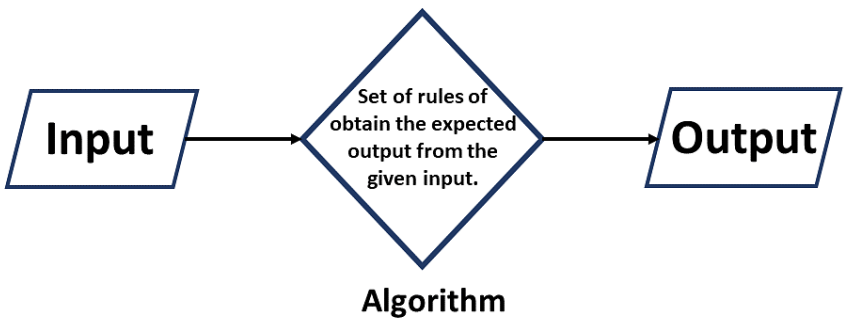
An algorithm is a step-by-step procedure that defines a set of instructions that must be carried out in a specific order to produce the desired result. Algorithms are generally developed independently of underlying languages, which means that an algorithm can be implemented in more than one programming language. Unambiguity, fineness, effectiveness, and language independence are some of the characteristics of an algorithm. The scalability and performance of an algorithm are the primary factors that contribute to its importance.

What is an Algorithm?

- An algorithm is a set of commands that must be followed for a computer to perform calculations or other problem-solving operations.
- According to its formal definition, an algorithm is a finite set of instructions carried out in a specific order to perform a particular task.
- It is not the entire program or code; it is simple logic to a problem represented as an informal description in the form of a flowchart or pseudocode.



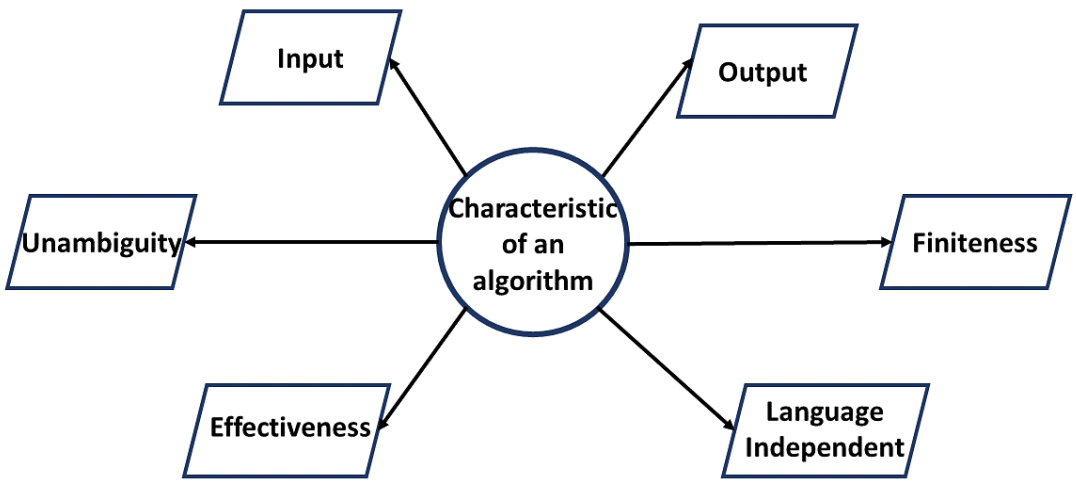
- Problem: A problem can be defined as a real-world problem or real-world instance problem for which you need to develop a program or set of instructions. An algorithm is a set of instructions.

- Algorithm: An algorithm is defined as a step-by-step process that will be designed for a problem.
- Input: After designing an algorithm, the algorithm is given the necessary and desired inputs.
- Processing unit: The input will be passed to the processing unit, producing the desired output.
- Output: The outcome or result of the program is referred to as the output.

After defining what an algorithm is, you will now look at algorithm characteristics.

Characteristics of an Algorithm

An algorithm has the following characteristics:



- Input: An algorithm requires some input values. An algorithm can be given a value other than 0 as input.

- Output: At the end of an algorithm, you will have one or more outcomes.
- Unambiguity: A perfect algorithm is defined as unambiguous, which means that its instructions should be clear and straightforward.
- Finiteness: An algorithm must be finite. Finiteness in this context means that the algorithm should have a limited number of instructions, i.e., the instructions should be countable.
- Effectiveness: Because each instruction in an algorithm affects the overall process, it should be adequate.
- Language independence: An algorithm must be language-independent, which means that its instructions can be implemented in any language and produce the same results.

Moving on in this What is an Algorithm tutorial, you will look at why you need an algorithm.

Why Do You Need an Algorithm?

You require algorithms for the following reasons:

Scalability

It aids in your understanding of scalability. When you have a sizable real-world problem, you must break it down into small steps to analyze it quickly.

Performance

The real world is challenging to break down into smaller steps. If a problem can be easily divided into smaller steps, it indicates that the problem is feasible.

After understanding what is an algorithm, why you need an algorithm, you will look at how to write one using an example.

How to Write an Algorithm?

- There are no well-defined standards for writing algorithms. It is, however, a problem that is resource-dependent. Algorithms are never written with a specific programming language in mind.
- As you all know, basic code constructs such as loops like do, for, while, all programming languages share flow control such as if-else, and so on. An algorithm can be written using these common constructs.
- Algorithms are typically written in a step-by-step fashion, but this is not always the case. Algorithm writing is a process that occurs after the problem domain has been well-defined. That is, you must be aware of the problem domain for which you are developing a solution.

Example

Now, use an example to learn how to write algorithms.

Problem: Create an algorithm that multiplies two numbers and displays the output.

Step 1 − Start

Step 2 − declare three integers x, y & z

Step 3 − define values of x & y

Step 4 − multiply values of x & y

Step 5 − store result of step 4 to z

Step 6 − print z

Step 7 − Stop

Algorithms instruct programmers on how to write code. In addition, the algorithm can be written as:

Step 1 − Start mul

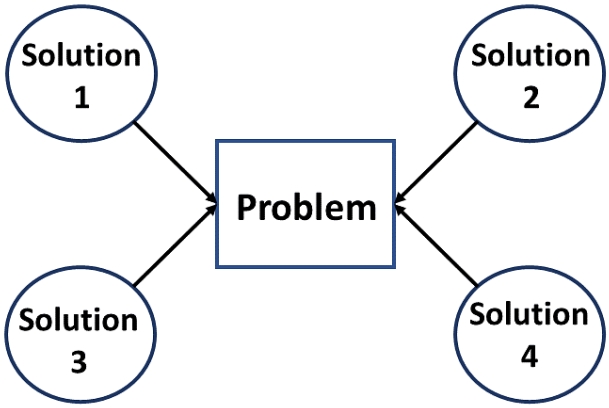
Step 2 − get values of x & y

Step 3 − z ← x * y

Step 4 − display z

Step 5 − Stop

In algorithm design and analysis, the second method is typically used to describe an algorithm. It allows the analyst to analyze the algorithm while ignoring all unwanted definitions easily. They can see which operations are being used and how the process is progressing. It is optional to write step numbers. To solve a given problem, you create an algorithm. A problem can be solved in a variety of ways.



As a result, many solution algorithms for a given problem can be derived. The following step is to evaluate the proposed solution algorithms and implement the most appropriate solution.

As you progress through this "what is an Algorithm" tutorial, you will learn about some of the components of an algorithm.

Factors of an Algorithm

The following are the factors to consider when designing an algorithm:

- Modularity: This feature was perfectly designed for the algorithm if you are given a problem and break it down into small-small modules or small-small steps, which is a basic definition of an algorithm.
- Correctness: An algorithm's correctness is defined as when the given inputs produce the desired output, indicating that the algorithm was designed correctly. An algorithm's analysis has been completed correctly.

- Maintainability: It means that the algorithm should be designed in a straightforward, structured way so that when you redefine the algorithm, no significant changes are made to the algorithm.
- Functionality: It takes into account various logical steps to solve a real-world problem.
- Robustness: Robustness refers to an algorithm's ability to define your problem clearly.
- User-friendly: If the algorithm is difficult to understand, the designer will not explain it to the programmer.
- Simplicity: If an algorithm is simple, it is simple to understand.
- Extensibility: Your algorithm should be extensible if another algorithm designer or programmer wants to use it.

You will now see why an algorithm is so essential after understanding some of its components.

Importance of an Algorithm

There are two factors in which the algorithm is fundamental:

Theoretical Significance

When you are given a real-world problem, you must break it down into smaller modules. To deconstruct the problem, you must first understand all of its theoretical aspects.

Practical Significance

As you all know, theory cannot be completed without practical application. As a result, the significance of algorithms can be considered both theoretically and practically.

As you progress through this "what is an algorthim" tutorial, you will see algorithmic approaches.

Approaches of an Algorithm

Following consideration of both the theoretical and practical importance of designing an algorithm, the following approaches were used:

- **Brute Force Algorithm**

This algorithm uses the general logic structure to design an algorithm. It is also called an exhaustive search algorithm because it exhausts all possibilities to provide the required solution. There are two kinds of such algorithms:

1. Optimizing: Finding all possible solutions to a problem and then selecting the best one, will terminate if the best solution is known.
2. Sacrificing: It will stop as soon as the best solution is found.

- **Divide and Conquer**

This is a straightforward algorithm implementation. It enables you to create an algorithm in a step-by-step fashion. It deconstructs the algorithm to solve the problem in various ways. It allows you to divide the problem into different methods, generating valid output for valid input. This accurate output is forwarded to another function.

- **Greedy Algorithm**

This is an algorithm paradigm that makes the best choice possible on each iteration in the hopes of choosing the best solution. It is simple to set up and has a shorter execution time. However, there are very few cases where it is the best solution.

- **Dynamic Programming**

It improves the efficiency of the algorithm by storing intermediate results. It goes through five steps to find the best solution to the problem:

1. It divides the problem into subproblems to find the best solution.
2. After breaking down the problem into subproblems, it finds the best solution from these subproblems.
3. Memorization is the process of storing the results of subproblems.
4. Reuse the result to prevent it from being recomputed for the same subproblems.
5. Finally, it computes the complex program's output.

- **Branch and Bound Algorithm**

Only integer programming problems can be solved using the branch and bound algorithm. This method divides all feasible solution sets into smaller subsets. These subsets are then evaluated further to find the best solution.

- **Randomized Algorithm**

As with a standard algorithm, you have predefined input and output. Deterministic algorithms have a defined set of information and required results and follow some described steps. They are more efficient than non-deterministic algorithms.

- **Backtracking**

It is an algorithmic procedure that recursively and discards the solution if it does not satisfy the constraints of the problem.

Following your understanding of what is an algorith, and its approaches, you will now look at algorithm analysis.

Analysis of an Algorithm

The algorithm can be examined at two levels: before and after it is created. The two algorithm analyses are as follows:

- **Priori Analysis**

In this context, priori analysis refers to the theoretical analysis of an algorithm performed before implementing the algorithm. Before implementing the algorithm, various factors such as processor speed, which does not affect the implementation, can be considered.

- **Posterior Analysis**

In this context, posterior analysis refers to a practical analysis of an algorithm. The algorithm is implemented in any programming language to perform the experimental research. This analysis determines how much running time and space is required.

Moving on in this "what is an algorithm" tutorial, you will now look at the complexity of an algorithm.

The Complexity of an Algorithm

The algorithm's performance can be measured in two ways:

Time Complexity

The amount of time required to complete an algorithm's execution is called time complexity. The big O notation is used to represent an algorithm's time complexity. The asymptotic notation for describing time complexity, in this case, is big O notation. The time complexity is calculated primarily by counting

the number of steps required to complete the execution. Let us look at an example of time complexity.

```
mul = 1;

// Suppose you have to calculate the multiplication of n numbers.

for i=1 to n

mul = mul *1;

// when the loop ends, then mul holds the multiplication of the n numbers

return mul;
```

The time complexity of the loop statement in the preceding code is at least n, and as the value of n escalates, so does the time complexity. While the code's complexity, i.e., returns mul, will be constant because its value is not dependent on the importance of n and will provide the result in a single step. The worst-time complexity is generally considered because it is the maximum time required for any given input size.

Space Complexity

The amount of space an algorithm requires to solve a problem and produce an output is called its space complexity. Space complexity, like time complexity, is expressed in big O notation.

The space is required for an algorithm for the following reasons:

1. To store program instructions.
2. To store track of constant values.

3. To store track of variable values.
4. To store track of function calls, jumping statements, and so on.

Space Complexity = Auxiliary Space + Input Size

Finally after understanding what is an algorithm, its analysis and approches, you will look at different types of algorithms.

Types of Algorithms

There are two types of algorithms:

- Search Algorithm
- Sort Algorithm

Search Algorithm

Every day, you look for something in your daily life. Similarly, in the case of a computer, a large amount of data is stored in the computer, and whenever a user requests data, the computer searches for that data in the memory and returns it to the user. There are primarily two methods for searching data in an array:

The searching algorithm is of two types:

- ***Linear Search***

Linear search is a simple algorithm that begins searching for an element or a value at the beginning of an array and continues until the required element is not found. It compares the element to be searched with all the elements in an array; if a match is found, the element index is returned; otherwise, -1 is returned. This algorithm can be applied to an unsorted list.

- ***Binary Search***

A binary algorithm is the most basic algorithm, and it searches for elements very quickly. It is used to find an element in a sorted list. To implement the binary algorithm, the elements must be stored in sequential order or sorted. If the elements are stored randomly, binary search cannot be implemented.

Sort Algorithm

Sorting algorithms rearrange elements in an array or a given data structure in ascending or descending order. The comparison operator decides the new order of the elements.

Now that you have completed the tutorial on "what is an algorithm," you will summarise what you have learned so far.

What is algorithm and why analysis of it is important?

In the analysis of the algorithm, it generally focused on CPU (time) usage, Memory usage, Disk usage, and Network usage. All are important, but the most concern is about the CPU time. Be careful to differentiate between:

- **Performance:** How much time/memory/disk/etc. is used when a program is run. This depends on the machine, compiler, etc. as well as the code we write.
- **Complexity:** How do the resource requirements of a program or algorithm scale, i.e. what happens as the size of the problem being solved by the code gets larger.

Note: Complexity affects performance but not vice-versa.

Algorithm Analysis:

Algorithm analysis is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem. Analysis of algorithms is the determination of the amount of time and space resources required to execute it.

Why Analysis of Algorithms is important?

- To predict the behavior of an algorithm without implementing it on a specific computer.
- It is much more convenient to have simple measures for the efficiency of an algorithm than to implement the algorithm and test the efficiency every time a certain parameter in the underlying computer system changes.
- It is impossible to predict the exact behavior of an algorithm. There are too many influencing factors.
- The analysis is thus only an approximation; it is not perfect.
- More importantly, by analyzing different algorithms, we can compare them to determine the best one for our purpose.

Types of Algorithm Analysis:

1. Best case
2. Worst case
3. Average case

- **Best case:** Define the input for which algorithm takes less time or minimum time. In the best case calculate the lower bound of an algorithm. Example: In the linear search when search data is present at the first location of large data then the best case occurs.
- **Worst Case**: Define the input for which algorithm takes a long time or maximum time. In the worst calculate the upper bound of an algorithm. Example: In the linear search when search data is not present at all then the worst case occurs.
- **Average case**: In the average case take all random inputs and calculate the computation time for all inputs.

  And then we divide it by the total number of inputs.

Average case = all random case time / total no of case

Asymptotic Notation and 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.

Worst, Average and Best Case Analysis of Algorithms

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 to consider 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 to consider 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
// 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:

```cpp
// C++ implementation of the approach
#include <bits/stdc++.h>
using namespace std;

int getSum(int arr[], int n)
{
    if (n % 2 == 0) // (n) is even
    {
        return 0;
    }
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += arr[i];
    }
    return sum; //  (n) is odd
}

// Driver's Code
int main()
{
    // Declaring two array one of length odd and other of
    // length even;
```

```
    int arr[4] = { 1, 2, 3, 4 };
    int a[5] = { 1, 2, 3, 4, 5 };

    // Function call
    cout << getSum(arr, 4)
         << endl; // print 0 because (n) is even
    cout << getSum(a, 5)
         << endl; // print sum because (n) is odd
}
// This code is contributed by Suruchi Kumari
```

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.

Types of 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.

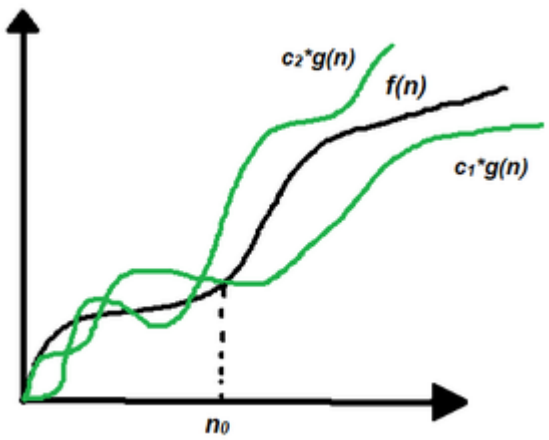
There are mainly three asymptotic notations:

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 ≥ n0). The definition of theta also requires that f(n) must be non-negative for values of n greater than n0.

The execution time serves as both a lower and upper bound on the algorithm's time complexity.

It exist as both, most, and least boundaries for a given input value.

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 = Θ(n^3)**, the dropping lower order terms is always fine because there will always be a number(n) after which Θ(n^3) has higher values than Θ(n^2) irrespective of the constants involved. For a given function g(n), we denote Θ(g(n)) is following set of functions. **Examples :**

*{ 100 , log (2000) , 10^4 } belongs to **Θ(1)***

*{ (n/4) , (2n+3) , (n/100 + log(n)) } belongs to **Θ(n)***

*{ (n^2+n) , (2n^2) , (n^2+log(n))} belongs to **Θ(n^2)***

Note: Θ provides exact bounds.

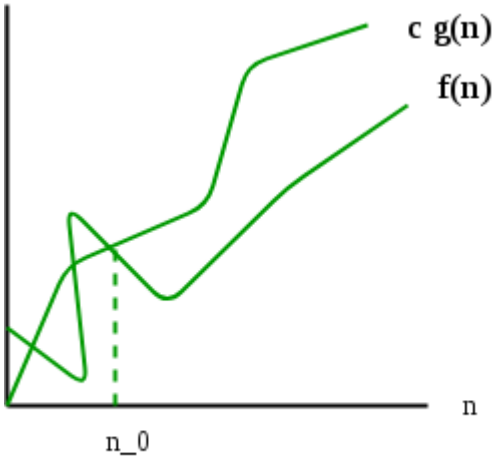
2. Big-O Notation (O-notation):

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 ≤ f(n) ≤ cg(n) for all n ≥ n0

It returns the highest possible output value (big-O)for a given input.

The execution time serves as an upper bound on the algorithm's time complexity.



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 ≤ f(n) ≤ cg(n) for all n ≥ 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 $\Theta(n^2)$.
- The best case time complexity of Insertion Sort is $\Theta(n)$.

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 .***

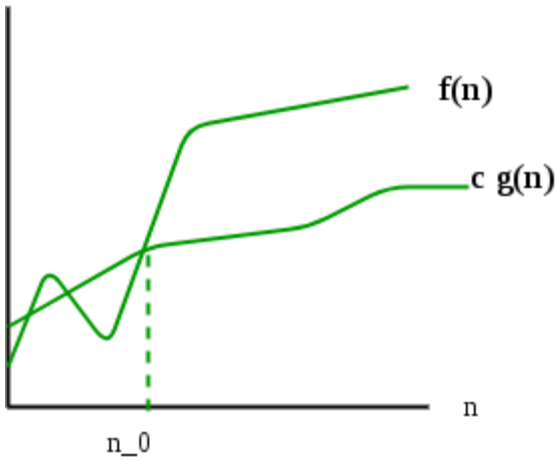
3. Omega Notation (Ω-Notation):

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

The execution time serves as a both lower bound on the algorithm's time complexity.

It is defined as the condition that allows an algorithm to complete statement execution in the shortest amount of time.

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) = n² ; O(n²) 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) = n² and g(n) = n²

then, f(n) = Θ(n²) and g(n) = Θ(n²)

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) = n²

then n is O(n²) and n² 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) = Ω(f(n))*

What is Algorithmic Efficiency, and Why is it Important?

Algorithm efficiency relates to how many resources a computer needs to expend to process an algorithm. The efficiency of an algorithm needs to be determined to ensure it can perform without the risk of crashes or severe delays. If an algorithm is not efficient, it is unlikely to be fit for its purpose.

We measure an algorithm's efficiency by how many resources are used to process it. An efficient algorithm uses minimal resources to perform its functions.

Algorithms can be used for various functions, including machine learning algorithms, protection against cybercrime, and making the internet safer. When browsing the web, you should also always be aware of how to protect yourself from identity theft and other criminal activity.

What are the Main Ways to Measure Algorithm Efficiency?

This section will discuss the two main measures for calculating algorithm efficiency. These are time complexity and space complexity. However, a direct comparison cannot be made between them; therefore, we need to consider a combination of the two.

Space complexity

Simply put, space complexity refers to the amount of memory needed during the execution of a program compared to the function input, i.e., the amount of memory on a computer or device that is required. The memory type could include registers, cache, RAM, virtual memory, and secondary memory.

When analyzing space complexity, you need to consider four key factors:

- The memory required to hold the code for the algorithm
- The memory required to input the data
- The memory required to output the data (some algorithms, such as sorting algorithms, do not need memory to output data and usually just rearrange the input data).
- The memory required for working space while the algorithm is calculated. This can include local variables and any stack space that is needed.

Mathematically, the space equals the sum of the two components below:

- **A variable part** that includes structured variables dependent on the problem the algorithm is trying to solve.
- **A fixed part** that is independent of the problem and consists of instruction space, the space for constants, fixed-size structure variables, and simple variables.

Therefore, space complexity S(a) of any algorithm is calculated as follows:

S(a) = c (the fixed part) + v(i) (the variable part which depends on an instance characteristic i)

Time complexity

Time complexity is the total time required to execute an algorithm, and it depends on all of the same factors used by space complexity, but these are broken down into a numerical function.

This measure can be useful when comparing different algorithms, mainly when large quantities of data are being processed. However, if the amount of data is small, more detailed estimates will be needed when comparing the performance of algorithms. Time complexity is less effective when an algorithm uses parallel processing.

Time complexity is defined as T(num). It is measured by the number of steps, as long as each step equates to the constant time.

Algorithm time complexity cases

An algorithm's complexity is defined by some key criteria; namely, the movement of the data and the comparison of keys - for example, how many times the data is moved and the key is compared. When measuring the complexity of an algorithm, we use three cases:

- Best case time complexity
- Average case time complexity
- Worst-case time complexity

The Process of Calculating Algorithm Efficiency

Two stages need to be completed when accurately calculating the efficiency of an algorithm - theoretical analysis and benchmarking (measuring the performance). We will provide a summary of each stage below.

Theoretical analysis

In relation to algorithms, theoretical analysis is usually the process of estimating an algorithm's complexity in an asymptotic manner (approaching a value or curve arbitrarily closely). The most common way of describing the number of resources an algorithm uses is by using Donald Knuth's Big O notation.

Using the Big O notation, programmers will measure algorithms to ensure they scale efficiently, regardless of input data size.

Benchmarking (measuring performance)

When analyzing and testing new algorithms or software, benchmarks are used to measure their performance. This helps to gauge an algorithm's efficiency compared to other well-performing algorithms.

Let's take a sorting algorithm as an example. Using benchmarks set by a previous version of the sorting algorithm can determine how efficient the current algorithm is, using known data while also taking into account its functional improvements.

Benchmarking also allows analysis teams to compare algorithm speed against various other programming languages to establish if improvements can be made. In fact, benchmarking can be implemented in various ways to measure performance against any predecessors or similar software.

Implementation Challenges

Implementing a new algorithm can sometimes impact its overall efficiency. This could be down to the chosen programming language, the compiler options used, the operating system, or just how the algorithm has been coded. In particular, the compiler used for a specific language can greatly impact speed.
When measuring space and time complexity, not everything can be dictated by the programmer. Issues such as cache locality and coherence, data alignment and granularity, multi-threading, and simultaneous multitasking can all impact performance, regardless of the programming language used or how the algorithm is written.

The processor used to run the software can also cause problems; some processors may support vector or parallel processing, whereas others may not. In some cases, utilizing such capabilities may not always be possible, making the algorithm less efficient and requiring some reconfiguration.

Finally, the instruction set used by a processor (e.g., ARM or x86-64) may affect how quickly instructions are processed on various devices. This makes it difficult to optimize compilers due to the number of different hardware combinations that need to be considered.

Conclusion

An algorithm needs to be processed very quickly, so it must perform as flawlessly as possible. This is why every new algorithm goes through testing to

calculate its efficiency, using theoretical analysis and benchmarking to compare it against other algorithms.

Time and space complexity are the two main measures for calculating algorithm efficiency, determining how many resources are needed on a machine to process it. Where time measures how long it takes to process the algorithm, space measures how much memory is used.

Unfortunately, numerous challenges, such as the programming language used, the processor, the instruction set, etc., can arise during the implementation process, causing headaches for programmers.

Despite this, time and space complexity have proven to be very effective ways of measuring algorithm efficiency.

Time-Space Trade-Off in Algorithms

In this article, we will discuss Time-Space Trade-Off in Algorithms. A tradeoff 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 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 **N[th] Fibonacci term** using recursion from the above recurrence relation.

Below is the implementation using recursion:

```
// C++ program to find Nth Fibonacci
// number using recursion
#include <iostream>
using namespace std;

// Function to find Nth Fibonacci term
int Fibonacci(int N)
{
    // Base Case
    if (N < 2)
        return N;

    // Recursively computing the term
    // using recurrence relation
    return Fibonacci(N - 1) + Fibonacci(N - 2);
}
// Driver Code
```

```
int main()
{
    int N = 5;
    // Function Call
    cout << Fibonacci(N);
    return 0;
}
```

Output:

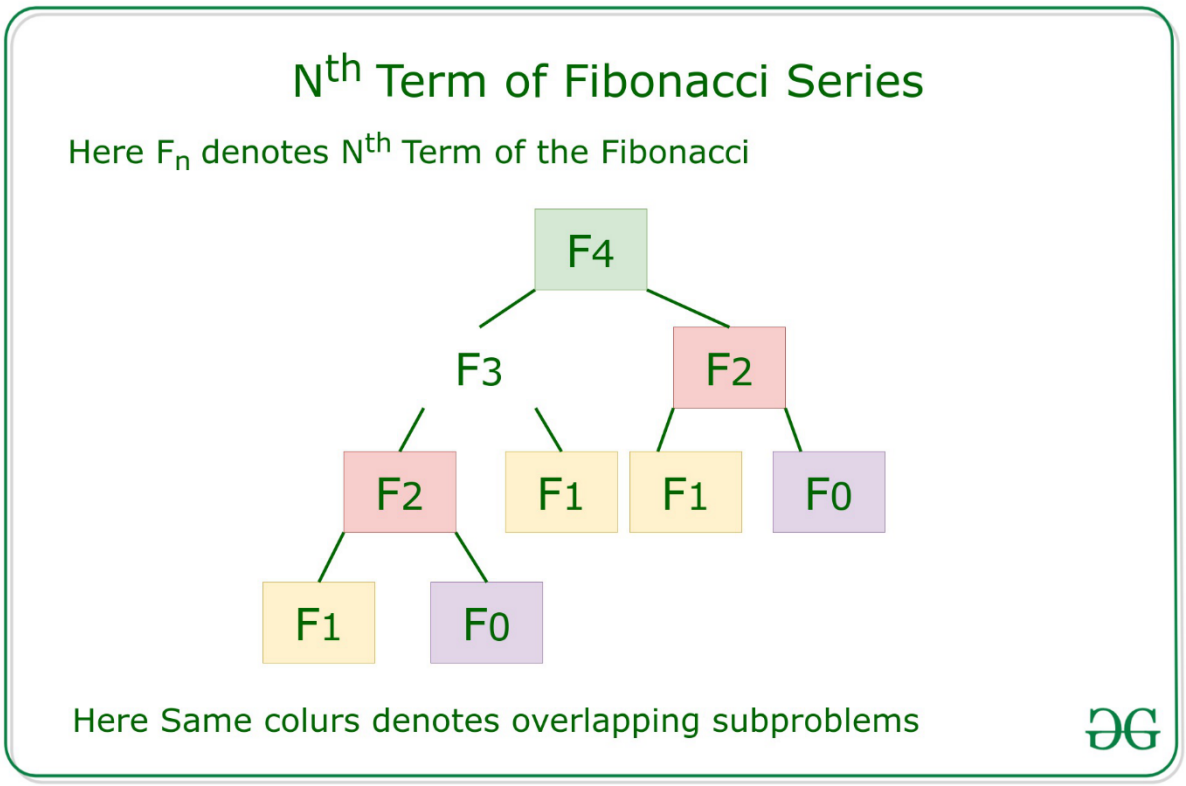
5

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:



Below is the implementation of the above approach:

- C++
- Java
- Python3
- C#

```
// C++ program to find Nth Fibonacci

// number using recursion

#include <iostream>

using namespace std;

// Function to find Nth Fibonacci term

int Fibonacci(int N)

{
```

```
    int f[N + 2];
    int i;
    // 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 = 2; i <= N; i++) {

        // 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
int main()
{
    int N = 5;
```

```
    // Function Call
    cout << Fibonacci(N);

    return 0;
}
```

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.

How to analyse Complexity of Recurrence Relation

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.

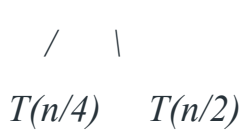
$$
\begin{aligned}
T(n) &= 2T(n/2) + n \\
&<= 2cn/2Log(n/2) + n \\
&= cnLogn - cnLog2 + n \\
&= cnLogn - cn + n \\
&<= cnLogn
\end{aligned}
$$

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.

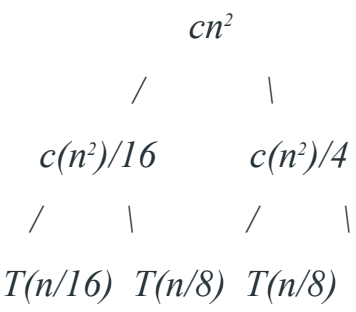
For example, consider the recurrence relation

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

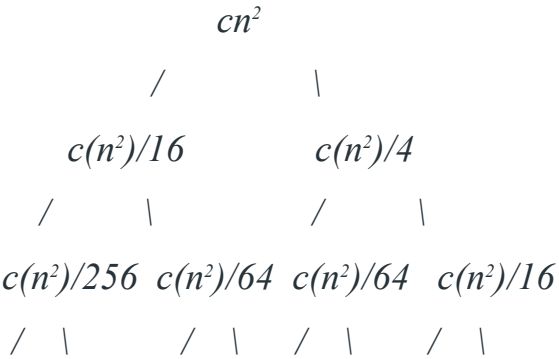


If we further break down the expression T(n/4) and T(n/2),

we get the following recursion tree.



Breaking down further gives us following



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 $b > 1$

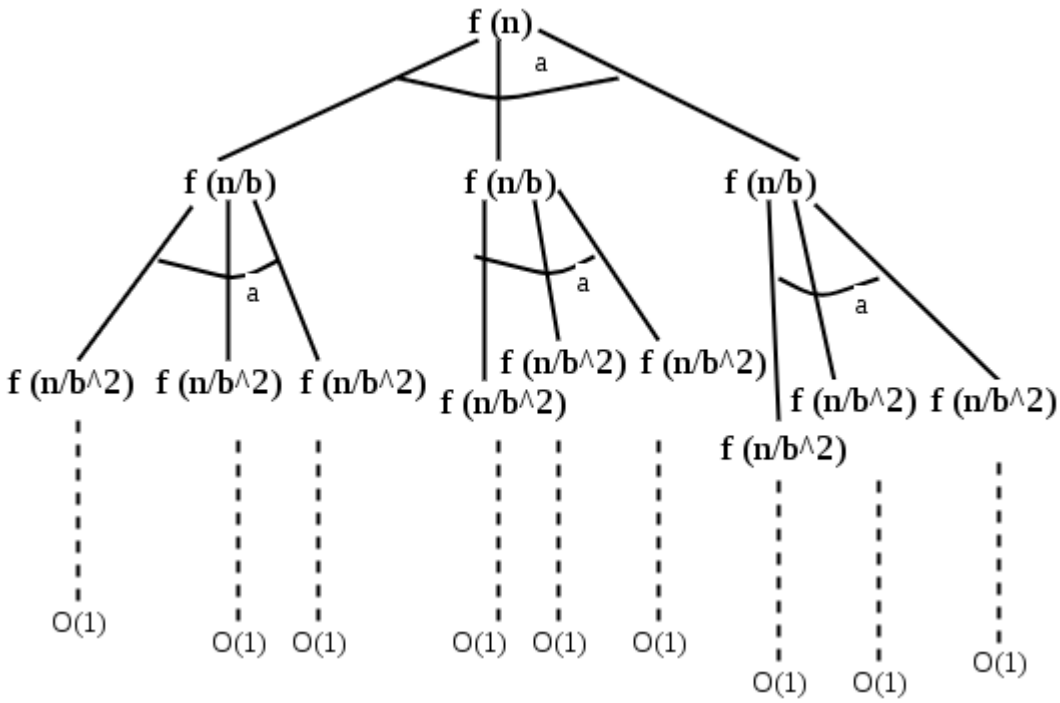
There are the following three cases:

- If $f(n) = O(n^c)$ where $c < Log_b a$ then $T(n) = \Theta(n^{Log_b a})$

- If $f(n) = \Theta(n^c)$ where $c = \text{Log}_b a$ then $T(n) = \Theta(n^c \text{Log } n)$
- If $f(n) = \Omega(n^c)$ where $c > \text{Log}_b a$ 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 $\text{Log}_b a$. And the height of the recurrence tree is $\text{Log}_b n$



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 $T(n) = \Theta(n^c Log^{k+1} n)$