# Rizvi College Of Arts Science and Commerce

# Fundamental Of Algorithm

## Unit-First

### Class: - SY B.Sc.(CS)

**Dr. Ruchi Gupta**

**Assistant Professor**

**CS\IT Department**

## B. Sc. (COMPUTER SCIENCE) Semester – IV

**Course: USCS401** **TOPICS (Credits: 02 Lectures/Week: 03)**

### Fundamentals of Algorithms

**UNIT-I**

**Introduction to algorithm**, why to analysis algorithm, Running time analysis, How to Compare Algorithms, Rate of Growth, Commonly Used Rates of Growth, Types of Analysis,

**Asymptotic Notation**, Big-O Notation, Omega-Ω Notation, Theta-Θ Notation, Asymptotic Analysis, Properties of Notations, commonly used Logarithms and Summations, Performance, characteristics of algorithms, Master Theorem for Divide and Conquer, Divide and Conquer

**Master Theorem:** Problems & Solutions, Master Theorem for Subtract and Conquer Recurrences, Method of Guessing and Confirming

## Introduction to Algorithm

- That term was coined by a Persian author, Abu. Jafar Mohammad bin Musa al-Khowarizmi.
- An algorithm is a set of self-contained sequence of instructions or actions that contains finite space or sequence and that will give us a result to a specific problem in a finite amount of time.
- A procedure that reduces the solution of some class of problems to a series of rote steps which, if followed to the letter, and as far as may be necessary, is bound to:
  - Always give some answer rather than ever give no answer;
  - Always give the right answer and never give a wrong answer;
  - Always be completed in a finite number of steps, rather than in an infinite number;
  - Work for all instances of problems of the class.
- It is a logical and mathematical approach to solve or crack a problem using any possible method.
- An algorithm is an effective, efficient and best method which can be used to express solution of any problem within a finite amount of space and time and in a well-defined formal language.
- Starting from an initial state the instructions describe a process or computational process that, when executed, proceeds through a finite number of well-defined successive states, eventually producing "output" and terminating at a final ending state.
- In other words, we can say that,
  - Step by step procedure for solving any problem is known as algorithm.
  - An algorithm is a finite set of instructions that, if followed, accomplishes a particular task.
  - An algorithm is a sequence of computational steps that transform the input into a valuable or required output.
  - Any special method of solving a certain kind of problem is known as algorithm.

## Criteria of Algorithm

All Algorithms must satisfy the following criteria -

### 1) Input

There are more quantities that are extremely supplied.

### 2) Output

At least one quantity is produced.

### 3) Definiteness

Each instruction of the algorithm should be clear and unambiguous.

**4) Finiteness**

The process should be terminated after a finite number of steps.

**5) Effectiveness**

Every instruction must be basic enough to be carried out theoretically or by using paper and pencil.

**Properties of Algorithm**

Simply writing the sequence of instructions as an algorithm is not sufficient to accomplish certain task. It is necessary to have following properties associated with an algorithm.

**Non Ambiguity**

Each step in an algorithm should be non-ambiguous. That means each instruction should be clear and precise. The instruction in any algorithm should not denote any conflicting meaning. This property also indicates the effectiveness of algorithm.

**Range of Input**

The range of input should be specified. This is because normally the algorithm is input driven and if the range of input is not being specified then algorithm can go in an infinite state.

**Multiplicity**

The same algorithm can be represented into several different ways. That means we can write in simple English the sequence of instruction or we can write it in form of pseudo code. Similarly, for solving the same problem we can write several different algorithms.

**Speed**

The algorithms written using some specified ideas. Bus such algorithm should be efficient and should produce the output with fast speed.

**Finiteness**

The algorithm should be finite. That means after performing required operations it should be terminate

**Types of algorithm**

Well there are many types of algorithm but the most fundamental types of algorithm are:

1. Recursive algorithms
2. Dynamic programming algorithm
3. Backtracking algorithm
4. Divide and conquer algorithm

5. Greedy algorithm
6. Brute Force algorithm
7. Randomized algorithm

**1) Simple recursive algorithm**

Solves the base case directly and then recurs with a simpler or easier input every time (A base value is set at the starting for which the algorithm terminates).

It is use to solve the problems which can be broken into simpler or smaller problems of same type.

**Example:**

Factorial using recursion

**2) Dynamic programming algorithm**

A dynamic programming algorithm (also known as dynamic optimization algorithm) remembers the past result and uses them to find new result means it solve complex problems by breaking it down into a collection of simpler subproblems, then solving each of those subproblems only once ,and storing their solution for future use instead of recomputing their solutions again.

**Example:**

Fibonacci sequence,

**3) Backtracking algorithm**

How about we learn backtracking using an example so let's say we have a problem "Monk" and we divide it into four smaller problems "M, R, A, A". It may be the case that the solution of these problems did not get accepted as the solution of "Monk".

In fact we did not know on which one it depends. So we will check each one of them one by one until we find the solution for "Monk".

So basically we attempt solving a subproblem but if we did not reach the desired solution undo whatever you have done and start from the scratch again until you find the solution.

**Example:**

**Queens Problem**

**4) Divide and conquer algorithm**

Divide and conquer consist of two parts first of all it divides the problems into smaller subproblems of the same type and solve them solve them recusively and then combine them to to form the solution of the original problem.

**Example:**

Quick sort, Merge sort

**5) Greedy algorithm**

Greedy algorithm is an algorithm that solves the problem by taking optimal solution at the local level (without regards for any consequences) with the hope of finding optimal solution at the global level.

Greedy algorithm is used to find the optimal solution but it is not necessary that you will definitely find the optimal solution by following this algorithm.

Like there are some problems for which an optimal solution does not exist (currently) these are called NP complete problem.

**Example:**

**Huffman tree, Counting money**

**6) Brute force algorithm**

A brute force algorithm simply tries all the possibilities until a satisfactory solution is found.

Such types of algorithm are also used to find the optimal (best) solution as it checks all the possible solutions.

And also used for finding a satisfactory solution (not the best), simply stop as soon as a solution of the problem is found.

**Example:**

**Exact string matching algorithm**

**7) Randomized algorithm**

A randomized algorithm uses a random number at least once during the computation to make a decision.

**Example:**

Quick sort

As we use random number to choose the pivot point.

**Advantages of the Algorithm**

- It identifies the solution process, decision points and variables required to solve the problem.

- It helps in dividing a huge problem into smaller manageable steps of the solution.
- The analysis and specification of the process lead to the efficiency.
- Separation of the steps divides labour and development expertise.

## Disadvantages of the Algorithm

- At a specific point, the algorithm terminates.
- Inability to solve problems that generate non-computational results.
- Consumes a lot of time.

## Expressing Algorithms

### Pseudocode

Pseudocode is the expressive form of the algorithm or a way to describe an algorithm. It is a combination of natural language and high-level programming practices which represent the fundamental concept behind a general implementation of a data structure or algorithm. Pseudocode incorporated the natural language when the details are insignificant with the standard programming language constructs to obtain more clarity. However, we cannot execute Pseudocode on a computer, but it models the actual programming code along with a similar level of detail.

Pseudocode is also written using some specific words and characters, which are shown below:

- To begin the comment double forward slash are used "//".
- Matching braces "{ and }" are used to present blocks where a compound statement (set of simple statements) can be illustrated as a block and terminated by a semicolon";". The body of a procedure constructs a block as well.
- All the identifiers start with a letter and the datatype of the variables are not declared explicitly.
- An assignment statement is used for the assigning values to the variables.
- To produce the boolean values (i.e., true and false) the logical operators and, or and not and the relational operators <, ≤, =, =, ≥ and > are provided.
- Input and output are presented by read and write instructions.
- "if and then" expressions are used to express a conditional statement.

## Algorithm vs. Pseudocode

| BASIS FOR COMPARISON | ALGORITHM | PSEUDOCODE |
|---|---|---|
| Comprehensibility | Quite hard to understand | Easy to interpret |
| Uses | Complicated programming language | Combination of programming language and natural language |
| Debugging | Moderate | Simpler |
| Ease of construction | Complex | Easier |

## Flowchart

- It is nothing but a manner of representing an algorithm.
- It is also known as the flow diagram, which illustrates a process or a detailed series of steps needed to produce a specific output.
- A flow chart is comprised of the different symbols and control lines to connect those symbols. Each symbol specifies distinct functions.
- It is extremely useful in programming because it simplifies the complicated algorithm and converts it into the understandable pictorial representation.
- With the help of the flowchart, the application designer can easily segregate the different components of the process. It facilitates the analysation by providing the step-by-step process of the problem.

## Algorithm vs. Flowchart

| BASIS FOR COMPARISON | ALGORITHM | FLOW CHART |
|---|---|---|
| Basic | Includes sequence of steps which depicts the procedure of the solution. | An information diagram made up of different shapes shows the data flow. |
| Comprehensibility | Hard to understand | Easily interpreted |
| Uses | Text | Symbols |
| Implements | No rules are employed. | Predefined rules are implemented. |
| Debugging | Easier | Difficult |
| Ease of construction | Perplexing | Simple |

**Need of Algorithm**

1. To understand the basic idea of the problem.
2. To find an approach to solve the problem.
3. To improve the efficiency of existing techniques.
4. To understand the basic principles of designing the algorithms.
5. To compare the performance of the algorithm with respect to other techniques.
6. It is the best method of description without describing the implementation detail.
7. The Algorithm gives a clear description of requirements and goal of the problem to the designer.
8. A good design can produce a good solution.
9. To understand the flow of the problem.
10. To measure the behaviour (or performance) of the methods in all cases (best cases, worst cases, average cases)
11. With the help of an algorithm, we can also identify the resources (memory, input-output) cycles required by the algorithm.
12. With the help of algorithm, we convert art into a science.
13. To understand the principle of designing.
14. We can measure and analyse the complexity (time and space) of the problems concerning input size without implementing and running it; it will reduce the cost of design

**Analysis of Algorithms**

**Why Analyse an Algorithm?**

The most straightforward reason for analyzing an algorithm is to discover its characteristics in order to evaluate its suitability for various applications or compare it with other algorithms for the same application. Moreover, the analysis of an algorithm can help us understand it better, and can suggest informed improvements. Algorithms tend to become shorter, simpler, and more elegant during the analysis process.

**Computational Complexity.**

The branch of theoretical computer science where the goal is to classify algorithms according to their efficiency and computational problems according to their inherent difficulty is known as *computational complexity*. Paradoxically, such classifications are typically not useful for predicting performance or for comparing algorithms in practical applications because they focus on order-of-growth worst-case performance. In this book, we focus on analyses that *can* be used to predict performance and compare algorithms.

**Analysis of Algorithms.**

A complete analysis of the running time of an algorithm involves the following steps:

- Implement the algorithm completely.
- Determine the time required for each basic operation.
- Identify unknown quantities that can be used to describe the frequency of execution of the basic operations.
- Develop a realistic model for the input to the program.
- Analyze the unknown quantities, assuming the modelled input.

- Calculate the total running time by multiplying the time by the frequency for each operation, then adding all the products.

Classical algorithm analysis on early computers could result in exact predictions of running times. Modern systems and algorithms are much more complex, but modern analyses are informed by the idea that exact analysis of this sort could be performed *in principle*.
The term analysis of algorithms is used to describe approaches to the study of the performance of algorithms, we will perform the following types of analysis:

- The *worst-case runtime complexity* of the algorithm is the function defined by the maximum number of steps taken on any instance of size a.
- The *best-case runtime complexity* of the algorithm is the function defined by the minimum number of steps taken on any instance of size a.
- The *average case runtime complexity* of the algorithm is the function defined by an average number of steps taken on any instance of size a.
- The *amortized runtime complexity* of the algorithm is the function defined by a sequence of operations applied to the input of size a and averaged over time.

## Complexity of Algorithm

It is very convenient to classify algorithm based on the relative amount of time or relative amount of space they required and specify the growth of time/space requirement as a function of input size.
An algorithm is said to be efficient and fast, if it takes less time to execute and consumes less memory space. The performance of an algorithm is measured on the basis of following properties:

1. Time Complexity
2. Space Complexity

### Space Complexity

It's the amount of memory space required by the algorithm, during the course of its execution. Space complexity must be taken seriously for multi-user systems and in situations where limited memory is available. Whenever a solution to a problem is written some memory is required to complete. For any algorithm memory may be used for the following:

1. Variables (This include the constant values, temporary values)
2. Program Instruction
3. Execution

***Space complexity*** is the amount of memory used by the algorithm (including the input values to the algorithm) to execute and produce the result.

Sometime **Auxiliary Space** is confused with Space Complexity. But Auxiliary Space is the extra space or the temporary space used by the algorithm during its execution.

**Space Complexity = Auxiliary Space + Input space**

### Memory Usage while Execution

While executing, algorithm uses memory space for three reasons:

1. **Instruction Space**

It's the amount of memory used to save the compiled version of instructions.

2. **Environmental Stack**
   Sometimes an algorithm (function) may be called inside another algorithm (function). In such a situation, the current variables are pushed onto the system stack, where they wait for further execution and then the call to the inside algorithm (function) is made.
   For example, If a function A() calls function B() inside it, then all th variables of the function A() will get stored on the system stack temporarily, while the function B() is called and executed inside the funciton A().
3. **Data Space**
   Amount of space used by the variables and constants.

But while calculating the **Space Complexity** of any algorithm, we usually consider only **Data Space** and we neglect the **Instruction Space** and **Environmental Stack**.

### Calculating the Space Complexity

For calculating the space complexity, we need to know the value of memory used by different type of data type variables, which generally varies for different operating systems, but the method for calculating the space complexity remains the same.

| Type | Size |
|---|---|
| bool, char, unsigned char, signed char, __int8 | 1 byte |
| __int16, short, unsigned short, wchar_t, __wchar_t | 2 bytes |
| float, __int32, int, unsigned int, long, unsigned long | 4 bytes |
| double, __int64, long double, long long | 8 bytes |

### Time Complexity

- Time Complexity is a way to represent the amount of time required by the program to run till its completion.
- It's generally a good practice to try to keep the time required minimum, so that our algorithm completes it's execution in the minimum time possible
- Time complexity of an algorithm signifies the total time required by the program to run till its completion.
- The time complexity of algorithms is most commonly expressed using the big O notation. It's an asymptotic notation to represent the time complexity. We will study about it in detail in the next tutorial.
- Time Complexity is most commonly estimated by counting the number of elementary steps performed by any algorithm to finish execution.

Like in the example above, for the first code the loop will run n number of times, so the time complexity will be n at least and as the value of n will increase the time taken will also increase. While for the second code, time complexity is constant, because it will never be dependent on the value of n, it will always give the result in 1 step. And since the algorithm's performance may vary with different types of input data, hence for an algorithm we usually

use the worst-case Time complexity of an algorithm because that is the maximum time taken for any input size.

**Types of Notations for Time Complexity**

Now we will discuss and understand the various notations used for Time Complexity.

- Big Oh denotes "fewer than or the same as" <expression> iterations.
- Big Omega denotes "more than or the same as" <expression> iterations.
- Big Theta denotes "the same as" <expression> iterations.
- Little Oh denotes "fewer than" <expression> iterations.
- Little Omega denotes "more than" <expression> iterations.

Understanding Notations of Time Complexity with Example

- Oh (expression) is the set of functions that grow slower than or at the same rate as expression. It indicates the maximum required by an algorithm for all input values. It represents the worst case of an algorithm's time complexity.
- Omega (expression) is the set of functions that grow faster than or at the same rate as expression. It indicates the minimum time required by an algorithm for all input values. It represents the best case of an algorithm's time complexity.
- Theta (expression) consist of all the functions that lie in both O(expression) and Omega(expression). It indicates the average bound of an algorithm. It represents the average case of an algorithm's time complexity.

These are the possible running times:

- Worst-case running time - the algorithm finds the number at the end of the list or determines that the number isn't in the list. It went through the entire list so it took *linear time*. The worst-case running time is usually what is examined.
- Best-case running time - the algorithm gets lucky and finds the number on the first check. It took constant time regardless of the input size. Since we are not usually that lucky, we don't usually care about the best-case running time.
- Expected-case running time - the algorithm finds the number halfway through the list (assuming the number is in the input). We sometimes care about the expected-case, though it can be harder to calculate than the worst-case.

**Typical Complexities of an Algorithm**

- **Constant Complexity:**
  It imposes a complexity of **O(1)**. It undergoes an execution of a constant number of steps like 1, 5, 10, etc. for solving a given problem. The count of operations is independent of the input data size.
- **Logarithmic Complexity:**
  It imposes a complexity of **O(log(N))**. It undergoes the execution of the order of log(N) steps. To perform operations on N elements, it often takes the logarithmic base as 2.
  For N = 1,000,000, an algorithm that has a complexity of O(log(N)) would undergo 20

steps (with a constant precision). Here, the logarithmic base does not hold a necessary consequence for the operation count order, so it is usually omitted.

- **Linear Complexity:**
  - It imposes a complexity of **O(N)**. It encompasses the same number of steps as that of the total number of elements to implement an operation on N elements.
    For example, if there exist 500 elements, then it will take about 500 steps.
    Basically, in linear complexity, the number of elements linearly depends on the number of steps. For example, the number of steps for N elements can be N/2 or 3*N.
  - It also imposes a run time of **O(n*log(n))**. It undergoes the execution of the order N*log(N) on N number of elements to solve the given problem.
    For a given 1000 elements, the linear complexity will execute 10,000 steps for solving a given problem.
- **Quadratic Complexity:** It imposes a complexity of **O($n^2$)**. For N input data size, it undergoes the order of $N^2$ count of operations on N number of elements for solving a given problem.
  If N = 100, it will endure 10,000 steps. In other words, whenever the order of operation tends to have a quadratic relation with the input data size, it results in quadratic complexity. For example, for N number of elements, the steps are found to be in the order of $3*N^2/2$.
- **Cubic Complexity:** It imposes a complexity of **O($n^3$)**. For N input data size, it executes the order of $N^3$ steps on N elements to solve a given problem.
  For example, if there exist 100 elements, it is going to execute 1,000,000 steps.
- **Exponential Complexity:** It imposes a complexity of **O($2^n$), O(N!), O($n^k$), ….** For N elements, it will execute the order of count of operations that is exponentially dependable on the input data size.
  For example, if N = 10, then the exponential function $2^N$ will result in 1024. Similarly, if N = 20, it will result in 1048 576, and if N = 100, it will result in a number having 30 digits. The exponential function N! grows even faster; for example, if N = 5 will result in 120. Likewise, if N = 10, it will result in 3,628,800 and so on.

**Asymptotic Analysis of algorithms (Growth of function)**

Resources for an algorithm are usually expressed as a function regarding input. Often this function is messy and complicated to work. To study Function growth efficiently, we reduce the function down to the important part.

Let f (n) = $an^2+bn+c$

In this function, the $n^2$ term dominates the function that is when n gets sufficiently large.

Dominate terms are what we are interested in reducing a function, in this; we ignore all constants and coefficient and look at the highest order term concerning n.

**Asymptotic notation:**

The word **Asymptotic** means approaching a value or curve arbitrarily closely (i.e., as some sort of limit is taken).

**Asymptotic analysis**

It is a technique of representing limiting behavior. The methodology has the applications across science. It can be used to analyze the performance of an algorithm for some large data set.

1. In computer science in the analysis of algorithms, considering the performance of algorithms when applied to very large input datasets

The simplest example is a function $f(n) = n^2+3n$, the term 3n becomes insignificant compared to $n^2$ when n is very large. The function "$f(n)$ is said to be **asymptotically equivalent** to $n^2$ as $n \rightarrow \infty$", and here is written symbolically as $f(n) \sim n^2$.

**Asymptotic notations** are used to write fastest and slowest possible running time for an algorithm. These are also referred to as 'best case' and 'worst case' scenarios respectively.

"In asymptotic notations, we derive the complexity concerning the size of the input. (Example in terms of n)"

"These notations are important because without expanding the cost of running the algorithm, we can estimate the complexity of the algorithms."

**Why is Asymptotic Notation Important?**

1. They give simple characteristics of an algorithm's efficiency.

2. They allow the comparisons of the performances of various algorithms.

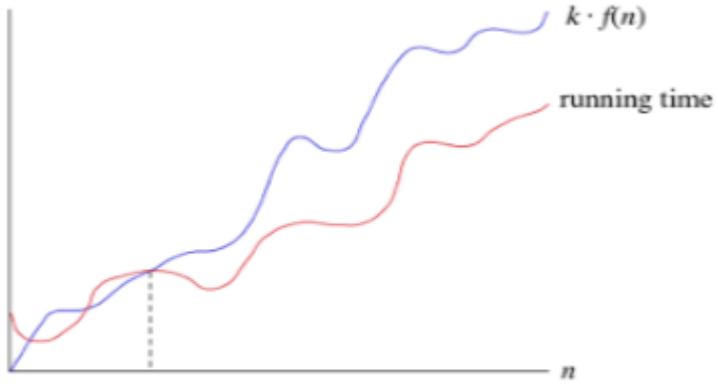
**Asymptotic Notations:**

Asymptotic Notation is a way of comparing function that ignores constant factors and small input sizes. Three notations are used to calculate the running time complexity of an algorithm:

**1. Big-oh notation:** Big-oh is the formal method of expressing the upper bound of an algorithm's running time. It is the measure of the longest amount of time. The function **f (n) = O (g (n))** [read as "f of n is big-oh of g of n"] if and only if exist positive constant c and such that

1. f (n) ⩽ k.g (n)f(n)⩽k.g(n) **for** n>n0n>n0 in all **case**

Hence, function g (n) is an upper bound for function f (n), as g (n) grows faster than f (n)




ASYMPTOTIC UPPER BOUND

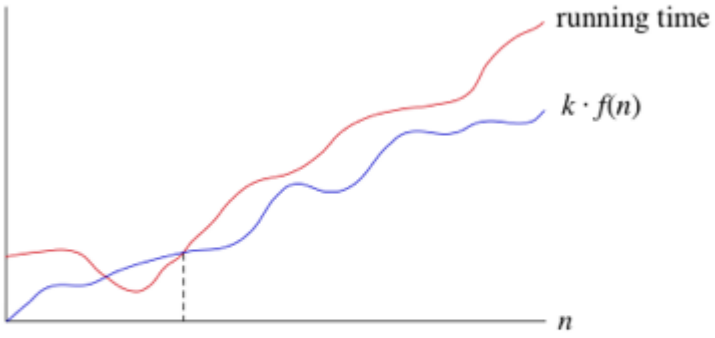
For Example:

1. 1. 3n+2=O(n) as 3n+2≤4n **for** all n≥2
2. 2. 3n+3=O(n) as 3n+3≤4n **for** all n≥3

Hence, the complexity of **f(n)** can be represented as O (g (n))

**2. Omega () Notation:** The function f (n) = **Ω** (g (n)) [read as "f of n is omega of g of n"] if and only if there exists positive constant c and $n_0$ such that

F (n) ≥ k* g (n) for all n, n≥ $n_0$




ASYMPTOTIC LOWER BOUND

For Example:

f (n) =$8n^2$+2n-3≥$8n^2$-3

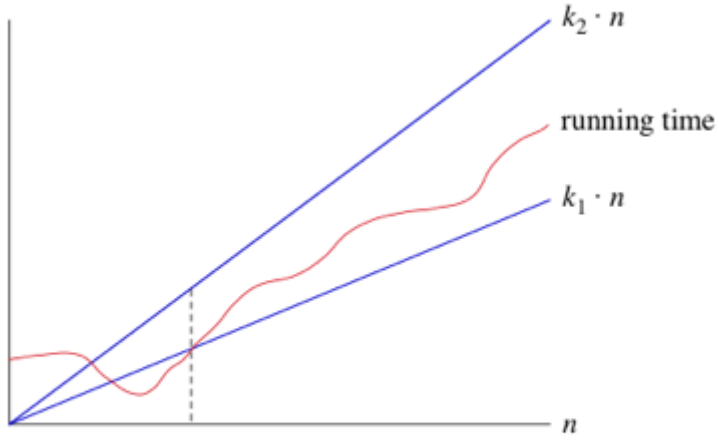
=$7n^2$+($n^2$-3)≥$7n^2$ (g(n))

Thus, $k_1$=7

Hence, the complexity of **f (n)** can be represented as **Ω** (g (n))

**3. Theta (θ):** The function f (n) = θ (g (n)) [read as "f is the theta of g of n"] if and only if there exists positive constant $k_1$, $k_2$ and $k_0$ such that

$k_1 * g(n) \leq f(n) \leq k_2\, g(n)$ for all $n, n \geq n_0$




**ASYMPTOTIC TIGHT BOUND**

For Example:

$3n+2 = \theta(n)$ as $3n+2 \geq 3n$ and $3n+2 \leq 4n$, for $n$

$k_1=3, k_2=4$, and $n_0=2$

Hence, the complexity of f (n) can be represented as θ (g(n)).

The Theta Notation is more precise than both the big-oh and Omega notation. The function f (n) = θ (g (n)) if g(n) is both an upper and lower bound.

## Analyzing Algorithm Control Structure

To analyze a programming code or algorithm, we must notice that each instruction affects the overall performance of the algorithm and therefore, each instruction must be analyzed separately to analyze overall performance. However, there are some algorithm control structures which are present in each programming code and have a specific asymptotic analysis. Some Algorithm Control Structures are:

1. **Sequencing**
2. **If-then-else**
3. **for loop**
4. **While loop**

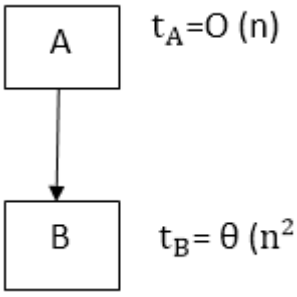
**1. Sequencing:**

Suppose our algorithm consists of two parts A and B. A takes time $t_A$ and B takes time $t_B$ for computation. The total computation "$t_A + t_B$" is according to the sequence rule. According to maximum rule, this computation time is (max ($t_A$,$t_B$)).

**Example:**

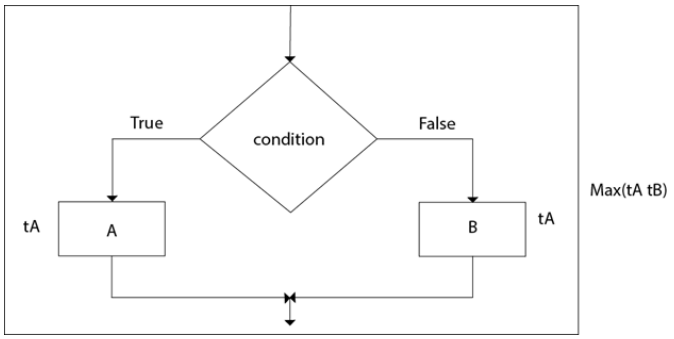
Suppose $t_A = O(n)$ and $t_B = \theta(n^2)$.

Then, the total computation time can be calculated as




Computation Time = $t_A + t_B$
= (max ($t_A$,$t_B$)
= (max (O (n), θ ($n^2$)) = θ ($n^2$)

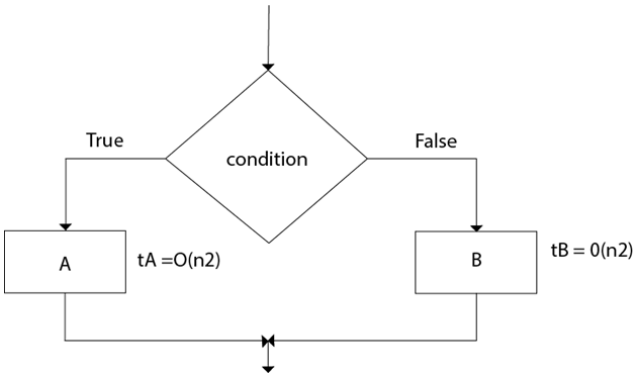
**2. If-then-else:**




The total time computation is according to the condition rule-"if-then-else." According to the maximum rule, this computation time is max ($t_A$,$t_B$).

**Example:**

Suppose $t_A$ = O ($n^2$) and $t_B$ = θ ($n^2$)
Calculate the total computation time for the following:




Total Computation = (max ($t_A$,$t_B$))
= max (O ($n^2$), θ ($n^2$) = θ ($n^2$)

**3. For loop:**

The general format of for loop is:

**For (initialization; condition; updation)**
**Statement(s);**

**Complexity of for loop**:

The outer loop executes N times. Every time the outer loop executes, the inner loop executes M times. As a result, the statements in the **inner** loop execute a total of N * M times. Thus, the total complexity for the two loops is O (N2)

Consider the following loop:

```
for i ← 1 to n
 {
      P (i)
 }
```

If the computation time $t_i$ for ( $P_I$) various as a function of "i", then the total computation time for the loop is given not by a multiplication but by a sum i.e.

```
For i ← 1 to n
 {
      P (i)
 }
```

Takes $\sum_{i=1}^{n} t_i$ time, i.e. $\sum_{j=1}^{n} \theta(1) = \theta \sum_{i=1}^{n} \theta(n)$

If the algorithms consist of nested "for" loops, then the total computation time is

```
For i ← 1 to n
 {
    For j ←  1 to n
   {
    P (ij)
   }
 }
```

$\sum_{i=1}^{n} \sum_{j=1}^{n} t_{ij}$

**Example:**

Consider the following "for" loop, Calculate the total computation time for the following:

```
1. For i ← 2 to n-1
2.    {
3.       For j ← 3 to i
4.        {
5.                Sum ← Sum+A [i] [j]
6.             }
7.          }
```

**Solution:**

The total Computation time is:

$\sum_{i=2}^{n-1} \sum_{j=3}^{i} t_{ij} = \sum_{i=2}^{n-1} \sum_{j=3}^{i} \theta(1)$

$\sum_{i=2}^{n-1} \theta(i)$

$= \theta(\sum_{i=2}^{n-1} i) = \theta\left(\frac{m^2}{2}\right) + \theta(m)$

$= \theta(m^2)$

**4. While loop:**

The Simple technique for analyzing the loop is to determine the function of variable involved whose value decreases each time around. Secondly, for terminating the loop, it is necessary that value must be a positive integer. By keeping track of how many times the value of function decreases, one can obtain the number of repetition of the loop. The other approach for analyzing "while" loop is to treat them as recursive algorithms.

Algorithm:

1. [Initialize] Set k: =1, LOC: =1 and MAX: = DATA [1]
2. Repeat steps 3 and 4 **while** K≤N
3. **if** MAX<DATA [k],then:
   Set LOC: = K and MAX: = DATA [k]
4. Set k: = k+1
   [End of step 2 loop]
5. Write: LOC, MAX
6. EXIT

**Example:**

The running time of algorithm array Max of computing the maximum element in an array of n integer is O (n).

**Solution:**

array Max (A, n)

1. Current max ← A [0]
2. For i ← 1 to n-1
3. **do if** current max < A [i]
4. then current max ← A [i]
5. **return** current max.

The number of primitive operation t (n) executed by this algorithm is at least.

2 + 1 + n +4 (n-1) + 1=5n

2 + 1 + n + 6 (n-1) + 1=7n-2

The best case T(n) =5n occurs when A [0] is the maximum element. The worst case T(n) = 7n-2 occurs when element are sorted in increasing order.

We may, therefore, apply the big-Oh definition with c=7 and $n_0$=1 and conclude the running time of this is O (n)

**Recurrence Relation**

A recurrence is an equation or inequality that describes a function in terms of its values on smaller inputs. To solve a Recurrence Relation means to obtain a function defined on the natural numbers that satisfy the recurrence.

**For Example**, the Worst Case Running Time T(n) of the MERGE SORT Procedures is described by the recurrence.

$$T(n) = \theta(1) \text{ if } n=1$$

$$2T\left(\frac{n}{2}\right) + \theta(n) \text{ if } n>1$$

There are four methods for solving Recurrence:

1. **Substitution Method**
2. **Iteration Method**
3. **Recursion Tree Method**
4. **Master Method**

---

**1. Substitution Method:**

The Substitution Method Consists of two main steps:

1. Guess the Solution.

2. Use the mathematical induction to find the boundary condition and shows that the guess is correct.

**For Example1** Solve the equation by Substitution Method.

$$T(n) = T\left(\frac{n}{2}\right) + n$$

We have to show that it is asymptotically bound by O (log n).

**Solution:**

For T (n) = O (log n)

We have to show that for some constant c

1. T (n) ≤c logn.

Put this in given Recurrence Equation.

$$T(n) \le c \log\left(\frac{n}{2}\right) + 1$$

$$\le c \log\left(\frac{n}{2}\right) + 1 = c\,\text{logn} - c\log_2 2 + 1$$

$$\le c\,\text{logn for } c \ge 1$$

Thus **T (n) =O logn**.

**Example2** Consider the Recurrence

$$T(n) = 2T\left(\frac{n}{2}\right) + n \quad n>1$$

Find an Asymptotic bound on T.

**Solution:**

We guess the solution is O (n (logn)).Thus for constant 'c'.
T (n) ≤c n logn
Put this in given Recurrence Equation.
Now,

$$T(n) \le 2c\left(\frac{n}{2}\right)\log\left(\frac{n}{2}\right) + n$$

$$\le cn\log n - cn\log 2 + n$$

$$= cn\log n - n(c\log 2 - 1)$$

$$\le cn\log n \text{ for } (c \ge 1)$$

Thus **T (n) = O (n logn)**.

## 2. Iteration Methods

It means to expand the recurrence and express it as a summation of terms of n and initial condition.

**Example1:** Consider the Recurrence

1. T (n) = 1 **if** n=1
2. = 2T (n-1) **if** n>1

**Solution:**

T (n) = 2T (n-1)
= 2[2T (n-2)] = $2^2$T (n-2)
= 4[2T (n-3)] = $2^3$T (n-3)
= 8[2T (n-4)] = $2^4$T (n-4) (Eq.1)

Repeat the procedure for i times

T (n) = $2^i$ T (n-i)
Put n-i=1 or i= n-1 in (Eq.1)
T (n) = $2^{n-1}$ T (1)
= $2^{n-1}$ .1 {T (1) =1 .....given}
= $2^{n-1}$

**Example2:** Consider the Recurrence

1. T (n) = T (n-1) +1 and T (1) = θ (1).

**Solution:**

T (n) = T (n-1) +1
= (T (n-2) +1) +1 = (T (n-3) +1) +1+1
= T (n-4) +4 = T (n-5) +1+4
= T (n-5) +5= T (n-k) + k
Where k = n-1
T (n-k) = T (1) = θ (1)
T (n) = θ (1) + (n-1) = 1+n-1=n= θ (n).

## 3. Recursion Tree Method

1. Recursion Tree Method is a pictorial representation of an iteration method which is in the form of a tree where at each level nodes are expanded.

2. In general, we consider the second term in recurrence as root.

3. It is useful when the divide & Conquer algorithm is used.

4. It is sometimes difficult to come up with a good guess. In Recursion tree, each root and child represents the cost of a single subproblem.

5. We sum the costs within each of the levels of the tree to obtain a set of pre-level costs and then sum all pre-level costs to determine the total cost of all levels of the recursion.

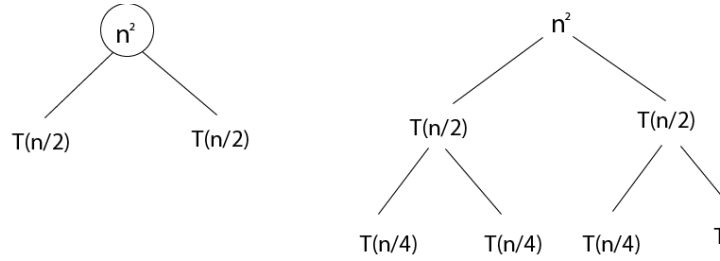
6. A Recursion Tree is best used to generate a good guess, which can be verified by the Substitution Method.

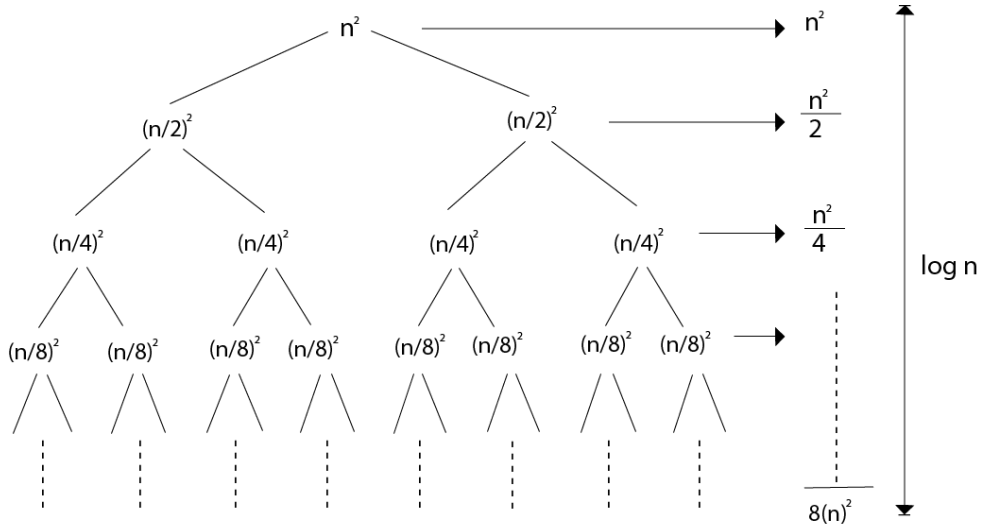
**Example 1**

Consider $T(n) = 2T\left(\frac{n}{2}\right) + n^2$

We have to obtain the asymptotic bound using recursion tree method.

**Solution:** The Recursion tree for the above recurrence is




$$T(n) = n^2 + \frac{n^2}{2} + \frac{n^2}{4} + \ldots\ldots \log n \text{ times.}$$

$$\leq n^2 \sum_{i=0}^{\infty} \left(\frac{1}{2^i}\right)$$

$$\leq n^2 \left(\frac{1}{1-\frac{1}{2}}\right) \leq 2n^2$$

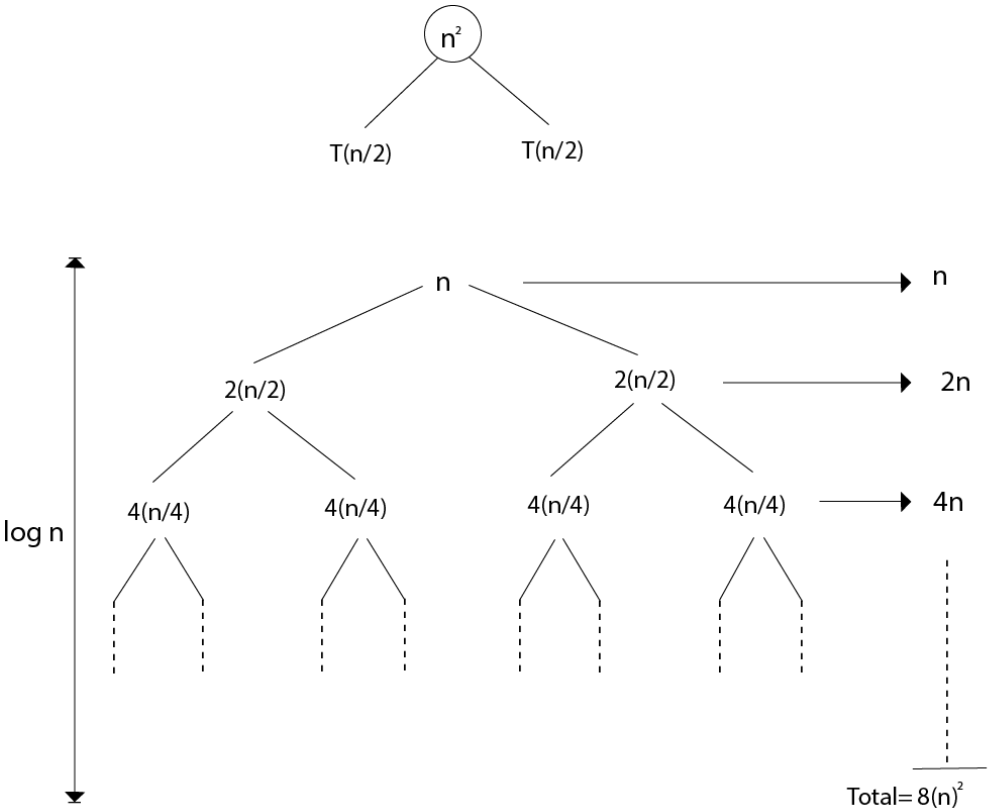
$$\mathbf{T(n) = \theta n^2}$$

**Example 2:** Consider the following recurrence

$$T(n) = 4T\left(\frac{n}{2}\right) + n$$

Obtain the asymptotic bound using recursion tree method.

**Solution:** The recursion trees for the above recurrence




We have $n + 2n + 4n + \ldots\ldots \log_2 n$ times

$$= n\,(1 + 2 + 4 + \ldots\ldots \log_2 n \text{ times})$$

$$= n\frac{(2^{\log_2 n} - 1)}{(2-1)} = \frac{n(n-1)}{1} = n^2 - n = \theta(n^2)$$

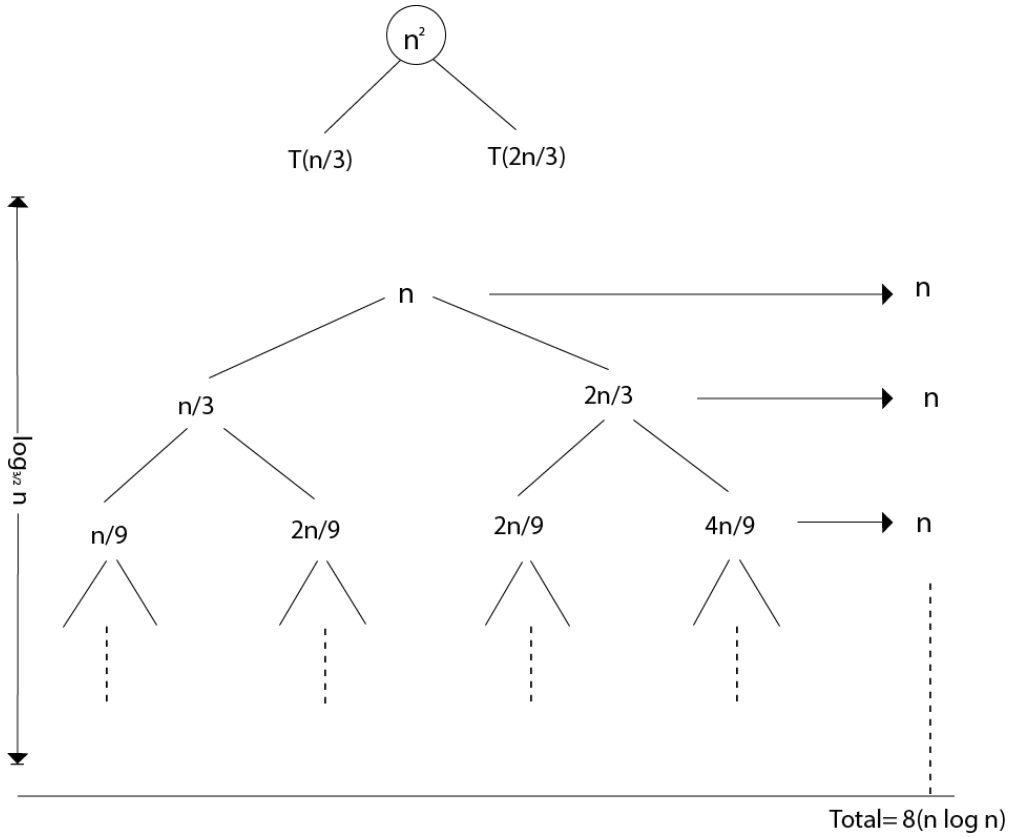
$$\mathbf{T(n) = \theta(n^2)}$$

**Example 3:** Consider the following recurrence

$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$$

Obtain the asymptotic bound using recursion tree method.

**Solution:** The given Recurrence has the following recursion tree




When we add the values across the levels of the recursion trees, we get a value of n for every level. The longest path from the root to leaf is

$$n \longrightarrow \frac{2}{3}n \longrightarrow \left(\frac{2}{3}\right) n \longrightarrow \ldots 1$$

Since $\left(\frac{2}{3}\right) n=1$ when $i=\log \frac{3}{2}\ n$.

Thus the height of the tree is $\log \frac{3}{2}\ n$.

$$T(n) = n + n + n + \ldots\ldots + \log \frac{3}{2}\, n \text{ times.} = \boldsymbol{\theta(n \log n)}$$

## 4. Master Method

The Master Method is used for solving the following types of recurrence

$T(n) = a\ T\left(\frac{n}{b}\right) + f(n)$ with $a \geq 1$ and $b \geq 1$ be constant & f(n) be a function and $\frac{n}{b}$ can be interpreted as

Let T (n) is defined on non-negative integers by the recurrence.

$$T(n) = a\ T\left(\frac{n}{b}\right) + f(n)$$

In the function to the analysis of a recursive algorithm, the constants and function take on the following significance:

- n is the size of the problem.
- a is the number of subproblems in the recursion.
- n/b is the size of each subproblem. (Here it is assumed that all subproblems are essentially the same size.)
- f (n) is the sum of the work done outside the recursive calls, which includes the sum of dividing the problem and the sum of combining the solutions to the subproblems.
- It is not possible always bound the function according to the requirement, so we make three cases which will tell us what kind of bound we can apply on the function.

## Master Theorem:

It is possible to complete an asymptotic tight bound in these three cases:

$$T(n)=\begin{cases} \Theta\left(n^{\log_b a}\right) & f(n)=O\left(n^{\log_b a-\varepsilon}\right) \\ \Theta\left(n^{\log_b a}\log n\right) & f(n)=\Theta\left(n^{\log_b a}\right) \\ \Theta(f(n)) & f(n)=\Omega\left(n^{\log_b a+\varepsilon}\right)\text{AND} \\ & af(n/b)<cf(n)\ \text{forlarge}n \end{cases} \begin{matrix} \\ \\ \varepsilon>0 \\ c<1 \end{matrix}$$

**Case1: If f (n) =** $O\left(n^{\log_b a-\varepsilon}\right)$ **for some constant ε >0, then it follows that:**

**T (n) = Θ** $\left(n^{\log_b a}\right)$

**Example:**

T (n) = 8 T $\left(\frac{n}{2}\right)+1000n^2$ apply master theorem on it.

**Solution:**

Compare T (n) = 8 T $\left(\frac{n}{2}\right)+1000n^2$ with

T (n) = a T $\left(\frac{n}{b}\right) + f(n)$ *with* $a \geq 1$ *and* $b > 1$

a = 8, b=2, f (n) = 1000 $n^2$, $\log_b a = \log_2 8 = 3$

Put all the values in: f (n) = $O\left(n^{\log_b a - \varepsilon}\right)$

1000 $n^2$ = O ($n^{3-\varepsilon}$ )

If we choose ε=1, we get: 1000 $n^2$ = O ($n^{3-1}$) = O ($n^2$)

Since this equation holds, the first case of the master theorem applies to the given recurrence relation, thus resulting in the conclusion:

T (n) = Θ $\left(n^{\log_b a}\right)$

Therefore: T (n) = Θ ($n^3$)

**Case 2: If it is true, for some constant k ≥ 0 that:**

**F (n) = Θ** $\left(n^{\log_b a} \log^k n\right)$ **then it follows that: T (n) = Θ** $\left(n^{\log_b a} \log^{k+1} n\right)$

**Example:**

T (n) = 2 $T\left(\frac{n}{2}\right) + 10n$ , solve the recurrence by using the master method.

As compare the given problem with T (n) = a T $\left(\frac{n}{b}\right) + f(n)$ *with* $a \geq 1$ *and* $b > 1$ a = 2, b=2, k=0, f (n) = 10n, $\log_b a = \log_2 2 = 1$

Put all the values in f (n) =Θ $\left(n^{\log_b a} \log^k n\right)$, we will get

10n = Θ ($n^1$) = Θ (n) which is true.

**Therefore:** T (n) = Θ $\left(n^{\log_b a} \log^{k+1} n\right)$

= Θ (n log n)

**Case 3: If it is true f(n) = Ω** $\left(n^{\log_b a + \varepsilon}\right)$ **for some constant ε >0 and it also true that: a f** $\left(\frac{n}{b}\right) \leq cf(n)$ **for some constant c<1 for large value of n ,then :**

1. T (n) = Θ((f (n))

**Example:** Solve the recurrence relation:

T (n) = 2 $T\left(\frac{n}{2}\right) + n^2$

**Solution:**

Compare the given problem with T (n) = a T $\left(\frac{n}{b}\right) + f(n)$ *with* $a \geq 1$ *and* $b > 1$

a= 2, b =2, f (n) = $n^2$, $\log_b a = \log_2 2 = 1$

Put all the values in f (n) = Ω $\left(n^{\log_b a+\varepsilon}\right)$ ..... (Eq. 1)

If we insert all the value in (Eq.1), we will get

$n^2 = \Omega(n^{1+\varepsilon})$ put ε =1, then the equality will hold.

$n^2 = \Omega(n^{1+1}) = \Omega(n^2)$

Now we will also check the second condition:

$2\left(\frac{n}{2}\right)^2 \leq cn^2 \Rightarrow \frac{1}{2}n^2 \leq cn^2$

If we will choose c =1/2, it is true:

$\frac{1}{2}n^2 \leq \frac{1}{2}n^2$ ∀ n ≥1

So it follows: T (n) = Θ ((f (n))

T (n) = $\Theta(n^2)$

## Master Theorem for divide and conquer recurrences

Master Theorem is used to determine running time of algorithms (divide and conquer algorithms) in terms of asymptotic notations.
Consider a problem that be solved using recursion.

**function f**(input x size n)
**if**(n < k)
solve x directly and return
**else**
divide x into **a** subproblems of size n/b
call f recursively to solve each subproblem
Combine the results of all sub-problems

The above algorithm divides the problem into **a** subproblems, each of size n/b and solve them recursively to compute the problem and the extra work done for problem is given by f(n), i.e., the time to create the subproblems and combine their results in the above procedure.
So, according to master theorem the runtime of the above algorithm can be expressed as:
**T(n) = aT(n/b) + f(n)**

where n = size of the problem
a = number of subproblems in the recursion and a >= 1
n/b = size of each subproblem
f(n) = cost of work done outside the recursive calls like dividing into subproblems and cost of combining them to get the solution.

Not all recurrence relations can be solved with the use of the master theorem i.e. if

- T(n) is not monotone, ex: T(n) = sin n
- f(n) is not a polynomial, ex: $T(n) = 2T(n/2) + 2^n$

This theorem is an advance version of master theorem that can be used to determine running time of divide and conquer algorithms if the recurrence is of the following form :-

$$T(n) = aT(n/b) + \theta(n^k \log^p n)$$

where n = size of the problem
a = number of subproblems in the recursion and a >= 1
n/b = size of each subproblem
b > 1, k >= 0 and p is a real number.

Then,

1. if $a > b^k$, then $T(n) = \theta(n^{\log_b a})$
2. if $a = b^k$, then
   (a) if p > -1, then $T(n) = \theta(n^{\log_b a} \log^{p+1} n)$
   (b) if p = -1, then $T(n) = \theta(n^{\log_b a} \log\log n)$
   (c) if p < -1, then $T(n) = \theta(n^{\log_b a})$
3. if $a < b^k$, then
   (a) if p >= 0, then $T(n) = \theta(n^k \log^p n)$
   (b) if p < 0, then $T(n) = \theta(n^k)$

**Time Complexity Analysis –**

- **Example-1: Binary Search –** T(n) = T(n/2) + O(1)
  a = 1, b = 2, k = 0 and p = 0
  $b^k = 1$. So, $a = b^k$ and p > -1 [Case 2.(a)]
  $T(n) = \theta(n^{\log_b a} \log^{p+1} n)$
  $T(n) = \theta(\log n)$
- **Example-2: Merge Sort –** T(n) = 2T(n/2) + O(n)
  a = 2, b = 2, k = 1, p = 0
  $b^k = 2$. So, $a = b^k$ and p > -1 [Case 2.(a)]
  $T(n) = \theta(n^{\log_b a} \log^{p+1} n)$
  $T(n) = \theta(n \log n)$
- **Example-3:** $T(n) = 3T(n/2) + n^2$
  a = 3, b = 2, k = 2, p = 0
  $b^k = 4$. So, $a < b^k$ and p = 0 [Case 3.(a)]
  $T(n) = \theta(n^k \log^p n)$
  $T(n) = \theta(n^2)$

- **Example-4:** $T(n) = 3T(n/2) + \log^2 n$
  $a = 3, b = 2, k = 0, p = 2$
  $b^k = 1$. So, $a > b^k$ [Case 1]
  $T(n) = \theta(n^{\log_b a})$
  $T(n) = \theta(n^{\log_2 3})$

- **Example-5:** $T(n) = 2T(n/2) + n\log^2 n$
  $a = 2, b = 2, k = 1, p = 2$
  $b^k = 2$. So, $a = b^k$ [Case 2.(a)]
  $T(n) = \theta(n^{\log_b a}\log^{p+1} n)$
  $T(n) = \theta(n^{\log_2 2}\log^3 n)$
  $T(n) = \theta(n\log^3 n)$

- **Example-6:** $T(n) = 2^n T(n/2) + n^n$
  This recurrence can't be solved using above method since function is not of form $T(n) = aT(n/b) + \theta(n^k \log^p n)$

## Master Theorem For Subtract and Conquer Recurrences

Master theorem is used to determine the Big – O upper bound on functions which possess recurrence, i.e which can be broken into sub problems.

**Master Theorem For Subtract and Conquer Recurrences**:

Let T(n) be a function defined on positive n as shown below:

$$T(n) \leq \begin{cases} c, & \text{if } n \leq 1, \\ aT(n-b) + f(n), & n > 1, \end{cases},$$

for some constants c, a>0, b>0, k>=0 and function f(n). If f(n) is $O(n^k)$, then

1. If a<1 then $T(n) = O(n^k)$
2. If a=1 then $T(n) = O(n^{k+1})$
3. if a>1 then $T(n) = O(n^k a^{n/b})$

**Proof of above theorem( By substitution method )**:

From above function, we have:

$T(n) = aT(n-b) + f(n)$
$T(n-b) = aT(n-2b) + f(n-b)$
$T(n-2b) = aT(n-3b) + f(n-2b)$

Now,

$T(n-b) = a^2T(n-3b) + af(n-2b) + f(n-b)$
$T(n) = a^3T(n-3b) + a^2f(n-2b) + af(n-b) + f(n)$
$T(n) = \Sigma^{i=0 \text{ to } n}\, a^i f(n-ib)$ + constant, where f(n-ib) is O(n-ib)
$T(n) = O(n^k \Sigma^{i=0 \text{ to } n/b}\, a^i)$
Where,
If a<1 then $\Sigma^{i=0 \text{ to } n/b}\, a^i = O(1)$, $T(n) = O(n^k)$
If a=1 then $\Sigma^{i=0 \text{ to } n/b}\, a^i = O(n)$, $T(n) = O(n^{k+1})$
If a>1 then $\Sigma^{i=0 \text{ to } n/b}\, a^i = O(a^{n/b})$, $T(n) = O(n^k a^{n/b})$

**Time complexity Analysis:**

The recursive function can be defined as, $T(n) = T(n-1) + T(n-2)$

- For Worst Case, Let $T(n-1) \approx T(n-2)$
  $T(n) = 2T(n-1) + c$
  where,$f(n) = O(1)$
  $\therefore k=0, a=2, b=1;$
  $T(n) = O(n^0 2^{n/1})$
  $= O(2^n)$
- For Best Case, Let $T(n-2) \approx T(n-1)$
  $T(n) = 2T(n-2) + c$
  where,$f(n) = O(1)$
  $\therefore k=0, a=2, b=2;$
  $T(n) = O(n^0 2^{n/2})$
  $= O(2^{n/2})$
- **Example-1**:
  $T(n) = 3T(n-1), n>0$
  $= c, n<=0$
  Sol:a=3, b=1, f(n)=0 so k=0;

  Since a>0, $T(n) = O(n^k a^{n/b})$
  $T(n)= O(n^0 3^{n/1})$
  $T(n)= 3^n$
- **Example-2**:
  $T(n) = T(n-1) + n(n-1)$, if $n>=2$
  $= 1$, if $n=1$
  Sol:a=1, b=1, f(n)=n(n-1) so k=2;

  Since a=1, $T(n) = O(n^{k+1})$
  $T(n)= O(n^{2+1})$
  $T(n)= O(n^3)$
- **Example-3**:
  $T(n) = 2T(n-1) – 1$, if $n>0$
  $= 1$, if $n<=0$
  Sol: This recurrence can't be solved using above method since function is not of form **T(n) = aT(n-b) + f(n)**