right of the pivot (from `pivot_index + 1` to `right_index`).

Here's a step-by-step walk-through:

1. Check if there's one or zero elements. If yes, stop.
2. Divide the array into two parts by choosing a pivot and rearranging elements.
3. Recursively sort the left part.
4. Recursively sort the right part.

In code, you can test the implementation with:

```
array = [0, 5, 2, 1, 6, 3]
```

```
sortable_array = SortableArray.new(array)
sortable_array.quickSort!(0, array.length - 1)
p sortable_array.array
```

This concise code performs the Quicksort algorithm to sort the given array, following the divide-and-conquer strategy mentioned earlier.

### The Efficiency of Quicksort

A partition in Quicksort consists of two main operations: comparisons and swaps.

1. **Comparisons:** Each element in the array is compared to the pivot. There are  $N$  comparisons where  $N$  is the number of elements in the partition.
2. **Swaps:** The number of swaps depends on the arrangement of the data. At most, there can be  $N/2$  swaps, but on average, it's about  $N/4$ .

The combined steps of comparisons and swaps give us about  $1.25N$  steps for  $N$  elements. In terms of Big O notation, this is simplified to  $O(N)$  for a single partition. Quicksort's efficiency depends on how the partitions are divided. Here are the main scenarios:

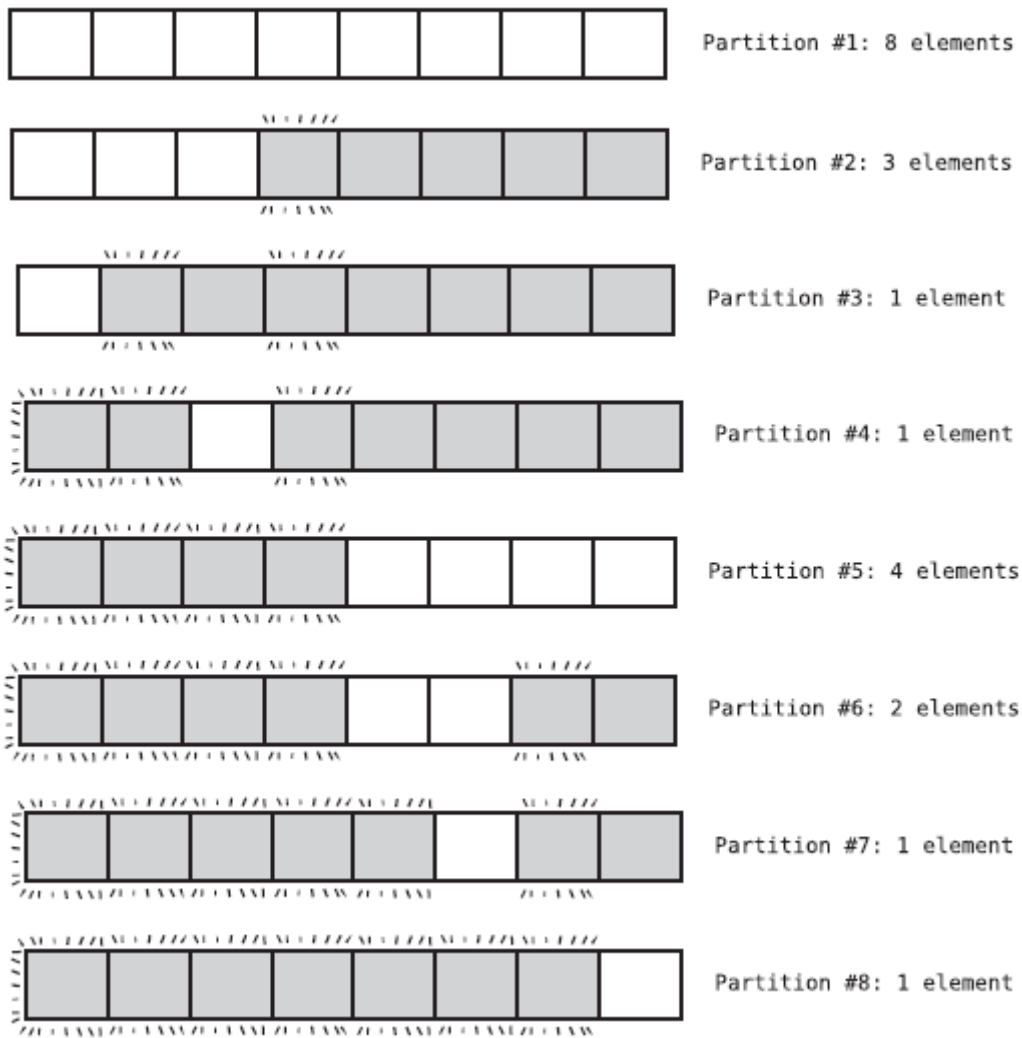
1. **Best Case (Balanced Partitions):** If the partitions are consistently divided in half, the time complexity is  $O(N \log N)$ . This occurs when the pivot is chosen in such a way that it divides the array into nearly equal halves.
2. **Average Case:** Most of the time, partitions are not exactly balanced, but the average time complexity is still generally  $O(N \log N)$ .
3. **Worst Case (Unbalanced Partitions):** If the partitions are consistently unbalanced (e.g., if the pivot is always the smallest or largest element), the time complexity can degrade to  $O(N^2)$ . This worst-case scenario is quite rare, especially if precautions are taken in choosing the pivot.

### Quicksort from a Bird's-Eye View

Imagine a typical Quicksort on an array with eight elements. You would perform several partitions on subarrays of different sizes: 8, 4, 3, 2, and four times on arrays of size 1.

#### Breaking Down the Steps:

1. Partition the original array of 8 elements.
2. Partition subarrays of sizes 4, 3, and 2.
3. Partition four more subarrays, each of size 1.



Since a partition takes about  $N$  steps for  $N$  elements in each subarray, you can add the sizes of all subarrays together to get the total number of steps:

- 8 elements
- 3 elements
- 1 element
- 1 element
- 4 elements
- 2 elements
- 1 element
- 1 element
- Total = About 21 steps

Quicksort's steps increase as the size of the array grows. Here are some approximate steps for different array sizes:



N	Quicksort Steps (approx.)
4	8
8	24
16	64
32	160

### What Does This Mean?

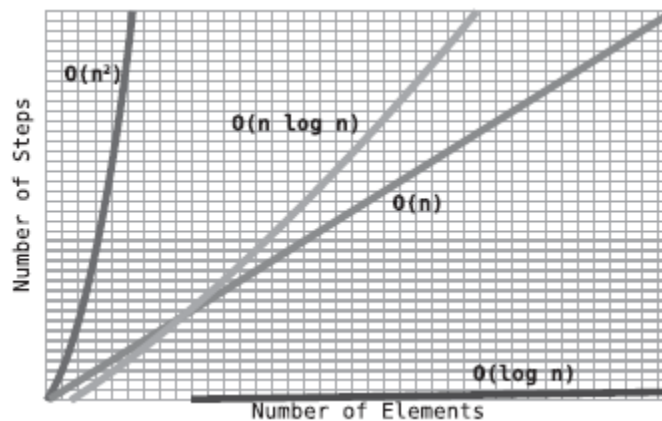
Quicksort partitions the array and its subarrays, sorting them with efficiency that often comes close to  $O(N \log N)$  in average or best-case scenarios. The exact number of steps can vary, but the table above shows a reasonable approximation of how the steps grow with the size of the array. The beauty of Quicksort lies in its recursive partitioning of the array and subarrays, which often makes it one of the most efficient sorting algorithms in practice.

### The Big O of Quicksort

Quicksort is an algorithm that is categorized under Big O Notation as  $O(N \log N)$ .

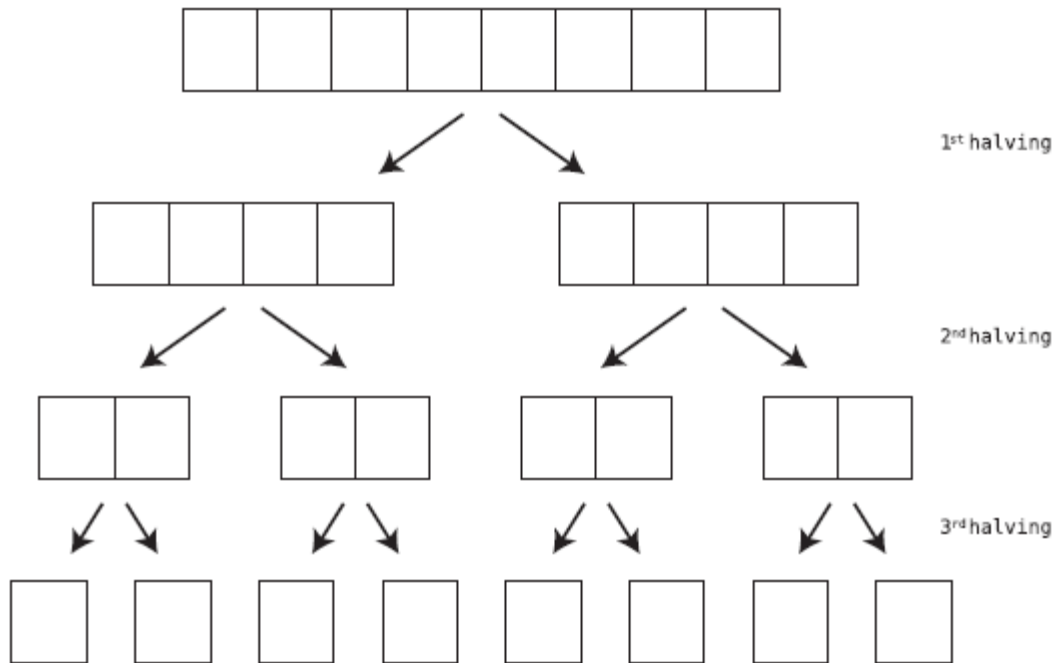
### Understanding Quicksort's Big O:

- **N**: The number of elements in the array.
- **Log N**: The number of times you can divide the array in half until you reach individual elements (subarrays of size 1).
- **N \* log N**: For each halving ( $\log N$ ), a partition is performed on all the subarrays, whose elements add up to N.



### Breakdown:

1. **Partition the Array**: Each partition breaks the array into two roughly equal subarrays.
2. **Number of Halvings**: You continue breaking down the array until each subarray has only one element. This takes  $\log N$  halvings.



3. **Steps Required:** You perform  $N$  steps for each halving, so the total number of steps is  $N$  multiplied by  $\log N$  ( $N * \log N$ ).

$N$	$\log N$	$N * \log N$	Quicksort Steps (approx.)
4	2	8	8
8	3	24	24
16	4	64	64
32	5	160	160

This is an approximation, and in a real-world scenario, things may not be as clean as this. The pivot (used for partitioning) doesn't always break the array into two even halves, and there may be an extra  $O(N)$  partition at the beginning.

Quicksort is an  $O(N \log N)$  algorithm, a category that places it among some of the most efficient sorting algorithms, especially when the pivot ends up roughly in the middle of the array in the average case. This notation captures the essence of the algorithm: breaking the array down into halves and sorting them, in a total of  $N * \log N$  steps.

### Quicksort in the Worst-Case Scenario

Quicksort's performance varies depending on the scenario. Here's how Quicksort functions in different cases, especially focusing on the worst-case scenario:

#### Best-Case Scenario:

- **Pivot Placement:** Pivot ends up in the middle of the subarray after each partition.
- **Occurrence:** Generally occurs when values are well-mixed.
- **Efficiency:**  $O(N \log N)$ .

#### Worst-Case Scenario:

- **Pivot Placement:** Pivot ends up on one side of the subarray (e.g., in ascending or descending order).
- **Result:** Increased number of comparisons.
- **Efficiency Calculation:** Partitions of  $N + (N - 1) + (N - 2) + \dots + 1 = N^2 / 2$  steps.
- **Efficiency:**  $O(N^2)$ .

### Quicksort vs. Insertion Sort

	Best Case	Average Case	Worst Case
Insertion Sort	$O(N)$	$O(N^2)$	$O(N^2)$
Quicksort	$O(N \log N)$	$O(N \log N)$	$O(N^2)$

### Key Takeaway:

- **Identical Worst Cases:** Both Insertion Sort and Quicksort have the same worst-case efficiency.
- **Average Scenario Superiority:** Quicksort is generally preferred over Insertion Sort because it is much faster in average scenarios ( $O(N \log N)$  compared to  $O(N^2)$ ).
- **Common Usage:** Quicksort is often used in programming languages' built-in sorting functions due to its average-case efficiency.

### Quickselect

Quickselect is a clever algorithm designed to find a specific order statistic, like the tenth-lowest or fifth-highest value, in an unsorted array. Here's how it works:

1. **Identify What You Want to Find:** Determine the specific value you're looking for, such as the second-lowest value within the array.
2. **Partition the Array:** Just like in Quicksort, partition the entire array. The pivot value will end up in its correct spot in the sorted array.
3. **Assess the Pivot's Position:** Look at where the pivot ended up. If it's in the fifth cell, for example, you know it's the fifth-lowest value.
4. **Focus on the Relevant Subarray:** If you're looking for the second-lowest value, you know it's somewhere to the left of the fifth-lowest value. You can ignore everything to the right of the pivot, just like in binary search.
5. **Repeat the Partitioning:** Continue to partition the relevant subarray, always focusing on the side where the desired value must be.
6. **Find the Correct Value:** After enough partitions, the desired value will end up in its correct spot, and you can retrieve it.

### Key Points:

- **Efficiency:** Unlike sorting the entire array (which would take  $O(N \log N)$ ), Quickselect can find the value more efficiently by repeatedly dividing the array and focusing only on relevant parts.

- **Similarity to Quicksort and Binary Search:** Quickselect combines concepts from both these algorithms, using partitioning and repeatedly narrowing down the search space.
- **No Need to Sort:** One of Quickselect's advantages is that it can find the correct value without having to sort the entire array, making it a more efficient choice for this specific task.

In a scenario where you're looking for a specific rank order value (like a percentile or median), Quickselect provides a more streamlined solution compared to sorting the entire array. It's a targeted search that leverages partitioning to find exactly what you're looking for, without unnecessary work.

### The Efficiency of Quickselect

The efficiency of the Quickselect algorithm is  $O(N)$  for average scenarios. Here's a more straightforward explanation to understand why:

1. **Partitioning:** Quickselect involves partitioning an array into smaller subarrays, much like Quicksort. Each partitioning takes about  $N$  steps, where  $N$  is the size of the subarray being partitioned.
2. **Total Steps:** For an array of eight elements, the total steps taken for partitioning are  $8 + 4 + 2 = 14$ . Similarly, for 64 elements, it's  $64 + 32 + 16 + 8 + 4 + 2 = 126$ , and so on.
3. **Roughly  $2N$  Steps:** Essentially, for an array with  $N$  elements, the total steps required are about  $2N$ . This can be expressed as  $N + (N/2) + (N/4) + (N/8) + \dots + 2$ , which sums to approximately  $2N$  steps.
4. **Ignoring Constants in Big O:** Since Big O notation focuses on how the runtime grows with input size, rather than the precise count of operations, the constant factor of 2 is ignored.

By focusing on the growth rate and disregarding the constant, we arrive at an average-case efficiency of  $O(N)$  for Quickselect. This makes it a linear-time algorithm on average, meaning that its running time grows linearly with the size of the input array.

### Code Implementation: Quickselect

Following is an implementation of a `quickselect!` method that can be dropped into the `SortableArray` class described earlier. You'll note that it's very similar to the `quicksort!` method:

```

def quickselect!(kth_lowest_value, left_index, right_index)
  # If we reach a base case - that is, that the subarray has one cell,
  # we know we've found the value we're looking for:
  if right_index - left_index <= 0
    return @array[left_index]
  end

  # Partition the array and grab the index of the pivot:
  pivot_index = partition!(left_index, right_index)

  # If what we're looking for is to the left of the pivot:
  if kth_lowest_value < pivot_index
    # Recursively perform quickselect on the subarray to
    # the left of the pivot:
    quickselect!(kth_lowest_value, left_index, pivot_index - 1)
  # If what we're looking for is to the right of the pivot:
  elsif kth_lowest_value > pivot_index
    # Recursively perform quickselect on the subarray to
    # the right of the pivot:
    quickselect!(kth_lowest_value, pivot_index + 1, right_index)
  else # if kth_lowest_value == pivot_index
    # if after the partition, the pivot position is in the same spot
    # as the kth lowest value, we've found the value we're looking for
    return @array[pivot_index]
  end
end
end

```

Here's a step-by-step example of how to use this method:

- **Initialization:** An array is created and then wrapped in a **SortableArray** class.
- **Calling the Method:** The method **quickselect!** is called with three arguments: the index of the value you're looking for (e.g., **1** for the second-to-lowest value), the starting index of the array (**0**), and the ending index of the array (**array.length - 1**).
- **Result:** The method returns the value at the specified index (in this case, the second-to-lowest value) in the unsorted array.

### Sorting as a Key to Other Algorithms

Sorting is a crucial part of many algorithms, and some of the fastest sorting methods, such as Quicksort and Mergesort, have an efficiency of  $O(N \log N)$ . This efficiency is important because it impacts other algorithms that use sorting. A good example of this is finding duplicates in an array. Without sorting, you might use nested loops, which would be  $O(N^2)$  efficiency, or a different method with lower time complexity but more memory usage. With sorting, you can improve the efficiency to  $O(N \log N)$ :

1. **Sort the Array:** First, you sort the array so that identical numbers are next to each other. For example, an array [5, 9, 3, 2, 4, 5, 6] would become [2, 3, 4, 5, 5, 6, 9].
2. **Look for Duplicates:** Then, you run a loop through the sorted array, checking if any adjacent numbers are the same. If any are, you've found a duplicate, and the process ends.
3. **Efficiency Analysis:** Sorting typically takes  $O(N \log N)$  time, and the loop through the array takes  $O(N)$  time. In Big O notation, you only keep the highest order, so the overall efficiency is  $O(N \log N)$ .

Here's a JavaScript example that illustrates this process:

```
function hasDuplicateValue(array) {
  // Presort the array:
  // (In JavaScript, the following usage of the sort function
  // is required to ensure that the numbers are in numerical order,
  // instead of "alphabetical" order.)
  array.sort((a, b) => (a < b) ? -1 : 1);

  // Iterate through the values in the array up until the last one:
  for(let i = 0; i < array.length - 1; i++) {

    // If the value is identical to the next value
    // in the array, we found a duplicate:
    if(array[i] == array[i + 1]) {
      return true;
    }
  }

  // If we get to the end of the array without having
  // returned true, it means there are no duplicates:
  return false;
}
```

This approach of using sorting as a part of a solution can make algorithms more efficient. Whenever sorting is involved in an algorithm, the minimum efficiency will generally be  $O(N \log N)$ , which can be a significant improvement over other methods.

## Wrapping Up

The Quicksort and Quickselect algorithms are recursive algorithms that present beautiful and efficient solutions to thorny problems. They're great examples of how a non-obvious but well-thought-out algorithm can boost performance. Now that we've seen some more advanced algorithms, we're now going to head in a new direction and explore a trove of additional data structures. Some of these data structures have operations that involve recursion, so we'll be fully prepared to tackle those now. Besides for being really interesting, we'll see that each data structure has a special power that can bring significant advantages to a variety of applications.

## Exercises

1. Given an array of positive numbers, write a function that returns the greatest product of any three numbers. The approach of using three nested loops would clock in at  $O(N^3)$ , which is very slow. Use sorting to implement the function in a way that it computes at  $O(N \log N)$  speed. (There are even faster implementations, but we're focusing on using sorting as a technique to make code faster.)

You can sort the array and then multiply the top three numbers together or multiply the lowest two numbers (in case of negative numbers) with the greatest one. The greatest product of any three numbers can be obtained as:

```
function greatestProduct(arr) {
  arr.sort((a, b) => b - a);
```

```

    return Math.max(arr[0] * arr[1] * arr[2], arr[0] * arr[arr.length - 1] *
        arr[arr.length - 2]);
}

```

2. The following function finds the “missing number” from an array of integers. That is, the array is expected to have all integers from 0 up to the array’s length, but one is missing. As examples, the array, [5, 2, 4, 1, 0] is missing the number 3, and the array, [9, 3, 2, 5, 6, 7, 1, 0, 4] is missing the number 8. Here’s an implementation that is  $O(N^2)$  (the includes method alone is already  $O(N)$ , since the computer needs to search the entire array to find n):

```

function findMissingNumber(array) {
  for(let i = 0; i < array.length; i++) {
    if(!array.includes(i)) {
      return i;
    }
  }
  // If all numbers are present:
  return null;
}

```

Use sorting to write a new implementation of this function that only takes  $O(N \log N)$ . (There are even faster implementations, but we’re focusing on using sorting as a technique to make code faster.)

By sorting the array first and then finding the missing number, the function's time complexity can be reduced to  $O(N \log N)$ :

```

function findMissingNumber(array) {
  array.sort((a, b) => a - b);
  for (let i = 0; i < array.length; i++) {
    if (array[i] !== i) {
      return i;
    }
  }
  return null; // if all numbers are present
}

```

3. Write three different implementations of a function that finds the greatest number within an array. Write one function that is  $O(N^2)$ , one that is  $O(N \log N)$ , and one that is  $O(N)$ .

**$O(N^2)$ :**

```

function greatestNumberN2(arr) {
  for (let i = 0; i < arr.length; i++) {
    let isGreatest = true;

```

```

    for (let j = 0; j < arr.length; j++) {
      if (arr[j] > arr[i]) {
        isGreatest = false;
        break;
      }
    }
    if (isGreatest) return arr[i];
  }
}

```

**O(N log N):**

```

function greatestNumberNLogN(arr) {
  arr.sort((a, b) => b - a);
  return arr[0];
}

```

**O(N):**

```

function greatestNumberN(arr) {
  let max = arr[0];
  for (let i = 1; i < arr.length; i++) {
    if (arr[i] > max) {
      max = arr[i];
    }
  }
  return max;
}

```

The first function uses two nested loops to compare each number to every other number in the array. The second function sorts the array and returns the first element, and the third function iterates through the array once to find the maximum value.

## CHAPTER 14: Node-Based Data Structures

In the upcoming chapters, we'll look into various data structures built on one idea: the node. **Nodes** are data pieces scattered in the computer's memory, and using them leads to enhanced performance in organizing and accessing data. This chapter focuses on the linked list, the most basic structure using nodes, and lays the



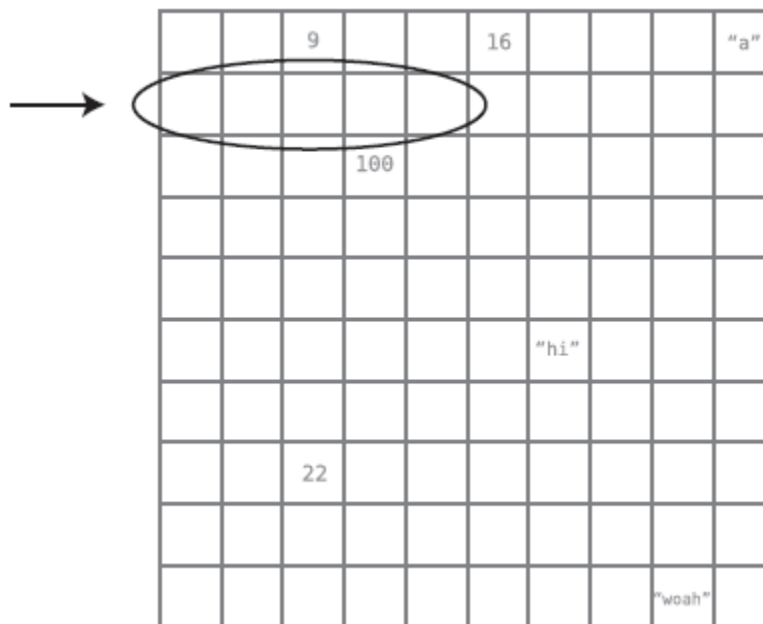
groundwork for what's coming next. Linked lists are similar to arrays but have unique efficiency trade-offs that can improve performance in specific cases.

## Linked Lists

Linked lists and arrays are both data structures that represent lists of items. Though they appear similar, they function quite differently inside a computer.

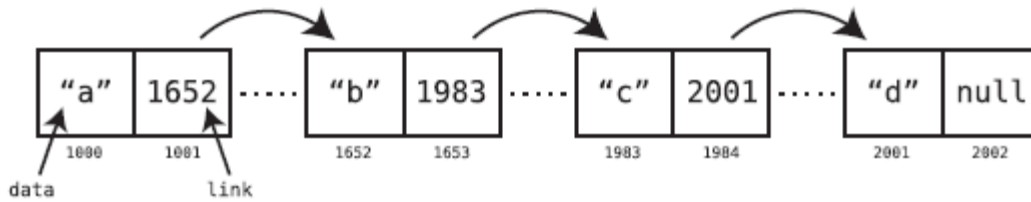
### Arrays:

- Arrays work by finding a continuous group of empty cells in the computer's memory to store data.
- If you ask the computer to find a value at a specific index, it can do so in one step, as the computer knows the starting memory address of the array. For instance, if the array starts at memory address 1000, to find index 4, the computer will directly access memory address 1004.
- Arrays need contiguous memory space, which can become an issue as the array size grows.



### Linked Lists:

- Unlike arrays, linked lists do not require contiguous memory; their data can be scattered across different cells in the computer's memory.
- These scattered pieces of connected data are called nodes, and each node represents one item in the list.
- The question arises, if nodes aren't next to each other, how does the computer recognize them as part of the same list? The answer lies in the design of linked lists. Each node includes extra information, specifically the memory address of the next node. This is known as a link.
- A linked list uses two memory cells for each piece of data; the first holds the actual data, and the second acts as a link to the next node. When the list ends, the final node's link contains null.



- Since the computer knows where the linked list begins, it can follow each link to assemble the entire list, similar to stringing beads.
- The flexibility in memory usage gives linked lists an advantage over arrays, although this is managed by your programming language, so you may not often think about it.

While arrays require one uninterrupted space in memory, linked lists can use bits and pieces scattered all over. This makes linked lists more adaptable but also introduces differences in how they're handled and accessed.

### Implementing a Linked List

Creating a Linked List in Ruby Implementing a linked list can be done easily, even if some languages, like Java, have them pre-built. Here's how you can build a linked list using Ruby, using two classes, **Node** and **LinkedList**.

#### 1. Create the Node Class:

```
class Node
  attr_accessor :data, :next_node
  def initialize(data)
    @data = data
  end
end
```

- The **Node** class contains two attributes: **data** (the primary value, like "a") and **next\_node** (the link to the next node).
- You can create nodes and link them together like this:

```
node_1 = Node.new("once")
node_2 = Node.new("upon")
node_3 = Node.new("a")
node_4 = Node.new("time")

node_1.next_node = node_2
node_2.next_node = node_3
node_3.next_node = node_4
```

- Here, **next\_node** doesn't refer to an actual memory address but to another **Node** instance. Still, it creates a list of strings "once", "upon", "a", "time".

#### 2. Create the LinkedList Class:

```

class LinkedList
  attr_accessor :first_node
  def initialize(first_node)
    @first_node = first_node
  end
end

```

- This class helps to keep track of the first node of the list.
- To reference the linked list, you write:

```
list = LinkedList.new(node_1)
```

- The **list** variable provides access to the linked list's first node.

### 3. Important Characteristics:

- A linked list only allows immediate access to its first node, leading to differences in handling compared to arrays.
- While similar in appearance to arrays, linked lists vary significantly in how they perform certain operations like reading, searching, insertion, and deletion.

#### Reading

Reading an element from an array is fast and takes constant time,  $O(1)$ . However, reading from a linked list is a bit more complex and not as quick. Here's why:

- In a linked list, the computer only knows the memory address of the first node. The other nodes could be anywhere in memory.
- To read, say, the third item, the computer must first access the first node, follow its link to the second node, and then follow the second node's link to the third node.
- Essentially, to read a specific node, you must start at the beginning and follow the chain of links until you reach the desired node.
- In the worst case, if you want to read the last node in a list of  $N$  nodes, it would take  $N$  steps, resulting in a reading efficiency of  $O(N)$ .

This makes linked lists less efficient in reading compared to arrays, which can read any element instantly. While this may seem like a disadvantage, it's important to remember that linked lists have other characteristics that can make them advantageous in different scenarios.

#### Code Implementation: Linked List Reading

```

def read(index)
  # We begin at the first node of the list:
  current_node = first_node
  current_index = 0

```

```

while current_index < index do
  # We keep following the links of each node until we get to the
  # index we're looking for:
  current_node = current_node.next_node
  current_index += 1

  # If we're past the end of the list, that means the
  # value cannot be found in the list, so return nil:
  return nil unless current_node
end

return current_node.data
end

```

- The **read** method starts with the first node and iteratively follows the links to the desired index.
- The **current\_node** variable keeps track of the current node being accessed.
- The **current\_index** variable keeps track of the index of the current node being accessed, starting at 0.
- A while loop continues as long as **current\_index** is less than the specified index.
  - Inside the loop, the method sets **current\_node** to the next node in the list and increments **current\_index**.
  - If **current\_node** becomes nil, meaning that the desired index is out of bounds, the method returns nil.
- Once the loop ends, the method returns the data of the current node, which is now the node at the desired index.

For example, calling **list.read(3)** would access the fourth node in the list (since indexes start at 0), follow its links to get to the correct node, and return its data. This code is specific to reading from a linked list and reflects the fact that you must traverse the list from the beginning to reach a particular index.

## Searching

Searching in a linked list involves looking for a specific value and returning its index, and it has a speed of  $O(N)$ . Similar to reading, the search process in a linked list starts with the first node and follows the links to subsequent nodes, inspecting each value one by one. Since you must traverse each node to find the desired value, the search operation takes linear time, making it the same as a linear search on an array.

### Code Implementation: Linked List Search

The search operation for a linked list can be implemented in Ruby using a method called **index\_of**, which accepts the value we're searching for.

1. **Starting Point:** We begin at the first node of the list and initialize a variable to keep track of the current index.
2. **Searching:** We then enter a loop that continues until the end of the list. Inside the loop, we compare the current node's data to the value we're searching for:
  - **Value Found:** If they match, we return the current index, as we've found the value.

- **Move to Next Node:** If they don't match, we move to the next node by following the link and increment the current index by 1.

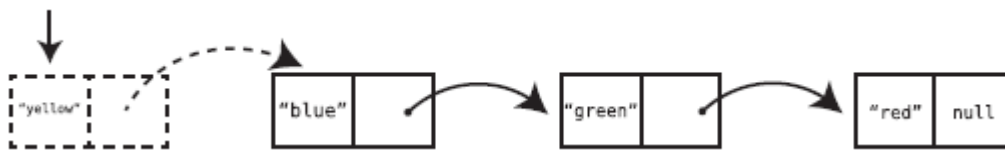
3. **Not Found:** If we reach the end of the list without finding the value, we return **nil**.

## Insertion

Inserting an element in a linked list offers a performance advantage in certain situations compared to arrays.

### 1. Inserting at the Beginning ( $O(1)$ ):

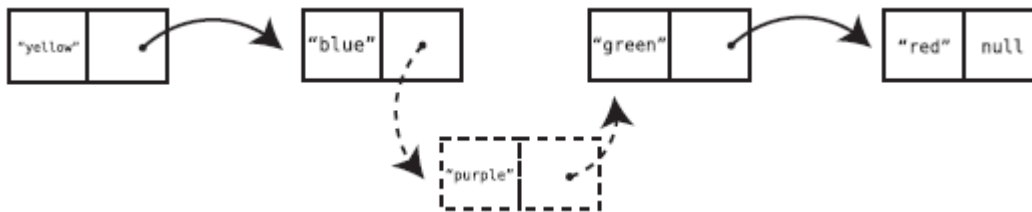
- Linked lists can insert an element at the beginning in just one step.
- For example, to add "yellow" at the beginning, create a new node with "yellow," link it to the current first node (e.g., "blue"), and update the first\_node attribute of the linked list.



- Arrays, on the other hand, would require shifting all elements, resulting in  $O(N)$  efficiency for this operation.

### 2. Inserting at Other Positions ( $O(N)$ ):

- Though the actual insertion in a linked list takes only one step, getting to the desired index takes up to  $N$  steps.
- For example, to insert "purple" between "blue" and "green," you must first reach the "blue" node ( $O(N)$ ) and then change its link to point to the new "purple" node ( $O(1)$ ).



- The total efficiency for this operation in a linked list is  $O(N)$ , the same as arrays.

### 3. Comparison with Arrays:

Scenario	Array	Linked List
Insert at beginning	Worst case	Best case
Insert at middle	Average case	Average case
Insert at end	Best case	Worst case

Linked lists are excellent for insertions at the beginning of the list, which can be done in constant time. This advantage, combined with the flexibility of not needing to shift data like in an array, highlights a specific area where linked lists can outperform arrays.

## Code Implementation: Linked List Insertion

The `insert_at_index` method is added to the `LinkedList` class to insert a value at a specific index.

```
def insert_at_index(index, value)
  # We create the new node with the provided value:
  new_node = Node.new(value)

  # If we are inserting at the beginning of the list:
  if index == 0
    # Have our new node link to what was the first node:
    new_node.next_node = first_node
    # Establish that our new node is now the list's first node:
    self.first_node = new_node
    return
  end

  # If we are inserting anywhere other than the beginning:

  current_node = first_node
  current_index = 0

  # First, we access the node immediately before where the
  # new node will go:
  while current_index < (index - 1) do
    current_node = current_node.next_node
    current_index += 1
  end

  # Have the new node link to the next node:
  new_node.next_node = current_node.next_node

  # Modify the link of the previous node to point to
  # our new node:
  current_node.next_node = new_node
end
```

1. **Create a New Node:** A new node with the given value is created.
2. **Inserting at the Beginning:**
  - If the index is 0 (beginning of the list), the new node's next node is set to the current first node.
  - The first node of the list is updated to the new node.
3. **Inserting at Other Indices:**
  - Start with the first node.
  - Use a while loop to reach the node immediately before the index where you want to insert.
  - Update the link of the new node to point to the node after the current node.
  - Change the link of the current node to point to the new node.

**Example Usage:** To insert "purple" at index 3:

```
list.insert_at_index(3, "purple")
```

## Deletion

Deletion in linked lists offers efficiency, especially at the beginning of the list. Here's how it works:

### 1. Deleting at the Beginning:

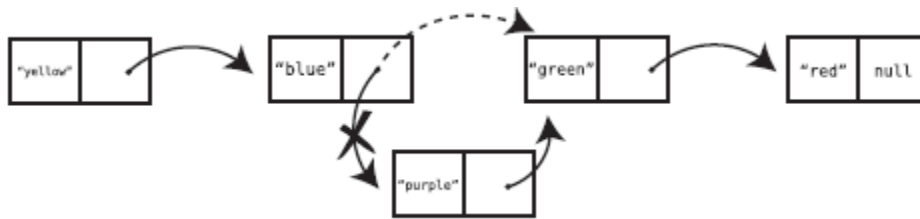
- Just change the `first_node` of the linked list to point to the second node.
- Example: To delete "once", make the linked list begin at "upon".
- This is an  $O(1)$  operation, taking just one step.

### 2. Deleting at the End:

- The deletion takes one step as you nullify the link of the second-to-last node.
- However, getting to that node takes  $N$  steps, making it an  $O(N)$  operation.

### 3. Deleting in the Middle:

- More complex as you need to access the node before the one you're deleting, then change its link to point to the node after the one you're deleting.
- Example: To delete "purple", change the "blue" node's link from "purple" to "green".



Here's a comparison between arrays and linked lists for deletion:

Situation	Array	Linked List
Delete at beginning	Worst case	Best case
Delete at middle	Average case	Average case
Delete at end	Best case	Worst case

The process of deleting a node in a linked list involves merely unlinking it from the list. The node itself may still exist in memory, depending on how the programming language handles garbage collection. Deleting from a linked list can be efficient, especially at the beginning, and is performed by updating links between nodes.

## Code Implementation: [Linked List Deletion](#)

Here's how the deletion operation works in a linked list, specifically using the `delete_at_index` method:

```

def delete_at_index(index)
  # If we are deleting the first node:
  if index == 0
    # Simply set the first node to be what is currently the second node:
    self.first_node = first_node.next_node
    return
  end

  current_node = first_node
  current_index = 0

  # First, we find the node immediately before the one we
  # want to delete and call it current_node:
  while current_index < (index - 1) do
    current_node = current_node.next_node
    current_index += 1
  end

  # We find the node that comes after the one we're deleting:
  node_after_deleted_node = current_node.next_node.next_node

  # We change the link of the current_node to point to the
  # node_after_deleted_node, leaving the node we want
  # to delete out of the list:
  current_node.next_node = node_after_deleted_node
end

```

### 1. If Deleting the First Node:

- If the index is 0, meaning you want to delete the first node, simply set the first node to be the current second node.
- This is achieved with the code **self.first\_node = first\_node.next\_node**.
- This case is handled very simply, taking just one step.

### 2. If Deleting Anywhere Else:

- A loop is used to find the node immediately before the one you want to delete, called **current\_node**.
- Find the node after the one you're deleting (**node\_after\_deleted\_node**), which is two nodes after the **current\_node**.
- Modify the link of the **current\_node** to point to **node\_after\_deleted\_node**, skipping over the node you want to delete.

This method effectively removes the node you want to delete from the list by changing the links, leaving it out of the list. The implementation is somewhat similar to the insert method but is adapted for the delete operation.

## Efficiency of Linked List Operations

Efficiency is an important aspect to consider when choosing between linked lists and arrays. Here's a comparison of both data structures based on common operations:



Operation	Array	Linked list
Reading	$O(1)$	$O(N)$
Search	$O(N)$	$O(N)$
Insertion	$O(N)$ ( $O(1)$ at end)	$O(N)$ ( $O(1)$ at beginning)
Deletion	$O(N)$ ( $O(1)$ at end)	$O(N)$ ( $O(1)$ at beginning)

So, why use linked lists when they appear to be slower or equal in performance compared to arrays, especially in reading?

- **The Advantage:** The actual steps for inserting or deleting are incredibly fast ( $O(1)$ ) in a linked list.
- **The Catch:** This advantage is particularly relevant when inserting or deleting at the beginning of the list. Elsewhere, the time spent accessing the desired node diminishes this advantage.
- **The Special Cases:** In certain situations where you've already accessed the right node for another reason, linked lists can be extremely efficient. This is where the true power of linked lists can shine, making them a suitable choice depending on the specific use case.

### Linked Lists in Action

The example below of processing a list of email addresses highlights a situation where linked lists demonstrate a significant advantage over arrays. Here's a comparison between the two when dealing with this particular task:

**Scenario:** You have a list of 1,000 email addresses and need to delete 100 invalid ones.

#### 1. Using an Array:

- **Reading Steps:** 1,000 (one for each email address).
- **Deletion Steps:** Up to 100,000. Each deletion might require up to 1,000 shifts to close the gap.
- **Total Steps:** Approximately 101,000.

#### 2. Using a Linked List:

- **Reading Steps:** 1,000 (same as with an array).
- **Deletion Steps:** Just 100. Changing a node's link for deletion is a single step.
- **Total Steps:** Only 1,100.

### Why Linked Lists Shine Here:

- Linked lists don't require shifting other data when making an insertion or deletion. So, in this example, where deletion is a frequent operation, linked lists are drastically more efficient.
- With arrays, the constant shifting of elements adds significant overhead, making the operation much slower in comparison.

Linked lists are a powerful tool when you need to move through a list and perform frequent insertions or deletions.

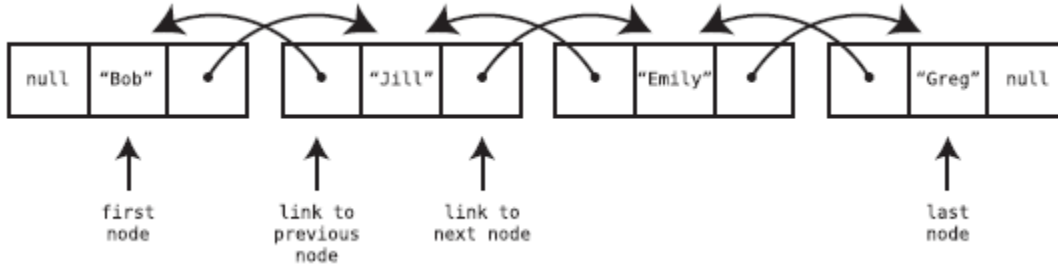
## Doubly Linked Lists

A doubly linked list is a specific type of linked list that grants additional functionality compared to a classic singly linked list. Here's a straightforward explanation of what a doubly linked list is and how it works:

### Doubly Linked List

#### 1. Structure:

- **Nodes:** Each node contains data and has two links: one pointing to the next node and the other pointing to the previous node.



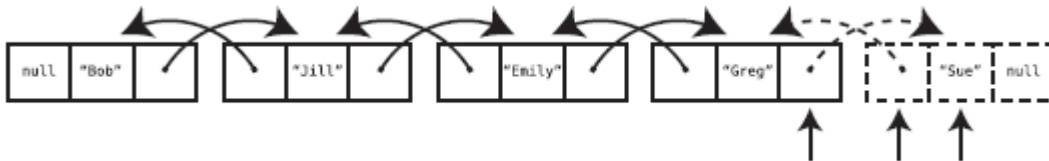
- **First and Last Nodes:** A doubly linked list keeps track of both the first and last nodes, allowing direct access to them.

#### 2. Operations:

- **Reading, Insertion, Deletion:** Both the beginning and end of the list can be accessed in  $O(1)$  time complexity. This allows for fast reading, insertion, and deletion at either end.

#### 3. Example of Insertion at the End:

- Create a new node (e.g., "Sue").
- Have its `previous_node` point to what used to be the `last_node` (e.g., "Greg").
- Change the `next_node` of the `last_node` (e.g., "Greg") to point to the new node (e.g., "Sue").
- Declare the new node (e.g., "Sue") to be the `last_node` of the linked list.



Doubly linked lists offer advantages over classic singly linked lists, such as the ability to access both the beginning and end of the list in constant time. This results in more efficient operations when dealing with both ends of the list. The inclusion of `previous_node` pointers in each node allows for more flexible navigation and manipulation within the list.

## Code Implementation: [Doubly Linked List Insertion](#)

Here's the code implementation for inserting a value at the end of a doubly linked list, along with what makes it unique:

```

def insert_at_end(value)
  new_node = Node.new(value)

  # If there are no elements yet in the linked list:
  if !first_node
    @first_node = new_node
    @last_node = new_node
  else # If the linked list already has at least one node:
    new_node.previous_node = @last_node
    @last_node.next_node = new_node
    @last_node = new_node
  end
end
end

```

### Method: insert\_at\_end(value)

1. **Create New Node:** A new node is created with the given value.
2. **Check for Existing Nodes:**
  - If there are no nodes in the linked list, both the **first\_node** and **last\_node** are set to the new node.
  - If there is at least one node in the linked list, proceed to the next steps.
3. **Linking New Node:**
  - Set the `previous_node` link of the new node to point to the current last node.
  - Change the `next_node` link of the current last node to point to the new node.
4. **Update Last Node:** The last node of the doubly linked list instance is updated to be the new node.

### Unique Features of Doubly Linked List

- **Moving Forward and Backward:** Unlike a "classic" linked list, a doubly linked list allows navigation in both directions, thanks to the `previous_node` links. You can start with either the first or last node and move through the list.

The `insert_at_end` method adds a new node to the end of a doubly linked list, handling both the case where the list is empty and where it already has nodes. The method ensures proper linking between nodes, allowing for both forward and backward navigation through the list. This bidirectional movement is a feature that sets doubly linked lists apart from their singly linked counterparts.

### Queues as Doubly Linked Lists

A queue is a data structure that follows the "First In, First Out" (FIFO) principle, where items are added at the end and removed from the beginning. Let's look at how a doubly linked list can be an excellent choice for implementing a queue:

### Key Features of Doubly Linked Lists:

- **Immediate Access:** You can quickly access both the front and end of the list.
- **O(1) Insertions:** Adding data at the front or end takes constant time,  $O(1)$ .

- **O(1) Deletions:** Deleting data from the front or end also takes constant time,  $O(1)$ .

### Why Doubly Linked Lists are Perfect for Queues:

- **Insert at the End:** Since a doubly linked list can insert data at the end in  $O(1)$  time, this aligns with a queue's need to add items at the end.
- **Delete from the Beginning:** A doubly linked list can delete data from the front in  $O(1)$  time, matching a queue's requirement to remove items from the beginning.

### Comparison with Arrays:

- Arrays can also be used to implement a queue, but they have limitations:
  - Insertions at the end are  $O(1)$ , which is good for queues.
  - However, deletions from the beginning are  $O(N)$ , making them less efficient for queues compared to doubly linked lists.

Doubly linked lists provide a robust and efficient way to implement queues. Their ability to insert and delete items at both ends in constant time aligns perfectly with the queue's FIFO structure. This makes them a preferred choice over arrays for implementing queues, where deletions at the beginning might be less efficient.

### Code Implementation: Queue Built Upon a Doubly Linked List

Here's code that demonstrates how a queue can be built using a doubly linked list:

```
class Node
  attr_accessor :data, :next_node, :previous_node
  def initialize(data)
    @data = data
  end
end
```

```

class DoublyLinkedList
  attr_accessor :first_node, :last_node

  def initialize(first_node=nil, last_node=nil)
    @first_node = first_node
    @last_node = last_node
  end

  def insert_at_end(value)
    new_node = Node.new(value)

    # If there are no elements yet in the linked list:
    if !first_node
      @first_node = new_node
      @last_node = new_node
    else # If the linked list already has at least one node:
      new_node.previous_node = @last_node
      @last_node.next_node = new_node
      @last_node = new_node
    end
  end

  def remove_from_front
    removed_node = @first_node
    @first_node = @first_node.next_node
    return removed_node
  end
end

class Queue
  attr_accessor :queue

  def initialize
    @data = DoublyLinkedList.new
  end

  def enqueue(element)
    @data.insert_at_end(element)
  end

  def dequeue
    removed_node = @data.remove_from_front
    return removed_node.data
  end

  def read
    return nil unless @data.first_node
    return @data.first_node.data
  end
end

```

### Classes Involved:

1. **Node**: Represents a single item in the list. It holds the data and has pointers to the next and previous nodes.
2. **DoublyLinkedList**: Manages the linked list operations including inserting at the end and removing from the front.
3. **Queue**: Utilizes the DoublyLinkedList class to create the queue functionality.

## Main Methods:

- **initialize**: Sets up the initial state for the classes.
- **insert\_at\_end**: Adds a new node to the end of the linked list.
- **remove\_from\_front**: Removes the first node from the linked list and returns it.
- **enqueue**: Adds an element to the end of the queue.
- **dequeue**: Removes and returns the data of the first element from the queue.
- **read**: Returns the data of the first element in the queue.

## How It Works:

- **Enqueue (Insertion)**: When you want to add an element to the queue, the **enqueue** method calls the **insert\_at\_end** method of `DoublyLinkedList`, which handles adding the new node at the end.
- **Dequeue (Deletion)**: When you want to remove an element from the queue, the **dequeue** method calls the **remove\_from\_front** method of `DoublyLinkedList`, which deletes the first node and returns its data.

## Key Benefits:

- Both insertion and deletion operations are carried out at constant time  $O(1)$ , making this a highly efficient way to implement a queue.

## Example of Using Queue:

1. Create a new queue.
2. Add elements using **enqueue**.
3. Read or remove elements using **dequeue** or **read**.

## Wrapping Up

As we've seen, the subtle differences between arrays and linked lists unlock new ways to make our code faster than ever. By looking at linked lists, you've also learned the concept of nodes. However, the linked list is only the simplest of node-based data structures. In the next chapters, you'll learn about node-based structures that are both more complex and more interesting—and will reveal new worlds about how nodes can yield tremendous power and efficiency.

## Exercises

The following exercises provide you with the opportunity to practice with linked lists.

1. Add a method to the classic `LinkedList` class that prints all the elements of the list.

We can iterate through the list and print each element. Here's a method that does this:

```
def print_elements
  current_node = @first_node
```

```

while current_node
  puts current_node.data
  current_node = current_node.next_node
end
end

```

2. Add a method to the DoublyLinkedList class that prints all the elements of the list in reverse order.

We can start from the last node and use the previous\_node link to move backwards, printing each element:

```

def print_elements_reverse
  current_node = @last_node
  while current_node
    puts current_node.data
    current_node = current_node.previous_node
  end
end

```

3. Add a method to the classic LinkedList class that returns the last element from the list. Assume you don't know how many elements are in the list.

We need to iterate through the list until we reach the last node:

```

def get_last_element
  current_node = @first_node
  current_node = current_node.next_node while current_node.next_node
  current_node.data
end

```

4. Here's a tricky one. Add a method to the classic LinkedList class that reverses the list. That is, if the original list is A -> B -> C, all of the list's links should change so that C -> B -> A.

We can change the next\_node pointers of every node to reverse the list:

```

def reverse
  previous_node = nil
  current_node = @first_node
  while current_node
    next_node = current_node.next_node

```

```
        current_node.next_node = previous_node
        previous_node = current_node
        current_node = next_node
    end

    @first_node = previous_node
end
```

5. Here's a brilliant little linked list puzzle for you. Let's say you have access to a node from somewhere in the middle of a classic linked list, but not the linked list itself. That is, you have a variable that points to an instance of Node, but you don't have access to the LinkedList instance. In this situation, if you follow this node's link, you can find all the items from this middle node until the end, but you have no way to find the nodes that precede this node in the list.

Write code that will effectively delete this node from the list. The entire remaining list should remain complete, with only this node removed.

This is a tricky one. Without access to the previous node, we can't simply unlink the current node. But we can copy the data from the next node to the current node and then delete the next node:

```
def delete_middle_node(node)
  return if node.nil? || node.next_node.nil?
  node.data = node.next_node.data
  node.next_node = node.next_node.next_node
end
```

Please note that the code above won't work if you are trying to delete the last node from the list, as there's no next node to copy data from. It's a solution specifically for a node somewhere in the middle of the list.



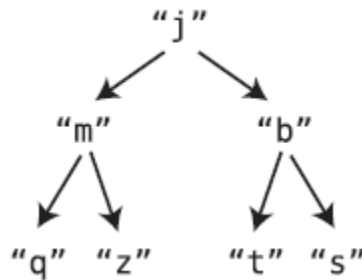
## CHAPTER 15: Speeding Up All the Things with Binary Search Trees

If you want to organize your data in a specific sequence, you might consider using a sorting method, like Quicksort. But these methods can take time, specifically  $O(N \log N)$ , so keeping the data sorted to begin with is often smarter. Ordered arrays help keep data sorted. They offer quick reading ( $O(1)$ ) and searching ( $O(\log N)$  with binary search). However, inserting or deleting items is slow, taking  $O(N)$  time, as all greater values must shift over.

If you need a really fast data structure but don't need to keep things in order, a hash table works well since it offers  $O(1)$  speed for searching, inserting, and deleting. But if you need both speed and order, neither ordered arrays nor hash tables are perfect. The binary search tree is a better solution for maintaining order and still providing relatively quick search, insertion, and deletion functions.

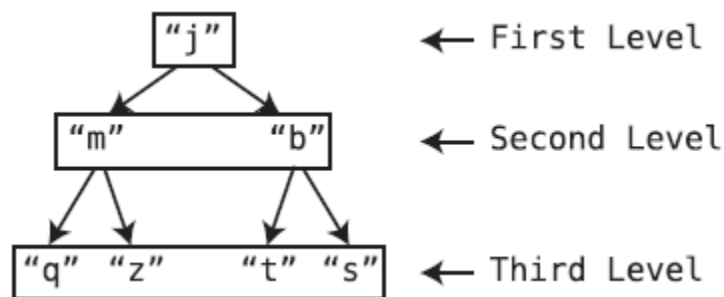
### Trees

In the last chapter, you learned about linked lists where each node connects to one other node. Trees are similar but can link to multiple nodes. Imagine a tree where each node connects to two others. We can visually represent it without showing memory addresses.



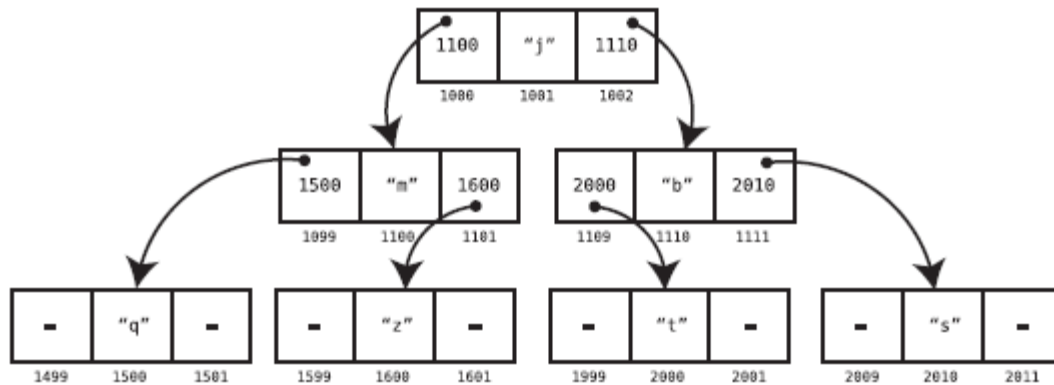
Here's some tree terminology:

- The top node is called the "root."
- A "parent" node is linked to "children" nodes, and these relationships are like those in a family tree.
- Nodes can have "descendants" (all nodes stemming from it) and "ancestors" (all nodes it stems from).
- Trees have "levels," which are the rows in the tree.



- A tree is "balanced" if each node's subtrees have the same number of nodes. If one subtree has more nodes than the other, the tree is "imbalanced."

For example, in a balanced tree, every node has two subtrees with an equal number of nodes. If the root's right subtree has more nodes than its left subtree, the tree is imbalanced.



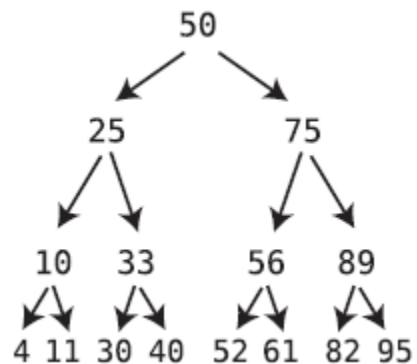
## Binary Search Trees

In a binary tree, each node can have zero, one, or two children. A binary search tree (BST) follows additional rules:

- Every node has at most one "left" (lesser value) child and one "right" (greater value) child.
- A node's "left" descendants have values less than the node, and its "right" descendants have values greater.

Here's how you can picture a binary search tree:

- If there's a node with the number 50, all left descendants will be less than 50, and all right descendants will be greater.
- Each node follows this pattern.



A binary tree might not always be a binary search tree. For instance, if a node has two "left" children, it can't be a binary search tree because a BST can have at most one left child and one right child.

You could implement a tree node in Python with a simple class:

```
class TreeNode:
    def __init__(self, val, left=None, right=None):
        self.value = val
        self.leftChild = left
        self.rightChild = right
```

Then you can create a tree:

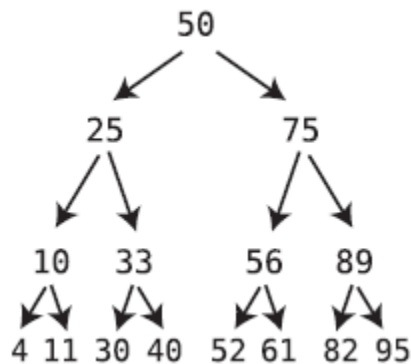
```
node1 = TreeNode(25)
node2 = TreeNode(75)
root = TreeNode(50, node1, node2)
```

Because of this unique structure, searching for values in a binary search tree is typically very efficient.

## Searching

Here's a simple explanation of how you can search within a binary search tree (BST):

1. Start at the root node as the "current node."
2. Check the value at the current node.
3. If it's the value you're looking for, you've found it!
4. If the value you're looking for is less, search in the left subtree.
5. If it's more, search in the right subtree.
6. Repeat steps 2-5 until you find the value or reach the bottom of the tree. If you hit the bottom, the value isn't in the tree.



For example, if you want to search for the number 61:

- Start at the root (say, 50).
- Since 61 is greater than 50, search the right child.
- Move to the next level (say, 75). Since 61 is less than 75, check the left child.
- Move to the next level (say, 56). Since 61 is greater than 56, check the right child.
- You've found 61!

In this example, it took four steps to find the value. This shows how the structure of the BST helps to eliminate unnecessary searches quickly, making the search process efficient.

## The Efficiency of Searching a Binary Search Tree

Searching in a binary search tree (BST) is efficient because each step eliminates half of the remaining nodes. When you decide whether to search the left or right subtree, you exclude the other half from the search. This

pattern of eliminating half the remaining values with each step is characteristic of algorithms that run in  $O(\log N)$  time complexity. In a perfectly balanced binary search tree, where the left and right subtrees have the same number of nodes, this efficiency holds true. However, it's essential to note that this describes the best-case scenario. If the tree is not balanced, the efficiency might vary.

## Log(N) Levels

In a balanced binary tree, the search operation is  $O(\log N)$ , and here's another way to understand why:

1. In a balanced binary tree, every level adds approximately twice the number of nodes as the previous one. So if there are  $N$  nodes in the tree, it will have around  $\log N$  levels (or rows).
2. For instance, if you have a tree with four complete levels, there will be 15 nodes. If you add another level, you'll roughly double the size of the tree by adding 16 new nodes.
3. The pattern of  $\log N$  emerges because each new level accommodates about twice the number of nodes. So, in a tree with 31 nodes, it will take about five levels (since  $\log 31$  is roughly 5).
4. When you search a binary search tree, each step takes you down one level, so it will take up to  $\log N$  steps to find a particular value.
5. Binary search within an ordered array also has the same efficiency as  $O(\log N)$  since each step eliminates half of the remaining values. But binary search trees are more efficient than ordered arrays in other operations like insertion, which we'll explore later.

## Code Implementation: Searching a Binary Search Tree

Here's a description of the code to search a binary search tree using recursion in Python:

```
def search(searchValue, node):  
    # Base case: If the node is nonexistent  
    # or we've found the value we're looking for:  
    if node is None or node.value == searchValue:  
        return node  
  
    # If the value is less than the current node, perform  
    # search on the left child:  
    elif searchValue < node.value:  
        return search(searchValue, node.leftChild)  
  
    # If the value is greater than the current node, perform  
    # search on the right child:  
    else: # searchValue > node.value  
        return search(searchValue, node.rightChild)
```

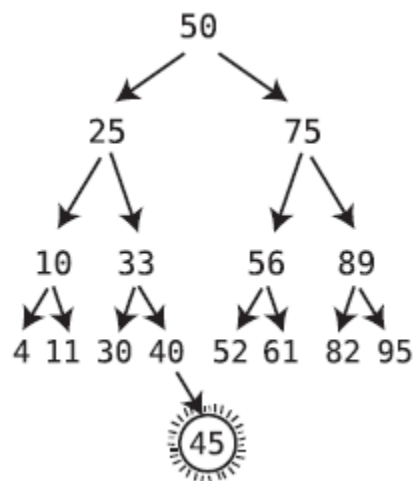
1. **Base Cases:** If the node is **None** (meaning we've reached a dead end in the tree) or the value at the current node matches the search value, the search ends and the node is returned.
2. **Search Left Child:** If the search value is less than the value at the current node, the search continues on the left child of the current node. The search function is called recursively with the left child.

3. **Search Right Child:** If the search value is greater than the value at the current node, the search continues on the right child of the current node. The search function is called recursively with the right child.
4. **How It Works:** The function handles four possibilities:
  - If the node is **None**, the search value is not in the tree, and **None** is returned.
  - If the node value equals the search value, the node is returned.
  - If the search value is less than the node value, the function is called recursively on the left child.
  - If the search value is greater than the node value, the function is called recursively on the right child.
5. **What It Returns:** The function returns the node containing the search value if found; otherwise, it returns **None**.

## Insertion

Inserting a value into a binary search tree is a straightforward process that leverages the structure of the tree.

1. **Start at the Root:** Begin the search for the insertion point at the root of the tree.
2. **Navigate the Tree:** Compare the value to be inserted with the current node's value:
  - If it's less, move to the left child.
  - If it's greater, move to the right child.
  - Repeat this process until you reach a node that has no children in the direction you need to move.
3. **Insert the Value:** Attach the new value as a child to the node where the search ended. If the value to be inserted is less than the node, insert it as a left child. If greater, insert it as a right child.



4. **Efficiency:** The insertion process takes  $O(\log N)$  steps to find the correct location plus one more step to perform the insertion itself. So in total, it takes  $(\log N)+1$  steps. In Big O Notation, this is  $O(\log N)$ , as constants are ignored.

5. **Comparison with Ordered Arrays:** While binary search trees have both search and insertion at  $O(\log N)$ , ordered arrays have search at  $O(\log N)$  but insertion at  $O(N)$  due to the shifting of elements to make room. This makes binary search trees more efficient for operations involving frequent changes to the data.

The key advantage of binary search trees over ordered arrays is that they enable more efficient insertions, while still providing fast search capabilities. This is particularly valuable in situations where data is frequently added or modified.

### Code Implementation: Binary Search Tree Insertion

Here's an explanation that inserts a new value into a binary search tree:

```
def insert(value, node):
    if value < node.value:
        # If the left child does not exist, we want to insert
        # the value as the left child:
        if node.leftChild is None:
            node.leftChild = TreeNode(value)
        else:
            insert(value, node.leftChild)
    elif value > node.value:
        # If the right child does not exist, we want to insert
        # the value as the right child:
        if node.rightChild is None:
            node.rightChild = TreeNode(value)
        else:
            insert(value, node.rightChild)
```

1. **Determine Direction:** Check if the value to be inserted is less than or greater than the value of the current node.
  - If less, you'll insert to the left.
  - If greater, you'll insert to the right.
2. **Insert to the Left:**
  - If there's no left child, simply create a new node with the value and set it as the left child.
  - If there is a left child, make a recursive call to the **insert** function with the left child as the current node.
3. **Insert to the Right:**
  - If there's no right child, create a new node with the value and set it as the right child.
  - If there is a right child, make a recursive call to the **insert** function with the right child as the current node.

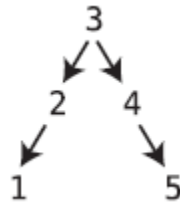
## The Order of Insertion

The order in which values are inserted into a binary search tree (BST) can affect the tree's balance and, consequently, the efficiency of search operations within the tree.

1. **Inserting Sorted Data:** If you insert sorted data into a tree, like the numbers 1, 2, 3, 4, 5 in that order, the tree will become a linear structure. It looks more like a linked list rather than a balanced tree, making search operations take  $O(N)$  time, where  $N$  is the number of nodes.



2. **Inserting Randomized Data:** If the same values were inserted in a different order, like 3, 2, 4, 1, 5, the tree could be evenly balanced. In this balanced structure, search operations would take  $O(\log N)$  time.



3. **Balanced vs Imbalanced Trees:**

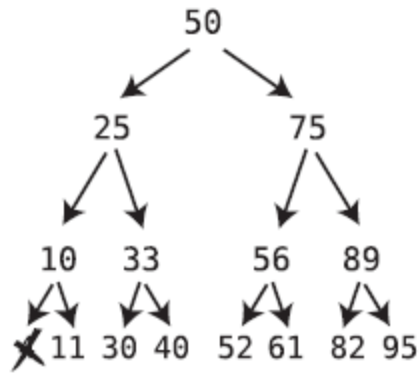
- A perfectly balanced tree allows for the most efficient search, with a time complexity of  $O(\log N)$ .
- An imbalanced tree, particularly one that is linear, will have a search time complexity of  $O(N)$ .

4. **Conversion from an Ordered Array:** If you're converting an ordered array into a BST, it might be beneficial to randomize the order of the data first to help ensure that the tree is well-balanced.

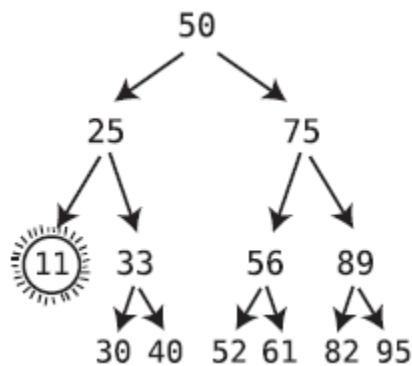
## Deletion

Deletion in a binary search tree (BST) is a more complex operation and depends on the node's child structure. Here's how it works:

1. **Deleting a Node with No Children:** If the node you want to delete doesn't have any children (it's a leaf node), you can simply remove it from the tree. This is the simplest case.



2. **Deleting a Node with One Child:** If the node you want to delete has only one child, you can delete the node and connect its parent to its only child. This means that the child takes the place of the deleted node, preserving the rest of the tree.



The deletion process gets more complex when the node you want to delete has two children, but in the examples above, we are focusing on nodes with zero or one child. The basic principles for these two cases are straightforward:

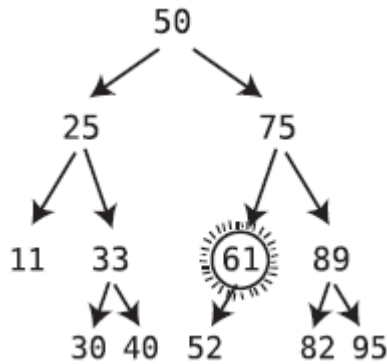
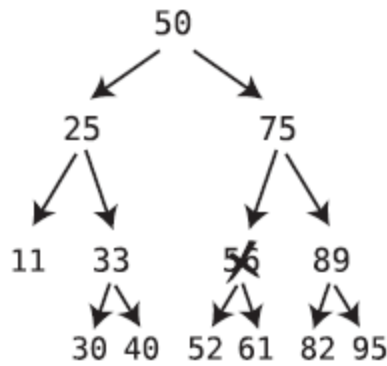
- If there are no children, delete the node directly.
- If there is one child, ensure that the child takes the place of the deleted node to keep the remaining parts of the tree connected.

### Deleting a Node with Two Children

Deleting a node with two children in a binary search tree is a more challenging scenario. Here's how you can handle it:

1. **Find the Successor Node:** When you want to delete a node with two children, you need to find a "successor" node. This node is the smallest one that is larger than the node you want to delete. If you arranged the node and all its descendants in ascending order, the successor would be the next number after the node you want to delete.
2. **Replace with Successor:** Once you find the successor node, you replace the node you want to delete with this successor. In the given example, if you want to delete the node with the value 56, and you have children with values 52 and 61, the successor would be 61. You would then replace the 56 with the 61.



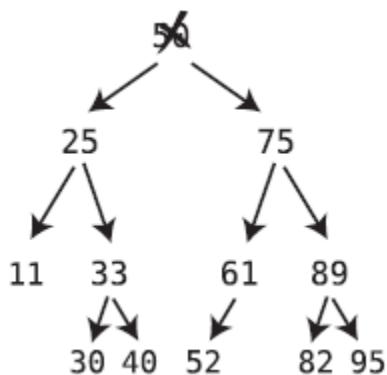


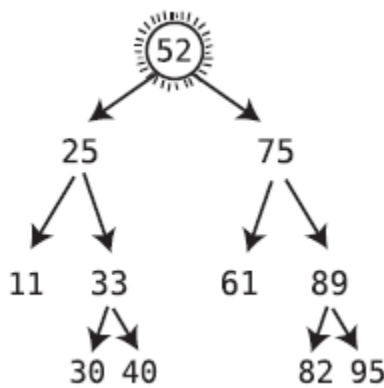
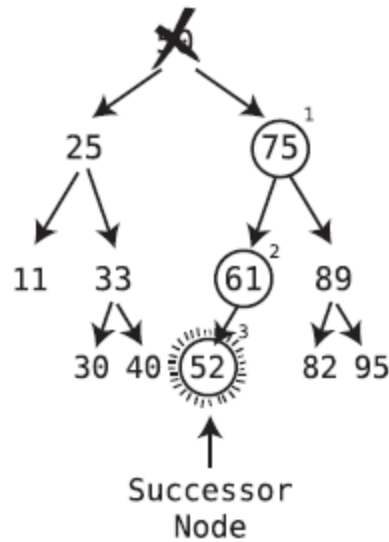
### Finding the Successor Node

Finding the successor node in a binary search tree when deleting a node can seem a bit tricky, especially if the node to be deleted is high up in the tree. Here's a simplified explanation of the process:

1. **Start at the Right Child:** When you want to delete a node, start by visiting its right child.
2. **Go Left Until You Can't:** From there, keep on moving to the left child of each subsequent node until you reach a node that doesn't have a left child.
3. **That's the Successor:** The value you reach is the successor node, the one you will use to replace the deleted node.

For example, if you're deleting the root node with a value of 50 and its right child is 75, you would start there, and then keep going left until you reach a node with no left child, let's say 52. That 52 is the successor node, and you replace the 50 with it, making it the new root.





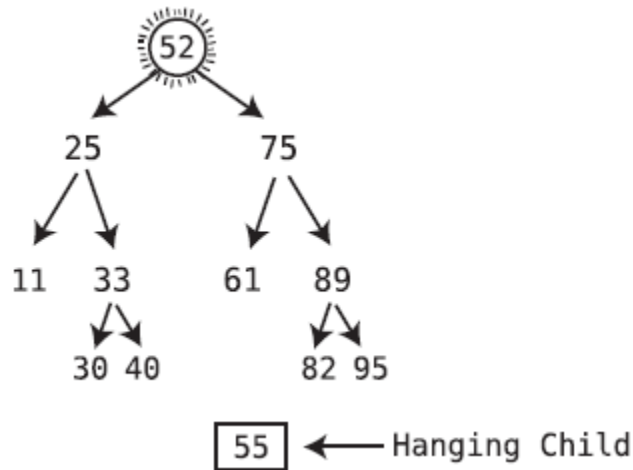
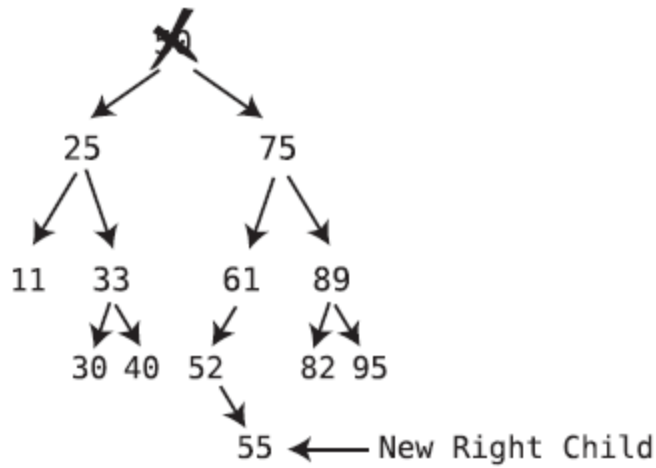
### Successor Node with a Right Child

Dealing with the successor node that has a right child of its own adds a little more complexity to the deletion process in a binary search tree. Here's how this works:

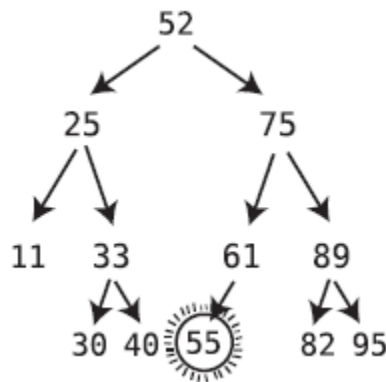
1. **Identify the Successor Node:** This is the node with the smallest value that's greater than the value of the node you're deleting. You typically find it by going to the right child of the deleted node and then continually going to the left child until you find a node with no left child.
2. **Plug in the Successor:** Replace the deleted node with the successor node. If the successor node has a right child (in our example, let's say it's 55), you'll be temporarily leaving it without a parent.
3. **Move the Right Child of the Successor:** To fix this, you'll need to take the former right child of the successor node (55) and turn it into the left child of the former parent of the successor node (in our example, 61).
4. **Check the Tree:** Ensure that the new structure maintains the binary search tree properties.

Here's an example:

- If you were to delete the root node and find 52 as the successor node with a right child of 55, you'd first replace the root with 52.



- Then, you'd make the 55 (formerly the right child of 52) the left child of 61, which was the original parent of 52.



### The Complete Deletion Algorithm

The complete deletion algorithm for a binary search tree can be summarized into the following set of rules:

1. **No Children:** If the node being deleted doesn't have any children, simply remove it from the tree.
2. **One Child:** If the node being deleted has only one child, remove the node and connect its child to the deleted node's parent.
3. **Two Children:** If the node being deleted has two children, it's a bit more complex:

- a. **Find the Successor Node:** Identify the successor node, which is the smallest value greater than the deleted node. Start from the right child of the deleted node and keep going left until there's no left child.
- b. **Replace with Successor:** Replace the deleted node with the successor node.
- c. **Handle Successor's Right Child:** If the successor node has a right child, make it the left child of the successor node's former parent.

#### Code Implementation: [Binary Search Tree Deletion](#)

The code for deletion from a binary search tree is rather detailed and involved.

```

def delete(valueToDelete, node):
    # The base case is when we've hit the bottom of the tree,
    # and the parent node has no children:
    if node is None:
        return None

    # If the value we're deleting is less or greater than the current node,
    # we set the left or right child respectively to be
    # the return value of a recursive call of this
    # very method on the current
    # node's left or right subtree.
    elif valueToDelete < node.value:
        node.leftChild = delete(valueToDelete, node.leftChild)

        # We return the current node (and its subtree if existent) to
        # be used as the new value of its parent's left or right child:
        return node
    elif valueToDelete > node.value:
        node.rightChild = delete(valueToDelete, node.rightChild)
        return node

    # If the current node is the one we want to delete:
    elif valueToDelete == node.value:

        # If the current node has no left child, we delete it by
        # returning its right child (and its subtree if existent)
        # to be its parent's new subtree:
        if node.leftChild is None:
            return node.rightChild

            # (If the current node has no left OR right child, this ends up
            # being None as per the first line of code in this function.)

        elif node.rightChild is None:
            return node.leftChild

        # If the current node has two children, we delete the current node
        # by calling the lift function (below),
        # which changes the current node's
        # value to the value of its successor node:
        else:
            node.rightChild = lift(node.rightChild, node)
            return node

def lift(node, nodeToDelete):
    # If the current node of this function has a left child,
    # we recursively call this function to continue down
    # the left subtree to find the successor node.
    if node.leftChild:
        node.leftChild = lift(node.leftChild, nodeToDelete)
        return node

    # If the current node has no left child, that means the current node
    # of this function is the successor node, and we take its value
    # and make it the new value of the node that we're deleting:
    else:
        nodeToDelete.value = node.value
        # We return the successor node's right child to be now used
        # as its parent's left child:
        return node.rightChild

```

1. **Base Case:** If the node doesn't exist (reached the bottom of the tree), just return **None**.
2. **Recursive Cases:**
  - If **valueToDelete** is less than the current node's value, recursively call the delete function on the left child.
  - If **valueToDelete** is greater than the current node's value, recursively call the delete function on the right child.
  - If **valueToDelete** equals the current node's value, deletion must occur.
3. **Actual Deletion:**
  - **No Children:** If the current node has no children, return **None**.
  - **One Child:** If the node has only one child (left or right), return that child.
  - **Two Children:** Find the successor node using a function called **lift**. Replace the current node's value with the successor's value and adjust the children accordingly.

#### Deletion Function Breakdown:

- **Finding the Successor (lift function):** This function finds the successor node (smallest value greater than the deleted value) and takes its value, moving its right child into its parent's left child if applicable.
- **If No Left Child:** Delete the current node by returning its right child.
- **If No Right Child:** Delete the current node by returning its left child.
- **If Two Children:** Apply the **lift** function to find and use the successor node, returning the current node after replacement.

#### Key Points:

- **Recursive Nature:** The algorithm makes use of recursive calls, going down the left or right subtree depending on the value to be deleted.
- **Handling Children:** Care is taken to adjust children nodes properly to maintain the binary search tree properties.
- **Successor Handling:** The **lift** function ensures that the successor node's value is used for replacement and that its children are handled correctly.

#### The Efficiency of Binary Search Tree Deletion

Like search and insertion, deleting from trees is also typically  $O(\log N)$ . This is because deletion requires a search plus a few extra steps to deal with any hanging children. Contrast this with deleting a value from an ordered array, which is  $O(N)$  due to shifting elements to the left to close the gap of the deleted value.

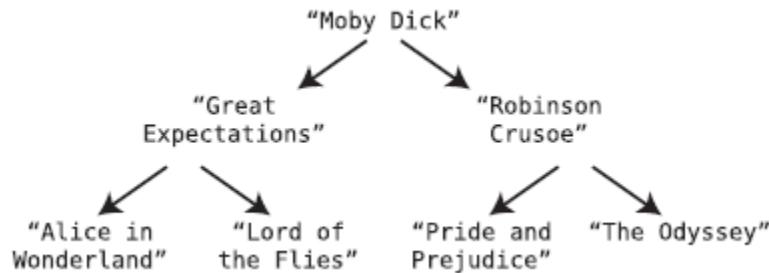
#### Binary Search Trees in Action

##### What BSTs Do:

- They offer quick operations like searching, inserting, and deleting (usually  $O(\log N)$  time complexity).
- They are more efficient than ordered arrays for frequent insertions and deletions.

### Why You Might Use Them:

- Imagine you're creating an app for managing a large list of book titles.
- You want to print the titles alphabetically, allow constant updates, and search within the list.



### Binary Search Trees vs. Ordered Arrays:

- **Ordered Arrays:** Good for static lists where you mostly search. They're fast for searching but slow for inserting and deleting.
- **Binary Search Trees:** Ideal for dynamic lists where data changes frequently. They provide quick search, insert, and delete operations.

### Example Scenario:

- You're building an app with a list of millions of book titles.
- You need to make real-time changes, such as add or remove titles.
- A binary search tree is suitable because it can handle these changes efficiently.
- Titles are organized in the tree based on alphabetical order, with "lower" alphabet titles on the left and "greater" on the right.

If you are working with large amounts of data that need frequent updates, BSTs might be a more efficient choice compared to ordered arrays.

### Binary Search Tree Traversal

Here's how to print the book titles stored in a binary search tree (BST) in alphabetical order using a method called inorder traversal.

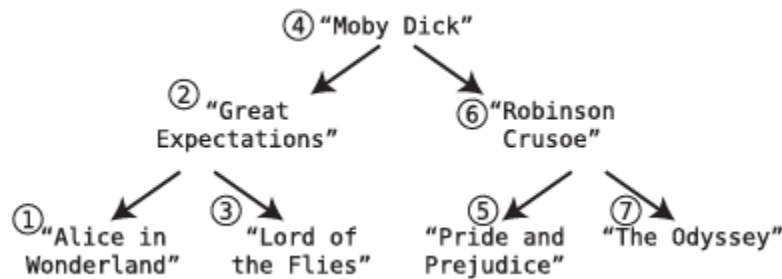
#### What is Inorder Traversal?

Inorder traversal is a way to visit all the nodes in a BST, in a specific order that will align with alphabetical order for a collection of book titles. In this case, it's done using a recursive method.

#### How Does It Work?

1. **Visit Left Child:** Start by visiting the left child of the current node, and continue doing so recursively until you reach a node with no left child.

2. **Print the Current Node:** Once you've reached the leftmost node, print its value (in this case, a book title).
3. **Visit Right Child:** Then, visit the right child of the current node, and continue doing so recursively until you reach a node with no right child.
4. **Continue Recursively:** Keep repeating these steps for each node in the tree until you've printed all the titles.



### Python Code:

Here's a simple Python function to accomplish this:

```

def traverse_and_print(node):
    if node is None:
        return
    traverse_and_print(node.leftChild)
    print(node.value)
    traverse_and_print(node.rightChild)
  
```

### How Does It Execute?

- The function will start with the root of the tree, such as "Moby Dick."
- It will traverse left to "Great Expectations," then left again to "Alice in Wonderland," printing it since there's no left child.
- It will then print "Great Expectations," and continue to "Lord of the Flies," printing it as well.
- This process continues for the whole tree, visiting each node in alphabetical order and printing it.

### Time Complexity:

Since every node in the tree must be visited, the time complexity for this operation is  $O(N)$ , where  $N$  is the total number of nodes. Inorder traversal in a BST provides a way to print all the book titles in alphabetical order efficiently, following the logical flow from the leftmost to the rightmost nodes in the tree.

### Wrapping Up

The binary search tree is a powerful node-based data structure that provides order maintenance, while also offering fast search, insertion, and deletion. It's more complex than its linked list cousin, but it offers tremendous value. However, the binary search tree is just one type of tree. There are many different kinds of trees, and each brings unique advantages to specialized situations. In the next chapter, we're going to discover another tree that will bring unique speed advantages to a specific, but common scenario.



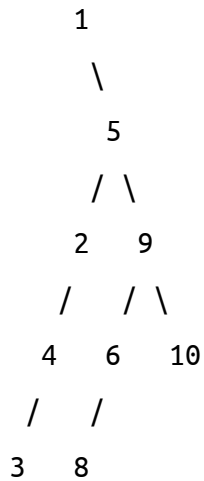
## Exercises

The following exercises provide you with the opportunity to practice with binary search trees.

1. Imagine you were to take an empty binary search tree and insert the following sequence of numbers in this order: [1, 5, 9, 2, 4, 10, 6, 3, 8]. Draw a diagram showing what the binary search tree would look like.

Remember, the numbers are being inserted in the order presented here.

By inserting the numbers [1, 5, 9, 2, 4, 10, 6, 3, 8] into an empty binary search tree in the given order, the tree would look like:



2. If a well-balanced binary search tree contains 1,000 values, what is the maximum number of steps it would take to search for a value within it?

The maximum number of steps required to search for a value would be the height of the tree. In a balanced binary tree, the height is  $\log_2 N$ , so for 1,000 values, it would be  $\log_2 1000$ , which is approximately 10.

3. Write an algorithm that finds the greatest value within a binary search tree.

To find the greatest value in a binary search tree, you can traverse the tree to the rightmost node.

Here's a simple algorithm to do this:

```
def find_greatest_value(node):
    while node.rightChild is not None:
        node = node.rightChild
    return node.value
```

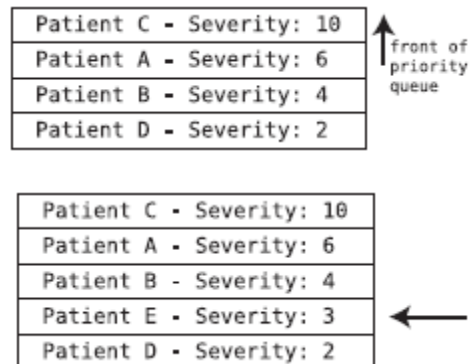
## CHAPTER 16: Keeping Your Priorities Straight with Heaps

We previously explored the concept of trees and their various forms, including binary search trees. Each type of tree offers its own set of benefits and drawbacks, making it crucial to select the most suitable one for a given task. This chapter will focus on heaps, a unique tree structure designed to quickly identify the largest or smallest elements in a dataset. Additionally, we will delve into another topic: priority queues.

### Priority Queues

Priority queues build on what you learned about queues, which process items First In, First Out (FIFO). In a queue, data is added at the end and removed from the front, keeping the original order of insertion. A **priority queue** works like a regular queue for deletions and access but adds data like an ordered array. This means that when data is inserted, it's sorted to ensure a specific order. Deletions and access only happen from the front, but insertions maintain the ordered sequence.

Imagine a hospital emergency room. Patients aren't treated based on arrival time but the severity of symptoms. Someone with a life-threatening injury gets immediate attention, even if they arrived after a patient with a minor issue. Suppose patients are ranked from 1 to 10 in urgency, with 10 most critical. A priority queue ensures that the most urgent patients are treated first. If a new patient arrives with a lower severity, they are placed in the proper spot in the priority queue.



A priority queue is an abstract data type that can be implemented using other structures like an ordered array. When using an array, specific constraints are applied:

- Insertions must maintain the proper order.
- Deletions occur only from the end of the array, representing the front of the priority queue.

Analyzing efficiency, deletions are quick ( $O(1)$ ) because they occur at the end of the array. Insertions are slower ( $O(N)$ ), as they may require shifting elements to maintain order. The array-based priority queue's deletion speed is acceptable, but the insertion speed may slow down applications if there are many items. Therefore, computer scientists developed a more efficient structure for priority queues called the heap.

### Heaps

Heaps are a special category of trees, and we'll be looking at a specific kind called the binary heap. In general, a binary tree is one where each node has at most two children. Binary heaps have two main types: max-heap

and min-heap. For our purposes, we'll focus on the max-heap, but the difference between the two is minimal. A binary max-heap, often referred to simply as a **heap**, follows two main rules:

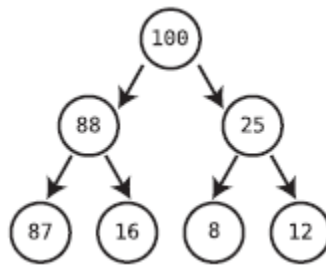
1. The value of each node must be greater than the values of its descendant nodes. This is known as the heap condition.
2. The tree must be complete, meaning every level of the tree is filled except possibly the last one, and all the nodes are as far left as possible.

These rules define the structure and behavior of a heap, ensuring that it's always easy to find the maximum value in a max-heap.

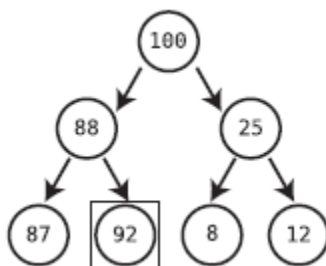
### The Heap Condition

The heap condition in a binary heap specifies that the value of each node must be greater than all its descendants. In a valid max-heap:

- Each node is greater than any of its children.
- For example, if the root node is 100, no descendant is greater than 100. If another node is 88, it's greater than both its children, and so on.



An invalid heap doesn't meet this condition. For instance, if a child node 92 is greater than its parent 88, it violates the heap condition.

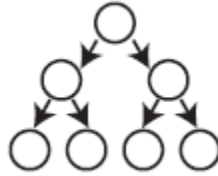


A binary heap is different from a binary search tree, where each node's right child is greater than it. In a heap, no descendants are greater than the node itself. There's also a min-heap, where each node must be smaller than its descendants, but we're focusing on the max-heap. The only difference between max-heap and min-heap is this reversal in the condition. Otherwise, the fundamental structure and concept remain the same.

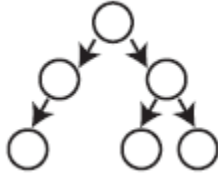
### Complete Trees

The second rule of heaps is that the tree needs to be complete. A complete tree is one where all levels are filled with nodes, except possibly for the last row, which must be filled from left to right with no gaps. Here's a breakdown:

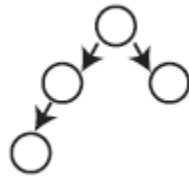
- A complete tree has every level filled with nodes. If the last row is not filled, it should have no gaps on the left.



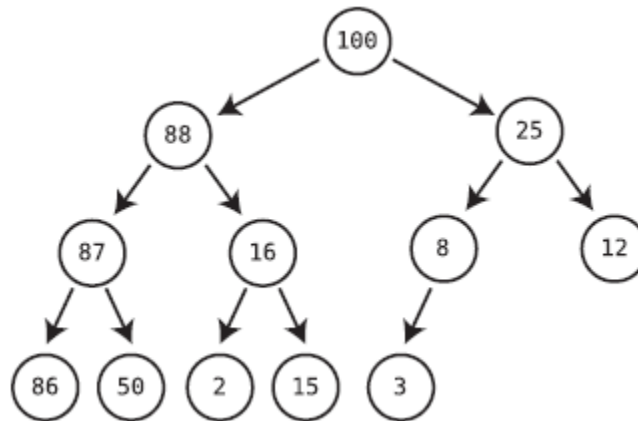
- A tree that's missing a node in any level except the last is not considered complete.



- Even if the last row is not entirely filled, the tree is still complete if all empty positions are to the right.



In the context of a heap, both the heap condition (each node being greater than its descendants) and the complete tree condition must be met. Gaps in the bottom row are acceptable as long as they are to the very right of the tree. This combination of conditions helps the heap efficiently maintain its order and makes operations like insertions and deletions more predictable.

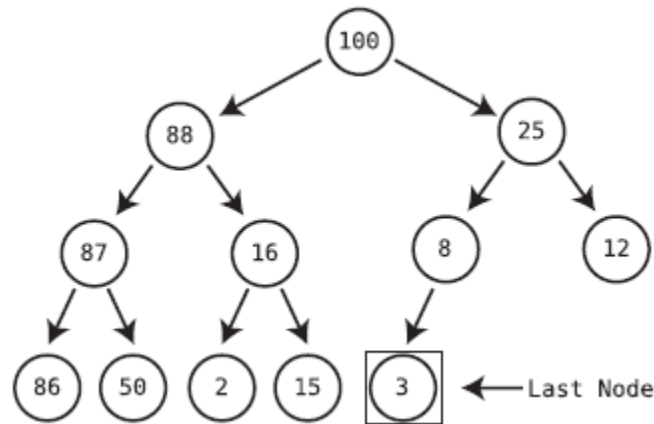


## Heap Properties

Heaps have some interesting properties that are worth noting:

1. **Weak Ordering:** Although the heap condition creates a certain order (descendants cannot be greater than ancestors), this is not enough for efficient searching. If you want to find a specific value in the heap, you don't have clear guidance on whether to look in the left or right descendants of a node. This makes heaps weakly ordered, unlike binary search trees.

- Root Node Value:** In a max-heap, the root node always contains the greatest value, while in a min-heap, it contains the smallest value. This property makes heaps a good fit for priority queues, where you often want to access the item with the highest priority.
- Last Node:** The last node in a heap is the rightmost node in its bottom level. This can be important in understanding the structure of the heap and in implementing operations.

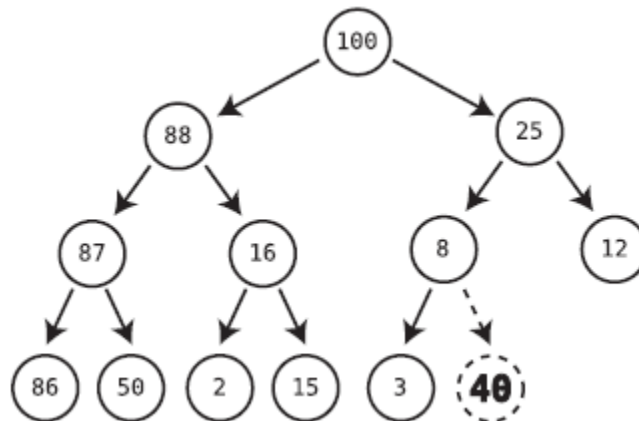


- Primary Operations:** Heaps mainly deal with two operations: inserting and deleting. Searching is usually not implemented in the context of heaps, as it would require inspecting each node. There might also be an optional "read" operation to view the root node's value.

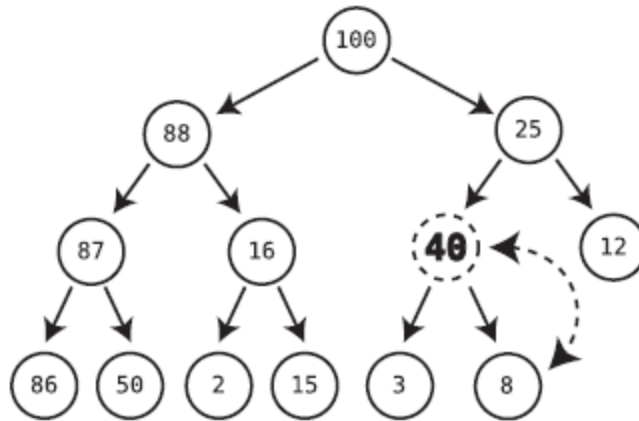
### Heap Insertion

Inserting a new value into a heap follows a clear algorithm that ensures the heap condition and the complete tree property are maintained.

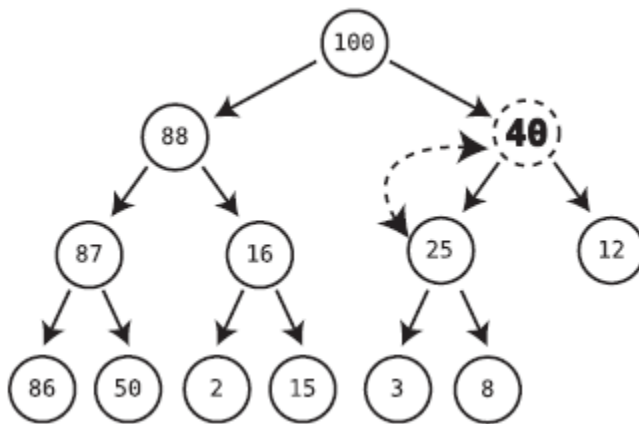
- Create and Place the New Node:** Add a new node with the value you want to insert at the rightmost spot in the bottom level. This becomes the last node in the heap and ensures the tree remains complete.



- Compare with Parent:** Check if the new node's value is greater than its parent. If so, swap the new node with the parent.



3. **Repeat Comparisons:** Continue comparing the new node with its parent and swapping if necessary, moving the new node upward through the heap until it has a parent with a value greater than itself.



4. **Settling the Node:** The process of moving the new node up is called "trickling" the node through the heap. It can move up to the right or left but will always continue upward until it settles into the correct position that maintains the heap's properties.
5. **Efficiency:** This insertion process has a time complexity of  $O(\log N)$ , as the tree is organized into about  $\log(N)$  rows, and at most, the new value might need to trickle up to the top row.

### Looking for the Last Node

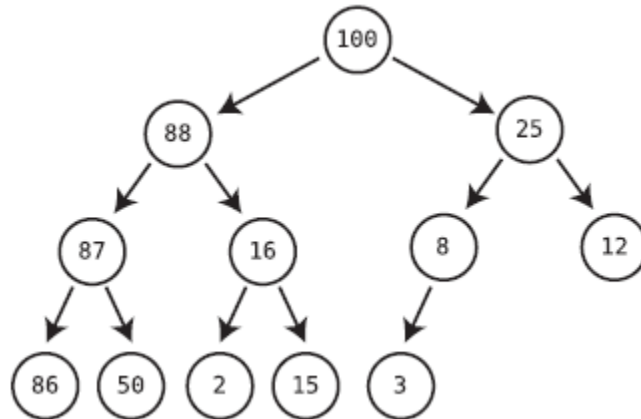
The heap insertion process has a challenging aspect, known as finding the last node or the next available spot for the new value.

1. **The Challenge:** When inserting a new value, it needs to be placed as the last node in the heap. Visually, this might seem simple, but computers don't "see" the heap as rows and columns. They only know the root node and can follow links to child nodes.
2. **Different Scenarios:** Depending on the structure of the heap, the next available spot might be on the left or right descendants of the root. This means there's no straightforward path to find this spot without inspecting each node.
3. **No Easy Solution:** It's not easily possible to search through the heap or efficiently find the last node without examining every node.

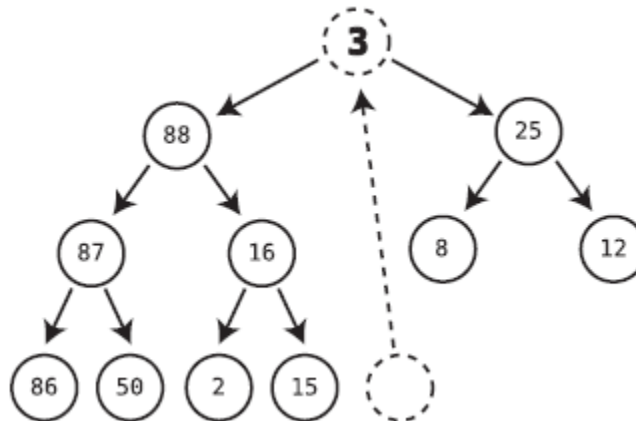
4. **The Problem of the Last Node:** This issue is significant enough to be referred to as "the Problem of the Last Node." This problem stands as a complexity in the otherwise straightforward insertion algorithm.

### Heap Deletion

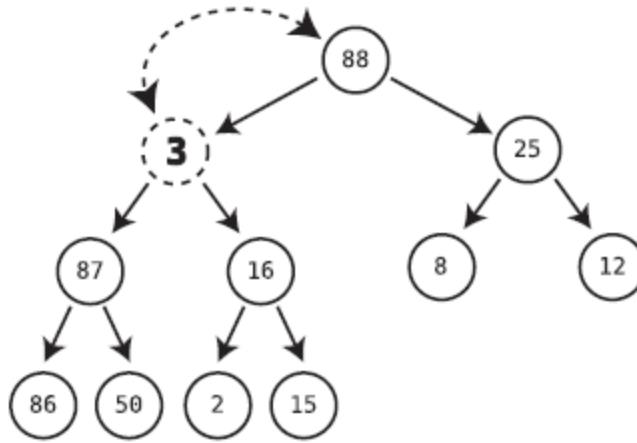
Heap deletion involves removing the root node, as this is the highest-priority item in a priority queue. Here's how the deletion process works in a heap:



1. **Move the Last Node to the Root:** The algorithm begins by taking the last node of the heap and placing it where the root node was, thus removing the original root node.



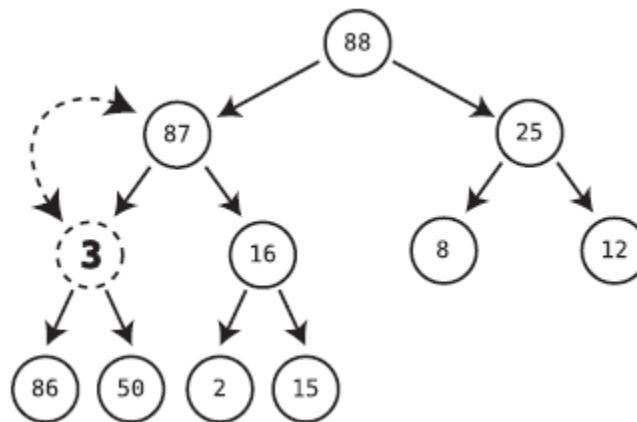
2. **Trickle Down:** Next, the heap needs to be reorganized by "trickling" the new root node down to its proper place. This process involves comparing the new root node with its children and swapping it with the larger child until it's in the correct position.



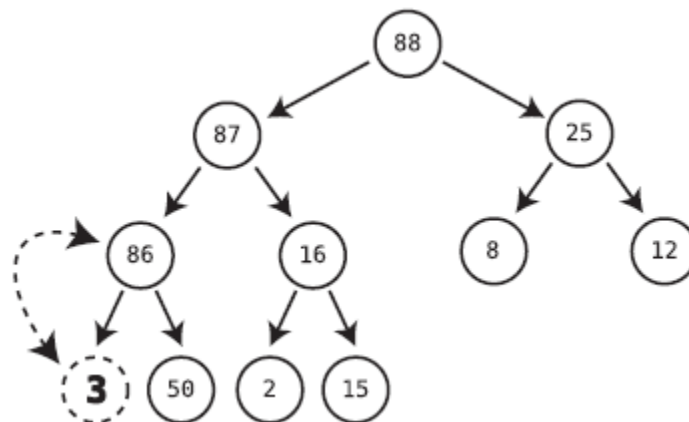
3. **Detailed Steps of Trickling Down:**

a. **Check the Children:** Examine both children of the new root node to see which one is larger.

b. **Compare and Swap:** If the new root node is smaller than the larger of the two children, swap it with that larger child.

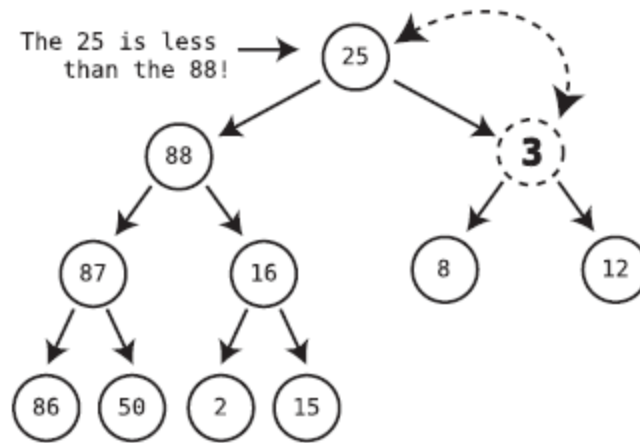


c. **Repeat:** Continue these steps until the new root node has no children who are greater than it.



4. **Why Swap with the Larger Child?:** Swapping with the larger child ensures that the heap condition is not violated. Swapping with the smaller child could lead to a situation where a child node is greater than its parent, breaking the heap condition.





5. **Time Complexity:** The deletion operation in a heap has a time complexity of  $O(\log N)$ , as the new root node must be trickled down through all  $\log(N)$  levels of the heap.

### Heaps vs. Ordered Arrays

Heaps and ordered arrays are two structures that can be used to implement priority queues, and each has its own strengths and weaknesses. Here's a comparison:

	Ordered Array	Heap	Ordered Array	Heap
Insertion	$O(N)$	$O(\log N)$	Slow	Very fast
Deletion	$O(1)$	$O(\log N)$	Extremely fast	Very fast

While ordered arrays offer extremely fast deletion, their insertion is slow. Heaps, on the other hand, provide very fast performance for both insertion and deletion. This leads to the reason why heaps are generally considered a better choice:

- **Consistency:** Heaps are consistently very fast for both primary operations, while ordered arrays are sometimes extremely fast and sometimes slow.
- **Equal Proportion of Operations:** In scenarios like a priority queue (e.g., in an emergency room), both insertions and deletions occur frequently and should be fast. A slow operation would make the whole system inefficient.

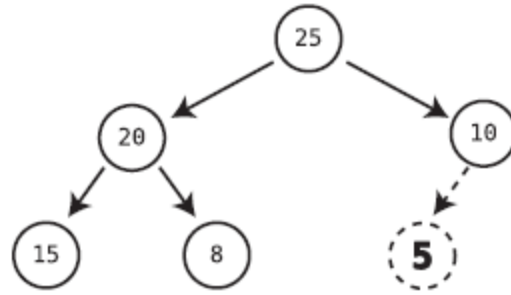
When looking at the overall performance and consistency, heaps tend to be the preferred choice for implementing priority queues, as they ensure both insertions and deletions are performed quickly.

### The Problem of the Last Node...Again

The Problem of the Last Node in a heap is a recurring challenge both in the process of inserting a new node and in deleting the root node. It raises the question: How can we consistently find the last node of a heap? Let's simplify the complexity of this problem and its importance:

#### 1. Importance of the Last Node:

- **Insertion:** When inserting a new value into a heap, placing it in the last node's position ensures that the heap stays complete and well-balanced.



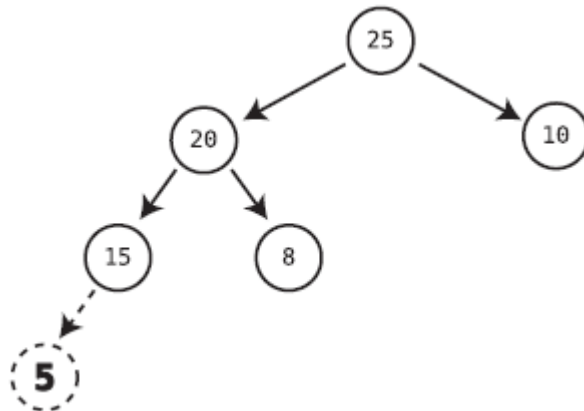
- **Deletion:** When deleting the root node, replacing it with the last node maintains the heap's balance.

## 2. Why Balance Matters:

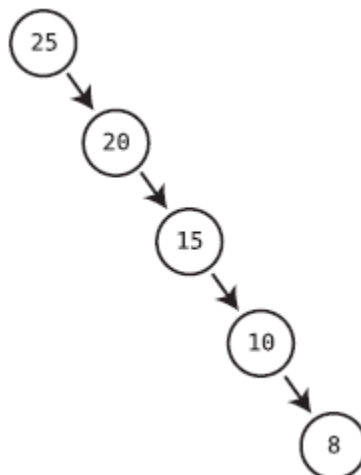
- A well-balanced heap is crucial for achieving efficient operations ( $O(\log N)$ ). An imbalanced heap could lead to slower operations ( $O(N)$ ).
- If we were to insert or delete nodes in ways that do not consider the last node's position, the heap could quickly become imbalanced, affecting its efficiency.

## 3. Example Scenarios:

- **Imbalanced Insertion:** If we insert new nodes at the bottom leftmost spot, the heap becomes imbalanced.



- **Imbalanced Deletion:** If we always move the bottom rightmost node into the root position, the heap also becomes imbalanced.

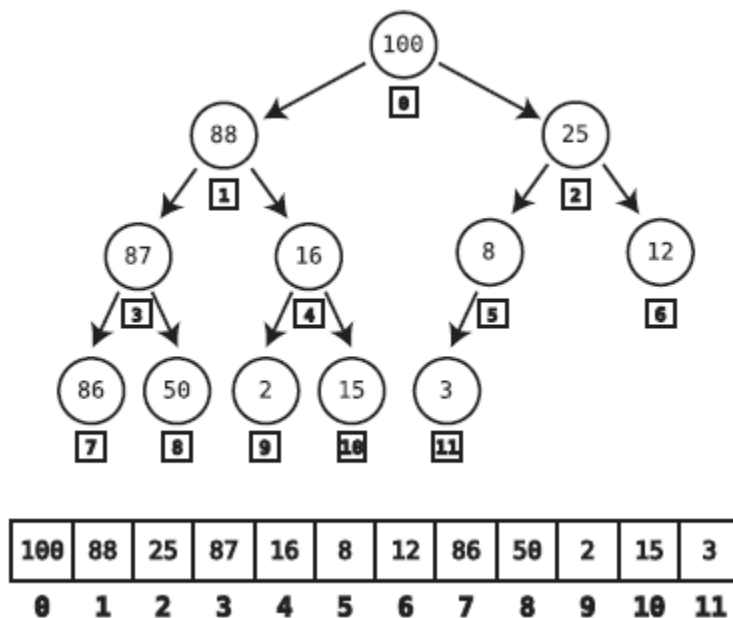


## Arrays as Heaps

Heaps can be implemented using arrays, and this approach provides an elegant solution to the Problem of the Last Node. Here's how this works:

### 1. Array Implementation of Heap:

- Instead of treating the heap as a collection of independent nodes connected by links, we can use an array.
- Each node is assigned to an index within the array, following a specific pattern.
- The root node is at index 0, and other nodes are assigned indices moving down a level and going from left to right.



### 2. Solving the Problem of the Last Node:

- By implementing the heap as an array, the last node always corresponds to the final element of the array.
- This makes finding the last node trivial and allows for efficient insertion at the end of the array to make it the last node.

### 3. Example of Array-Based Heap in Code:

- In Ruby, you might begin implementing a heap as a class with an array. Here's an illustration:

```

class Heap
  def initialize
    @data = []
  end

  def root_node
    return @data.first
  end

  def last_node
    return @data.last
  end
end

```

- The **root\_node** method returns the first item of the array (root), and the **last\_node** method returns the last value of the array (last node).

Using an array to represent a heap is an intuitive and efficient approach. It ensures easy access to the last node, a crucial element for heap operations, and allows for simple and clear coding of the heap's structure.

### Traversing an Array-Based Heap

Traversing an array-based heap can be done efficiently using simple mathematical formulas that relate the parent and child nodes. Here's how this is achieved:

#### 1. Array-Based Heap Structure:

- The heap is represented as an array, and the nodes are assigned indexes according to a specific pattern.

#### 2. Formulas to Navigate the Heap:

- **Left Child:** To find the left child of a node at a given index, use the formula **(index \* 2) + 1**.
- **Right Child:** To find the right child, use the formula **(index \* 2) + 2**.
- **Parent Node:** To find the parent of a node at a given index, use the formula **(index - 1) / 2**. Note that you must use integer division, so any fractional part is discarded.

#### 3. Implementation in Code:

- These formulas can be implemented as methods in a Heap class, like so:

```

def left_child_index(index)
  return (index * 2) + 1
end

def right_child_index(index)
  return (index * 2) + 2
end

def parent_index(index)
  return (index - 1) / 2
end

```

- These methods allow you to navigate the array-based heap, accessing left and right children, or the parent of a given node, all by using the corresponding index.

#### 4. Why This Works:

- By following the specific pattern for assigning indexes, these relationships between parent and child nodes always hold true.
- This enables the array to be treated as a tree, allowing efficient navigation and manipulation of the heap using simple arithmetic.

#### Code Implementation: [Heap Insertion](#)

The given code describes the heap insertion algorithm, and it can be broken down into the following key steps:

##### 1. Insert the New Value:

- Add the new value to the end of the array: `@data << value`.
- The new value becomes the last node in the heap.

##### 2. Identify the New Node's Index:

- Keep track of the index of the new node: `new_node_index = @data.length - 1`.

##### 3. Trickle Up the New Node:

- This process ensures that the heap's properties are maintained.
- A while loop is used to continually compare the new node with its parent.
- The loop runs until either the new node's index is 0 (meaning it's the root) or the new node is no longer greater than its parent.
- Inside the loop:
  - If the new node is greater than its parent, they are swapped.
  - The index of the new node is then updated to be the index of the parent.

##### 4. Loop Ends:

- Once the new node is no longer greater than its parent or it becomes the root, the loop ends.
- The new node is now in its proper place in the heap.

The insertion algorithm adds the new value to the end of the array (making it the last node of the heap), and then "trickles it up" by repeatedly swapping it with its parent until it reaches the correct position in the heap. This ensures that the heap's specific ordering property (e.g., a max heap where the parent is always greater than its children) is preserved.

#### Code Implementation: [Heap Deletion](#)

The Ruby implementation of deleting an item from a heap can be simplified into the following main components:

##### 1. Deleting the Root Node:

- Since deletion always occurs at the root node in a heap, the value of the last node in the array is moved to the root position: **@data[0] = @data.pop**.
- This action deletes the root node and replaces it with the last node.

## 2. Trickle the New Root Down:

- The new root node (termed as the "trickle node") may not be in its proper place, so it needs to be moved or "trickled down" the heap.
- Initialize the index of the trickle node: **trickle\_node\_index = 0**.

## 3. Trickle-Down Algorithm:

- A while loop is used to keep moving the trickle node down the heap as long as it has a child greater than itself.
- Within the loop:
  - Find the larger child of the trickle node using **calculate\_larger\_child\_index**: **larger\_child\_index = calculate\_larger\_child\_index(trickle\_node\_index)**.
  - Swap the trickle node with its larger child.
  - Update the index of the trickle node to the larger child's index: **trickle\_node\_index = larger\_child\_index**.

## 4. Helper Methods:

- **has\_greater\_child(index)**: Checks if the node at the given index has left or right children greater than itself.
- **calculate\_larger\_child\_index(index)**: Returns the index of the larger child of the node at the given index.

## 5. Completion:

- The loop continues until the trickle node doesn't have any children greater than itself, ensuring that it's in its proper place in the heap.

When deleting from the heap, the root node is removed and replaced with the last node. This new root is then "trickled down" to its correct position by repeatedly swapping it with its larger child until it's in the correct place in the heap. The implementation makes use of helper methods to make the code more concise and understandable.

## Alternate Heap Implementations

The heap implementation discussed earlier uses an array to represent the underlying structure, but that's not the only way to implement a heap. Here's a simpler look at the alternate ways to create a heap:

### 1. Array Implementation:

- This is the common approach used for heaps.

- The heap structure is represented using an array, where parent and child nodes can be accessed using simple arithmetic formulas.
- It's fascinating how an array can be used to emulate a tree structure.
- This method has an advantage in finding the last node easily, making it suitable for heaps.

## 2. **Linked Nodes Implementation:**

- It's also feasible to implement a heap using linked nodes.
- This method involves a different strategy to solve the problem of finding the last node, often involving binary numbers.
- Though not as common as the array method, it's an alternative way to represent a heap.

## Heaps as Priority Queues

A heap serves as an efficient implementation for priority queues, providing quick access to the highest-priority items. Here's how it works, compared to other implementations like ordered arrays:

### 1. **Purpose of Priority Queue:**

- In a priority queue, you can access and remove the highest-priority item immediately.
- In scenarios like an emergency room, the most urgent case needs to be addressed first.

### 2. **Heaps as Priority Queues:**

- Heaps are well-suited for priority queue implementation.
- The highest-priority item is always found at the root node of the heap.
- When the top-priority item is removed, the next-highest floats to the top.
- The heap offers fast insertions and deletions, both of which are  $O(\log N)$  time complexity.

### 3. **Comparison with Ordered Arrays:**

- Ordered arrays require slower insertions, at  $O(N)$  time, to maintain order.
- Heaps do not need to be perfectly ordered, which is actually an advantage.
- The weak ordering of heaps allows quick insertions, yet the structure is just enough to always access the greatest value when needed.

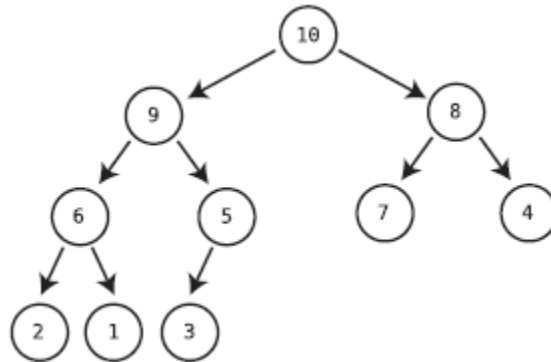
## Wrapping Up

So far, we've seen how different types of trees can optimize different types of problems. Binary search trees kept search fast while minimizing the cost of insertions, and heaps were the perfect tool for building priority queues. In the next chapter, we'll explore another tree that is used to power some of the most common text-based operations you use on a day-to-day basis.

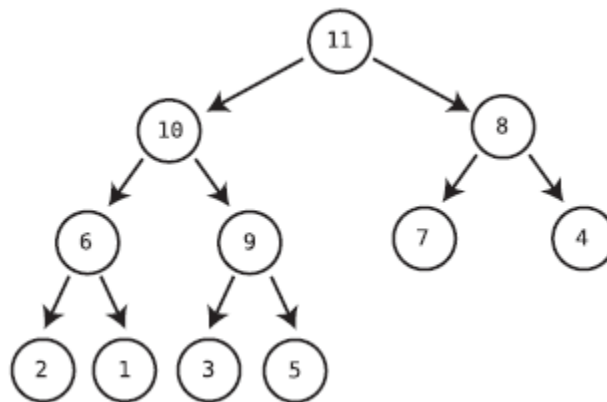
## Exercises

The following exercises provide you with the opportunity to practice with heaps.

1. Draw what the following heap would look like after we insert the value 11 into it:

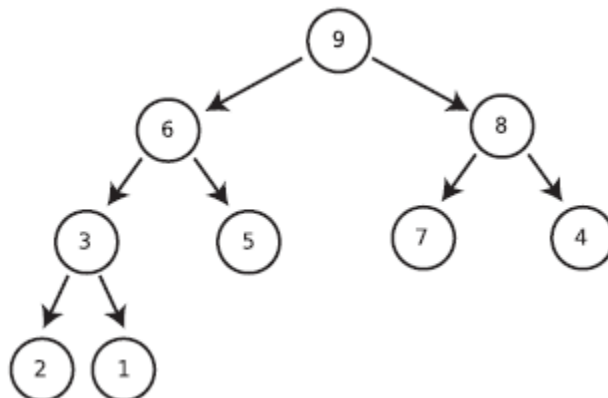


Remember, insertion in a heap is done by adding the value at the end and then performing a "trickle-up" operation to place it in the correct position.



2. Draw what the previous heap would look like after we delete the root node.

Deletion of the root in a heap involves replacing it with the last value, then performing a "trickle-down" operation to reorganize the heap.





3. Imagine you've built a brand-new heap by inserting the following numbers into the heap in this particular order: 55, 22, 34, 10, 2, 99, 68. If you then pop them from the heap one at a time and insert the numbers into a new array, in what order would the numbers now appear?

- You'll build a heap by inserting the given numbers: 55, 22, 34, 10, 2, 99, 68.
- Then, you'll remove the numbers from the heap one at a time (always removing the root), inserting them into a new array.
- Since heaps maintain the largest value at the root (for a max heap), the array would contain the numbers in descending order.

The resulting array after popping the elements would be [99, 68, 55, 34, 22, 10, 2].

## CHAPTER 17: It Doesn't Hurt to Trie

You know how your phone suggests words when you're typing something like "catn," and it might offer "catnip" or "catnap"? Ever wondered how that works? Your phone has a whole dictionary, and it needs to figure out what words you might be typing quickly. If the words were just randomly thrown into a list, your phone would have to look at every single word to find ones that start with "catn." That would be really slow.

If the words were sorted, like in alphabetical order, your phone could use a technique called binary search to find the right words much faster. But there's an even quicker way! A special kind of structure called a **"trie"** can be used. It helps your phone find the words you might be typing really quickly. Tries are used not only for text but can also help in other areas like working with IP addresses or phone numbers.

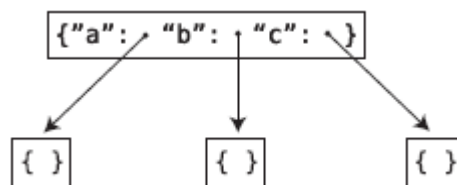
### Tries

The **trie** is a specific type of tree structure used mainly for things like autocomplete in text. Now, the name "trie" can be a bit confusing. Even though it comes from the word "retrieval," it doesn't sound like "tree." If it did, people might mix it up with the general term for tree structures. So, most folks pronounce "trie" like "try." Some people might call it a prefix tree or digital tree, but "trie" is still the name most people use.

One more thing to know is that the trie isn't as commonly explained as some other structures, and different people might use slightly different versions of it. The main ideas are usually the same, though, so don't worry too much about the differences.

### The Trie Node

The trie is like a tree, but unlike regular trees that only have two branches from each point (called binary trees), a trie can have as many branches as it needs. Imagine each part of the trie as a container (node) that holds a small table. The table's keys are letters like "a," "b," "c," etc., and the values are links to other containers or nodes.



Here's a way to understand it:

- The starting container, or root node, has a table with the keys "a," "b," and "c."
- Those keys link to other containers, which can also have tables with links to even more containers.
- It keeps going like this, building a network of linked containers.

In computer code (like Python), a trie node can be created with a simple class like this:

```
class TrieNode:
    def __init__(self):
        self.children = {}
```

This class has a dictionary called **children** that works like the small tables we talked about. If you were to look at the data from our starting container, it would show something like this:

```
{'a': <__main__.TrieNode instance at 0x108635638>,  
'b': <__main__.TrieNode instance at 0x108635878>,  
'c': <__main__.TrieNode instance at 0x108635ab8>}
```

Each key is a letter, and the value is the location of another container in the network.

## The Trie Class

The trie structure has many parts (nodes), but it needs something to keep track of where it all begins. That's where the Trie class comes in. Think of the Trie class as the main controller of the whole trie. It keeps track of the starting point or root node. Here's what it might look like in code:

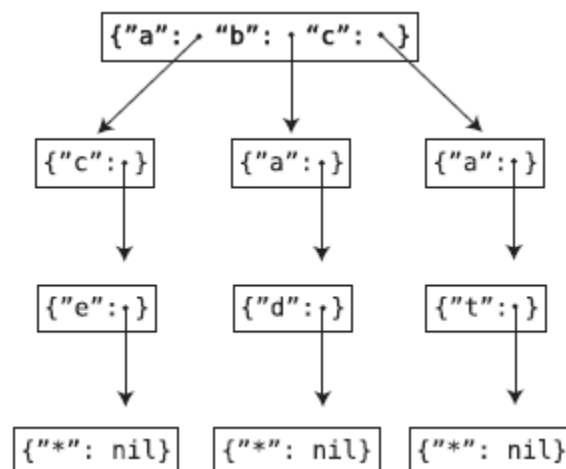
```
class Trie:  
    def __init__(self):  
        self.root = TrieNode()
```

When a new Trie is created, it starts with an empty container (TrieNode) at the root. This Trie class is like the handle to the whole tree, helping you know where it starts so you can work with everything inside.

## Storing Words

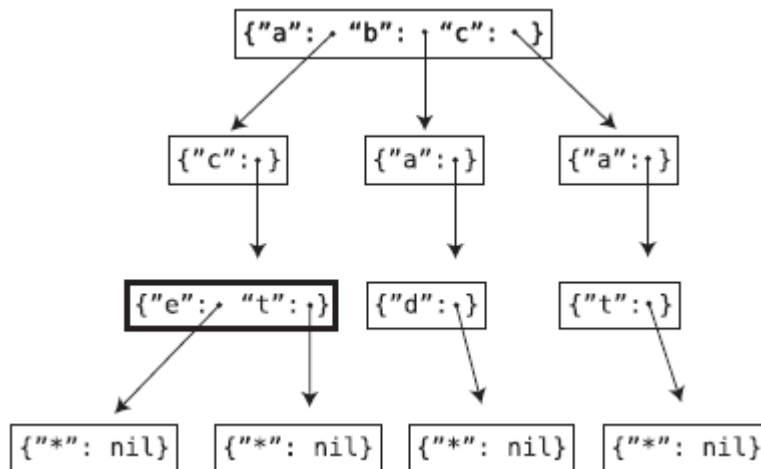
Imagine the trie is like a pathway to different words. It's storing words by having each letter of the word as a separate container (node) in the path. For example, to store the word "ace":

- You start at the root container and look at the "a" inside it.
- The "a" leads you to a next container with "c."
- The "c" leads you to a next container with "e."
- At the end of the path, there might be a special symbol like "\*" to say, "This is the end of a word."



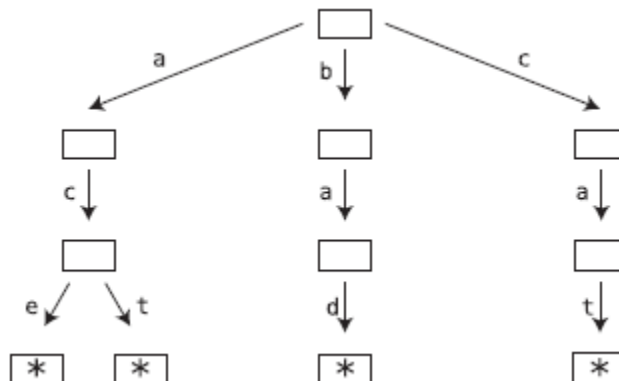
So, by connecting these containers, the trie has created a path spelling out "ace." Now, what if you want to add a word like "act" that shares some letters with "ace"?

- You can use the same "a" and "c" containers as before.
- But instead of going to "e," you add a new container with "t."



This way, the trie can efficiently store both "ace" and "act" without totally repeating the path. You can do the same for other words like "bad" and "cat." Each word creates a unique path from the root container to the end symbol, like a road map spelling out each word.

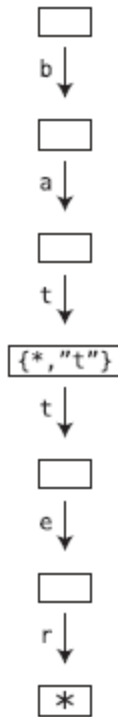
In some diagrams, you might see these paths drawn with arrows from one letter to the next. It's just a visual way to understand how the trie is linking all these letters together into words.



### The Need for the Asterisk

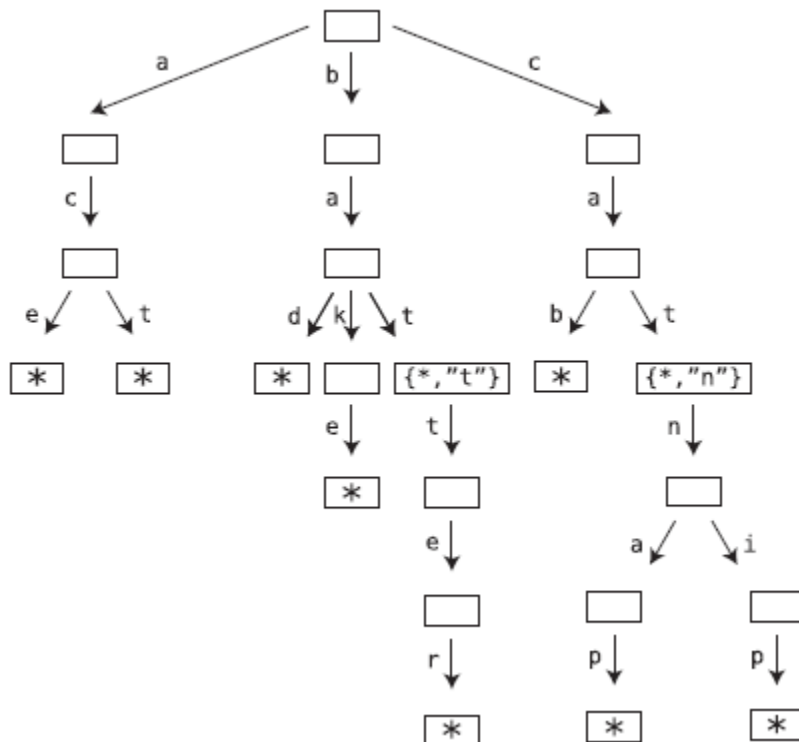
Think of the trie as a network of containers (nodes) that store parts of words. An asterisk ("\*") is used to show the end of a word. Let's look at the words "bat" and "batter" to understand why the asterisk is needed:

- "batter" includes the whole word "bat."
- If you were to follow the trie's path for "bat," you'd find an asterisk after the "t" to show that "bat" is a complete word.
- You'd also find another "t" that continues the path for "batter."



So the asterisk helps us see where one word ends, even if that word is part of a longer word. It's like a stop sign along the road that says, "This is a complete word by itself." In a diagram, you might see this with curly braces like {\*, "t"}, where the asterisk shows the end of "bat," and the "t" continues the path to "batter."

Now, imagine this network with lots of words like "ace," "act," "bad," "bake," "bat," "batter," "cab," "cat," "catnap," and "catnip." The trie connects all these words, using the asterisks to mark where each word ends.

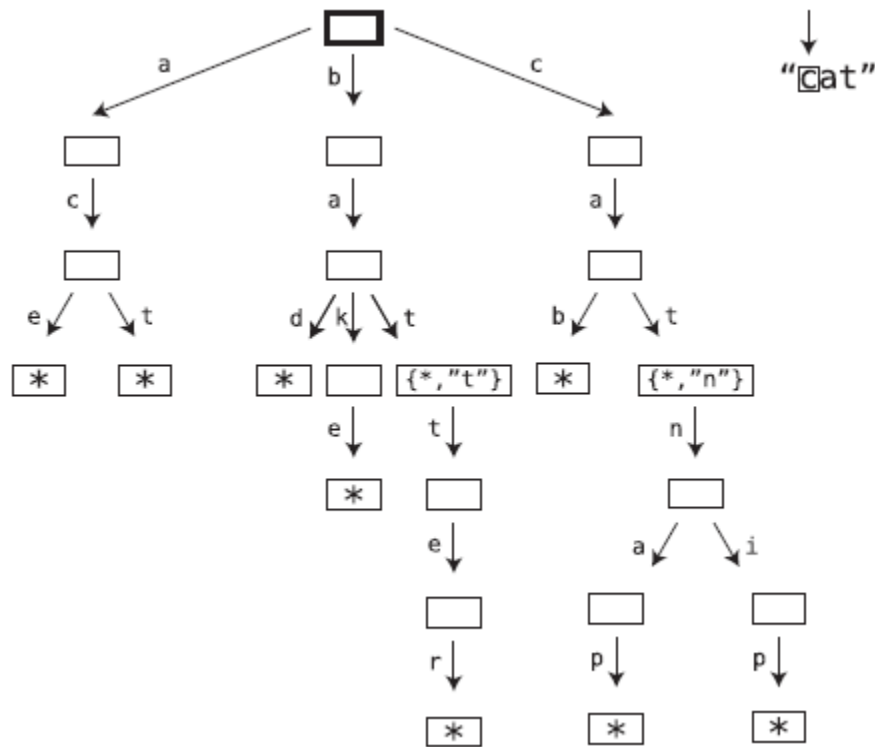


In real-life applications like your phone's autocomplete, tries might contain thousands of words. They use this clever system of paths and stop signs (the asterisks) to handle even the most common words in the language. It's a smart way to help you find and complete words quickly!

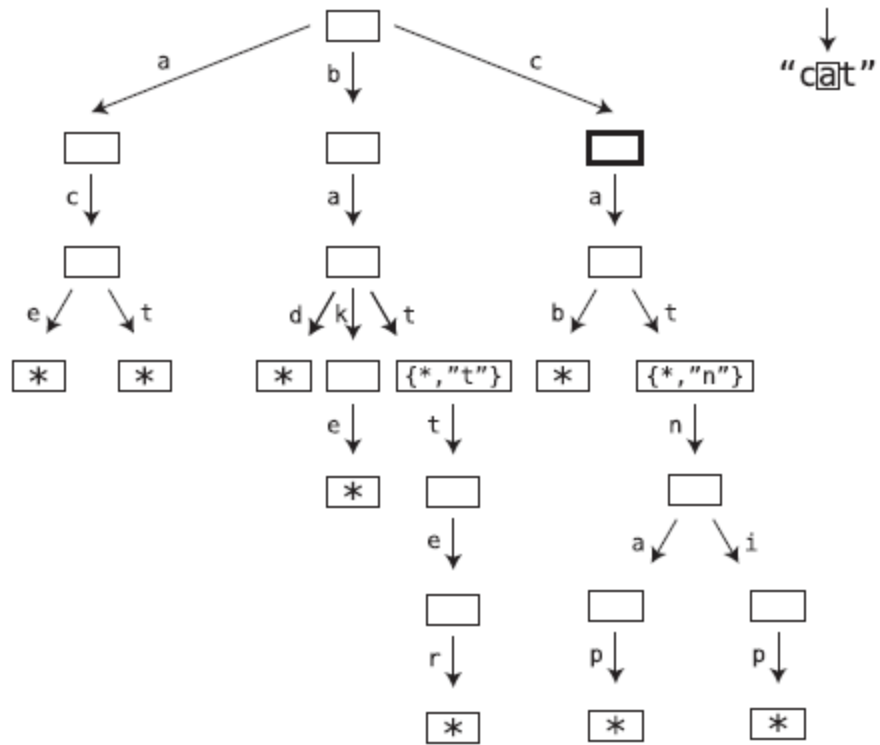
### Trie Search

Searching for a word or a prefix of a word in a trie is like following a treasure map. Here's how you can do it using the example of searching for the string "cat":

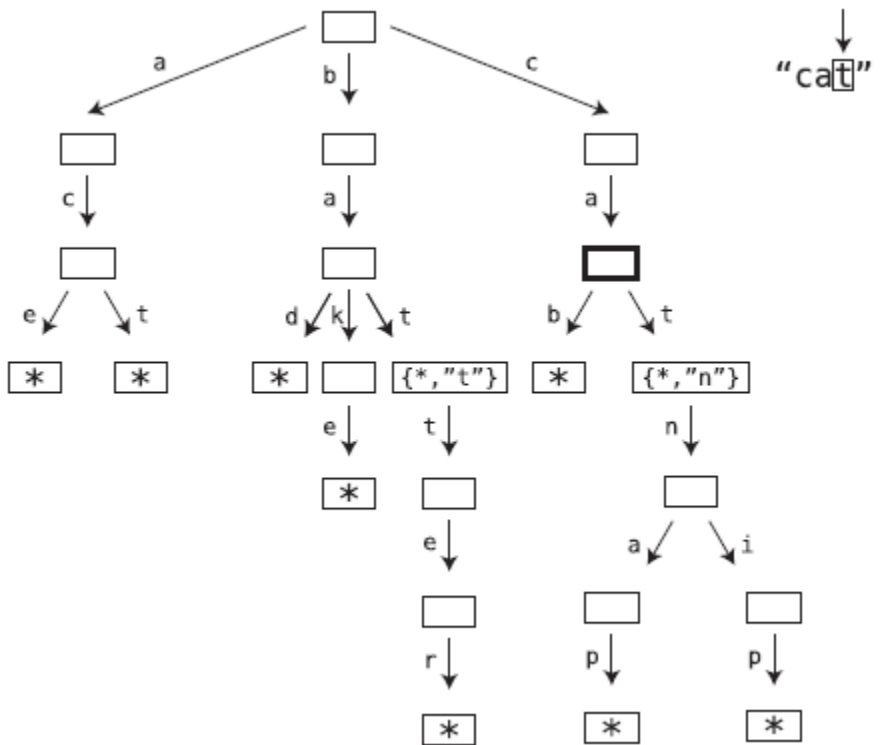
1. **Start at the Beginning:** Imagine you're at the starting point (the root node) of the trie, and you have the word "cat" to find.
2. **Follow the Path:** Look at the first letter, "c." Does the starting point have a path (or a child node) for "c"? If not, "cat" is not there. If yes, follow that path.



3. **Repeat with the Next Letter:** Now, you're at the new spot for "c." Look for the next letter, "a." Is there a path for "a"? If not, "cat" is not there. If yes, follow that path.



4. **Repeat Again:** Now, you're at the spot for "a." Look for the final letter, "t." Is there a path for "t"? If not, "cat" is not there. If yes, follow that path.



5. **Found or Not:** If you can follow the path for each letter in "cat" without hitting a dead end, then "cat" is in the trie. If you hit a dead end at any point, "cat" is not there.

## Code Implementation: Trie Search

Here's an explanation of code that adds a search method to the Trie class, allowing you to find a word or prefix in a trie:

1. **Start with the Root Node:** The search begins at the root of the trie (**currentNode = self.root**).
2. **Loop through the Characters:** For each character in the word you're looking for (**for char in word:**), do the following:
  - a. **Check for the Character:** See if there's a child node corresponding to that character (**if currentNode.children.get(char):**).
  - b. **If Found, Follow the Path:** If the character is found, update the current node to that child node (**currentNode = currentNode.children[char]**), and continue with the next character.
  - c. **If Not Found, Return None:** If the character is not found, the search ends, and the function returns None, indicating the word is not in the trie.
3. **Success:** If you make it through all the characters without hitting a dead end, you've found the word in the trie. You return the current node (**return currentNode**).

The returned current node can then be used for additional operations, such as building an autocomplete feature. Think of it as a maze: you start at the entrance, and for each letter in the word, you follow a path. If you reach a dead end, the word isn't there. If you successfully follow the path for each letter, you've found the word.

## The Efficiency of Trie Search

The efficiency of trie search is remarkable, and here's why:

1. **Focusing on Characters:** In a trie search, you look at each character of the search string individually.
2. **Hash Table Lookup:** For each character, you use the corresponding node's hash table to find the child node. This lookup takes constant time  $O(1)$  for each character.
3. **Number of Steps Equals Number of Characters:** The total number of steps equals the number of characters in the search string. For a word like "cat," that's just three steps.
4. **Comparison to Binary Search:** Binary search takes  $O(\log N)$  time, where  $N$  is the number of words. Trie search is often faster as it only depends on the length of the search string, not the number of words.
5. **Expressing in Big O Notation:** The algorithm's efficiency is represented as  $O(K)$ , where  $K$  is the number of characters in the search string. It's not exactly constant time but behaves similarly.
6. **Why It's Efficient:** Unlike many algorithms, trie search's speed isn't affected by the size of the data structure (number of nodes in the trie). It only depends on the size of the input (the search string). Even if the trie grows significantly, the search time for a three-character word will always be three steps.

The trie search is exceptionally efficient because it takes a constant amount of time for each character in the search string, and its performance isn't slowed down by the size of the trie.



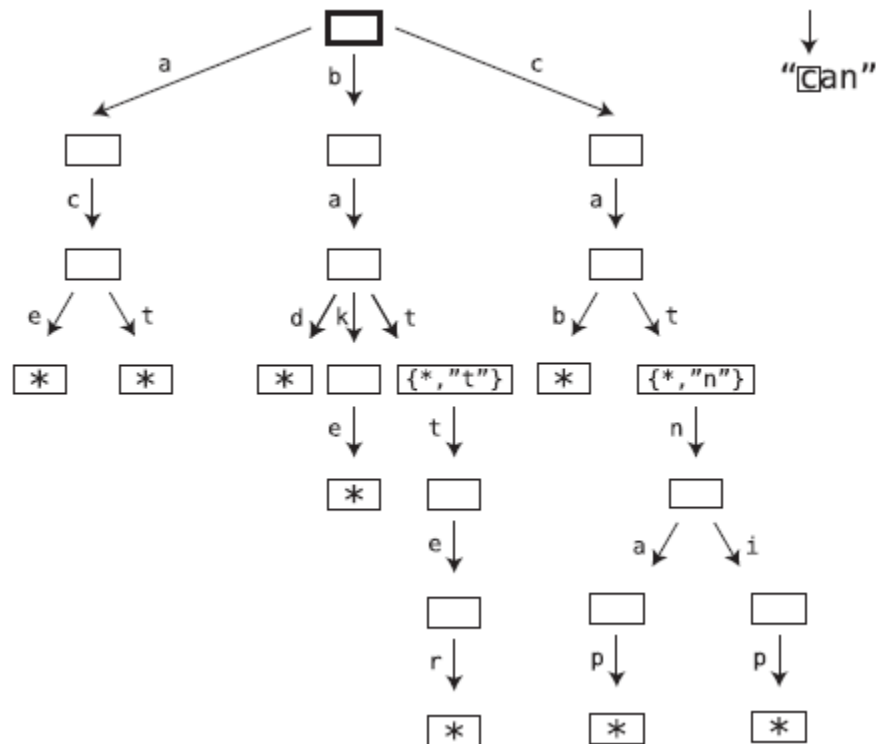
## Trie Insertion

Inserting a new word into a trie involves the following simplified steps:

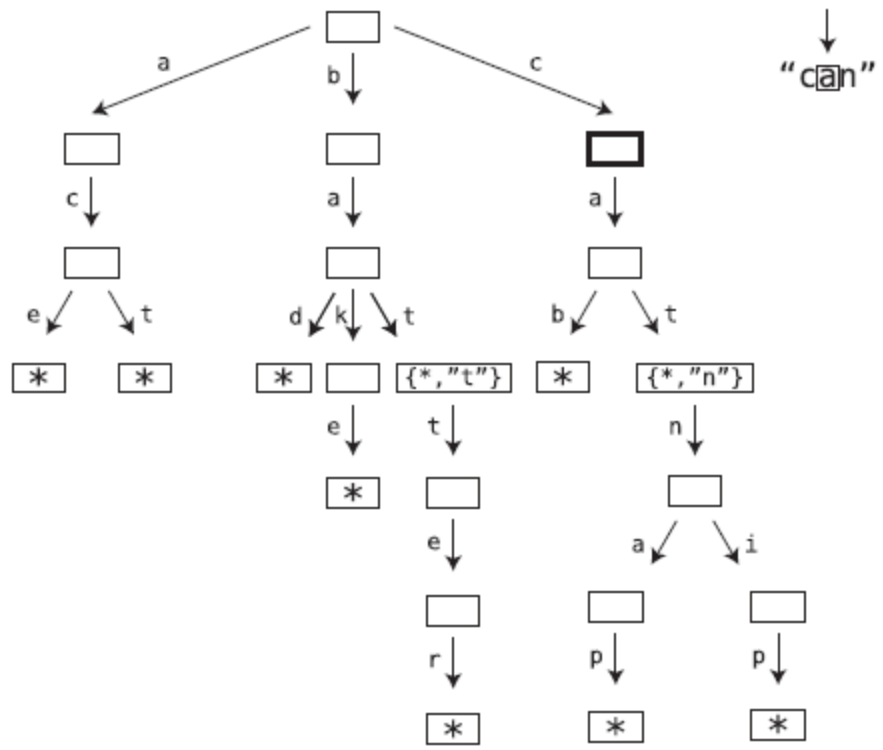
1. **Start at the Root Node:** A variable called **currentNode** is initialized to the root of the trie.
2. **Iterate Through the Word's Characters:** For each character in the word: a. If **currentNode** has a child with the character as a key, move to that child node. b. If no such child exists, create a new child node with the character, and move to it.
3. **Mark the Word as Complete:** After inserting the final character, add a "\*" child to the last node to signify that the word is complete.

Here's an example of how the word "can" is inserted:

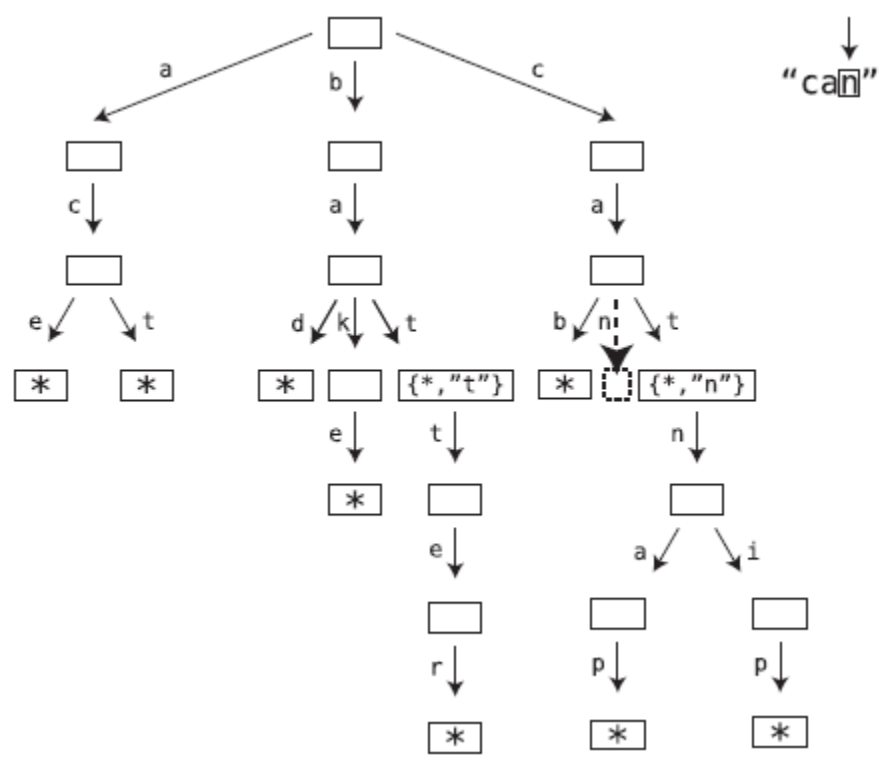
- Start at the root and find the "c" child. Move to that node.



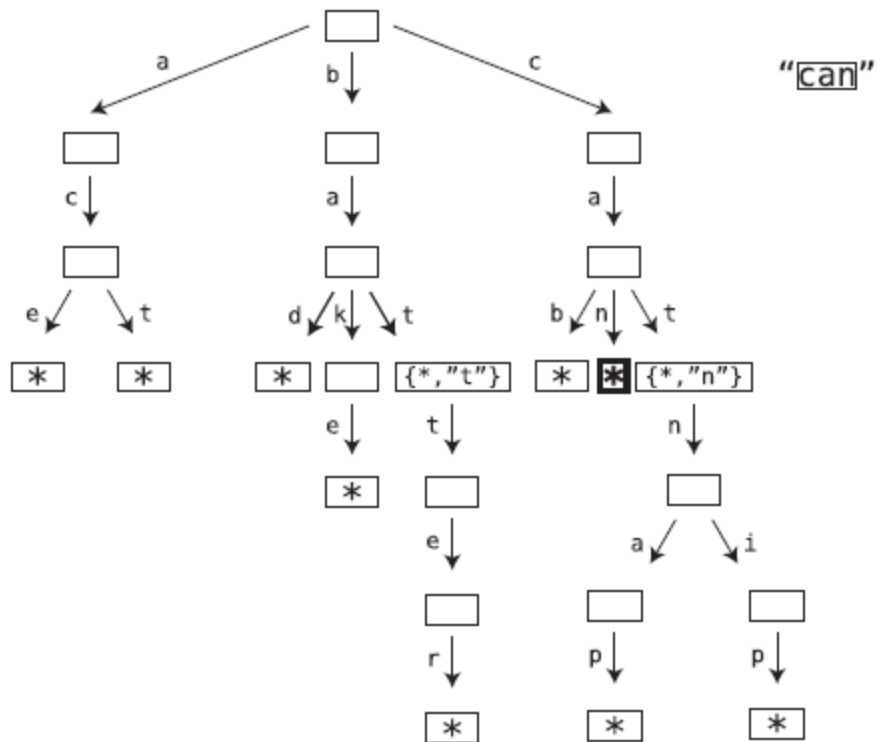
- In the next node, find the "a" child and move to that node.



- The next node doesn't have an "n" child, so create one and move to it.



- Add a "\*" child to the final node to complete the insertion.



By following these steps, you can insert any word into a trie. This process is quite efficient, closely resembling the trie search process, but with the added step of creating new nodes as needed.

### Code Implementation: Trie Insertion

The code implementation for inserting a word into a trie is relatively straightforward and can be done in  $O(K)$  time complexity, where  $K$  is the number of characters in the word:

1. **Initialize Current Node:** Start from the root of the trie and set it as the **currentNode**.
2. **Iterate Through the Word's Characters:** For each character in the word:
  - a. If a child node with the current character exists, move to that child node.
  - b. If no such child exists, create a new node with that character as a key and move to the newly created node.
3. **Mark the Word as Complete:** After inserting the entire word, add a "\*" key to the final node's children with a value of None.

### Building Autocomplete

We're just about ready to build a real autocomplete feature. To make this a tad easier, let's first build a slightly simpler function that we'll use to help us with this feature.

## Collecting All the Words

The `collectAllWords` method in the `Trie` class is designed to retrieve all the words in the trie. It can also start from a specific node, allowing us to collect words that start from that particular node. Here's a simplified explanation of how the method works:

1. **Setting the Current Node:** If no node is provided as an argument, the method starts from the root node of the trie. Otherwise, it starts from the specified node.
2. **Iterating through Children:** The method iterates through all the children of the current node.
3. **Base Case - Completing a Word:** If a "\*" key is encountered, the method recognizes that a complete word has been found and adds it to the list of words.
4. **Recursive Call - Building the Word:** If a regular character is found, the method recursively calls itself with the child node, adding the character to the building word and passing the words array.
5. **Returning the Words:** Finally, the method returns the collected words.

In code terms, the process looks like this:

```
def collectAllWords(self, node=None, word="", words=[]):
    # This method accepts three arguments. The first is the
    # node whose descendants we're collecting words from.
    # The second argument, word, begins as an empty string,
    # and we add characters to it as we move through the trie.
    # The third argument, words, begins as an empty array,
    # and by the end of the function will contain all the words
    # from the trie.

    # The current node is the node passed in as the first parameter,
    # or the root node if none is provided:
    currentNode = node or self.root

    # We iterate through all the current node's children:
    for key, childNode in currentNode.children.items():
        # If the current key is *, it means we hit the end of a
        # complete word, so we can add it to our words array:
        if key == "*":
            words.append(word)
        else: # If we're still in the middle of a word:
            # We recursively call this function on the child node.
            self.collectAllWords(childNode, word + key, words)

    return words
```

Here's a breakdown of what happens:

- **Initialize:** Start with the root node or the specified node and iterate through its children.
- **Build the Word:** If the child is not a "\*", recursively call the function with the child node and append the character to the building word.
- **Complete the Word:** If the child is a "\*", append the current word to the words list.
- **Return:** Once all children are processed, return the words list containing all the found words.

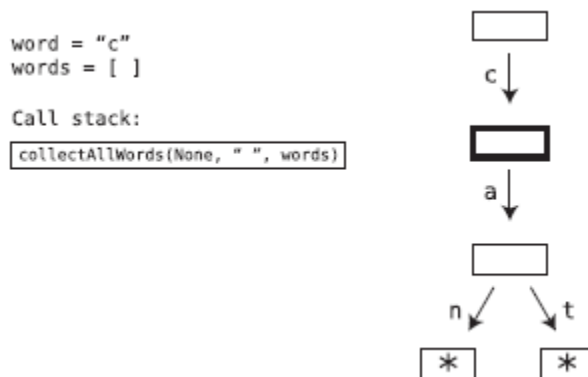
This approach leverages recursion to traverse the trie and collect the words, building them character by character. If no specific node is provided, the method will return the entire list of words in the trie. If a node is provided, it will return all words that start from that node.

### Recursion Walk-Through

Let's break down how the recursion works in the **collectAllWords** method using a visual walk-through with a simple trie containing the words "can" and "cat":



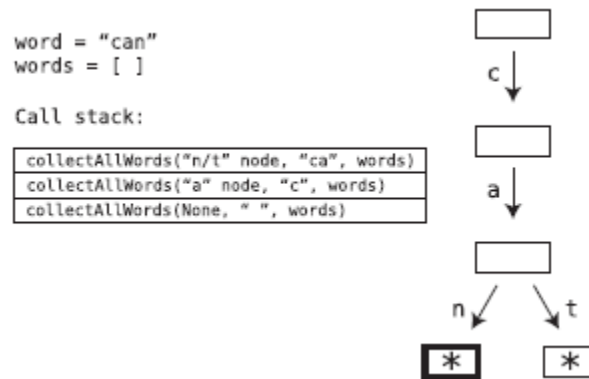
1. **Call 1:** Start at the root. The current word is empty, and no words have been collected yet. The root node has one child key "c", so we call **collectAllWords** on the child node corresponding to "c", with the word now as "c".



2. **Call 2:** In this call, we are at the "c" node. The current node has one child key "a", so we call **collectAllWords** on the "a" child node, with the word now as "ca".



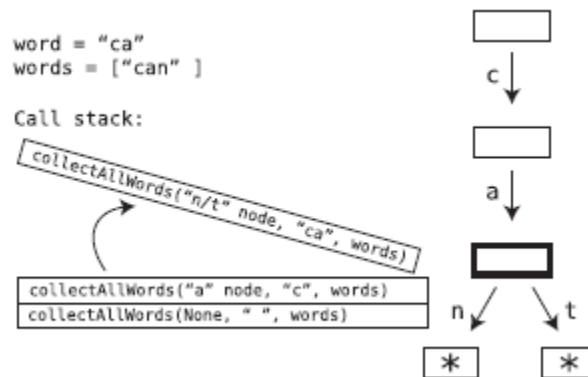
3. **Call 3:** Now at the node with children keys "n" and "t", we start with "n". We call `collectAllWords` on the "n" child, with the word "can".



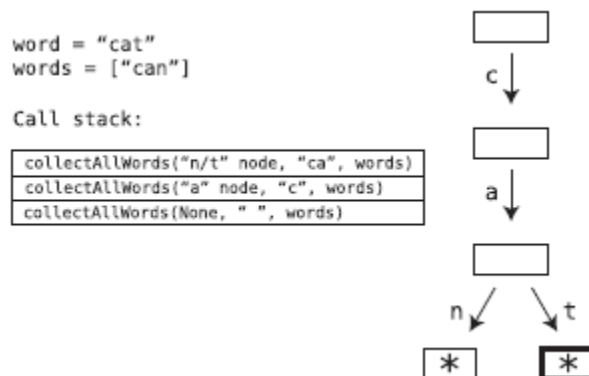
4. **Call 4:** At the node corresponding to "n", the only child key is "\*", so we know the word is complete. We add "can" to the words array.

`words = ["can"]`

5. **Call 5:** Returning to the previous call at the "n/t" node, we continue to the "t" key. The word is back to "ca", but the words array still contains "can" (as arrays remain the same object in memory).



6. **Call 6:** We call `collectAllWords` on the "t" child, with the word "cat".



7. **Call 7:** At the node corresponding to "t", the only child key is "\*", so we add "cat" to our words array. We return from each call, finally getting back to the original call, with the words array containing both "can" and "cat".

`words = ["can", "cat"]`

This walk-through shows how the function navigates through the trie, exploring each branch through recursive calls, and building the words as it goes. The use of recursion, combined with the manipulation of the word string and words array, allows the function to build and collect all the words in the trie. The key points here are the recursive traversal of the trie's structure, the building of words character by character, and the collection of complete words in the array, which is then returned.

### Completing Autocomplete

We want to create an autocomplete feature, where a user starts typing a word, and the system suggests possible completions for that word. To do this, we're utilizing a trie which has already been set up to store all the words we might want to suggest. Here's how the basic autocomplete method works:

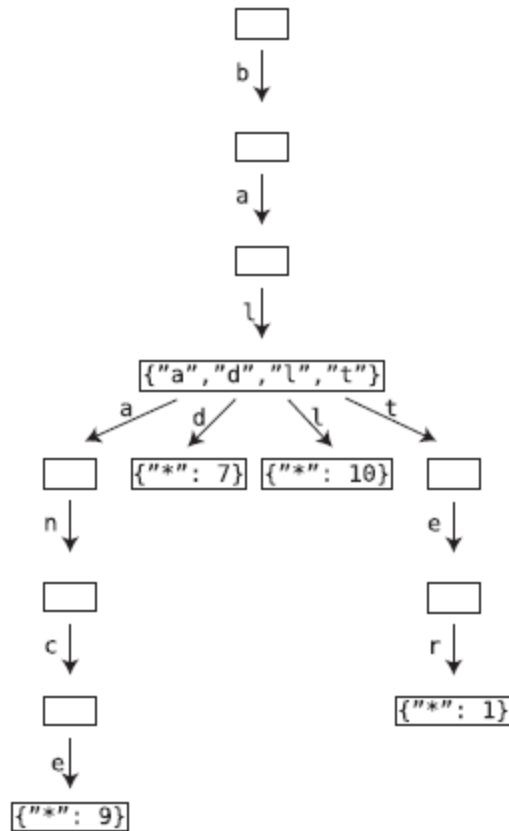
1. **Start with a Prefix:** The user types in the beginning of a word (e.g., "ca"), which is called the prefix. This prefix is the input to the autocomplete method.
2. **Search for the Prefix:** The method searches the trie to see if that prefix exists within the stored words. If not found, it returns None, meaning there are no possible completions.
3. **Find the Final Node of the Prefix:** If the prefix is found, the search method returns the last node in the trie that corresponds to the last character of the prefix.
4. **Collect All Words from That Node:** The method then calls **collectAllWords** on that final node of the prefix. This step explores all possible paths from that node, collecting all complete words that can be made by extending the prefix.
5. **Return the Possible Completions:** The method returns an array of all the words found, representing all the possible ways the user's prefix could be completed. These can be shown as suggestions to the user.

This is achieved by leveraging previously defined methods like **search** and **collectAllWords**, and putting them together to create the autocomplete functionality. It's a neat way of providing useful suggestions to users based on the prefix they've entered. It could be used, for example, in a search engine to provide search suggestions or in a text editor to assist with typing.

### Tries with Values: A Better Autocomplete

Now let's enhance the autocomplete feature by taking into account the popularity of words, instead of just suggesting every possible word that matches the prefix:

1. **Storing Popularity Scores:** In addition to keeping track of words, the trie now also stores a popularity score for each word, ranging from 1 to 10 (with 10 being the most popular).
2. **Modifying the Trie Structure:** Previously, the trie used a special symbol ("\*") to indicate the end of a word, with no additional information. Now, the value associated with that symbol will hold the popularity score. For example, "ball" might have a score of 10, while the more obscure word "balter" might have a score of 1.



3. **User Types a Prefix:** When a user starts typing a word (e.g., "bal"), the system will look for completions like "ball," "bald," "balance," and "balter."
4. **Collecting and Sorting Options:** The system will collect all matching words and their popularity scores, then sort them by popularity.
5. **Displaying the Most Popular Options:** Instead of showing all possible completions, the system will display only the most popular ones, such as the top 3 or 5. This makes the suggestions more targeted and less overwhelming for the user.
6. **Benefit:** This method makes autocomplete smarter by considering what the user is most likely trying to type. It's not just about matching the letters; it's about understanding common usage and presenting the most relevant options.

### Wrapping Up

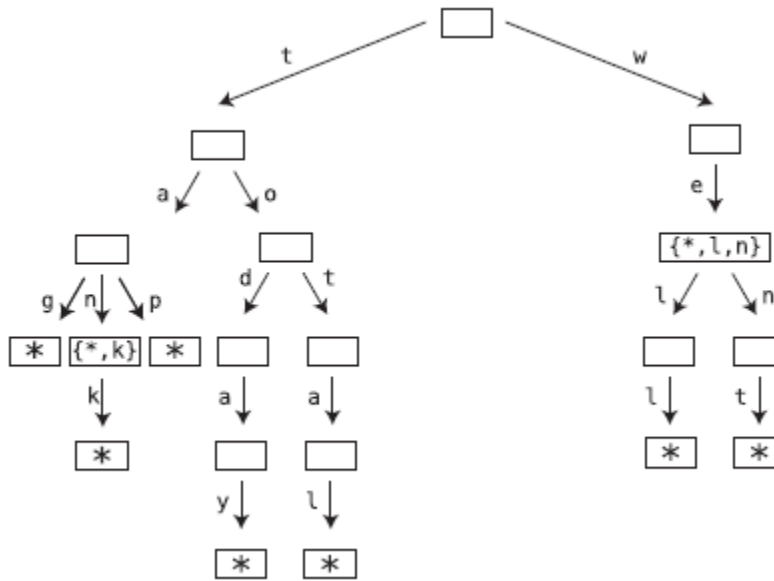
We've now covered three types of trees: binary search trees, heaps, and tries. There are many other types of trees as well, such as AVL trees, Red-Black trees, 2-3-4 trees, and plenty of others. Each tree has unique traits and behaviors that can be leveraged for specific situations. I encourage you to learn more about these various trees, but in any case, you now have a taste for how different trees can solve different problems. It's now time for the final data structure of the book. Everything you've learned about trees will help you understand graphs. Graphs are helpful in so many different situations, and that's why they're so popular. So, let's dive in.

### Exercises

The following exercises provide you with the opportunity to practice with tries.

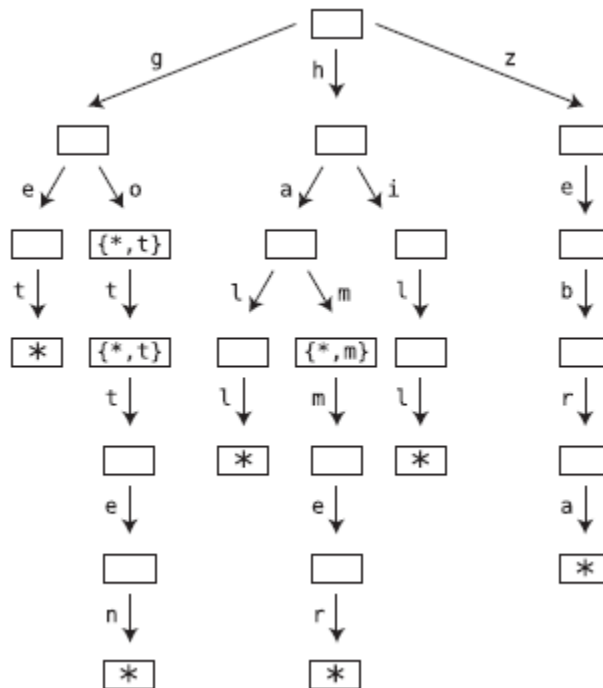


1. List all the words stored in the following trie:



tag, tan, tank, tap, today, total, we, well, went

2. Draw a trie that stores the following words: "get," "go," "got," "gotten," "hall," "ham," "hammer," "hill," and "zebra."



3. Write a function that traverses each node of a trie and prints each key, including all "\*" keys.

```
def print_trie(node):
    for key in node:
```

```

print(key)
if key != '*' and isinstance(node[key], dict):
    print_trie(node[key])

```

4. Write an *autocorrect* function that attempts to replace a user's typo with a correct word. The function should accept a string that represents text a user typed in. If the user's string is *not* in the trie, the function should return an alternative word that shares the longest possible prefix with the user's string.

For example, let's say our trie contained the words "cat," "catnap," and "catnip." If the user accidentally types in "catnar," our function should return "catnap," since that's the word from the trie that shares the longest prefix with "catnar." This is because both "catnar" and "catnap" share a prefix of "catna," which is five characters long. The word "catnip" isn't as good, since it only shares the shorter, four-character prefix of "catn" with "catnar."

One more example: if the user types in "caxasfdij," the function could return any of the words "cat," "catnap," and "catnip," since they all share the same prefix of "ca" with the user's typo.

If the user's string is found in the trie, the function should just return the word itself. This should be true even if the user's text is not a complete word, as we're only trying to correct typos, not suggest endings to the user's prefix.

```

def autocorrect(user_text, trie):
    current_node = trie
    last_matching_node = None
    last_matching_prefix = ""
    for char in user_text:
        if char in current_node:
            last_matching_node = current_node[char]
            last_matching_prefix += char
            current_node = current_node[char]
        else:
            break

    if last_matching_node and '*' in last_matching_node:
        return last_matching_prefix

    # Traverse the last matching node to find the word with the longest shared
    # prefix
    def find_word(node, prefix):
        if '*' in node:

```

```
    return prefix
for key in node:
    if key != '*':
        word = find_word(node[key], prefix + key)
        if word:
            return word

return find_word(last_matching_node, last_matching_prefix) if last_matching_node
else None
```

## CHAPTER 18: Connecting Everything with Graphs

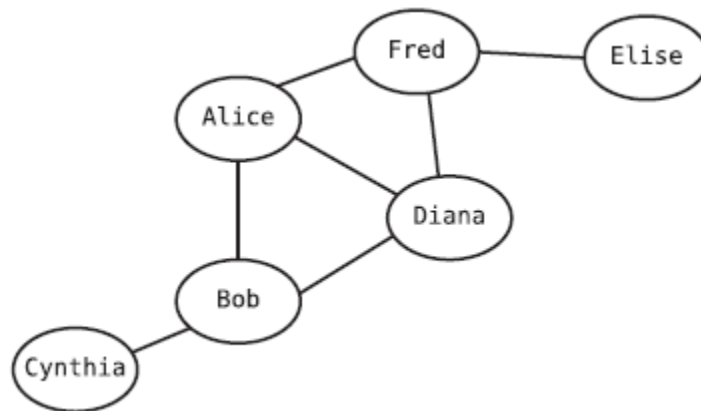
Imagine you're making a social network where people can be friends with each other. If Alice is friends with Bob, then Bob is also friends with Alice. You might first try to organize this information with a list of friendships like this:

```
friendships = [  
    ["Alice", "Bob"],  
    ["Bob", "Cynthia"],  
    ["Alice", "Diana"],  
    ["Bob", "Diana"],  
    ["Elise", "Fred"],  
    ["Diana", "Fred"],  
    ["Fred", "Alice"]  
]
```

Each pair of names in this list is a friendship. But this method has a problem. If you want to find out who Alice's friends are, the computer has to look through the entire list. This can take a long time, especially if there are many friendships. A better way to do this is by using something called a graph. A graph can show the friendships in a way that lets the computer find Alice's friends quickly and easily. It's much faster than the first method!

### Graphs

A *graph* is a data structure that specializes in relationships, as it easily conveys how data is connected. Here is a visualization of our social network, displayed as a graph:



Each person is represented by a node, and each line indicates a friendship with another person. If you look at Alice, for example, you can see that she is friends with Bob, Diana, and Fred, since her node has lines that connect to their nodes.

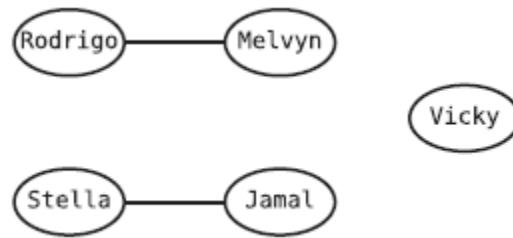
### Graphs vs. Trees

Graphs and trees are similar, but they have some key differences. Both consist of nodes connected to each other, but there are rules that distinguish them.

1. **Cycles:** Graphs might have cycles, meaning that you can start at one node and follow the connections to eventually get back to the same node. Trees can't have cycles. If there's a cycle, it's not a tree.

2. **Connections Between Nodes:** In a tree, every node must be connected to every other node, even indirectly. In a graph, this isn't a requirement. You might have some nodes that aren't connected to anything else.

Here's an example to clarify:



Imagine a social network where you have two groups of friends, and nobody from one group is friends with anyone from the other group. You could also have someone who just joined and doesn't have any friends yet.

- This would be a graph because there are nodes that aren't connected to everything else, and you could have cycles.
- But it wouldn't be a tree, since in a tree, every node must be connected to every other one, and there can't be any cycles.

## Graph Jargon

Let's break down the graph terminology:

- **Node:** In the world of graphs, what we usually call a "node" is referred to as a "vertex."
- **Lines Between Nodes:** The connections or lines between these vertices are called "edges."
- **Adjacent or Neighbors:** When two vertices are connected by an edge, they are said to be "adjacent" to each other or "neighbors."
- **Connected Graph:** If all the vertices in the graph are connected in some way, it's referred to as a "connected graph."

If you have a graph where "Alice" and "Bob" are connected, you'd say that the vertices of "Alice" and "Bob" are adjacent, since they share an edge. If every vertex is connected to at least one other, the graph is connected.

## The Bare-Bones Graph Implementation

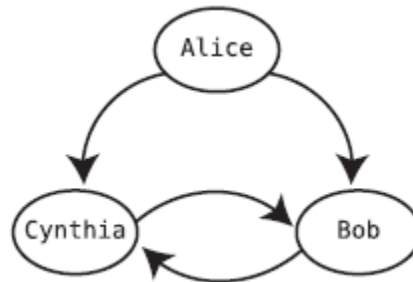
We can create a basic representation of a social network using a programming tool called a hash table. Here's an example using Ruby:

```
friends = {  
  "Alice" => ["Bob", "Diana", "Fred"],  
  "Bob" => ["Alice", "Cynthia", "Diana"],  
  "Cynthia" => ["Bob"],  
  "Diana" => ["Alice", "Bob", "Fred"],  
  "Elise" => ["Fred"],  
  "Fred" => ["Alice", "Diana", "Elise"]  
}
```

In this example, the names of people are the "keys," and their lists of friends are the "values." So "Alice" is friends with "Bob," "Diana," and "Fred." The great thing about this method is that it's really fast to find someone's friends. For example, if you want to find all of Alice's friends, you just use `friends["Alice"]`, and it instantly gives you the list of her friends. It's just one step, so it's very quick!

## Directed Graphs

In some social networks, relationships are not mutual. For example, a social network may allow Alice to "follow" Bob, but Bob doesn't have to follow Alice back. Let's construct a new graph that demonstrates who follows whom:



This is known as a **directed graph**. In this example, the arrows indicate the *direction* of the relationship. Alice follows both Bob and Cynthia, but no one follows Alice. We can also see that Bob and Cynthia follow each other. We can still use our simple hash-table implementation to store this data:

```
followees = {
  "Alice" => ["Bob", "Cynthia"],
  "Bob" => ["Cynthia"],
  "Cynthia" => ["Bob"]
}
```

The only difference here is that we are using the arrays to represent the people each person *follows*.

## Object-Oriented Graph Implementation

You can represent a graph using classes in Ruby:

### Vertex Class

```
class Vertex
  attr_accessor :value, :adjacent_vertices

  def initialize(value)
    @value = value
    @adjacent_vertices = []
  end

  def add_adjacent_vertex(vertex)
    @adjacent_vertices << vertex
  end
end
```

- A "vertex" class represents a point or node in the graph.

- Each vertex has a "value" (like a person's name) and a list of "adjacent vertices" (or connections to other vertices).
- You can add connections to other vertices using the **add\_adjacent\_vertex** method.

### Example Code

```
alice = Vertex.new("alice")
bob = Vertex.new("bob")
cynthia = Vertex.new("cynthia")

alice.add_adjacent_vertex(bob)
alice.add_adjacent_vertex(cynthia)
bob.add_adjacent_vertex(cynthia)
cynthia.add_adjacent_vertex(bob)
```

- You can create a new vertex by calling **Vertex.new("name")**.
- Then you can connect vertices by calling **add\_adjacent\_vertex** method on one and passing the other as an argument.
- The method makes sure not to enter an infinite loop by checking if a vertex is already in the list of adjacent vertices.

### Building a Social Network

- If you're building a social network, a vertex represents a person.
- You can create connections to represent friendships, and make sure that friendships are mutual (if Alice is friends with Bob, Bob is also friends with Alice).

### Types of Graphs

- **Directed Graph:** Connections go one way (e.g., Alice follows Bob, but Bob doesn't follow Alice).
- **Undirected Graph:** Connections are mutual (e.g., if Alice is friends with Bob, Bob is also friends with Alice).

### Types of Implementations

- **Adjacency List:** This approach uses a simple list (or array) to store a vertex's connections. The example above uses this method.
- **Adjacency Matrix:** This alternative uses two-dimensional arrays to represent connections. It can be useful in certain situations, but the example sticks to the adjacency list because it's more intuitive.

### Notes on Connectivity

- A "connected" graph means all vertices are connected to each other in some way. You can access all vertices from any vertex.
- A "disconnected" graph might have isolated vertices. In such cases, you might need additional data structures like an array to access all vertices.

## Graph Search

### What is Graph Search?

Graph search is a process to find a specific vertex (or point) within a graph. It helps us understand how two vertices are connected and what paths exist between them.

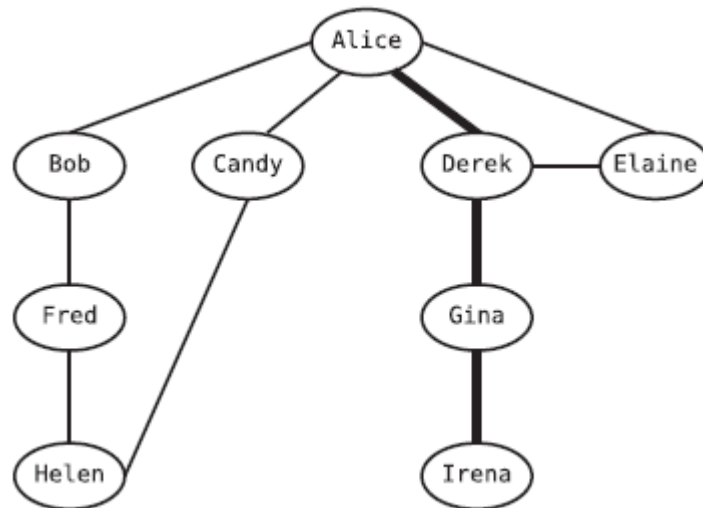
### Types of Search:

1. **Finding a Specific Vertex:** Like searching for a value in an array, you might look for a particular vertex within the graph.
2. **Finding a Path Between Two Vertices:** More commonly, graph search refers to finding a way from one specific vertex to another. You can think of it as navigating a maze or finding a route on a map.

### Examples of Search:

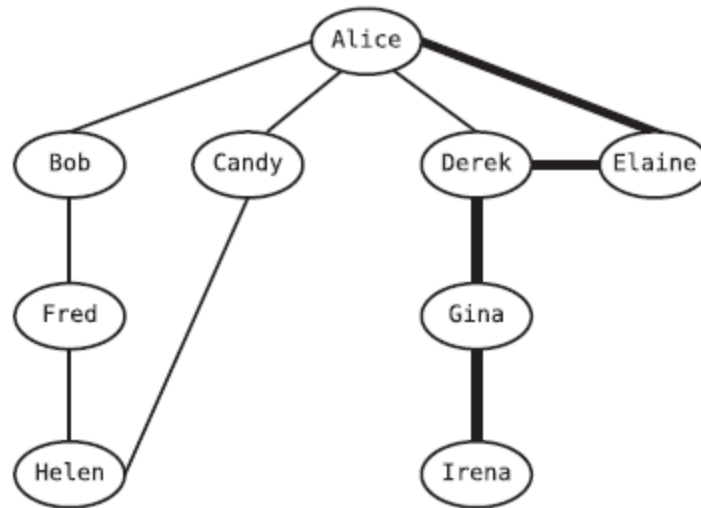
Consider a social network with different people as vertices and friendships as connections. Here's what a graph search might involve:

- **Starting Point:** You start with access to one vertex, like Alice.
- **Goal:** You're trying to find another specific vertex, like Irena.
- **Paths:** There might be multiple ways to get from Alice to Irena, like:
  - Alice -> Derek -> Gina -> Irena (Short Path)



- Alice -> Elaine -> Derek -> Gina -> Irena (Longer Path)





## Why Search a Graph?

Graph search can be useful for various purposes:

- **Finding a Vertex:** If you only have access to one random vertex, you can still search the entire graph to find any other vertex.
- **Checking Connection:** You can determine whether two vertices are connected. Like checking if Alice and Irena are friends or friends of friends.
- **Traversing the Graph:** Sometimes, you might want to go through the entire graph, applying an operation to each vertex. Like sending a notification to every person in a social network.

## Terms:

- **Vertex:** A point or node in the graph.
- **Edge:** A connection between two vertices.
- **Path:** A specific sequence of edges connecting two vertices.

Graph search is not just about finding a specific point; it's about understanding how the entire network is interconnected. It's like exploring a map, where the vertices are places, and the edges are roads, helping you discover different routes from one place to another. Whether you're looking for the shortest path or just exploring the entire network, graph search tools can guide you.

## Depth-First Search

### What is Depth-First Search?

Depth-First Search is a way to traverse or explore a graph. Think of it like exploring a maze and going as far as possible down each path before backtracking.

### How Does It Work?

The main steps of DFS are:

1. **Start:** Choose any vertex in the graph to start with.

2. **Mark as Visited:** Add this starting vertex to a list (or hash table) of visited vertices.
3. **Explore Neighbors:** Look at the vertices directly connected to the current one.
4. **Go Deeper:** If a neighboring vertex has not been visited, go to that vertex and repeat steps 2-4. This is the "depth-first" part, where you go as far as possible before moving to another branch.
5. **Avoid Cycles:** If you come across a vertex that's already been visited, ignore it. This prevents you from getting stuck in a loop.

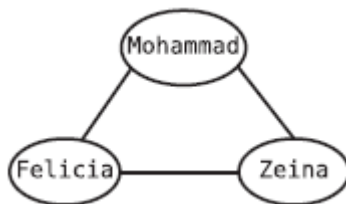
### Why Use Depth-First Search?

DFS can be used to find a particular vertex, or just to go through every vertex in the graph. It's useful for understanding how everything in the graph is connected.

### What Makes DFS Special?

- **Similar to Other Algorithms:** If you've seen binary tree traversal or filesystem traversal, DFS uses a similar approach.
- **Keeps Track of Visits:** Because graphs can have cycles (like a loop of friends who are all connected to each other), DFS must keep track of what's been visited to avoid going in circles.
- **Goes Deep:** Unlike some other approaches, DFS goes as far as possible down each path before moving on to another branch.

Imagine a social network graph where Mohammed is friends with Felicia, who is friends with Zeina, and Zeina is friends with Mohammed.

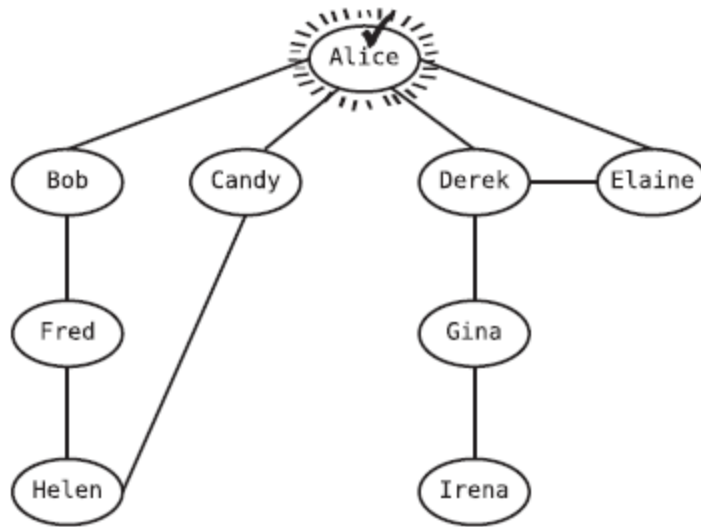


- **Without DFS:** If you didn't keep track of who you've already looked at, you could end up going in circles: Mohammed -> Felicia -> Zeina -> Mohammed -> and so on.
- **With DFS:** By marking each person as visited once you've looked at them, you can explore the whole network without getting stuck in a loop.

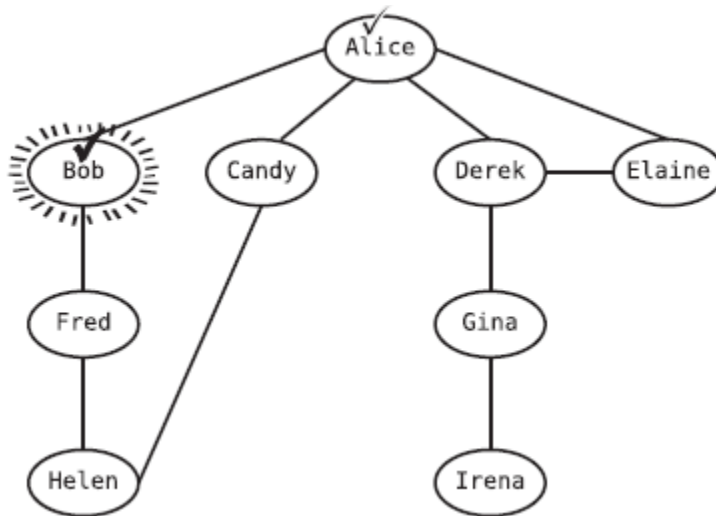
### Depth-First Search Walk-Through

#### Steps of the Walk-Through

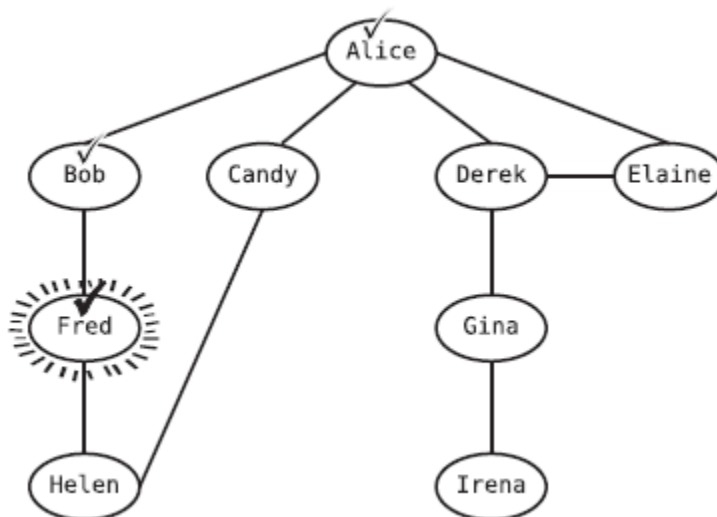
1. **Start with Alice:** We start at Alice and mark her as visited.



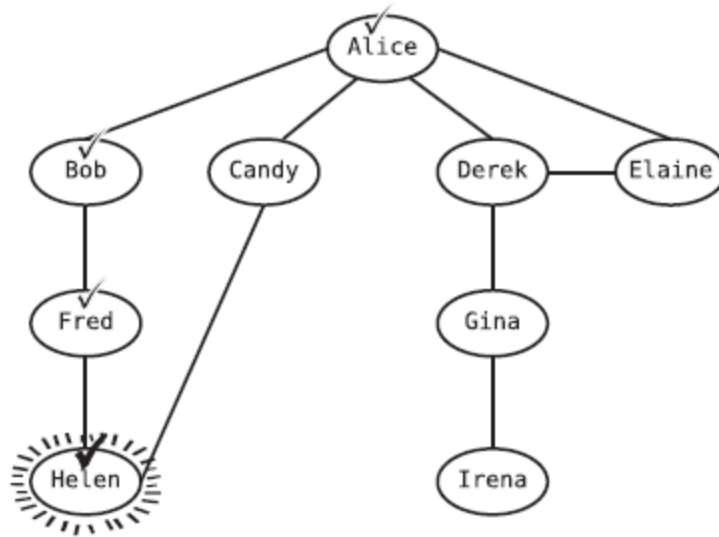
2. **Move to Bob:** We choose Bob from Alice's neighbors and mark him as visited.



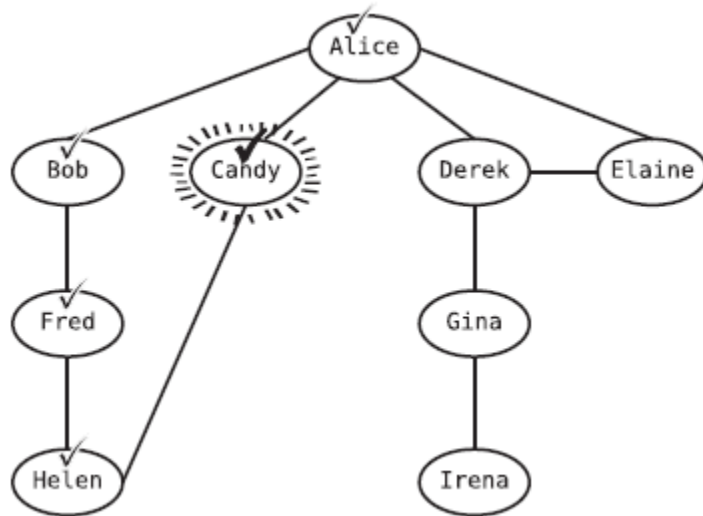
3. **Move to Fred:** We ignore Alice (already visited) and move to Bob's other neighbor, Fred. Mark Fred as visited.



4. **Move to Helen:** From Fred's neighbors, we move to Helen and mark her as visited.



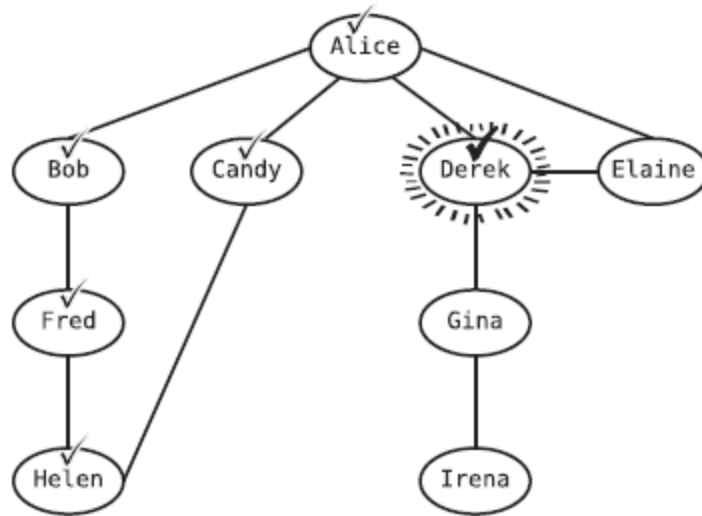
5. **Move to Candy:** From Helen's neighbors, we move to Candy, skipping Fred, and mark Candy as visited.



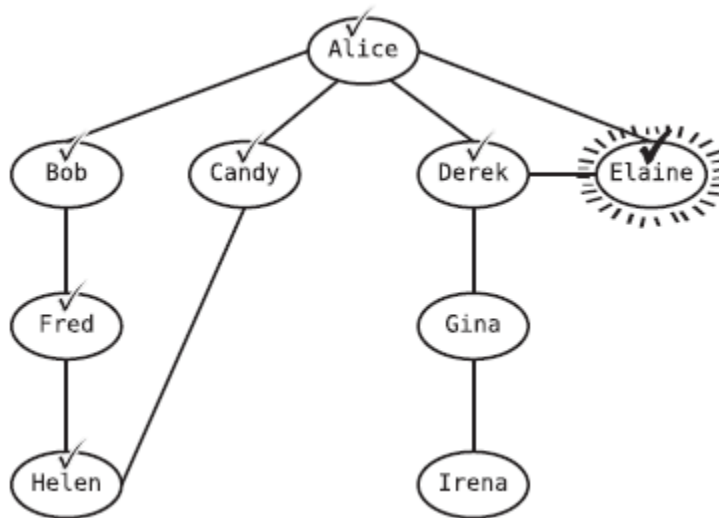
6. **Backtrack to Helen, Fred, Bob:** Since Candy has no unvisited neighbors, we backtrack, popping Helen, Fred, and Bob off the call stack.



7. **Continue with Alice's Neighbors:** Back at Alice, we continue with her unvisited neighbors, Derek, and Elaine.
8. **Move to Derek:** Perform a depth-first search on Derek, marking him as visited.

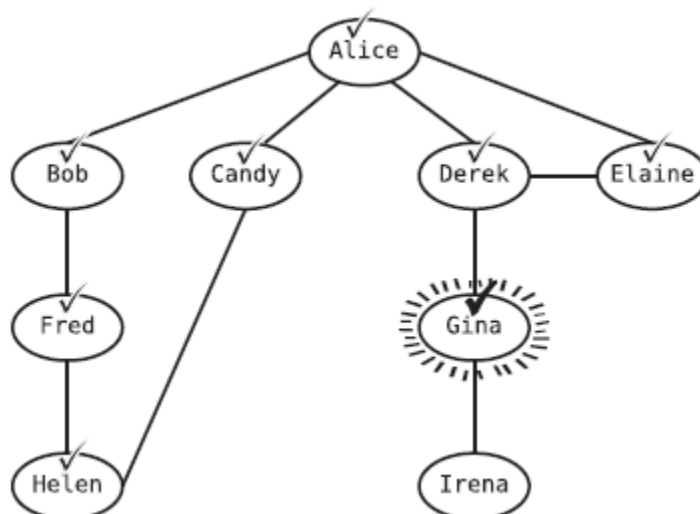


9. **Move to Elaine:** From Derek, move to Elaine and mark her as visited.



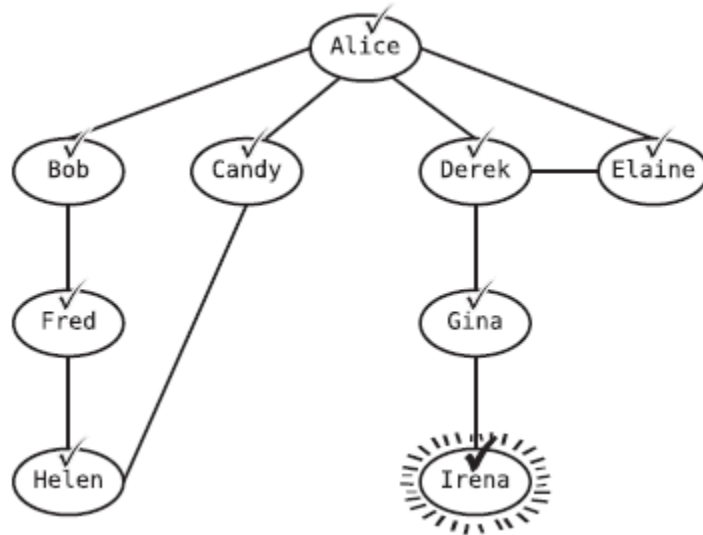
10. **Backtrack to Derek:** Since all of Elaine's neighbors are visited, backtrack to Derek.

11. **Move to Gina:** From Derek, move to his last unvisited neighbor, Gina, and mark her as visited.





12. **Move to Irena:** From Gina's neighbors, move to Irena and mark her as visited.



13. **Complete the Search:** Since Irena's only neighbor, Gina, is already visited, and no vertices remain in the call stack, the search is complete.

### What Happened?

- We started with one vertex and moved deeper into the graph, marking each new vertex as visited.
- When we reached a vertex with no unvisited neighbors, we backtracked.
- We used a "call stack" to keep track of where we were in the search, so we could return to previous vertices and continue exploring their neighbors.
- Eventually, we visited every connected vertex in the graph, and the search ended.

[Code Implementation: Depth-First Search](#)

### Depth-First Traversal Function

```

def dfs_traverse(vertex, visited_vertices={})
  # Mark vertex as visited by adding it to the hash table:
  visited_vertices[vertex.value] = true

  # Print the vertex's value, so we can make sure our traversal really works:
  puts vertex.value

  # Iterate through the current vertex's adjacent vertices:
  vertex.adjacent_vertices.each do |adjacent_vertex|

    # Ignore an adjacent vertex if we've already visited it:
    next if visited_vertices[adjacent_vertex.value]

    # Recursively call this method on the adjacent vertex:
    dfs_traverse(adjacent_vertex, visited_vertices)
  end
end

```

1. **Start with a vertex:** The function begins with a vertex and a hash table to keep track of the visited vertices.
2. **Mark the vertex as visited:** The current vertex's value is added to the hash table.
3. **Print the vertex:** The code prints the vertex's value to provide feedback.
4. **Iterate through the neighbors:** The code loops through each adjacent vertex (neighbor).
5. **Skip if visited:** If a neighbor has been visited already, the loop moves to the next neighbor.
6. **Recursively call with a neighbor:** If the neighbor has not been visited, the function calls itself with the neighbor.

## Depth-First Search Function

```

def dfs(vertex, search_value, visited_vertices={})
  # Return the original vertex if it happens to
  # be the one we're searching for:
  return vertex if vertex.value == search_value

  visited_vertices[vertex.value] = true

  vertex.adjacent_vertices.each do |adjacent_vertex|
    next if visited_vertices[adjacent_vertex.value]

    # If the adjacent vertex is the vertex we're searching for, just return
    # that vertex:
    return adjacent_vertex if adjacent_vertex.value == search_value

    # Attempt to find the vertex we're searching for by recursively calling
    # this method on the adjacent vertex:
    vertex_were_searching_for =
      dfs(adjacent_vertex, search_value, visited_vertices)

    # If we were able to find the correct vertex using the above recursion,
    # return the correct vertex:
    return vertex_were_searching_for if vertex_were_searching_for
  end

  # If we haven't found the vertex we're searching for:
  return nil
end

```

This second function builds on the first by adding a target search value. It follows the same basic steps, but with additional checks:

- If the current vertex matches the search value, return it.
- If an adjacent vertex matches the search value, return that vertex.
- If none match, the function returns nil.

## Breadth-First Search

Breadth-first search (BFS) is a method for exploring a graph, like a network of friends or connections. Rather than diving deep into one branch before moving to the next like Depth-First Search (DFS), BFS explores all the neighbors at the present depth before moving on to the neighbors of those neighbors at the next depth level. Here's how you can think of the algorithm:

### Algorithm for Breadth-First Traversal:

1. **Pick a Starting Point:** Choose any vertex in the graph as the starting point.
2. **Mark it Visited:** Record the starting vertex as visited by adding it to a hash table.
3. **Use a Queue:** Add the starting vertex to a queue. A queue is like a line at a store; the first one in is the first one out (FIFO - First In, First Out).
4. **Loop Until Queue is Empty:** As long as there are vertices in the queue, keep doing the following steps:
  - a. **Remove a Vertex from the Queue:** Take the first vertex out of the queue and call it the "current vertex."
  - b. **Look at the Neighbors:** Go through all the vertices that are adjacent to the current vertex.



- c. **Ignore Visited Neighbors:** If a neighboring vertex has already been visited, move to the next one.
- d. **Visit and Queue Unvisited Neighbors:** If a neighboring vertex hasn't been visited yet, mark it as visited and put it in the queue.

5. **Repeat Until Done:** Keep repeating the loop until the queue is empty, meaning you've visited all reachable vertices.

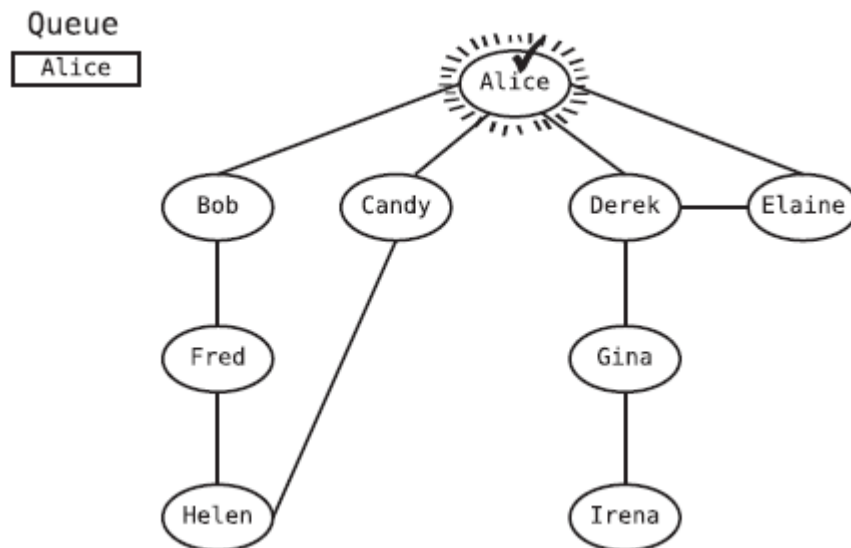
Imagine you are exploring a series of rooms connected by doors, and you want to visit every room. You start in one room, then visit each room connected to it, then each room connected to those rooms, and so on.

- You start in one room and mark it as visited.
- You make a list (queue) of rooms to visit next, starting with the rooms connected to your starting room.
- You visit the first room on your list, mark it as visited, and add its connected rooms to your list.
- You keep visiting rooms in the order they were added to the list, adding new connected rooms to the end of the list.
- You continue this process until you've visited every room on your list.

### Breadth-First Search Walk-Through

Let's break down the Breadth-First Search (BFS) traversal using a fictional social network of friends:

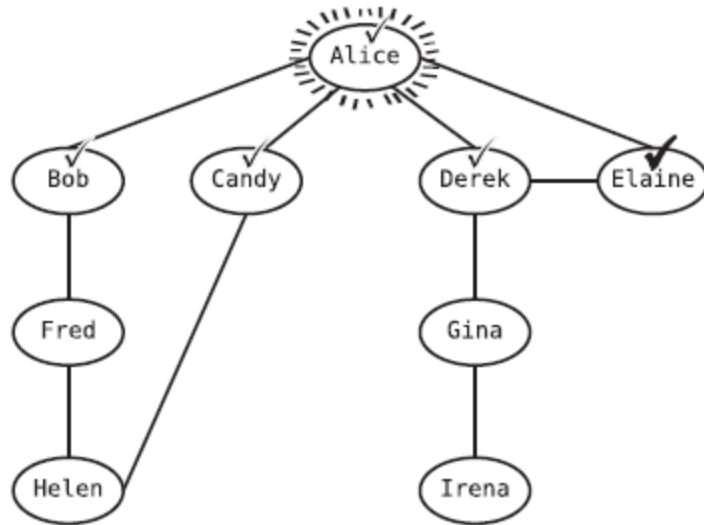
1. **Start with Alice:** Alice is our starting vertex. Mark her as visited and put her in the queue. At this point, the queue only has Alice, so it's emptied when we remove her.



2. **Look at Alice's Friends:** One by one, mark them as visited and add them to the queue: Bob, Candy, Derek, and Elaine.

### Queue

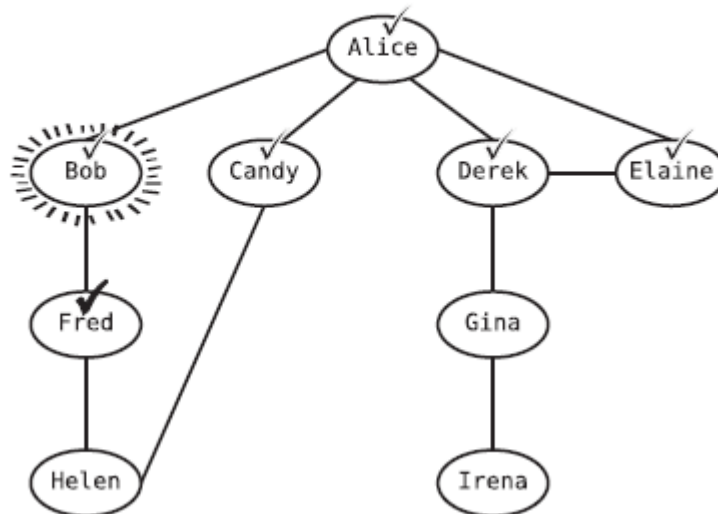
Bob	Candy	Derek	<b>Elaine</b>
-----	-------	-------	---------------



3. **Move to Bob:** Remove Bob from the queue (he's the first), and look at his friends. Ignore Alice since she's visited. Mark Fred as visited and add him to the queue.

### Queue

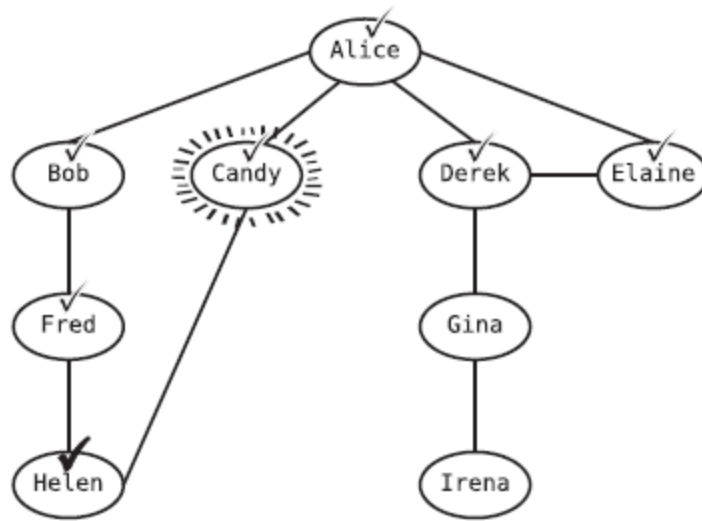
Candy	Derek	Elaine	<b>Fred</b>
-------	-------	--------	-------------



4. **Move to Candy:** Remove Candy from the queue and look at her friends. Ignore Alice and mark Helen as visited, adding her to the queue.

### Queue

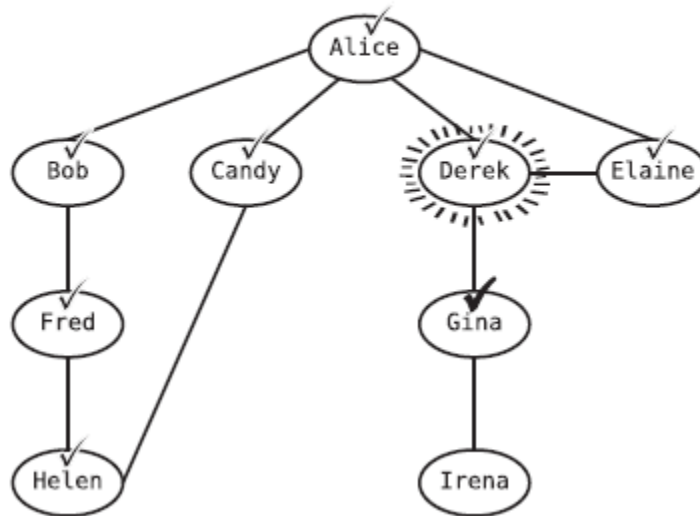
Derek	Elaine	Fred	<b>Helen</b>
-------	--------	------	--------------



5. **Move to Derek:** Remove Derek and look at his friends. Alice and Elaine are already visited, but Gina isn't. Mark Gina as visited and add her to the queue.

### Queue

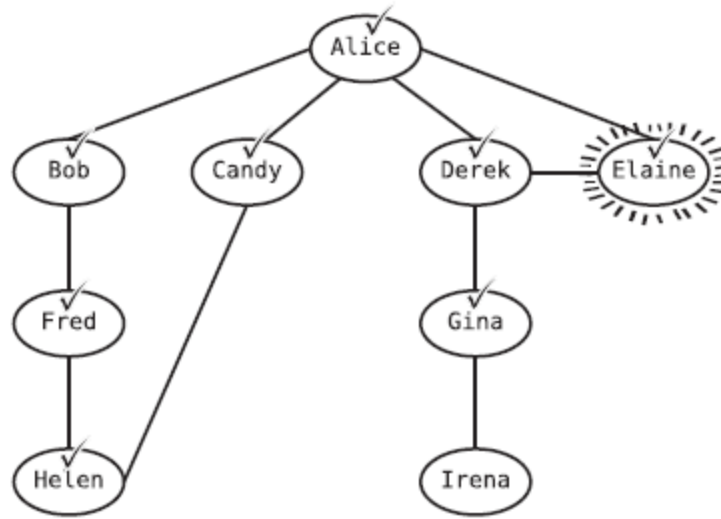
Elaine	Fred	Helen	<b>Gina</b>
--------	------	-------	-------------



6. **Move to Elaine:** Remove her from the queue, but all her adjacent vertices have been visited.

Queue

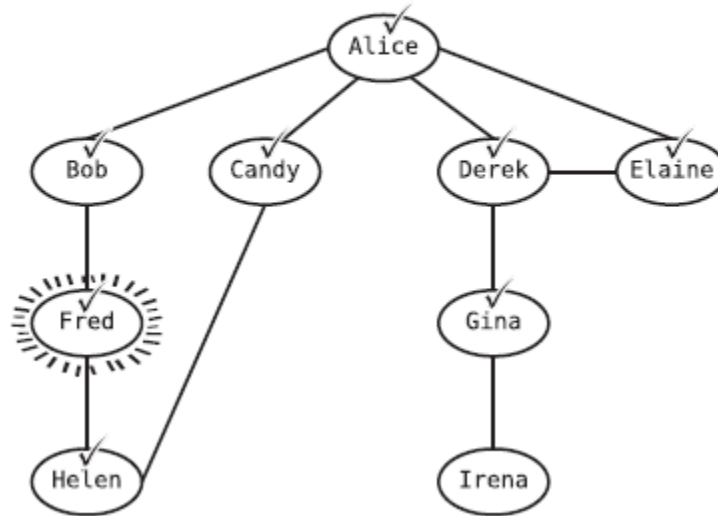
Fred	Helen	Gina
------	-------	------



7. **Move to Fred:** Remove him from the queue and look at his friends. Bob and Helen are already visited.

Queue

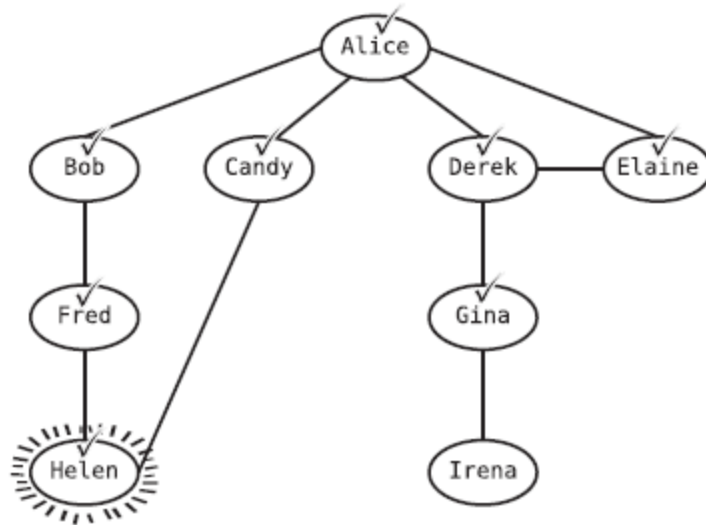
Helen	Gina
-------	------



8. **Move to Helen:** Remove her from the queue, but her friends have been visited.

Queue

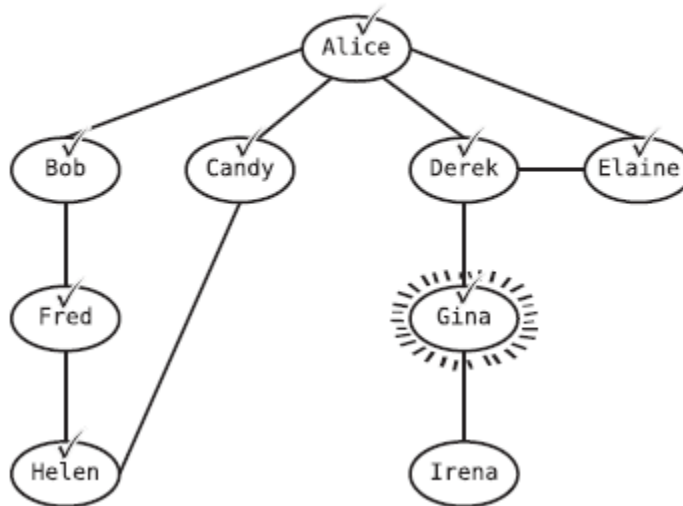
Gina



9. **Move to Gina:** Remove her from the queue and look at her friends. Derek is visited, but Irena isn't. Mark Irena as visited and add her to the queue.

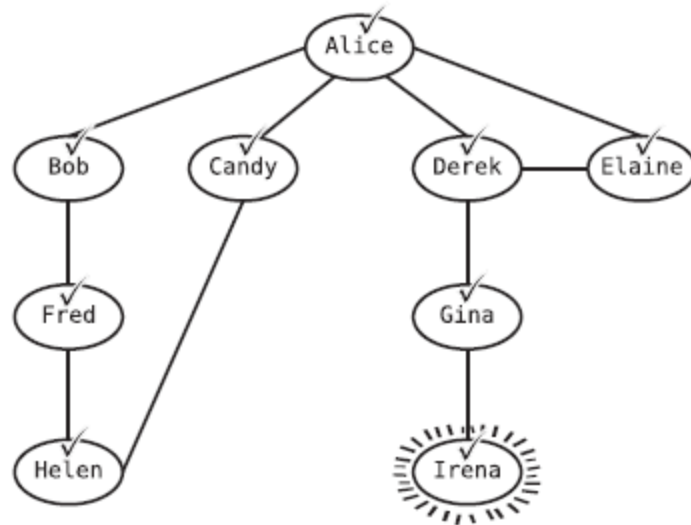
Queue

(Empty)



10. **Move to Irena:** Remove her from the queue and look at her only friend, Gina, who has already been visited.

Queue  
(Empty)



Now, the queue is empty, meaning that we've successfully visited every friend in this network.

#### Picture it like a Party:

- **Alice is the Host:** She invites her friends (Bob, Candy, Derek, and Elaine), and they get in line.
- **Bob's Turn:** He invites Fred, who gets in line.
- **Candy's Turn:** She invites Helen, who gets in line.
- **Derek's Turn:** He invites Gina, who gets in line.
- **Elaine, Fred, and Helen:** They don't invite anyone new.
- **Gina's Turn:** She invites Irena, who gets in line.
- **Irena's Turn:** She doesn't invite anyone new.

Everyone at the party has been greeted. We've made sure to greet all the friends of friends, all the way down the line. This is the essence of Breadth-First Search. It's like a party where everyone greets everyone else, and you make sure to greet your friends' friends too.

#### Code Implementation: [Breadth-First Search](#)

Let's break down the code for a Breadth-First Search (BFS) traversal:

```

def bfs_traverse(starting_vertex)
  queue = Queue.new
  visited_vertices = {}
  visited_vertices[starting_vertex.value] = true
  queue.enqueue(starting_vertex)
  # While the queue is not empty:
  while queue.read
    # Remove the first vertex off the queue and make it the current vertex:
    current_vertex = queue.dequeue

    # Print the current vertex's value:
    puts current_vertex.value

    # Iterate over current vertex's adjacent vertices:
    current_vertex.adjacent_vertices.each do |adjacent_vertex|

      # If we have not yet visited the adjacent vertex:
      if !visited_vertices[adjacent_vertex.value]

        # Mark the adjacent vertex as visited:
        visited_vertices[adjacent_vertex.value] = true

        # Add the adjacent vertex to the queue:
        queue.enqueue(adjacent_vertex)
      end
    end
  end
end

```

1. **Define the Method:** The method `bfs_traverse` takes a starting vertex as an argument.
2. **Create the Queue and Visited List:**
  - A queue (**queue**) is created to hold the vertices yet to be explored.
  - A hash table (**visited\_vertices**) is created to keep track of the visited vertices.
3. **Mark the Starting Vertex:**
  - The starting vertex is marked as visited.
  - The starting vertex is then added to the queue.
4. **Loop through the Queue:** As long as the queue isn't empty, the loop continues:
  - a. **Dequeue:** The first vertex is removed from the queue and becomes the current vertex.
  - b. **Print Vertex:** (Optional) Print the value to verify the algorithm is working.
  - c. **Look at Neighbors:** For each adjacent vertex of the current vertex:
    - **If Not Visited:** If the adjacent vertex hasn't been visited yet:
      - a. Mark it as visited.
      - b. Add it to the queue.
5. **Repeat:** The process continues until the queue is empty.

**What Does This Code Do?**

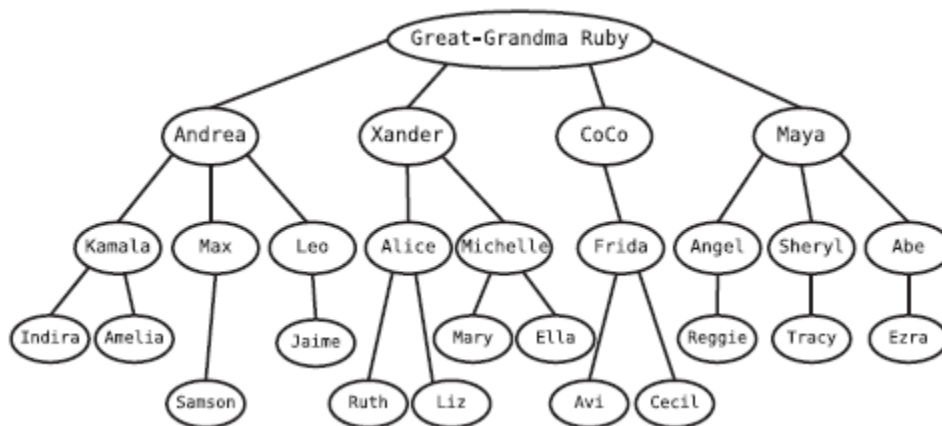
Imagine you want to explore a network of connected people, starting with one person and then moving on to their friends, then their friends' friends, and so on.

- **Starting with One Person:** You begin with one person (the starting vertex) and mark them as "visited."
- **Looking at Their Friends:** You then look at their friends (adjacent vertices), mark them as "visited" if you haven't done so already, and add them to a list (queue) to explore next.
- **Exploring Friends' Friends:** You move on to the next person in the list (queue) and repeat the process.
- **Continuing the Process:** You continue this process, exploring friends and friends of friends, until you've visited everyone.

## DFS vs. BFS

### Depth-First Search (DFS):

- **How It Works:** DFS delves deep into the graph, exploring as far as possible from the starting point before backtracking.
- **Strengths:** If you want to find something far from the starting vertex quickly, this might be the better method. In a family tree example, if you're looking for a distant relative, DFS may find them faster.



- **Weaknesses:** If you're looking for something close to the starting point, DFS can waste time exploring unnecessary areas. In a social network, you might explore friends of friends before finding all immediate connections.

### Breadth-First Search (BFS):

- **How It Works:** BFS explores all of the immediate neighbors of the starting vertex first, then gradually moves farther away.
- **Strengths:** If you want to find connections close to the starting point, BFS will find them first. In a social network, you'll discover all immediate friends before exploring further.
- **Weaknesses:** In a family tree example, if you're looking for a distant relative, BFS would explore all the closer relations first, which might be inefficient.

## Which to Use?



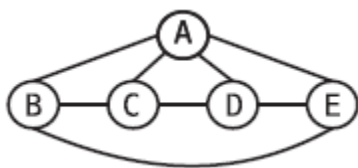
- **Staying Close:** If you want to explore things close to the starting vertex, use BFS.
- **Moving Far Away Quickly:** If you want to find something far from the starting point, use DFS.
- **Depends on Your Needs:** The choice depends on the specific scenario and what you're searching for in the graph. No one method is inherently better; it depends on the context of your search.

Think of DFS as a method that dives deep into a maze, looking for something far away, while BFS takes a more methodical approach, exploring everything close by before moving further away. You would choose between them based on whether you want to find something close or far from your starting point.

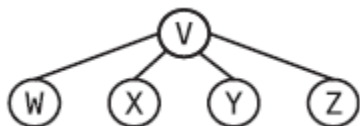
### The Efficiency of Graph Search

Analyzing the efficiency of graph search in terms of time complexity involves a slightly more intricate examination:

- **Vertices (N):** The number of vertices in the graph. In the worst case, you may have to visit every vertex in the graph. This would initially suggest a time complexity of  $O(N)$ .
- **Edges (E):** For each vertex you visit, you also have to check all of its adjacent vertices or neighbors. The number of these adjacent vertices can vary, and this is what introduces the complexity in analysis.
- **Combining Vertices and Edges:** In the example below, two different graphs with the same number of vertices required different numbers of steps to traverse due to the varying numbers of adjacent neighbors. This illustrates that it's not only the number of vertices that determines the efficiency, but also the relationships between them (i.e., the edges).



A: 4 steps to iterate over 4 neighbors  
 B: 3 steps to iterate over 3 neighbors  
 C: 3 steps to iterate over 3 neighbors  
 D: 3 steps to iterate over 3 neighbors  
 E: 3 steps to iterate over 3 neighbors  
 This yields 16 iterations.



V: 4 steps to iterate over 4 neighbors  
 W: 1 step to iterate over 1 neighbor  
 X: 1 step to iterate over 1 neighbor  
 Y: 1 step to iterate over 1 neighbor  
 Z: 1 step to iterate over 1 neighbor  
 This is a total of eight iterations.

### Time Complexity

If we are to accurately describe the time complexity, we need to take both vertices and edges into account. The appropriate Big O notation for both DFS and BFS would be:

- $O(N + E)$

Here,  $N$  is the number of vertices, and  $E$  is the total number of edges or connections between vertices. The "+ $E$ " part of the expression represents the time it takes to examine all the edges or connections in the graph, in addition to visiting all the vertices.

### Why This Matters

Understanding this complexity is crucial for determining how efficiently a graph can be traversed using these methods. It helps you predict how the algorithm will perform as the graph grows in size, both in terms of the number of vertices and the complexity of connections between them. It also illustrates why graph searches can be more complicated to analyze than linear or even tree-based structures, as the relationships between elements can be more intricate and less predictable.

### $O(V + E)$

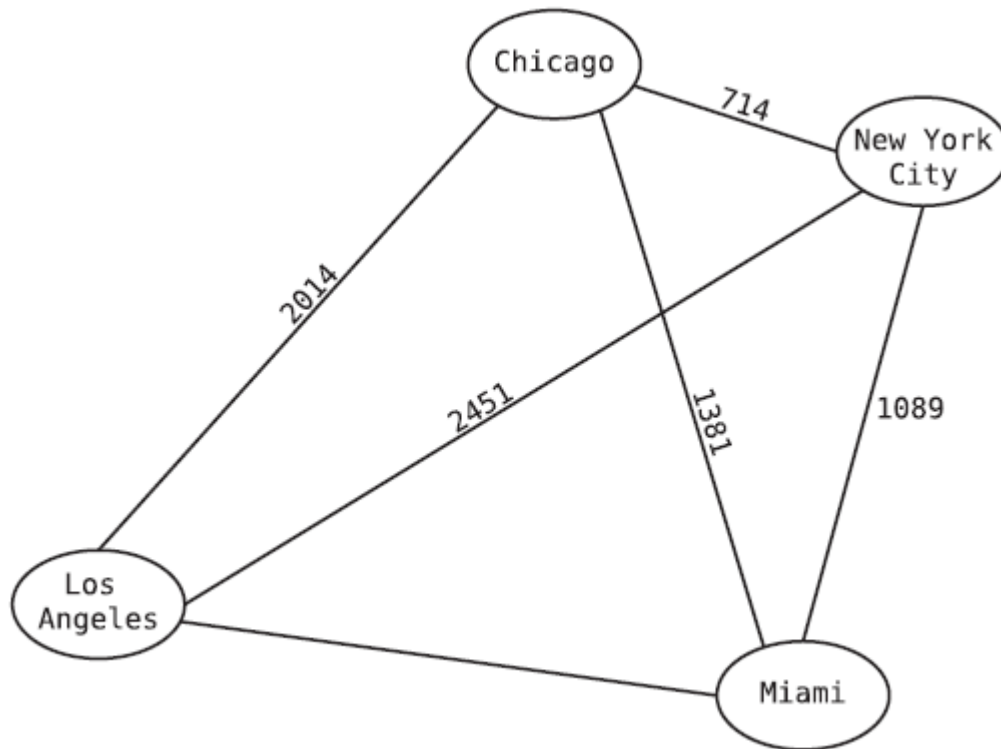
The concept of  $O(V + E)$  as the efficiency of graph search can be broken down into these terms:

1. **V and E:** Here,  $V$  stands for the number of vertices (or nodes) in the graph, and  $E$  stands for the number of edges (or connections) between these vertices.
2. **Counting Edges Twice:** The reason this analysis might seem confusing at first is that, in reality, each edge is considered twice during a typical graph search. That's because when you traverse from vertex  $V$  to  $W$ , and later from  $W$  to  $V$ , the same edge is touched twice.
3. **Why Not  $O(V + 2E)$ ?:** While the actual number of steps might be  $V + 2E$  (counting each edge twice), Big  $O$  Notation simplifies this by dropping constants. That's why we use  $O(V + E)$  instead. It's an approximation, but it's enough to understand the efficiency of the algorithm.
4. **Choosing the Right Search Method:** Both depth-first search (DFS) and breadth-first search (BFS) have this time complexity, but depending on the shape of the graph and what you're looking for, one might be more optimized than the other. Your choice of search method can help you avoid worst-case scenarios.
5. **Real-world Applications in Graph Databases:** Graphs are crucial in dealing with data that involves relationships (like social networks). Graph databases such as Neo4j, ArangoDB, and Apache Giraph specialize in storing and handling this kind of data efficiently.
6. **The Key Takeaway:** Graph search's time complexity of  $O(V + E)$  means that the efficiency of the search is determined by both the number of vertices and the number of edges. Increasing the number of edges will make the search take more steps, and the actual algorithm used might influence where in the graph the desired vertex is found.

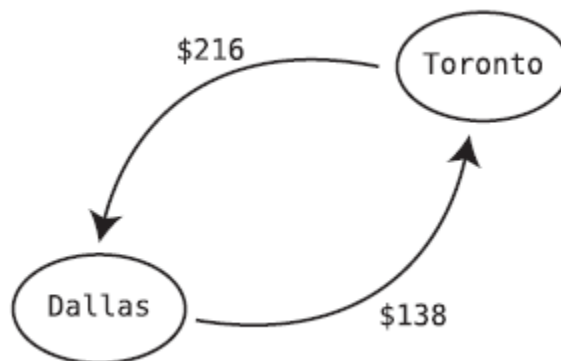
### Weighted Graphs

A weighted graph is a specific type of graph where the edges (or connections) between the vertices (or nodes) have a numerical value or "weight" associated with them. This weight can represent various attributes, depending on the context of the graph:

1. **Distances Between Cities:** In the weighted graph below representing a map, the weight on each edge represents the physical distance between two cities. For example, the edge between Chicago and New York City has a weight of 714, representing the 714 miles between those two cities.



2. **Cost of Flights:** Weighted graphs can also represent directional connections, where the weight has different meanings depending on the direction of the edge. For example, the cost of a flight from Dallas to Toronto might be \$138, while the return flight from Toronto to Dallas might be \$216. The graph below has two different weighted edges to represent these different costs.



Weighted graphs allow us to add more detailed information to a graph by assigning values to the connections between nodes. This information can represent distances, costs, time, or any other attribute that might be relevant to the particular problem or system being modeled.

### [Weighted Graphs in Code](#)

In coding, we can represent a weighted graph by making a slight adjustment to how we define the connections between vertices. Instead of using an array to store adjacent vertices, we use a hash table:

```

class WeightedGraphVertex
  attr_accessor :value, :adjacent_vertices

  def initialize(value)
    @value = value
    @adjacent_vertices = {}
  end

  def add_adjacent_vertex(vertex, weight)
    @adjacent_vertices[vertex] = weight
  end
end

```

1. **Class Definition:** A class named **WeightedGraphVertex** is defined to represent a vertex in the weighted graph.
2. **Attributes:** Each vertex has two attributes: **value** (the value of the vertex, like the city name) and **adjacent\_vertices** (a hash table to store neighboring vertices along with the weight of the connection).
3. **Initialization:** In the **initialize** method, the vertex is assigned a value, and the **adjacent\_vertices** hash table is initialized as an empty hash.
4. **Adding Adjacent Vertices:** The **add\_adjacent\_vertex** method is used to add an adjacent vertex and its weight. The adjacent vertex is stored as a key, and the weight is stored as the corresponding value in the hash table.
5. **Example Usage:** To create the Dallas–Toronto flight price graph mentioned earlier, two instances of **City** are created (for Dallas and Toronto), and the **add\_adjacent\_vertex** method is used to connect them, providing the prices (\$138 and \$216) as weights.

```

dallas = City.new("Dallas")
toronto = City.new("Toronto")

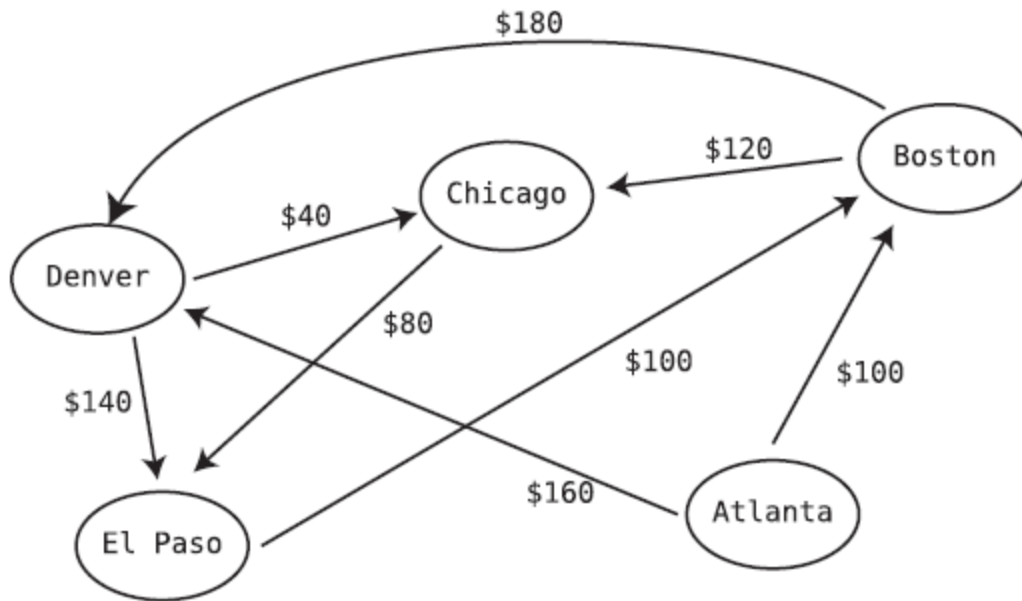
dallas.add_adjacent_vertex(toronto, 138)
toronto.add_adjacent_vertex(dallas, 216)

```

## The Shortest Path Problem

The Shortest Path Problem is a common challenge in graph theory where you want to find the path between two points that minimizes some measure, such as distance or cost. Here's an example of finding the cheapest flight:

1. **Problem Statement:** Imagine you're in Atlanta and want to fly to El Paso, but there's no direct flight. You can take different paths through other cities, but each path has a different price.



2. **Paths and Costs:** For example, going from Atlanta to Denver, and then to El Paso costs \$300, while a path from Atlanta to Denver to Chicago to El Paso costs only \$280.
3. **Finding the Cheapest Fare:** The challenge is to find the path that will cost you the least amount of money. You don't care how many stops you make; you just want the cheapest fare.
4. **Shortest Path Problem:** This is a classic example of the shortest path problem, where "shortest" means the path that minimizes the chosen measure (in this case, the cost).
5. **General Use:** The problem can be applied to various scenarios. If the graph's weights represented distances, the problem would be to find the path with the shortest distance instead of the cheapest cost.
6. **Weighted Graphs:** This problem is specifically relevant to weighted graphs, where the connections between points (or vertices) have associated values or "weights."

### Dijkstra's Algorithm

Numerous algorithms can solve the shortest path problem, and one of the most famous is one discovered by Edsger Dijkstra (pronounced "dike' struh") in 1959. Unsurprisingly, this algorithm is known as Dijkstra's algorithm. In the following section, we're going to use Dijkstra's algorithm to find the cheapest path in our city flights example.

### Dijkstra's Algorithm Setup

Dijkstra's Algorithm is a popular way to find the shortest path in a weighted graph:

1. **Goal:** You want to find not just the cheapest price from Atlanta to El Paso but the cheapest prices from Atlanta to all known cities.
2. **Creating a Table:** You need a way to keep track of the cheapest known prices from Atlanta to other cities. You create a table (or a hash table in code) to store this information.

3. **Initial State:** Initially, the table looks something like this:

<b>From Atlanta To:</b>	<b>City #1</b>	<b>City #2</b>	<b>City #3</b>	<b>Etc.</b>
	?	?	?	?

4. **Final State:** As the algorithm progresses, it will find the cheapest prices, and the table will fill in:

<b>Cheapest Price from Atlanta To:</b>	<b>Boston</b>	<b>Chicago</b>	<b>Denver</b>	<b>El Paso</b>
	\$100	\$200	\$160	\$280

5. **Code Representation:** In code, this information might look like:

```
{"Atlanta" => 0, "Boston" => 100, "Chicago" => 200,  
"Denver" => 160, "El Paso" => 280}
```

6. **Tracking the Path:** To also keep track of the actual path (not just the cost), you'll need another table, which might be called **cheapest\_previous\_stopover\_city\_table**. This table will help you know the specific path to take.

7. **Example of Path Tracking:** By the end of the algorithm, the path tracking table might look like:

<b>Cheapest Previous Stopover City from Atlanta:</b>	<b>Boston</b>	<b>Chicago</b>	<b>Denver</b>	<b>El Paso</b>
	Atlanta	Denver	Atlanta	Chicago

You're setting up a way to track both the cost and the path to reach any city from Atlanta. You start with an empty table and fill it in as you find the cheapest routes. Along the way, you also keep track of the actual paths taken, so you know how to achieve those cheapest prices.

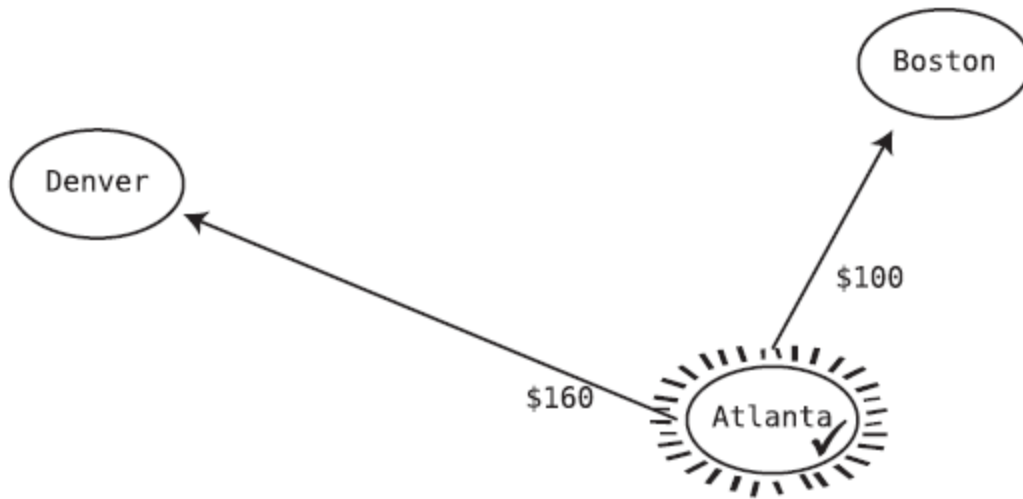
## Dijkstra's Algorithm Steps

Let's break down the steps of Dijkstra's algorithm:

1. **Start at the Initial City:** You begin by visiting the starting city, and this becomes your "current city."
2. **Check Adjacent Cities:** Look at the prices from the current city to all the cities directly connected to it.
3. **Update If Cheaper Path is Found:** If you find a cheaper price to an adjacent city from the starting city than what's already recorded, do the following:
  - a. **Update Prices:** Replace the old price with the cheaper price in the **cheapest\_prices\_table**.
  - b. **Update Path:** Record the path taken in the **cheapest\_previous\_stopover\_city\_table**.
4. **Choose the Next City:** Move to the unvisited city that has the cheapest price from the starting city. This becomes the new current city.
5. **Repeat the Process:** Continue repeating steps 2 through 4 until you've checked all the known cities.

## Dijkstra's Algorithm Walk-Through

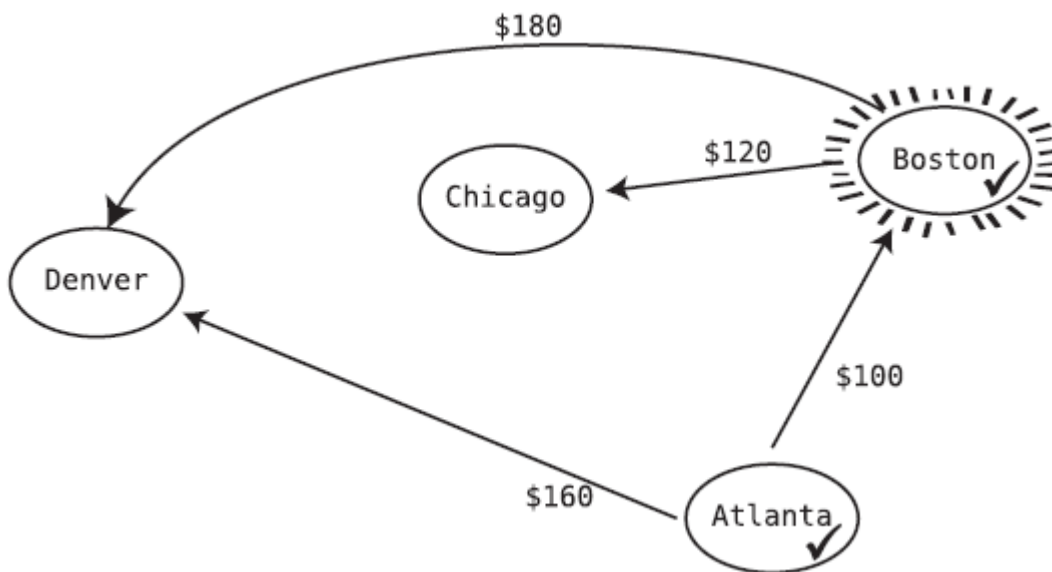
1. **Start with Atlanta:** The **cheapest\_prices\_table** only has Atlanta initially at a cost of \$0.
2. **Visit Atlanta:** Make Atlanta the **current\_city**.



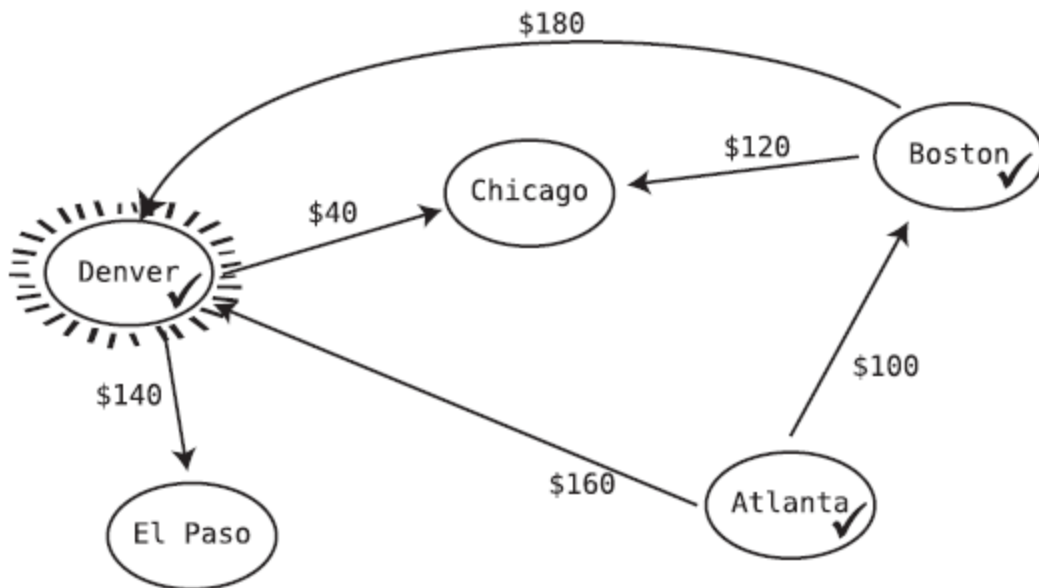
3. **Inspect Adjacent Cities:** Discover new cities by checking adjacent cities.
4. **Add Boston:** The cost from Atlanta to Boston is \$100, so add that to the tables.
5. **Add Denver:** The cost from Atlanta to Denver is \$160, add it to the tables.

From Atlanta To:	Boston	Denver
	\$0	\$160

6. **Move to Boston:** Since it's cheaper to go to Boston, make it the **current\_city**.



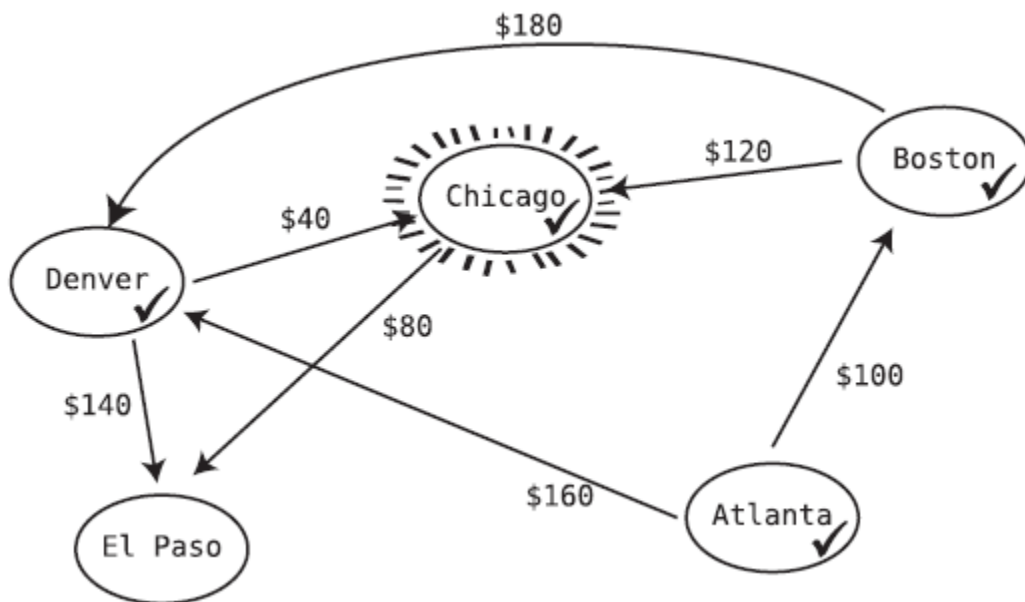
7. **Analyze Boston's Adjacent Cities:** Find that going to Chicago through Boston is \$220, and going to Denver through Boston is \$280.
8. **Choose Denver as the Next City:** It's cheaper to go from Atlanta to Denver (\$160).



9. **Inspect Denver's Adjacent Cities:** The cost from Denver to Chicago is \$40, so update the cost from Atlanta to Chicago to \$200. Find the cost from Denver to El Paso is \$140, add the cost from Atlanta to El Paso as \$300.

From Atlanta To:	Boston	Chicago	Denver	El Paso
	\$0	\$100	\$200	\$160
				\$300

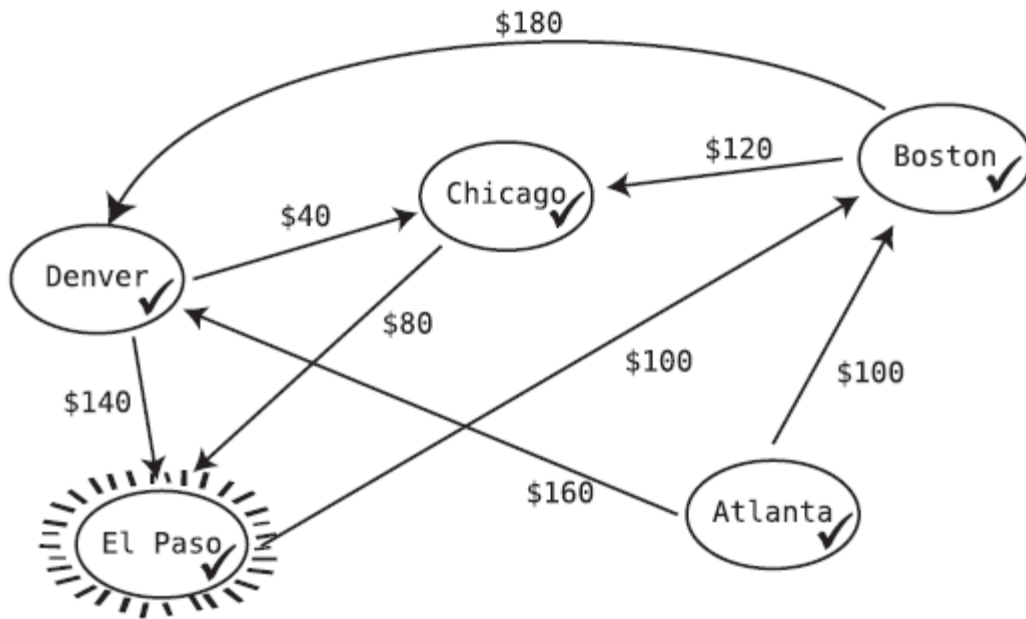
10. **Visit Chicago:** The cost from Chicago to El Paso is \$80. Update the total cost from Atlanta to El Paso to \$280.



From Atlanta To:	Boston	Chicago	Denver	El Paso
	\$0	\$100	\$200	\$160
				\$280

11. **Visit El Paso:** Analyze its outbound flight to Boston, but no update to the tables is necessary.





12. **Complete:** You now have all the information to find the cheapest path from Atlanta to any other city.

This walk-through helps in understanding how Dijkstra's algorithm operates by iterating over cities and constantly updating the cost tables as it discovers more efficient paths. It's an effective method for finding the shortest paths in weighted graphs, and in this case, the most cost-efficient routes between cities.

### Finding the Shortest Path

You want to know the exact path to fly from Atlanta to El Paso to get the lowest price. The final step in Dijkstra's algorithm allows you to do just that by using a table that stores the cheapest previous stopover city for each destination.

1. **Look at the Destination:** Start by looking at the destination city, El Paso, in the `cheapest_previous_stopover_city_table`.

<b>Cheapest Previous Stopover City from Atlanta:</b>	<b>Boston</b>	<b>Chicago</b>	<b>Denver</b>	<b>El Paso</b>
	Atlanta	Denver	Atlanta	Chicago

2. **Find the Previous Stopover:** The table shows Chicago is the immediate stop before flying to El Paso.
3. **Continue Backward:** Next, find Chicago's corresponding value in the table, which is Denver. It indicates that the cheapest route from Atlanta to Chicago involves stopping in Denver right before Chicago.
4. **Find the Start:** Finally, you'll find that the cheapest flight from Atlanta to Denver is to fly directly from Atlanta to Denver. Atlanta is the starting city, so you now have the complete route.
5. **Assemble the Path:** Put all the findings together, and you have your cheapest path:
  - Atlanta -> Denver -> Chicago -> El Paso

This backward chaining logic leads you to the exact sequence of cities to visit, from start to destination, ensuring you have the cheapest flight path. It's an elegant way of using the information stored during the algorithm to find the final path.

## Code Implementation: Dijkstra's Algorithm

```
class City
  attr_accessor :name, :routes

  def initialize(name)
    @name = name
    @routes = {}
  end

  def add_route(city, price)
    @routes[city] = price
  end
end

atlanta = City.new("Atlanta")
boston = City.new("Boston")
chicago = City.new("Chicago")
denver = City.new("Denver")
el_paso = City.new("El Paso")

atlanta.add_route(boston, 100)
atlanta.add_route(denver, 160)
boston.add_route(chicago, 120)
boston.add_route(denver, 180)
chicago.add_route(el_paso, 80)
denver.add_route(chicago, 40)
denver.add_route(el_paso, 140)
```

### 1. Classes Definition:

- **City:** Contains the name of the city and routes to other cities with their prices.
- **add\_route:** A method to add a route to another city with a given price.

### 2. Sample Code to Build Graph:

- Cities like Atlanta, Boston, Chicago, etc., are created.
- Routes between cities with specific prices are defined.

### 3. Dijkstra's Algorithm Implementation:

- The function **dijkstra\_shortest\_path** is used to find the shortest path between two cities.
- **Setup:** Create tables for tracking cheapest prices and previous stopover cities. Initialize arrays for visited and unvisited cities.
- **Loop Through Cities:**
  - Mark current city as visited, and delete it from unvisited cities.
  - Iterate through adjacent cities, adding them to unvisited cities if not visited.
  - Calculate price through the current city and update tables if a cheaper route is found.
  - Choose the next unvisited city with the cheapest price.

- **Build Shortest Path:**
    - Create an array for the shortest path.
    - Work backward from the final destination, adding each city to the array using the previous stopover city table.
    - Reverse the array to see the path from start to finish.
4. **Optimization Consideration:** The code could be optimized by using a priority queue instead of an array to handle unvisited cities, but this has been avoided to keep the code simple.
  5. **Generality:** Although the code talks about cities and prices, the names can be changed to apply to any weighted graph.

### The Efficiency of Dijkstra's Algorithm

Dijkstra's algorithm is a method for finding the shortest path in a weighted graph. Its specific code can vary, and different implementations will affect its time complexity. In the example above, an array was used to track unvisited cities. This can lead to a time complexity of  $O(V^2)$ , where  $V$  is the number of vertices. This occurs in the worst case where every vertex is connected to every other vertex.

Other implementations, like using a priority queue instead of an array, can result in faster running times. Various versions of Dijkstra's algorithm exist, each with its own specific time complexity that needs analysis. Despite these variations, Dijkstra's algorithm offers a significant advantage over the alternative of finding every possible path through the graph and selecting the fastest one. It provides a systematic way to find the shortest path, making it a beneficial choice.

### Wrapping Up

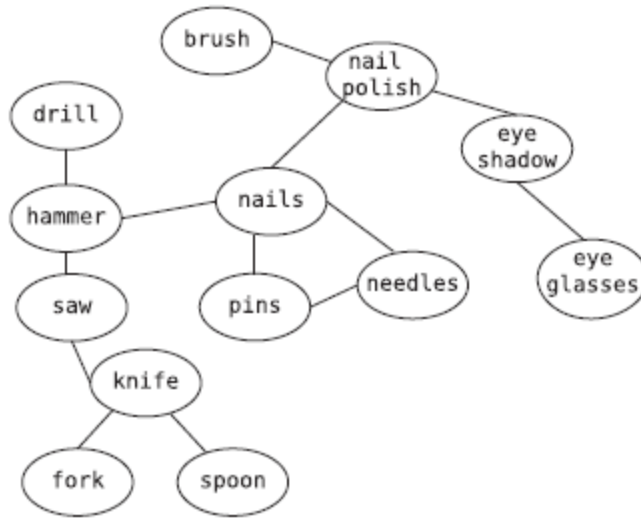
We are nearing the end of the book, and this chapter was about the last major data structure, graphs. Graphs are valuable tools for handling data involving relationships, making code faster, and solving complex problems. There's a wealth of fascinating algorithms related to graphs like minimum spanning tree, topological sort, bidirectional search, and others. This chapter lays the groundwork for exploring these subjects further.

Throughout the book, the main focus has been on the speed of code, measuring efficiency in terms of time and the number of algorithmic steps. But efficiency isn't only about speed; it also relates to how much memory an algorithm or data structure might use. The next chapter will teach you how to analyze code efficiency in terms of space.

### Exercises

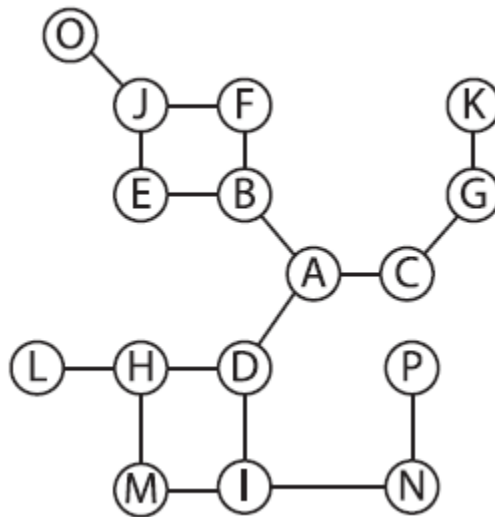
The following exercises provide you with the opportunity to practice with graphs.

1. The first graph on page 385 powers an e-commerce store's recommendation engine. Each vertex represents a product available on the store's website. The edges connect each product to other "similar" products the site will recommend to the user when browsing a particular item. If the user is browsing "nails," what other products will be recommended to the user?



Nail polish, needles, pins, and hammer.

2. If we perform depth-first search on the second graph on page 385 starting with the “A” vertex, what is the order in which we’ll traverse all the vertices? Assume that when given the choice to visit multiple adjacent vertices, we’ll first visit the node that is earliest in the alphabet.



A-B-E-J-F-O-C-G-K-D-H-L-M-I-N-P

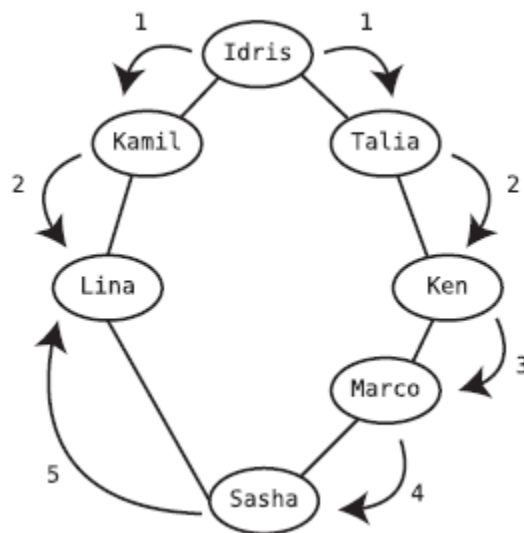
3. If we perform breadth-first search on the previous graph starting with the “A” vertex, what is the order in which we’ll traverse all the vertices? Assume that when given the choice to visit multiple adjacent vertices, we’ll first visit the node that is earliest in the alphabet.

A-B-C-D-E-F-G-H-I-J-K-L-M-N-O-P

4. In this chapter, I only provided the code for breadth-first traversal, as discussed in Breadth-First Search, on page 348. That is, the code simply printed the value of each vertex. Modify the code so that it performs an actual search for a vertex value provided to the function. (We did this for depth-first search.) That is, if the function finds the vertex it's searching for, it should return that vertex's value. Otherwise, it should return null.

```
def breadth_first_search(graph, start, search_value):  
    visited = set()  
    queue = [start]  
  
    while queue:  
        vertex = queue.pop(0)  
        if vertex == search_value:  
            return vertex  
        if vertex not in visited:  
            visited.add(vertex)  
            queue.extend(neighbor for neighbor in graph[vertex] if neighbor not in  
                        visited)  
  
    return None
```

5. In Dijkstra's Algorithm, on page 367, we saw how Dijkstra's algorithm helped us find the shortest path within a weighted graph. However, the concept of a shortest path exists within an unweighted graph as well. How? The shortest path in a classic (unweighted) graph is the path that takes the fewest number of vertices to get from one vertex to another. This can be especially useful in social networking applications. Take the example network that follows:



If we want to know how Idris is connected to Lina, we'd see that she's connected to her from two different directions. That is, Idris is a second-degree connection to Lina through Kamil, but she is also a fifth-degree

connection through Talia. Now, we're probably interested in how closely Idris is connected to Lina, so the fact that she's a fifth-degree connection is unimportant given that they're also second-degree connections.

Write a function that accepts two vertices from a graph and returns the shortest path between them. The function should return an array containing the precise path, such as ["Idris", "Kamil", "Lina"]. Hint: The algorithm may contain elements of both breadth-first search and Dijkstra's algorithm.

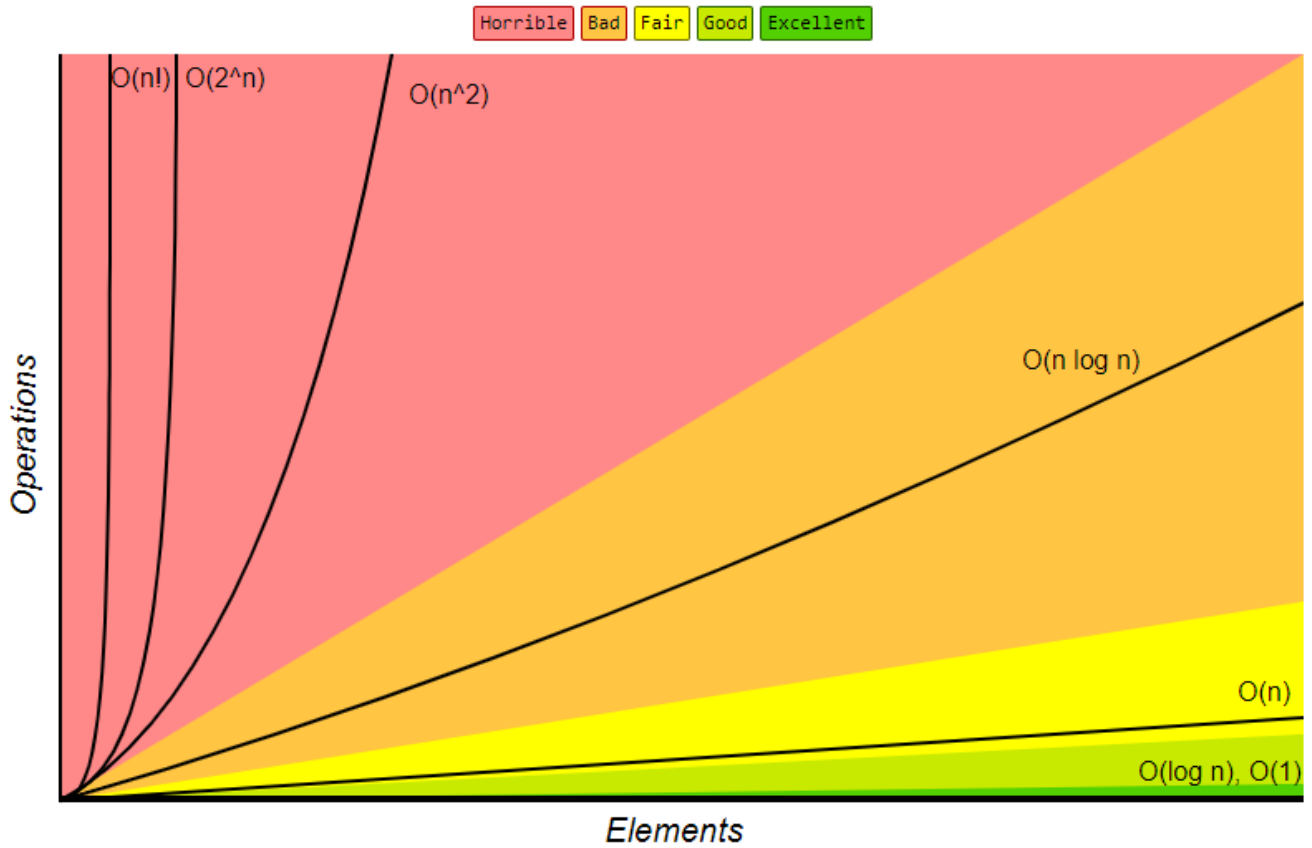
```
def shortest_path(graph, start, end):
    queue = [(start, [start])]
    visited = set([start])

    while queue:
        (vertex, path) = queue.pop(0)
        for next_vertex in graph[vertex] - set(path):
            if next_vertex == end:
                return path + [next_vertex]
            else:
                visited.add(next_vertex)
                queue.append((next_vertex, path + [next_vertex]))

    return None
```

This function takes the graph and the start and end vertices as input and returns an array with the shortest path, such as ["Idris", "Kamil", "Lina"], or None if no path is found. It's a simple breadth-first search tailored to track paths and return the shortest one.

## Big-O Complexity Chart



## Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	$\theta(1)$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Stack	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Queue	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Singly-Linked List	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Doubly-Linked List	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Skip List	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n \log(n))$
Hash Table	N/A	$\theta(1)$	$\theta(1)$	$\theta(1)$	N/A	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Binary Search Tree	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Cartesian Tree	N/A	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	N/A	$O(n)$	$O(n)$	$O(n)$	$O(n)$
B-Tree	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
Red-Black Tree	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
Splay Tree	N/A	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	N/A	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
AVL Tree	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
KD Tree	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$

# Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
<u>Quicksort</u>	$\Omega(n \log(n))$	$\theta(n \log(n))$	$O(n^2)$	$O(\log(n))$
<u>Mergesort</u>	$\Omega(n \log(n))$	$\theta(n \log(n))$	$O(n \log(n))$	$O(n)$
<u>Timsort</u>	$\Omega(n)$	$\theta(n \log(n))$	$O(n \log(n))$	$O(n)$
<u>Heapsort</u>	$\Omega(n \log(n))$	$\theta(n \log(n))$	$O(n \log(n))$	$O(1)$
<u>Bubble Sort</u>	$\Omega(n)$	$\theta(n^2)$	$O(n^2)$	$O(1)$
<u>Insertion Sort</u>	$\Omega(n)$	$\theta(n^2)$	$O(n^2)$	$O(1)$
<u>Selection Sort</u>	$\Omega(n^2)$	$\theta(n^2)$	$O(n^2)$	$O(1)$
<u>Tree Sort</u>	$\Omega(n \log(n))$	$\theta(n \log(n))$	$O(n^2)$	$O(n)$
<u>Shell Sort</u>	$\Omega(n \log(n))$	$\theta(n(\log(n))^2)$	$O(n(\log(n))^2)$	$O(1)$
<u>Bucket Sort</u>	$\Omega(n+k)$	$\theta(n+k)$	$O(n^2)$	$O(n)$
<u>Radix Sort</u>	$\Omega(nk)$	$\theta(nk)$	$O(nk)$	$O(n+k)$
<u>Counting Sort</u>	$\Omega(n+k)$	$\theta(n+k)$	$O(n+k)$	$O(k)$
<u>Cubesort</u>	$\Omega(n)$	$\theta(n \log(n))$	$O(n \log(n))$	$O(n)$

or