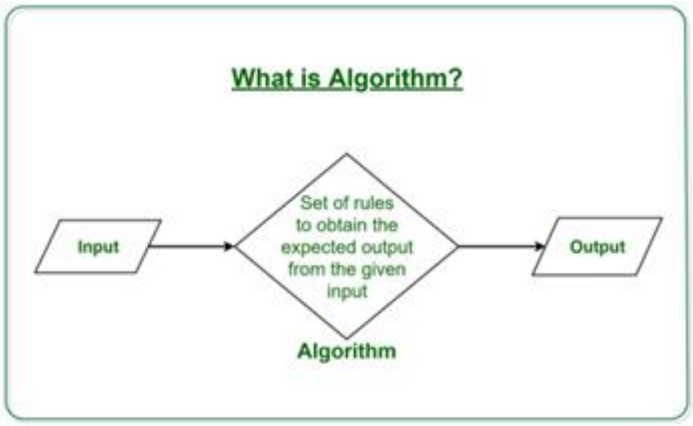
Module-I (08 Hrs)

Algorithm Introduction

An algorithm is an effective step-by-step procedure for solving a problem in a finite number of steps. In other words, it is a finite set of well-defined instructions or step-by-step description of the procedure written in human readable language for solving a given problem. An algorithm itself is division of a problem into small steps which are ordered in sequence and easily understandable. Algorithms are very important to the way computers process information, because a computer program is basically an algorithm that tells computer what specific tasks to perform in what specific order to accomplish a specific task. The same problem can be solved with different methods. So, for solving the same problem, different algorithms can be designed. In these algorithms, number of steps, time and efforts may vary more or less.


What is Algorithm?
Input
Set of rules to obtain the expected output from the given input
Output
Algorithm

Algorithm and its characteristics

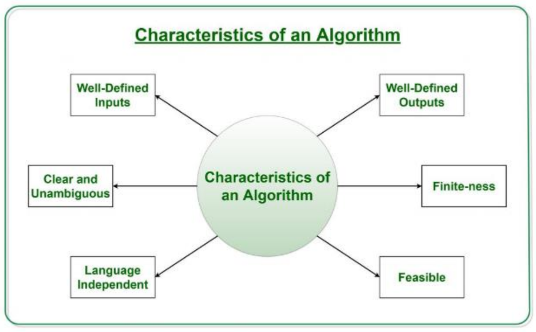
Design
Experiment
Implement
Analyze
Algorithms

In mathematics, computing, linguistics and related subjects, an algorithm is a sequence of finite instructions, often used for calculation and data processing. It is formally a type of effective method in which a list of well-defined instructions for completing a task will, when given an initial state, proceed through a well-defined series of successive states, eventually terminating in an end-state. The transition from one state to the next is not necessarily deterministic; some algorithms, known as probabilistic algorithms, incorporate randomness. A step-by-step method of solving a problem or making decisions, as in making a diagnosis.

An established mechanical procedure for solving certain mathematical problems.

Properties of the algorithm:

1. Finiteness. An algorithm must always terminate after a finite number of steps.
2. Definiteness. Each step of an algorithm must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case.
3. Input. An algorithm has zero or more inputs, i.e, quantities which are given to it initially before the algorithm begins.
4. Output. An algorithm has one or more outputs i.e, quantities which have a specified relation to the inputs.
5. Effectiveness. An algorithm is also generally expected to be effective. This means that all of the operations to be performed in the algorithm must be sufficiently basic that they can in principle be done exactly and in a finite length of time.



Complexity of Algorithm:
It is very convenient to classify algorithms based on the relative amount of time or relative amount of space they require and specify the growth of time /space requirements as a function of the input size. Thus, we have the notions of:

1. Time Complexity: Running time of the program as a function of the size of input

2. Space Complexity: Amount of computer memory required during the program execution, as a function of the input size.

Guidelines for Developing an Algorithm

Following guidelines must be followed while developing an algorithm :

1. An algorithm will be enclosed by START (or BEGIN) and STOP (or END).
2. To accept data from user, generally used statements are INPUT, READ, GET or OBTAIN.
3. To display result or any message, generally used statements are PRINT, DISPLAY, or WRITE.
4. Generally, COMPUTE or CALCULATE is used while describing mathematical expressions and based on situation relevant operators can be used.

Example of an Algorithm

Algorithm : Calculation of Simple Interest

```
Step 1: Start
Step 2: Read principle (P), time (T) and rate (R)
Step 3: Calculate I = P*T*R/100
Step 4: Print I as Interest
Step 5: Stop
```

Advantages of an Algorithm

Designing an algorithm has following advantages :

1. **Effective Communication:** Since algorithm is written in English like language, it is simple to understand step-by-step solution of the problems.
2. **Easy Debugging:** Well-designed algorithm makes debugging easy so that we can identify logical error in the program.
3. **Easy ann Efficient Coding:** An algorithm acts as a blueprint of a program and helps during program development.
4. **Independent of Programming Language:** An algorithm is independent of programming languages and can be easily coded using any high level language.

Disadvantages of an Algorithm

An algorithm has following disadvantages :

1. Developing algorithm for complex problems would be time consuming and difficult to understand.
2. Understanding complex logic through algorithms can be very difficult.

Types of Algorithms:

There are several types of algorithms available. Some important algorithms are:

1. Brute Force Algorithm: It is the simplest approach for a problem. A brute force algorithm is the first approach that comes to finding when we see a problem.

2. Recursive Algorithm: A recursive algorithm is based on recursion. In this case, a problem is broken into several sub-parts and called the same function again and again.

3. Backtracking Algorithm: The backtracking algorithm basically builds the solution by searching among all possible solutions. Using this algorithm, we keep on building the solution following criteria. Whenever a solution fails we trace back to the failure point and build on the next solution and continue this process till we find the solution or all possible solutions are looked after.

4. Searching Algorithm: Searching algorithms are the ones that are used for searching elements or groups of elements from a particular data structure. They can be of different types based on their approach or the data structure in which the element should be found.

5. Sorting Algorithm: Sorting is arranging a group of data in a particular manner according to the requirement. The algorithms which help in performing this function are called sorting algorithms. Generally sorting algorithms are used to sort groups of data in an increasing or decreasing manner.

6. Hashing Algorithm: Hashing algorithms work similarly to the searching algorithm. But they contain an index with a key ID. In hashing, a key is assigned to specific data.

7. Divide and Conquer Algorithm: This algorithm breaks a problem into sub-problems, solves a single sub-problem and merges the solutions together to get the final solution. It consists of the following three steps:

- Divide
- Solve
- Combine

8. Greedy Algorithm: In this type of algorithm the solution is built part by part. The solution of the next part is built based on the immediate benefit of the next part. The one solution giving the most benefit will be chosen as the solution for the next part.

9. Dynamic Programming Algorithm: This algorithm uses the concept of using the already found solution to avoid repetitive calculation of the same part of the problem. It divides the problem into smaller overlapping subproblems and solves them.

10. Randomized Algorithm: In the randomized algorithm we use a random number so it gives immediate benefit. The random number helps in deciding the expected outcome.

To learn more about the types of algorithms refer to the article about "**Types of Algorithms**".

How to Design an Algorithm?

In order to write an algorithm, the following things are needed as a pre-requisite:

1. The **problem** that is to be solved by this algorithm i.e. clear problem definition.
2. The **constraints** of the problem must be considered while solving the problem.
3. The **input** to be taken to solve the problem.
4. The **output** to be expected when the problem is solved.
5. The **solution** to this problem, is within the given constraints.

Then the algorithm is written with the help of the above parameters such that it solves the problem.
Example: Consider the example to add three numbers and print the sum.

- **Step 1: Fulfilling the pre-requisites**
  As discussed above, in order to write an algorithm, its pre-requisites must be fulfilled.
  1. **The problem that is to be solved by this algorithm**: Add 3 numbers and print their sum.
  2. **The constraints of the problem that must be considered while solving the problem**: The numbers must contain only digits and no other characters.
  3. **The input to be taken to solve the problem:** The three numbers to be added.
  4. **The output to be expected when the problem is solved:** The sum of the three numbers taken as the input i.e. a single integer value.

5. **The solution to this problem, in the given constraints:** The solution consists of adding the 3 numbers. It can be done with the help of '+' operator, or bit-wise, or any other method.

- **Step 2: Designing the algorithm**
  Now let's design the algorithm with the help of the above pre-requisites:
  Algorithm to add 3 numbers and print their sum:
  1. START
  2. Declare 3 integer variables num1, num2 and num3.
  3. Take the three numbers, to be added, as inputs in variables num1, num2, and num3 respectively.
  4. Declare an integer variable sum to store the resultant sum of the 3 numbers.
  5. Add the 3 numbers and store the result in the variable sum.
  6. Print the value of the variable sum
  7. END
- **Step 3: Testing the algorithm by implementing it.**
  In order to test the algorithm, let's implement it in C language.

What is an Algorithm? Definition, Types, Complexity, Examples

An algorithm is a well-defined sequential computational technique that accepts a value or a collection of values as input and produces the output(s) needed to solve a problem.

Or we can say that an algorithm is said to be accurate if and only if it stops with the proper output for each input instance.

Example:

Consider a box where no one can see what's happening inside, we say a black box.

We give input to the box and it gives us the output we need but the procedure that we might need to know behind the conversion of input to desired output is an ALGORITHM.

An algorithm is independent of the language used. It tells the programmer the logic used to solve the problem. So, it is a logical step-by-step procedure that acts as a blueprint to programmers.

Real-life examples that define the use of algorithms:

- Consider a clock. We know the clock is ticking but how does the manufacturer set those nuts and bolts so that it keeps on moving every 60 seconds, the min hand should move and every 60 mins, the hour hand should move? So to solve this problem, there must be an algorithm behind it.
- Seen someone cooking your favorite food for you? Is the recipe necessary for it? Yes, it is necessary as a recipe is a sequential procedure that turns a raw potato into a chilly potato. This is what an algorithm is: following a procedure to get the desired output. Is the sequence necessary to be followed? Yes, the sequence is the most important thing that has to be followed to get what we want.

Types of Algorithms:

- **Sorting algorithms:** Bubble Sort, insertion sort, and many more. These algorithms are used to sort the data in a particular format.
- **Searching algorithms:** Linear search, binary search, etc. These algorithms are used in finding a value or record that the user demands.
- **Graph Algorithms**: It is used to find solutions to problems like finding the shortest path between cities, and real-life problems like traveling salesman problems.

Why do we use algorithms?

Consider two kids, Aman and Rohan, solving the Rubik's Cube. Aman knows how to solve it in a definite number of steps. On the other hand, Rohan knows that he will do it but is not aware of the procedure. Aman solves the cube within 2 minutes whereas Rohan is still stuck and by the end of the day, he somehow managed to solve it (might have cheated as the procedure is necessary).

So the time required to solve with a procedure/algorithm is much more effective than that without any procedure. Hence the need for an algorithm is a must.

In terms of designing a solution to an IT problem, computers are fast but not infinitely fast. The memory may be inexpensive but not free. So, computing time is therefore a bounded resource and so is the space in memory. So we should use these resources wisely and algorithms that are efficient in terms of time and space will help you do so.

Creating an Algorithm:

Since the algorithm is language-independent, we write the steps to demonstrate the logic behind the solution to be used for solving a problem. But before writing an algorithm, keep the following points in mind:

- The algorithm should be clear and unambiguous.
- There should be 0 or more well-defined inputs in an algorithm.
- An algorithm must produce one or more well-defined outputs that are equivalent to the desired output. After a specific number of steps, algorithms must ground to a halt.
- Algorithms must stop or end after a finite number of steps.
- In an algorithm, step-by-step instructions should be supplied, and they should be independent of any computer code.

Example: algorithm to multiply 2 numbers and print the result:

Step 1: Start
Step 2: Get the knowledge of input. Here we need 3 variables; a and b will be the user input and c will hold the result.
Step 3: Declare a, b, c variables.
Step 4: Take input for a and b variable from the user.
Step 5: Know the problem and find the solution using operators, data structures and logic

We need to multiply a and b variables so we use * operator and assign the result to c.
That is c <- a * b

Step 6: Check how to give output, Here we need to print the output. So write print c
Step 7: End

Example 1: Write an algorithm to find the maximum of all the elements present in the array.
Follow the algorithm approach as below:

Step 1: Start the Program
Step 2: Declare a variable max with the value of the first element of the array.
Step 3: Compare max with other elements using loop.
Step 4: If max < array element value, change max to new max.
Step 5: If no element is left, return or print max otherwise goto step 3.
Step 6: End of Solution

Example 2: Write an algorithm to find the average of 3 subjects.
Follow the algorithm approach as below:

Step 1: Start the Program
Step 2: Declare and Read 3 Subject, let's say **S1, S2, S3**
Step 3: Calculate the sum of all the 3 Subject values and store result in Sum variable **(Sum = S1+S2+S3)**
Step 4: Divide Sum by 3 and assign it to Average variable. **(Average = Sum/3)**
Step 5: Print the value of **Average of 3 Subjects**
Step 6: End of Solution

Know about Algorithm Complexity:

An algorithm is analysed using Time Complexity and Space Complexity. Writing an efficient algorithm help to consume the minimum amount of time for processing the logic. For algorithm A, it is judged on the basis of two parameters for an input of size n :

- **Time Complexity:** Time taken by the algorithm to solve the problem. It is measured by calculating the iteration of loops, number of comparisons etc.
- **Space Complexity:** Space taken by the algorithm to solve the problem. It includes space used by necessary input variables and any extra space (excluding the space taken by inputs) that is used by the algorithm. For example, if we use a hash table (a kind of data structure), we need an array to store values so this is an extra space occupied, hence will count towards the space complexity of the algorithm. This extra space is known as Auxiliary Space.

Advantages of Algorithms

- **Easy to understand:** Since it is a stepwise representation of a solution to a given problem, it is easy to understand.
- **Language Independent:** It is not dependent on any programming language, so it can easily be understood by anyone.
- **Debug / Error Finding:** Every step is independent / in a flow so it will be easy to spot and fix the error.
- **Sub-Problems:** It is written in a flow so now the programmer can divide the tasks which makes them easier to code.

Disadvantages of Algorithms

- Creating efficient algorithms is time-consuming and requires good logical skills.
- Nasty to show branching and looping in algorithms.

Algorithms Design Techniques

An Algorithm is a procedure to solve a particular problem in a finite number of steps for a finite-sized input. The algorithms can be classified in various ways. They are:

1. Implementation Method
2. Design Method
3. Design Approaches
4. Other Classifications

In this article, the different algorithms in each classification method are discussed.

Classification by Implementation Method: There are primarily three main categories into which an algorithm can be named in this type of classification. They are:

1. **Recursion or Iteration:** A recursive algorithm is an algorithm which calls itself again and again until a base condition is achieved whereas iterative algorithms use loops and/or data structures like stacks, queues to solve any problem. Every recursive solution can be implemented as an iterative solution and vice versa.
   Example: The Tower of Hanoi is implemented in a recursive fashion while Stock Span problem is implemented iteratively.
2. **Exact or Approximate:** Algorithms that are capable of finding an optimal solution for any problem are known as the exact algorithm. For all those problems, where it is not possible to find the most optimized solution, an approximation algorithm is used. Approximate algorithms are the type of algorithms that find the result as an average outcome of sub outcomes to a problem.
   Example: For NP-Hard Problems, approximation algorithms are used. Sorting algorithms are the exact algorithms.
3. **Serial or Parallel or Distributed Algorithms:** In serial algorithms, one instruction is executed at a time while parallel algorithms are those in which we divide the problem into subproblems and execute them on different processors. If parallel algorithms are distributed on different machines, then they are known as distributed algorithms.

Classification by Design Method: There are primarily three main categories into which an algorithm can be named in this type of classification. They are:
Greedy Method: In the greedy method, at each step, a decision is made to choose the *local optimum*, without thinking about the future consequences.
Example: Fractional Knapsack, Activity Selection.

1. **Divide and Conquer:** The Divide and Conquer strategy involves dividing the problem into sub-problem, recursively solving them, and then recombining them for

the final answer.
Example: Merge sort, Quicksort.

2. **Dynamic Programming:** The approach of Dynamic programming is similar to divide and conquer. The difference is that whenever we have recursive function calls with the same result, instead of calling them again we try to store the result in a data structure in the form of a table and retrieve the results from the table. Thus, the overall time complexity is reduced. “Dynamic” means we dynamically decide, whether to call a function or retrieve values from the table.
   Example: 0-1 Knapsack, subset-sum problem.
3. **Linear Programming:** In Linear Programming, there are inequalities in terms of inputs and maximizing or minimizing some linear functions of inputs.
   Example: Maximum flow of Directed Graph
4. **Reduction(Transform and Conquer):** In this method, we solve a difficult problem by transforming it into a known problem for which we have an optimal solution. Basically, the goal is to find a reducing algorithm whose complexity is not dominated by the resulting reduced algorithms.
   Example: Selection algorithm for finding the median in a list involves first sorting the list and then finding out the middle element in the sorted list. These techniques are also called *transform and conquer.*
5. **Backtracking:** This technique is very useful in solving combinatorial problems that have *a single unique solution*. Where we have to find the correct combination of steps that lead to fulfillment of the task. Such problems have multiple stages and there are multiple options at each stage. This approach is based on exploring each available option at every stage one-by-one. While exploring an option if a point is reached that doesn’t seem to lead to the solution, the program control backtracks one step, and starts exploring the next option. In this way, the program explores all possible course of actions and finds the route that leads to the solution.
   Example: N-queen problem, maize problem.
6. **Branch and Bound:** This technique is very useful in solving combinatorial optimization problem that have *multiple solutions* and we are interested in find the most optimum solution. In this approach, the entire solution space is represented in the form of a state space tree. As the program progresses each state combination is explored, and the previous solution is replaced by new one if it is not the optimal than the current solution.
   Example: Job sequencing, Travelling salesman problem.

Classification by Design Approaches : There are two approaches for designing an algorithm. these approaches include

1. **Top-Down Approach :**
2. **Bottom-up approach**

- **Top-Down Approach:** In the top-down approach, a large problem is divided into small sub-problem. and keep repeating the process of decomposing problems until the complex problem is solved.
- **Bottom-up approach:** The bottom-up approach is also known as the reverse of top-down approaches.
  In approach different, part of a complex program is solved using a programming language and then this is combined into a complete program.

Other Classifications: Apart from classifying the algorithms into the above broad categories, the algorithm can be classified into other broad categories like:

1. **Randomized Algorithms:** Algorithms that make random choices for faster solutions are known as randomized algorithms.
   Example: Randomized Quicksort Algorithm.
2. **Classification by complexity:** Algorithms that are classified on the basis of time taken to get a solution to any problem for input size. This analysis is known as time complexity analysis.
   Example: Some algorithms take O(n), while some take exponential time.
3. **Classification by Research Area:** In CS each field has its own problems and needs efficient algorithms.
   Example: Sorting Algorithm, Searching Algorithm, Machine Learning etc.
4. **Branch and Bound Enumeration and Backtracking:** These are mostly used in Artificial Intelligence.

Analysis of algorithm:

Asymptotic Analysis (Based on input size) in Complexity Analysis of Algorithms

Asymptotic analysis of an algorithm refers to defining the mathematical boundation/framing of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case, and worst case scenario of an algorithm.

Asymptotic analysis is input bound i.e., if there's no input to the algorithm, it is concluded to work in a constant time. Other than the "input" all other factors are considered constant.

Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation. For example, the running time of one operation is computed as f(n) and may be for another operation it is computed as $g(n^2)$. This means the first operation running time will increase linearly with the increase in **n** and the running time of the second operation will increase exponentially when **n** increases. Similarly, the running time of both operations will be nearly the same if **n** is significantly small.

Usually, the time required by an algorithm falls under three types −

- **Best Case** − Minimum time required for program execution.
- **Average Case** − Average time required for program execution.
- **Worst Case** − Maximum time required for program execution.

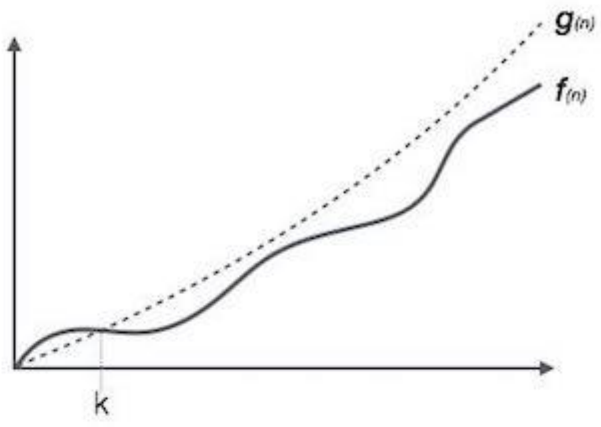
Asymptotic Notations

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

- O Notation
- Ω Notation
- θ Notation

Big Oh Notation, O

The notation O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.

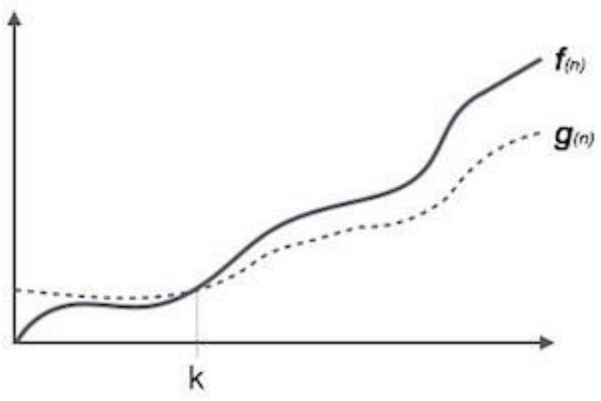


For example, for a function ***f*(n)**

```
O(f(n)) = { g(n) : there exists c > 0 and n₀ such that f(n) ≤ c.g(n) for
all n > n₀. }
```

Omega Notation, Ω

The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.

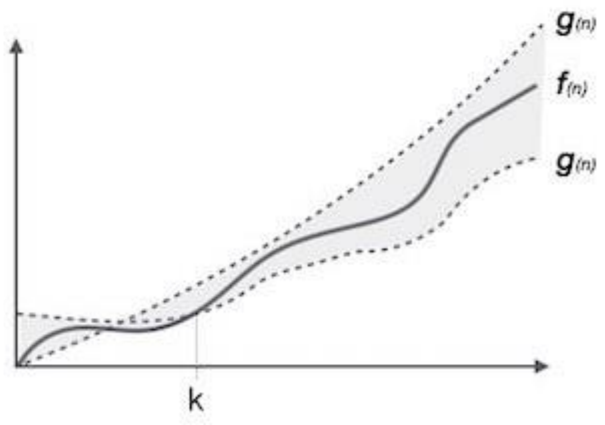


For example, for a function ***f*(n)**

```
Ω(f(n)) ≥ { g(n) : there exists c > 0 and n₀ such that g(n) ≤ c.f(n) for
all n > n₀. }
```

Theta Notation, θ

The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time. It is represented as follows −



`θ(f(n)) = { g(n) if and only if g(n) =  O(f(n)) and g(n) = Ω(f(n)) for all n > n₀. }`

Common Asymptotic Notations

Following is a list of some common asymptotic notations −

constant	−	O(1)
logarithmic	−	O(log n)
linear	−	O(n)
n log n	−	O(n log n)
quadratic	−	$O(n^2)$
cubic	−	$O(n^3)$
polynomial	−	$n^{O(1)}$
exponential	−	$2^{O(n)}$

Asymptotic Analysis is defined as the big idea that handles the above issues in analyzing algorithms. In Asymptotic Analysis, we **evaluate the performance of an algorithm in terms of input size** (we don't measure the actual running time). We calculate, how the time (or space) taken by an algorithm increases with the input size.

Why performance analysis?

There are many important things that should be taken care of, like user-friendliness, modularity, security, maintainability, etc. Why worry about performance? The answer to this is simple, we can have all the above things only if we have performance. So performance is like currency through which we can buy all the above things. Another reason for studying performance is – speed is fun! To summarize, performance == scale. Imagine a text editor that can load 1000 pages, but can spell check 1 page per minute OR an image editor that takes 1 hour to rotate your image 90 degrees left OR … you get it. If a software feature can not cope with the scale of tasks users need to perform – it is as good as dead.

Given two algorithms for a task, how do we find out which one is better?

One naive way of doing this is – to implement both the algorithms and run the two programs on your computer for different inputs and see which one takes less time. There are many problems with this approach for the analysis of algorithms.

- It might be possible that for some inputs, the first algorithm performs better than the second. And for some inputs second performs better.
- It might also be possible that for some inputs, the first algorithm performs better on one machine, and the second works better on another machine for some other inputs.

Asymptotic Analysis is the big idea that handles the above issues in analyzing algorithms. In Asymptotic Analysis, we **evaluate the performance of an algorithm in terms of input size** (we don't measure the actual running time). We calculate, how the time (or space) taken by an algorithm increases with the input size.

For example, let us consider the search problem (searching a given item) in a sorted array.

The solution to above search problem includes:

- **Linear Search** (order of growth is linear)
- **Binary Search** (order of growth is logarithmic).

To understand how Asymptotic Analysis solves the problems mentioned above in analyzing algorithms,

- let us say:
  - we run the Linear Search on a fast computer A and
  - Binary Search on a slow computer B and
  - pick the constant values for the two computers so that it tells us exactly how long it takes for the given machine to perform the search in seconds.
- Let's say the constant for A is 0.2 and the constant for B is 1000 which means that A is 5000 times more powerful than B.
- For small values of input array size n, the fast computer may take less time.
- **But, after a certain value of input array size, the Binary Search will definitely start taking less time compared to the Linear Search even though the Binary Search is being run on a slow machine**.

Input Size	Running time on A	Running time on B
10	2 sec	~ 1 h
100	20 sec	~ 1.8 h
10^6	~ 55.5 h	~ 5.5 h
10^9	~ 6.3 years	~ 8.3 h

- The reason is the order of growth of Binary Search with respect to input size is logarithmic while the order of growth of Linear Search is linear.
- **So the machine-dependent constants can always be ignored after a certain value of input size.**

Running times for this example:

- Linear Search running time in seconds on A: 0.2 * n
- Binary Search running time in seconds on B: 1000*log(n)

Does Asymptotic Analysis always work?

Asymptotic Analysis is not perfect, but that's the best way available for analyzing algorithms. For example, say there are two sorting algorithms that take 1000nLogn and 2nLogn time respectively on a machine. Both of these algorithms are asymptotically the same (order of growth is nLogn). So, With Asymptotic Analysis, we can't judge which one is better as we ignore constants in Asymptotic Analysis.

Also, in Asymptotic analysis, we always talk about input sizes larger than a constant value. It might be possible that those large inputs are never given to your software and an asymptotically slower algorithm always performs better for your particular situation. So, you may end up choosing an algorithm that is Asymptotically slower but faster for your software.

Please write comments if you find anything incorrect, or if you want to share more information about the topic discussed above

In the previous post, we discussed how Asymptotic analysis overcomes the problems of the naive way of analyzing algorithms. But let's take an overview of the **asymptotic notation** and learn about What is Worst, Average, and Best cases of an algorithm:

Popular Notations in Complexity Analysis of Algorithms

1. Big-O Notation

We define an algorithm's **worst-case** time complexity by using the Big-O notation, which determines the set of functions grows slower than or at the same rate as the expression. Furthermore, it explains the maximum amount of time an algorithm requires considering all input values.

2. Omega Notation

It defines the **best case** of an algorithm’s time complexity, the Omega notation defines whether the set of functions will grow faster or at the same rate as the expression. Furthermore, it explains the minimum amount of time an algorithm requires considering all input values.

3. Theta Notation

It defines the **average case** of an algorithm’s time complexity, the Theta notation defines when the set of functions lies in both **O(expression)** and **Omega(expression)**, then Theta notation is used. This is how we define a time complexity average case for an algorithm.

Measurement of Complexity of an Algorithm

Based on the above three notations of Time Complexity there are three cases to analyze an algorithm:

1. Worst Case Analysis (Mostly used)

In the worst-case analysis, we calculate the upper bound on the running time of an algorithm. We must know the case that causes a maximum number of operations to be executed. For Linear Search, the worst case happens when the element to be searched (x) is not present in the array. When x is not present, the search() function compares it with all the elements of arr[] one by one. Therefore, the worst-case time complexity of the linear search would be **O(n)**.

2. Best Case Analysis (Very Rarely used)

In the best case analysis, we calculate the lower bound on the running time of an algorithm. We must know the case that causes a minimum number of operations to be executed. In the linear search problem, the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on n). So time complexity in the best case would be Ω(1)

3. Average Case Analysis (Rarely used)

In average case analysis, we take all possible inputs and calculate the computing time for all of the inputs. Sum all the calculated values and divide the sum by the total number of inputs. We must know (or predict) the distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed (including the case of x not being present in the array). So we sum all the cases and divide the sum by (n+1). Following is the value of average-case time complexity.

Average Case Time = \sum_{i=1}^{n}\frac{\theta (i)}{(n+1)} = \frac{\theta (\frac{(n+1)*(n+2)}{2})}{(n+1)} = \theta (n)

Which Complexity analysis is generally used?

Below is the ranked mention of complexity analysis notation based on popularity:

1. Worst Case Analysis:

Most of the time, we do worst-case analyses to analyze algorithms. In the worst analysis, we guarantee an upper bound on the running time of an algorithm which is good information.

2. Average Case Analysis

The average case analysis is not easy to do in most practical cases and it is rarely done. In the average case analysis, we must know (or predict) the mathematical distribution of all possible inputs.

3. Best Case Analysis

The Best Case analysis is bogus. Guaranteeing a lower bound on an algorithm doesn't provide any information as in the worst case, an algorithm may take years to run.

Interesting information about asymptotic notations:

A) For some algorithms, all the cases (worst, best, average) are asymptotically the same. i.e., there are no worst and best cases.

- **Example:** Merge Sort does Θ(n log(n)) operations in all cases.

B) Where as most of the other sorting algorithms have worst and best cases.

- **Example 1:** In the typical implementation of Quick Sort (where pivot is chosen as a corner element), the worst occurs when the input array is already sorted and the best occurs when the pivot elements always divide the array into two halves.
- **Example 2:** For insertion sort, the worst case occurs when the array is reverse sorted and the best case occurs when the array is sorted in the same order as output.

Examples with their complexity analysis:

1. Linear search algorithm:

```
// C implementation of the approach

#include <stdio.h>


// Linearly search x in arr[].

// If x is present then return the index,

// otherwise return -1

int search(int arr[], int n, int x)

{

    int i;

    for (i = 0; i < n; i++) {

        if (arr[i] == x)
```

```
            return i;
    }
    return -1;
}

/* Driver's code*/
int main()
{
    int arr[] = { 1, 10, 30, 15 };
    int x = 30;
    int n = sizeof(arr) / sizeof(arr[0]);

    // Function call
    printf("%d is present at index %d", x,
           search(arr, n, x));

    getchar();
    return 0;
}
```

Output

```
30 is present at index 2
```

Time Complexity Analysis: (In Big-O notation)

- **Best Case:** O(1), This will take place if the element to be searched is on the first index of the given list. So, the number of comparisons, in this case, is 1.
- **Average Case:** O(n), This will take place if the element to be searched is on the middle index of the given list.
- **Worst Case:** O(n), This will take place if:
  - The element to be searched is on the last index
  - The element to be searched is not present on the list

2. In this example, we will take an array of length (n) and deals with the following cases :

- If (n) is even then our output will be 0
- If (n) is odd then our output will be the sum of the elements of the array.

Below is the implementation of the given problem:

```
#Python 3 implementation of the approach
```

```
def getsum(arr, n):
    if n % 2 == 0: # if (n) is even
        return 0

    Sum = 0
    for i in range(n):
        Sum += arr[i]
    return Sum # if (n) is odd

#Driver's Code
if __name__ == '__main__':
    arr1 = [1,2,3,4] # Declaring an array of even length
    n1 = len(arr1)
    arr2 = [1,2,3,4,5] # Declaring an array of odd length
    n2 = len(arr2)

#Function call
    print(getsum(arr1,n1)) # print 0 because (n) is even

    print(getsum(arr2,n2)) # print sum of array because (n) is odd

#This code is contributed by Syed Maruf Ali
```

Output

```
0
15
```

Time Complexity Analysis:

- **Best Case:** The order of growth will be **constant** because in the best case we are assuming that (n) is even.
- **Average Case:** In this case, we will assume that even and odd are equally likely, therefore Order of growth will be **linear**
- **Worst Case:** The order of growth will be **linear** because in this case, we are assuming that (n) is always odd.

Asymptotic Analysis (Based on input size) in Complexity Analysis of Algorithms

Asymptotic Analysis is defined as the big idea that handles the above issues in analyzing algorithms. In Asymptotic Analysis, we **evaluate the performance of an algorithm in terms of input size** (we don't measure the actual running time). We calculate, how the time (or space) taken by an algorithm increases with the input size.

Why performance analysis?

There are many important things that should be taken care of, like user-friendliness, modularity, security, maintainability, etc. Why worry about performance? The answer to this is simple, we can have all the above things only if we have performance. So performance is like currency through which we can buy all the above things. Another reason for studying performance is – speed is fun! To summarize, performance == scale. Imagine a text editor that can load 1000 pages, but can spell check 1 page per minute OR an image editor that takes 1 hour to rotate your image 90 degrees left OR … you get it. If a software feature can not cope with the scale of tasks users need to perform – it is as good as dead.

Given two algorithms for a task, how do we find out which one is better?

One naive way of doing this is – to implement both the algorithms and run the two programs on your computer for different inputs and see which one takes less time. There are many problems with this approach for the analysis of algorithms.

- It might be possible that for some inputs, the first algorithm performs better than the second. And for some inputs second performs better.
- It might also be possible that for some inputs, the first algorithm performs better on one machine, and the second works better on another machine for some other inputs.

Asymptotic Analysis is the big idea that handles the above issues in analyzing algorithms. In Asymptotic Analysis, we **evaluate the performance of an algorithm in terms of input size** (we don't measure the actual running time). We calculate, how the time (or space) taken by an algorithm increases with the input size.

For example, let us consider the search problem (searching a given item) in a sorted array.

The solution to above search problem includes:

- **Linear Search** (order of growth is linear)
- **Binary Search** (order of growth is logarithmic).

To understand how Asymptotic Analysis solves the problems mentioned above in analyzing algorithms,

- let us say:

    - we run the Linear Search on a fast computer A and
    - Binary Search on a slow computer B and
    - pick the constant values for the two computers so that it tells us exactly how long it takes for the given machine to perform the search in seconds.
- Let's say the constant for A is 0.2 and the constant for B is 1000 which means that A is 5000 times more powerful than B.
- For small values of input array size n, the fast computer may take less time.
- **But, after a certain value of input array size, the Binary Search will definitely start taking less time compared to the Linear Search even though the Binary Search is being run on a slow machine**.

Input Size	Running time on A	Running time on B
10	2 sec	~ 1 h
100	20 sec	~ 1.8 h
10^6	~ 55.5 h	~ 5.5 h
10^9	~ 6.3 years	~ 8.3 h

- The reason is the order of growth of Binary Search with respect to input size is logarithmic while the order of growth of Linear Search is linear.
- **So the machine-dependent constants can always be ignored after a certain value of input size.**

Running times for this example:

- Linear Search running time in seconds on A: 0.2 * n
- Binary Search running time in seconds on B: 1000*log(n)

Does Asymptotic Analysis always work?

Asymptotic Analysis is not perfect, but that's the best way available for analyzing algorithms. For example, say there are two sorting algorithms that take 1000nLogn and 2nLogn time respectively on a machine. Both of these algorithms are asymptotically the same (order of growth is nLogn). So, With Asymptotic Analysis, we can't judge which one is better as we ignore constants in Asymptotic Analysis.

Also, in Asymptotic analysis, we always talk about input sizes larger than a constant value. It might be possible that those large inputs are never given to your software and an asymptotically slower algorithm always performs better for your particular situation. So, you may end up choosing an algorithm that is Asymptotically slower but faster for your software.

Please write comments if you find anything incorrect, or if you want to share more information about the topic discussed above

What are Asymptotic Notations in Complexity Analysis of Algorithms

We have discussed Asymptotic Analysis, and Worst, Average, and Best Cases of Algorithms. The main idea of asymptotic analysis is to have a measure of the efficiency of algorithms that don't depend on machine-specific constants and don't require algorithms to be implemented and time taken by programs to be compared. Asymptotic notations are mathematical tools to represent the time complexity of algorithms for asymptotic analysis.

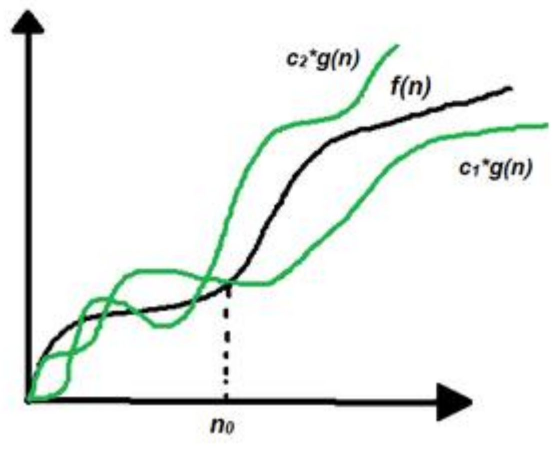
There are mainly three asymptotic notations:

1. **Big-O Notation (O-notation)**
2. **Omega Notation (Ω-notation)**
3. **Theta Notation (Θ-notation)**

1. Theta Notation (Θ-Notation):

Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the **average-case** complexity of an algorithm.

Let g and f be the function from the set of natural numbers to itself. The function f is said to be Θ(g), if there are constants c1, c2 > 0 and a natural number n0 such that c1* g(n) ≤ f(n) ≤ c2 * g(n) for all n ≥ n0



Theta notation

Mathematical Representation of Theta notation:

Θ (g(n)) = {f(n): there exist positive constants c1, c2 and n0 such that 0 ≤ c1 * g(n) ≤ f(n) ≤ c2 * g(n) for all n ≥ n0}

Note: Θ(g) is a set

The above expression can be described as if f(n) is theta of g(n), then the value f(n) is always between c1 * g(n) and c2 * g(n) for large values of n ($n \geq n0$). The definition of theta also requires that f(n) must be non-negative for values of n greater than n0.

A simple way to get the Theta notation of an expression is to drop low-order terms and ignore leading constants. For example**,** Consider the expression **$3n^3 + 6n^2 + 6000 = \Theta(n^3)$**, the dropping lower order terms is always fine because there will always be a number(n) after which $\Theta(n^3)$ has higher values than $\Theta(n^2)$ irrespective of the constants involved. For a given function g(n), we denote $\Theta(g(n))$ is following set of functions.

Examples :

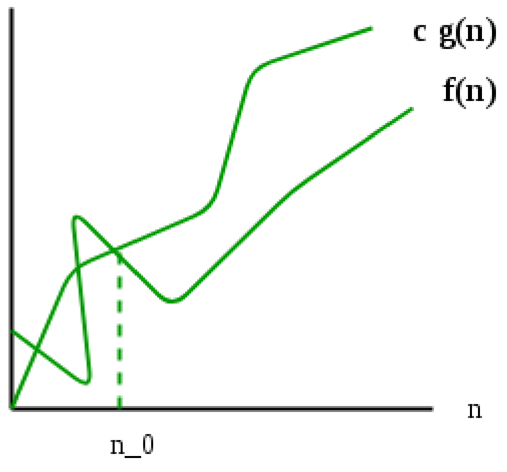
{ 100 , log (2000) , 10^4 } belongs to **$\Theta(1)$**
{ (n/4) , (2n+3) , (n/100 + log(n)) } belongs to **$\Theta(n)$**
{ (n^2+n) , (2n^2) , (n^2+log(n))} belongs to **$\Theta(n^2)$**

Note: Θ provides exact bounds.

2. Big-O Notation (O-notation):

Big-O notation represents the upper bound of the running time of an algorithm. Therefore, it gives the worst-case complexity of an algorithm.

If f(n) describes the running time of an algorithm, f(n) is O(g(n)) if there exist a positive constant C and n0 such that, $0 \leq f(n) \leq cg(n)$ for all $n \geq n0$



f(n) = O(g(n))

Mathematical Representation of Big-O Notation:

O(g(n)) = { f(n): there exist positive constants c and n0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n0$ }

For example**,** Consider the case of Insertion Sort. It takes linear time in the best case and quadratic time in the worst case. We can safely say that the time complexity of the Insertion

sort is O(n^2).
Note: O(n^2) also covers linear time.

If we use Θ notation to represent the time complexity of Insertion sort, we have to use two statements for best and worst cases:

- The worst-case time complexity of Insertion Sort is Θ(n^2).
- The best case time complexity of Insertion Sort is Θ(n).

The Big-O notation is useful when we only have an upper bound on the time complexity of an algorithm. Many times we easily find an upper bound by simply looking at the algorithm.

Examples :

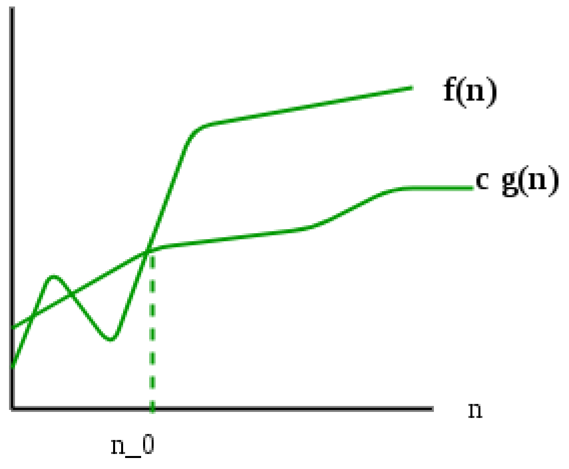
{ 100 , log (2000) , 10^4 } belongs to **O(1)**
U { (n/4) , (2n+3) , (n/100 + log(n)) } belongs to **O(n)**
U { (n^2+n) , (2n^2) , (n^2+log(n))} belongs to **O(n^2)**

Note: Here, **U represents union**, we can write it in these manner because **O provides exact or upper bounds .**

3. Omega Notation (Ω-Notation):

Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm.

Let g and f be the function from the set of natural numbers to itself. The function f is said to be Ω(g), if there is a constant c > 0 and a natural number n0 such that c*g(n) ≤ f(n) for all n ≥ n0



f(n) = Omega(g(n))

Mathematical Representation of Omega notation :

Ω(g(n)) = { f(n): there exist positive constants c and n0 such that 0 ≤ cg(n) ≤ f(n) for all n ≥ n0 }

Let us consider the same Insertion sort example here. The time complexity of Insertion Sort can be written as Ω(n), but it is not very useful information about insertion sort, as we are generally interested in worst-case and sometimes in the average case.

Examples :

{ (n^2+n) , (2n^2) , (n^2+log(n))} belongs to **Ω(n^2)**
U { (n/4) , (2n+3) , (n/100 + log(n)) } belongs to **Ω(n)**
U { 100 , log (2000) , 10^4 } belongs to **Ω(1)**

Note: Here, **U represents union,** we can write it in these manner because **Ω provides exact or lower bounds.**

Properties of Asymptotic Notations:

1. General Properties:

If **f(n)** is **O(g(n))** then **a*f(n)** is also **O(g(n))**, where **a** is a constant.

Example:

$f(n) = 2n^2+5$ is $O(n^2)$
then, $7*f(n) = 7(2n^2+5) = 14n^2+35$ is also $O(n^2)$.

Similarly, this property satisfies both Θ and Ω notation.

We can say,

If f(n) is Θ(g(n)) then a*f(n) is also Θ(g(n)), where a is a constant.
If f(n) is Ω (g(n)) then a*f(n) is also Ω (g(n)), where a is a constant.

2. Transitive Properties:

If **f(n)** is **O(g(n))** and **g(n)** is **O(h(n))** then **f(n) = O(h(n))**.

Example:

If $f(n) = n$, $g(n) = n^2$ and $h(n)=n^3$
n is $O(n^2)$ and n^2 is $O(n^3)$ then, n is $O(n^3)$

Similarly, this property satisfies both Θ and Ω notation.

We can say,

If f(n) is Θ(g(n)) and g(n) is Θ(h(n)) then f(n) = Θ(h(n)) .
If f(n) is Ω (g(n)) and g(n) is Ω (h(n)) then f(n) = Ω (h(n))

3. Reflexive Properties:

Reflexive properties are always easy to understand after transitive.
If f(n) is given then f(n) is O(f(n)). Since *MAXIMUM VALUE OF f(n) will be f(n) ITSELF!*
Hence x = f(n) and y = O(f(n) tie themselves in reflexive relation always.

Example:

$f(n) = n^2$; $O(n^2)$ i.e O(f(n))
Similarly, this property satisfies both Θ and Ω notation.

We can say that,

If f(n) is given then f(n) is Θ(f(n)).
If f(n) is given then f(n) is Ω (f(n)).

4. Symmetric Properties:

If **f(n)** is **Θ(g(n))** then **g(n)** is **Θ(f(n))**.

Example:

If(n) = n^2 and g(n) = n^2
then, f(n) = $\Theta(n^2)$ and g(n) = $\Theta(n^2)$

This property only satisfies for Θ notation.

5. Transpose Symmetric Properties:

If **f(n)** is **O(g(n))** then **g(n)** is **Ω (f(n))**.

Example:

If(n) = n , g(n) = n^2
then n is $O(n^2)$ and n^2 is Ω (n)

This property only satisfies O and Ω notations.

6. Some More Properties:

1. If **f(n) = O(g(n))** and **f(n) = Ω(g(n))** then **f(n) = Θ(g(n))**
2. If **f(n) = O(g(n))** and **d(n)=O(e(n))** then **f(n) + d(n) = O(max(g(n), e(n)))**

Example:

f(n) = n i.e O(n)
d(n) = n^2 i.e $O(n^2)$
then f(n) + d(n) = n + n^2 i.e $O(n^2)$

3. If **$f(n)=O(g(n))$** and **$d(n)=O(e(n))$** then **$f(n) * d(n) = O(g(n) * e(n))$**

Example:

$f(n) = n$ i.e $O(n)$
$d(n) = n^2$ i.e $O(n^2)$
then $f(n) * d(n) = n * n^2 = n^3$ i.e $O(n^3)$

Note: If $f(n) = O(g(n))$ then $g(n) = \Omega(f(n))$

Time-Space Trade-Off in Algorithms

In this article, we will discuss Time-Space Trade-Off in Algorithms. A trade-off is a situation where one thing increases and another thing decreases. It is a way to solve a problem in:

- Either in less time and by using more space, or
- In very little space by spending a long amount of time.

The best Algorithm is that which helps to solve a problem that requires less space in memory and also takes less time to generate the output. But in general, it is not always possible to achieve both of these conditions at the same time. The most common condition is an algorithm using a lookup table. This means that the answers to some questions for every possible value can be written down. One way of solving this problem is to write down the entire **lookup table**, which will let you find answers very quickly but will use a lot of space. Another way is to calculate the answers without writing down anything, which uses very little space, but might take a long time. Therefore, the more time-efficient algorithms you have, that would be less space-efficient.

Types of Space-Time Trade-off

- Compressed or Uncompressed data
- Re Rendering or Stored images
- Smaller code or loop unrolling
- Lookup tables or Recalculation

Compressed or Uncompressed data: A space-time trade-off can be applied to the problem of **data storage**. If data stored is uncompressed, it takes more space but less time. But if the data is stored compressed, it takes less space but more time to run the decompression algorithm. There are many instances where it is possible to directly work with compressed data. In that case of compressed bitmap indices, where it is faster to work with compression than without compression.

Re-Rendering or Stored images: In this case, storing only the source and rendering it as an image would take more space but less time i.e., storing an image in the cache is faster than re-rendering but requires more space in memory.

Smaller code or Loop Unrolling: Smaller code occupies less space in memory but it requires high computation time that is required for jumping back to the beginning of the loop at the end of each iteration. Loop unrolling can optimize execution speed at the cost of increased binary size. It occupies more space in memory but requires less computation time.

Lookup tables or Recalculation: In a lookup table, an implementation can include the entire table which reduces computing time but increases the amount of memory needed. It can recalculate i.e., compute table entries as needed, increasing computing time but reducing memory requirements.

For Example: In mathematical terms, the sequence $\mathbf{F_n}$ of the Fibonacci Numbers is defined by the recurrence relation:

$F_n = F_{n-1} + F_{n-2}$,
where, $F_0 = 0$ and $F_1 = 1$.

A simple solution to find the **Nth Fibonacci term** using recursion from the above recurrence relation.

Below is the implementation using recursion:

```
# Python3 program to find Nth Fibonacci

# number using recursion




# Function to find Nth Fibonacci term

def Fibonacci(N:int):

    # Base Case

    if (N < 2):

        return N


    # Recursively computing the term

    # using recurrence relation

    return Fibonacci(N - 1) + Fibonacci(N - 2)




# Driver Code

if __name__ == '__main__':

    N = 5


    # Function Call
    print(Fibonacci(N))
```

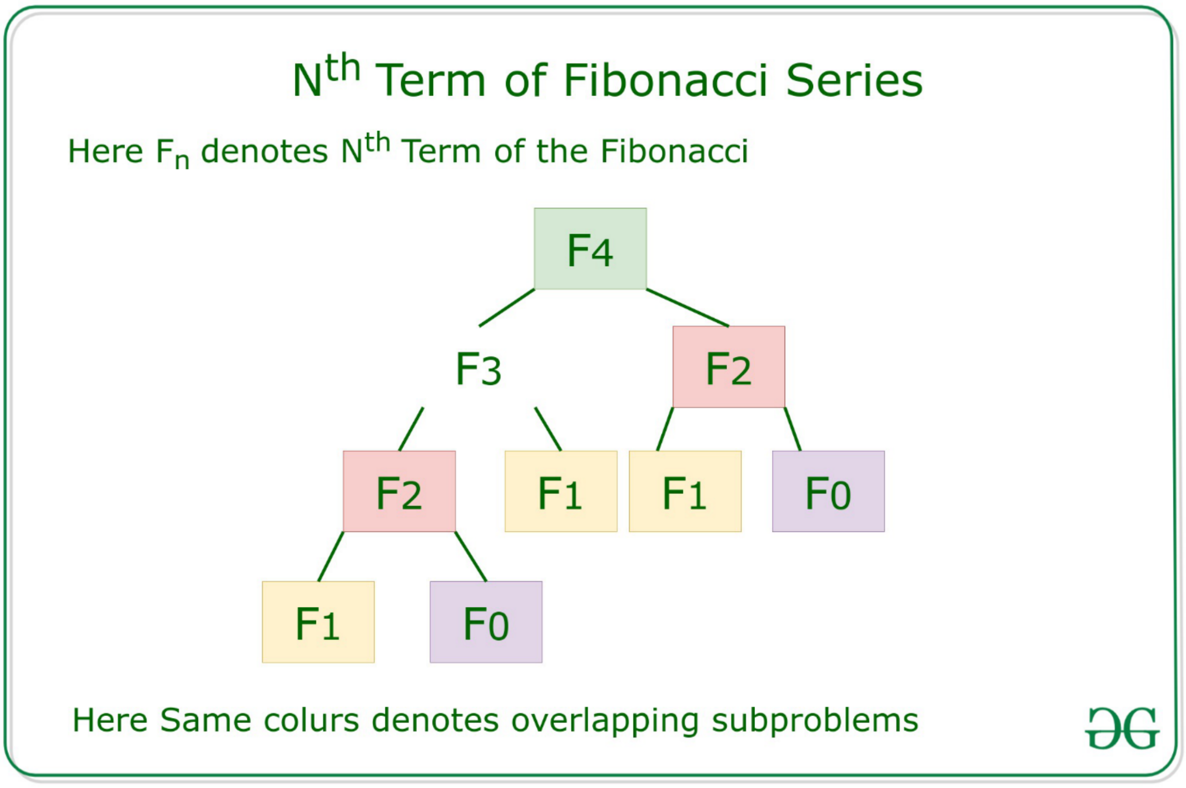
Output:

```
5
```

Time Complexity: *O(2^N)*
Auxiliary Space: *O(1)*

Explanation: The time complexity of the above implementation is exponential due to multiple calculations of the same subproblems again and again. The auxiliary space used is minimum. But our goal is to reduce the time complexity of the approach even it requires extra space. Below is the Optimized approach discussed.

Efficient Approach: To optimize the above approach, the idea is to use Dynamic Programming to reduce the complexity by memoization of the overlapping subproblems as shown in the below recursion tree:


N th Term of Fibonacci Series
Here Fn denotes Nth Term of the Fibonacci
F4
F3
F2
F2
F1
F1
F0
F1
F0
Here Same colurs denotes overlapping subproblems

Below is the implementation of the above approach:

```
# Python3 program to find Nth Fibonacci

# number using recursion


# Function to find Nth Fibonacci term
```

```
def Fibonacci(N):
    f=[0]*(N + 2)

    # 0th and 1st number of the
    # series are 0 and 1
    f[0] = 0
    f[1] = 1

    # Iterate over the range [2, N]
    for i in range(2,N+1) :

        # Add the previous 2 numbers
        # in the series and store it
        f[i] = f[i - 1] + f[i - 2]



    # Return Nth Fibonacci Number
    return f[N]



# Driver Code
if __name__ == '__main__':
    N = 5

    # Function Call
    print(Fibonacci(N))
```

Output:

```
5
```

Time Complexity: *O(N)*
Auxiliary Space: *O(N)*

Explanation: The time complexity of the above implementation is linear by using an auxiliary space for storing the overlapping subproblems states so that it can be used further when required.

Analysis of Algorithms | Set 2 (Worst, Average and Best Cases)

In the previous post, we discussed how Asymptotic analysis overcomes the problems of the naive way of analyzing algorithms. But let's take an overview of the **asymptotic notation** and learn about What is Worst, Average, and Best cases of an algorithm:

Popular Notations in Complexity Analysis of Algorithms

1. Big-O Notation

We define an algorithm's **worst-case** time complexity by using the Big-O notation, which determines the set of functions grows slower than or at the same rate as the expression. Furthermore, it explains the maximum amount of time an algorithm requires considering all input values.

2. Omega Notation

It defines the **best case** of an algorithm's time complexity, the Omega notation defines whether the set of functions will grow faster or at the same rate as the expression. Furthermore, it explains the minimum amount of time an algorithm requires considering all input values.

3. Theta Notation

It defines the **average case** of an algorithm's time complexity, the Theta notation defines when the set of functions lies in both **O(expression)** and **Omega(expression)**, then Theta notation is used. This is how we define a time complexity average case for an algorithm.

Measurement of Complexity of an Algorithm

Based on the above three notations of Time Complexity there are three cases to analyze an algorithm:

1. Worst Case Analysis (Mostly used)

In the worst-case analysis, we calculate the upper bound on the running time of an algorithm. We must know the case that causes a maximum number of operations to be executed. For Linear Search, the worst case happens when the element to be searched (x) is not present in the array. When x is not present, the search() function compares it with all the elements of arr[] one by one. Therefore, the worst-case time complexity of the linear search would be **O(n)**.

2. Best Case Analysis (Very Rarely used)

In the best case analysis, we calculate the lower bound on the running time of an algorithm. We must know the case that causes a minimum number of operations to be executed. In the linear search problem, the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on n). So time complexity in the best case would be Ω(1)

3. Average Case Analysis (Rarely used)

In average case analysis, we take all possible inputs and calculate the computing time for all of the inputs. Sum all the calculated values and divide the sum by the total number of inputs. We must know (or predict) the distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed (including the case of x not being present in the array). So we sum all the cases and divide the sum by (n+1). Following is the value of average-case time complexity.

Average Case Time = \sum_{i=1}^{n}\frac{\theta (i)}{(n+1)} = \frac{\theta (\frac{(n+1)*(n+2)}{2})}{(n+1)} = \theta (n)

Which Complexity analysis is generally used?

Below is the ranked mention of complexity analysis notation based on popularity:

1. Worst Case Analysis:

Most of the time, we do worst-case analyses to analyze algorithms. In the worst analysis, we guarantee an upper bound on the running time of an algorithm which is good information.

2. Average Case Analysis

The average case analysis is not easy to do in most practical cases and it is rarely done. In the average case analysis, we must know (or predict) the mathematical distribution of all possible inputs.

3. Best Case Analysis

The Best Case analysis is bogus. Guaranteeing a lower bound on an algorithm doesn’t provide any information as in the worst case, an algorithm may take years to run.

Interesting information about asymptotic notations:

A) For some algorithms, all the cases (worst, best, average) are asymptotically the same. i.e., there are no worst and best cases.

- **Example:** Merge Sort does Θ(n log(n)) operations in all cases.

B) Where as most of the other sorting algorithms have worst and best cases.

- **Example 1:** In the typical implementation of Quick Sort (where pivot is chosen as a corner element), the worst occurs when the input array is already sorted and the best occurs when the pivot elements always divide the array into two halves.
- **Example 2:** For insertion sort, the worst case occurs when the array is reverse sorted and the best case occurs when the array is sorted in the same order as output.

Examples with their complexity analysis:

1. Linear search algorithm:

```
# Python 3 implementation of the approach


# Linearly search x in arr[]. If x is present

# then return the index, otherwise return -1



def search(arr, x):

    for index, value in enumerate(arr):

        if value == x:

            return index

    return -1



# Driver's Code

if __name__ == '__main__':

    arr = [1, 10, 30, 15]

    x = 30


    # Function call

    print(x, "is present at index",

          search(arr, x))
```

Output

```
30 is present at index 2
```

Time Complexity Analysis: (In Big-O notation)

- **Best Case:** O(1), This will take place if the element to be searched is on the first index of the given list. So, the number of comparisons, in this case, is 1.
- **Average Case:** O(n), This will take place if the element to be searched is on the middle index of the given list.
- **Worst Case:** O(n), This will take place if:
    - The element to be searched is on the last index
    - The element to be searched is not present on the list

2. In this example, we will take an array of length (n) and deals with the following cases :

- If (n) is even then our output will be 0
- If (n) is odd then our output will be the sum of the elements of the array.

Below is the implementation of the given problem:

```
# Python 3 implementation of the approach



def getsum(arr, n):

    if n % 2 == 0:  # if (n) is even

        return 0


    Sum = 0

    for i in range(n):

        Sum += arr[i]

    return Sum # if (n) is odd




# Driver's Code

if __name__ == '__main__':

arr1 = [1, 2, 3, 4]  # Declaring an array of even length

n1 = len(arr1)

arr2 = [1, 2, 3, 4, 5]  # Declaring an array of odd length

n2 = len(arr2)
```

```
# Function call

print(getsum(arr1, n1))  # print 0 because (n) is even


print(getsum(arr2, n2))  # print sum of array because (n) is odd


# This code is contributed by Syed Maruf Ali
```

Output

```
0
15
```

Time Complexity Analysis:

- **Best Case:** The order of growth will be **constant** because in the best case we are assuming that (n) is even.
- **Average Case:** In this case, we will assume that even and odd are equally likely, therefore Order of growth will be **linear**
- **Worst Case:** The order of growth will be **linear** because in this case, we are assuming that (n) is always odd.

For more details, please refer: Design and Analysis of Algorithms. Please write comments if you find anything incorrect, or if you want to share more information about the topic discussed above.

How to Analyse Loops for Complexity Analysis of Algorithms

We have discussed Asymptotic Analysis, Worst, Average and Best Cases and Asymptotic Notations in previous posts. In this post, an analysis of iterative programs with simple examples is discussed.

Constant Time Complexity O(1):

The time complexity of a function (or set of statements) is considered as O(1) if it doesn’t contain a loop, recursion, and call to any other non-constant time function.
 i.e. set of non-recursive and non-loop statements

Example:

- swap() function has O(1) time complexity.
- A loop or recursion that runs a constant number of times is also considered O(1). For example, the following loop is O(1).

```
// Here c is a constant
for (int i = 1; i <= c; i++) {
    // some O(1) expressions
}
```

Linear Time Complexity O(n):

The Time Complexity of a loop is considered as O(n) if the loop variables are incremented/decremented by a constant amount. For example following functions have O(n) time complexity.

```
// Here c is a positive integer constant
for (int i = 1; i <= n; i += c) {
    // some O(1) expressions
}


for (int i = n; i > 0; i -= c) {
    // some O(1) expressions
}
```

Quadratic Time Complexity O(nc):

The time complexity is defined as an algorithm whose performance is directly proportional to the squared size of the input data, as in nested loops it is equal to the number of times the innermost statement is executed. For example, the following sample loops have O(n^2) time complexity

```
for (int i = 1; i <= n; i += c) {
    for (int j = 1; j <= n; j += c) {
        // some O(1) expressions
    }
}


for (int i = n; i > 0; i -= c) {
    for (int j = i + 1; j <= n; j += c) {
        // some O(1) expressions
    }
```

Example: Selection sort and Insertion Sort have O(n^2) time complexity.

Logarithmic Time Complexity O(Log n):

The time Complexity of a loop is considered as O(Logn) if the loop variables are divided/multiplied by a constant amount. And also for recursive calls in the recursive function, the Time Complexity is considered as O(Logn).

```
for (int i = 1; i <= n; i *= c) {
    // some O(1) expressions
}
for (int i = n; i > 0; i /= c) {
    // some O(1) expressions
}
// Recursive function
void recurse(n)
{
    if (n == 0)
        return;
    else {
        // some O(1) expressions
    }
    recurse(n - 1);
}
```

Example: Binary Search(refer iterative implementation) has O(Logn) time complexity.

Logarithmic Time Complexity O(Log Log n):

The Time Complexity of a loop is considered as O(LogLogn) if the loop variables are reduced/increased exponentially by a constant amount.

```
// Here c is a constant greater than 1
for (int i = 2; i <= n; i = pow(i, c)) {
    // some O(1) expressions
}
// Here fun is sqrt or cuberoot or any other constant root
for (int i = n; i > 1; i = fun(i)) {
    // some O(1) expressions
}
```

See this for mathematical details.

How to combine the time complexities of consecutive loops?

When there are consecutive loops, we calculate time complexity as a sum of the time complexities of individual loops.

```
for (int i = 1; i <= m; i += c) {

    // some O(1) expressions

}

for (int i = 1; i <= n; i += c) {

    // some O(1) expressions

}


Time complexity of above code is O(m)

        + O(n) which is O(m + n) If m

    == n,

    the time complexity becomes O(2n) which is O(n).
```

How to calculate time complexity when there are many if, else statements inside loops?

As discussed here, the worst-case time complexity is the most useful among best, average and worst. Therefore we need to consider the worst case. We evaluate the situation when values in if-else conditions cause a maximum number of statements to be executed.
For example, consider the linear search function where we consider the case when an element is present at the end or not present at all.
When the code is too complex to consider all if-else cases, we can get an upper bound by ignoring if-else and other complex control statements.

How to calculate the time complexity of recursive functions?

The time complexity of a recursive function can be written as a mathematical recurrence relation. To calculate time complexity, we must know how to solve recurrences. We will soon be discussing recurrence-solving techniques as a separate post.

Algorithms Cheat Sheet:

Algorithm	Best Case	Average Case	Worst Case
Selection Sort	O(n^2)	O(n^2)	O(n^2)
Bubble Sort	O(n)	O(n^2)	O(n^2)
Insertion Sort	O(n)	O(n^2)	O(n^2)

Tree Sort	O(nlogn)	O(nlogn)	O(n^2)
Radix Sort	O(dn)	O(dn)	O(dn)
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)
Heap Sort	O(nlogn)	O(nlogn)	O(nlogn)
Quick Sort	O(nlogn)	O(nlogn)	O(n^2)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)
Counting Sort	O(n+k)	O(n+k)	O(n+k)

Quiz on Analysis of Algorithms
For more details, please refer: Design and Analysis of Algorithms.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Analysis of recursive algorithms through recurrence relations:

- **Substitution method,**
- **Recursion tree method**
- **Masters' theorem.**

In the previous post, we discussed the analysis of loops. Many algorithms are recursive. When we analyze them, we get a recurrence relation for time complexity. We get running time on an input of size n as a function of n and the running time on inputs of smaller sizes. For example in Merge Sort, to sort a given array, we divide it into two halves and recursively repeat the process for the two halves. Finally, we merge the results. Time complexity of Merge Sort can be written as T(n) = 2T(n/2) + cn. There are many other algorithms like Binary Search, Tower of Hanoi, etc.

There are mainly three ways of solving recurrences:

Substitution Method:

We make a guess for the solution and then we use mathematical induction to prove the guess is correct or incorrect.

For example consider the recurrence T(n) = 2T(n/2) + n

We guess the solution as T(n) = O(nLogn). Now we use induction to prove our guess.

We need to prove that T(n) <= cnLogn. We can assume that it is true for values smaller than n.

```
T(n) = 2T(n/2) + n
     <= 2cn/2Log(n/2) + n
       =  cnLogn – cnLog2 + n
       =  cnLogn – cn + n
    <= cnLogn
```

Recurrence Tree Method:

In this method, we draw a recurrence tree and calculate the time taken by every level of the tree. Finally, we sum the work done at all levels. To draw the recurrence tree, we start from the given recurrence and keep drawing till we find a pattern among levels. The pattern is typically arithmetic or geometric series.

For example, consider the recurrence relation

$T(n) = T(n/4) + T(n/2) + cn^2$

```
            $cn^2$
        /       \
  T(n/4)     T(n/2)
```

If we further break down the expression T(n/4) and T(n/2),
we get the following recursion tree.

```
                    $cn^2$
              /              \
    $c(n^2)/16$          $c(n^2)/4$
   /         \            /         \
T(n/16)  T(n/8)  T(n/8)    T(n/4)
```

Breaking down further gives us following

```
                       $cn^2$
                /                 \
       $c(n^2)/16$              $c(n^2)/4$
    /           \                 /           \
$c(n^2)/256$  $c(n^2)/64$  $c(n^2)/64$   $c(n^2)/16$
 /    \             /    \       /    \         /    \
```

To know the value of T(n), we need to calculate the sum of tree
nodes level by level. If we sum the above tree level by level,

we get the following series T(n) = c(n^2 + 5(n^2)/16 + 25(n^2)/256) + ….
The above series is a geometrical progression with a ratio of 5/16.

To get an upper bound, we can sum the infinite series. We get the sum as $(n^2)/(1 – 5/16)$ which is $O(n^2)$

Master Method:

Master Method is a direct way to get the solution. The master method works only for the following type of recurrences or for recurrences that can be transformed into the following type.

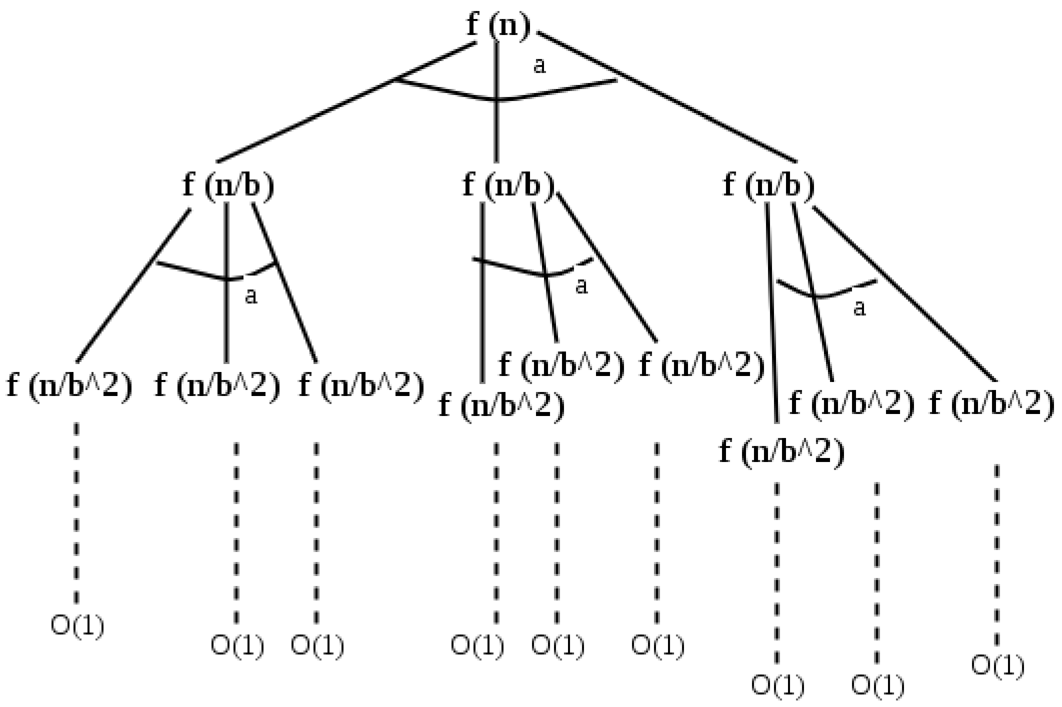
T(n) = aT(n/b) + f(n) where a >= 1 and b > 1

There are the following three cases:

- If $f(n) = O(n^c)$ where $c < \mathrm{Log}_b a$ then $T(n) = \Theta(n^{\mathrm{Log} ba})$
- If $f(n) = \Theta(n^c)$ where $c = \mathrm{Log}_b a$ then $T(n) = \Theta(n^c \mathrm{Log}\ n)$
- If $f(n) = \Omega(n^c)$ where $c > \mathrm{Log}_b a$ then $T(n) = \Theta(f(n))$

How does this work?

The master method is mainly derived from the recurrence tree method. If we draw the recurrence tree of T(n) = aT(n/b) + f(n), we can see that the work done at the root is f(n), and work done at all leaves is $\Theta(n^c)$ where c is $\mathrm{Log}_b a$. And the height of the recurrence tree is $\mathrm{Log}_b n$



In the recurrence tree method, we calculate the total work done. If the work done at leaves is polynomially more, then leaves are the dominant part, and our result becomes the work done at leaves (Case 1). If work done at leaves and root is asymptotically the same, then our result becomes height multiplied by work done at any level (Case 2). If work done at the root is asymptotically more, then our result becomes work done at the root (Case 3).

Examples of some standard algorithms whose time complexity can be evaluated using the Master Method

- Merge Sort: $T(n) = 2T(n/2) + \Theta(n)$. It falls in case 2 as c is 1 and $Log_b a]$ is also 1. So the solution is $\Theta(n\ Logn)$
- Binary Search: $T(n) = T(n/2) + \Theta(1)$. It also falls in case 2 as c is 0 and $Log_b a$ is also 0. So the solution is $\Theta(Logn)$

Notes:

- It is not necessary that a recurrence of the form $T(n) = aT(n/b) + f(n)$ can be solved using Master Theorem. The given three cases have some gaps between them. For example, the recurrence $T(n) = 2T(n/2) + n/Logn$ cannot be solved using master method.
- Case 2 can be extended for $f(n) = \Theta(n^c Log^k n)$
  If $f(n) = \Theta(n^c Log^k n)$ for some constant $k >= 0$ and $c = Log_b a$, then $T(n) = \Theta(n^c Log^{k+1} n)$

For more details, please refer: Design and Analysis of Algorithms.

Please write comments if you find anything incorrect, or if you want to share more information about the topic discussed above