A-3

Varun Khadayate A016 70362019028

Q1. Asymptotic notation are the expression that are used to represent the complexity of an algorithm

TYPES OF DATA STRUCTURE ASYMPTOTIC NOTATION

1. Big-O Notation - Big O notation specifies the worst case scenario.
2. Omega Notation (Ω) - Specifies the best case scenario
3. Theta Notation (Θ) - Represents the average complexity of an algorithm.

BIG-O NOTATION (O)

It represents the upper bound running time complexities of an algorithm. Here are some of its examples

a. O(1)

Represents complexity of an algorithm that always executes in same time or space regardless of the input data size.

Eg: Accessing array index (int num=arr[5])

b. O(n)

The execution time will depend on size of array. When the size of array increases, the execution time will also increase in the same proportion. It represents the complexity of an algorithm, whose performance will grow linearly to the size of input data.

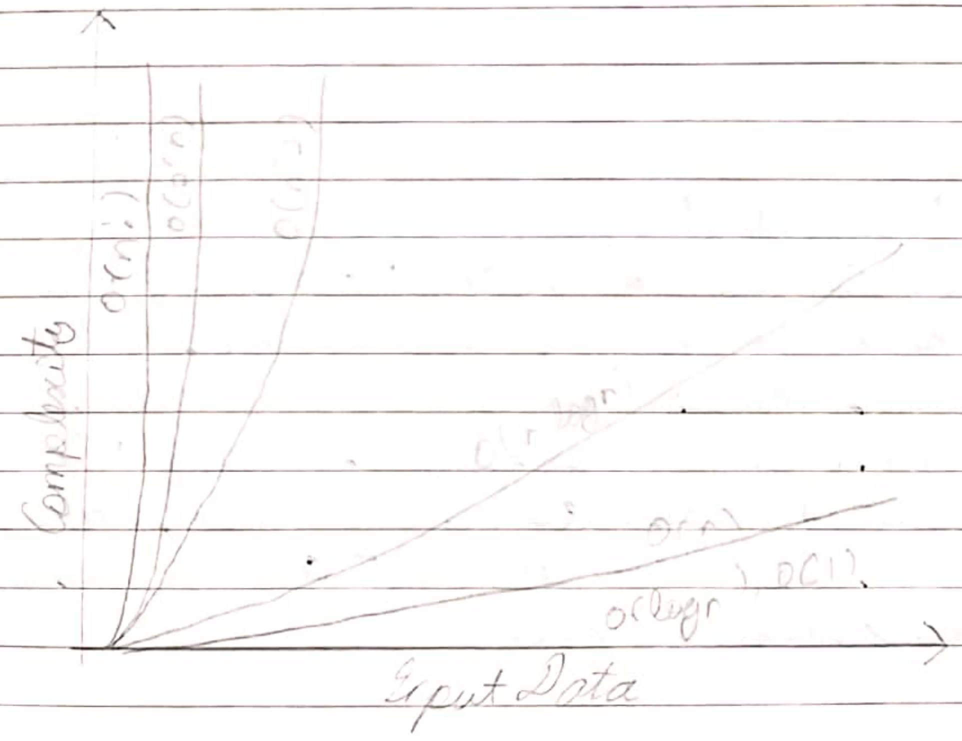
Eg: Traversing an array.

Varun A016 A-3

c. O(n^2)
Represents the complexity of an algorithm, whose performance is directly proportional to the square of the size of the input data.
Eg: Traversing a 2D array

Similarly there are other Big O notation such as:
Logarithmic Growth O(log n); log-linear growth O(n log n); Exponential growth O(2^n); factorial growth O(n!)

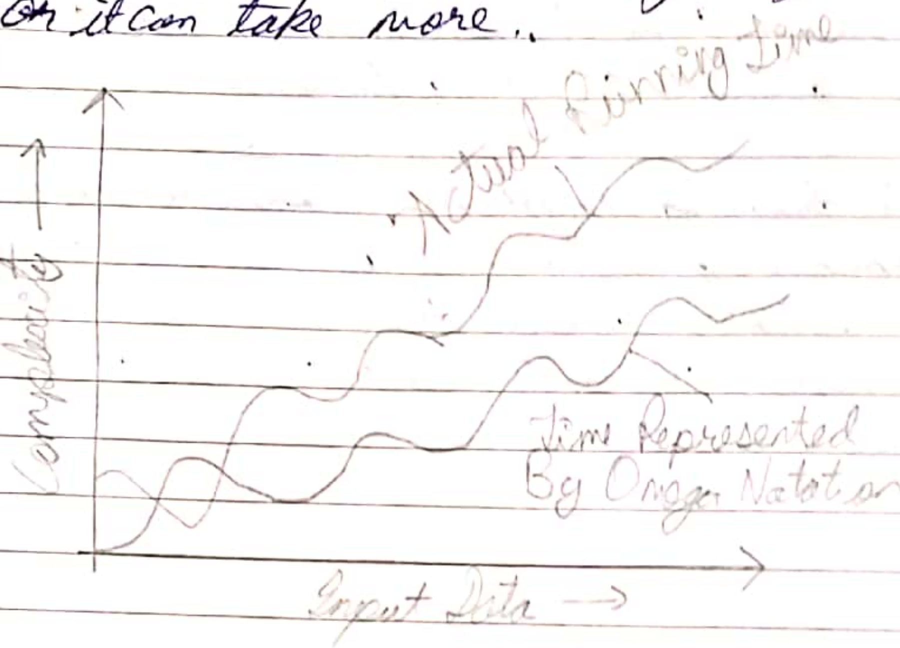
If we have to draw a diagram to ~~complete~~ compare the performance of algorithm denoted by these notation, then we would draw it like this:
O(1) < O(log n) < O(n) < O(n log n) < O(n^2) < O(n^3) < O(2^n) < O(n!)



Varun K A016 A3

Omega Notation (Ω)

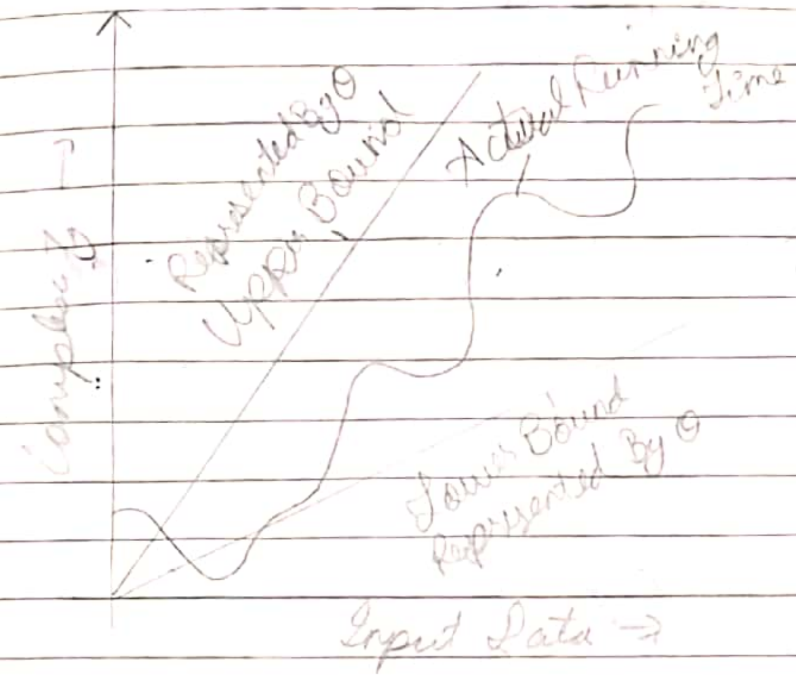
It represents the lower bound running time complexity of an algorithm. So if we represent a complexity of an algorithm in Omega Notation it means that the algorithm cannot be completed in less time than this, it would at-least take take the time represented by Omega Notation or it can take more.



Theta Notation (Θ)

Describes both Upper Bound and Lower Bound of an algorithm so we can say that it defines exact asymptotic behaviour. In the real case scenario the algorithm. In the real case scenario the algorith not always run on best and worst cases, the average running time lies between best and worst and can be represented by the Theta Notation.

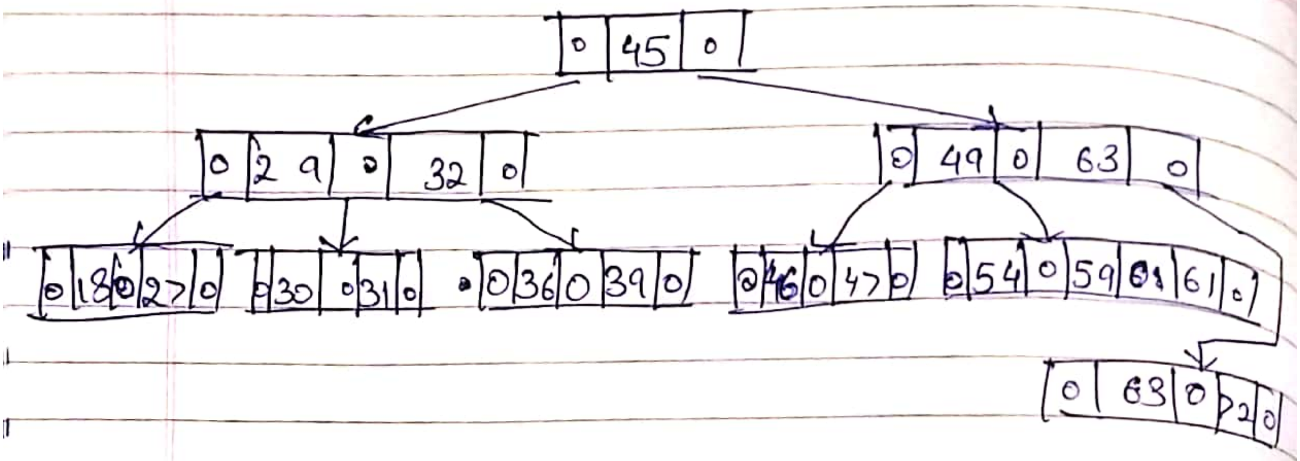
Varun K A016 A3



Q3. B TREE is ~~designing~~ designed to store stored data and allow search, insertion and deletion operation to be performed in logorathmic amortised time. A B tree of order m is a tree with all the properties of an M-way search tree. In addition it has following properties:-

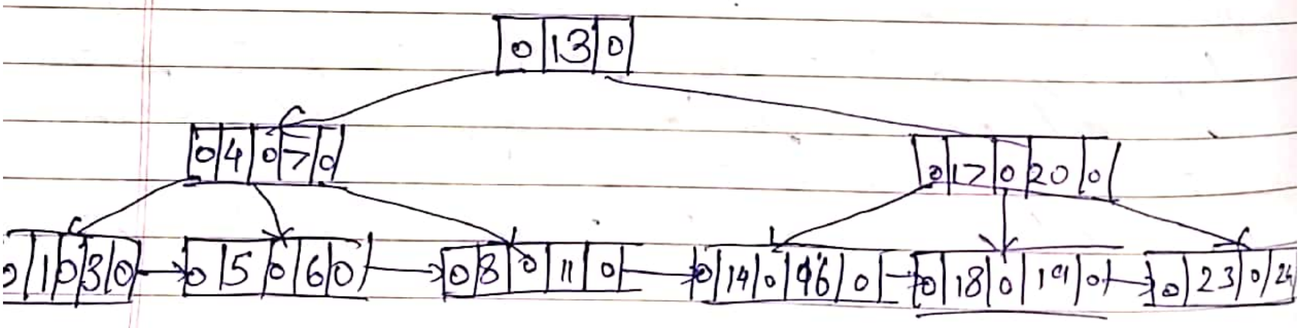
- Every node in the B tree has atmost m children.
- Every node in the B tree except the root node and leaf node has at least m/2 children. This conditions help to keep the tree bushy so that the path from the root node to leaf ~~no~~ is very short, even in a tree that stores a lot of data.
- The root node has at least 2 children if it is not a terminal node.
- All leaf nodes are at same level.

Varun K A016 A3



A B+ Tree is a variant of a B tree which store stored data in a way that allows for efficient insertion, retrieval and removal of records, each of which is identified by a key. A B+ Tree can be thought as a multilevel index in which the leaves make up a dense index and non-leaf node makeup a sparse index. The advantage are:-

- Records can be fetched of equal number of disk access.
- It can be used to perform wide range queries easily.
- Height of tree is less and balanced.
- Supports both random and sequential access to records.
- ~~Re~~ Keys are used for indexing.



Varun K A016 A3

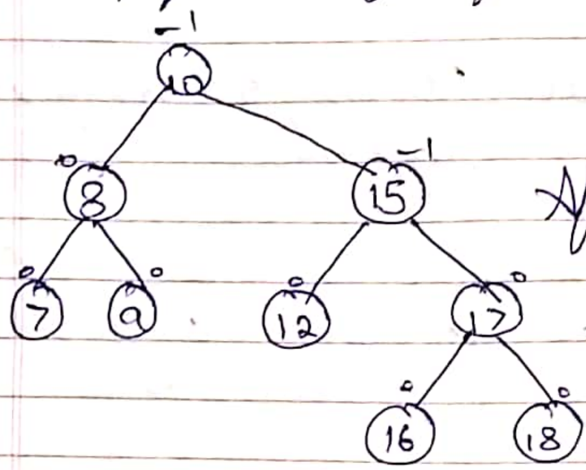
B-TREE	B+-TREE
- Search keys are not repeated	Stores redundant search keys.
- Data is stored in internal or leaf node	Data is stored only in leaf node.
- Searching takes more time as data may be found in leaf or non-leaf node	Searching of data is easy as the data can be found in leaf nodes only.
- Structure and operation are complicated	Structure and operation are simple.

Q4. There are 4 types of AVL Rotation:-
- LL[Left Left] Rotation
- RR[Right Right] Rotation
- LR[Left Right] Rotation
- RL[Right Left] Rotation

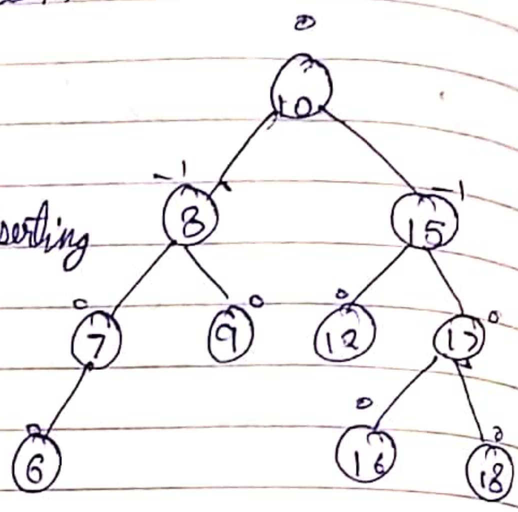
Varun K A016 A-3

- LL Rotation

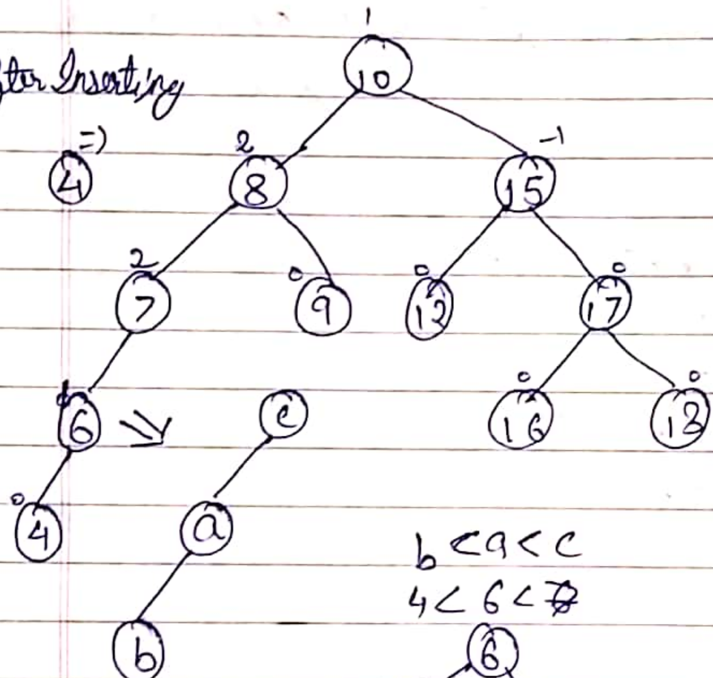
If a tree becomes unbalanced, when a node is inserted into the ~~right~~ left sub tree of the ~~right~~ left subtree, then we perform a ~~single~~ double left rotation.



After Inserting ⇒ 6

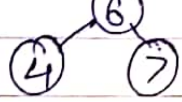


After Inserting ⇒ 4

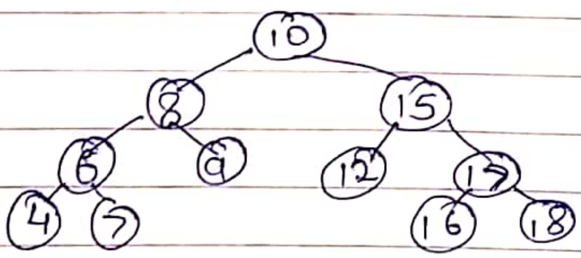


b < a < c

4 < 6 < ~~7~~



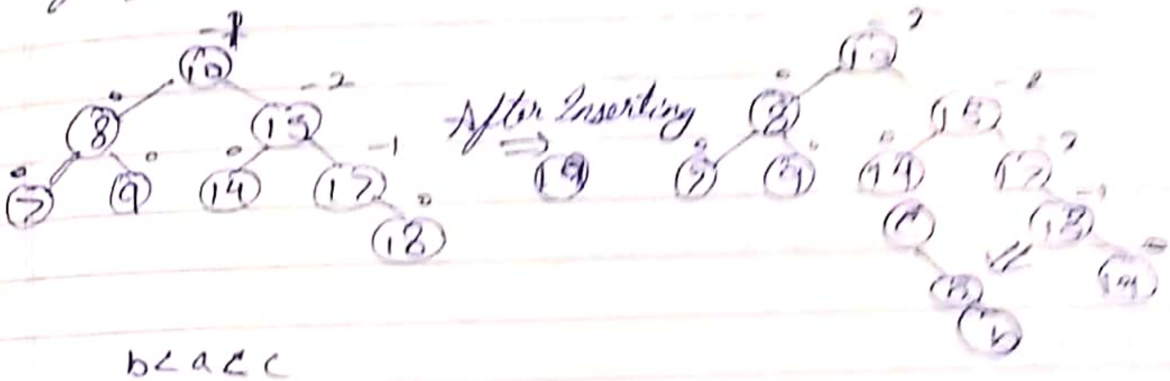
⇓



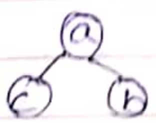
Varun K A016 A3

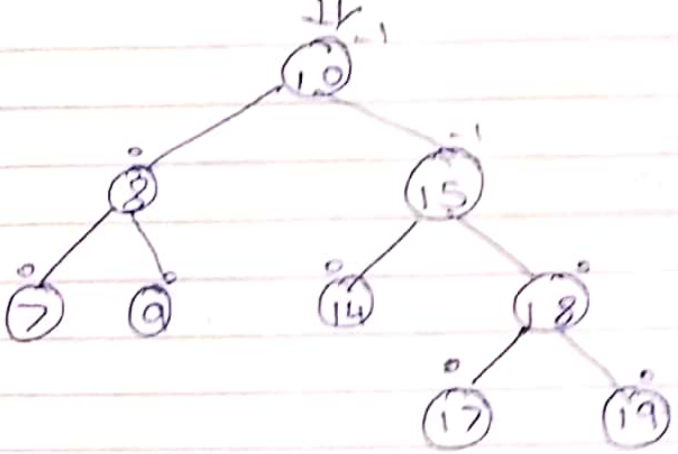
- RR Rotation

If a new node is inserted in the right subtree of the of right subtree of critical node.



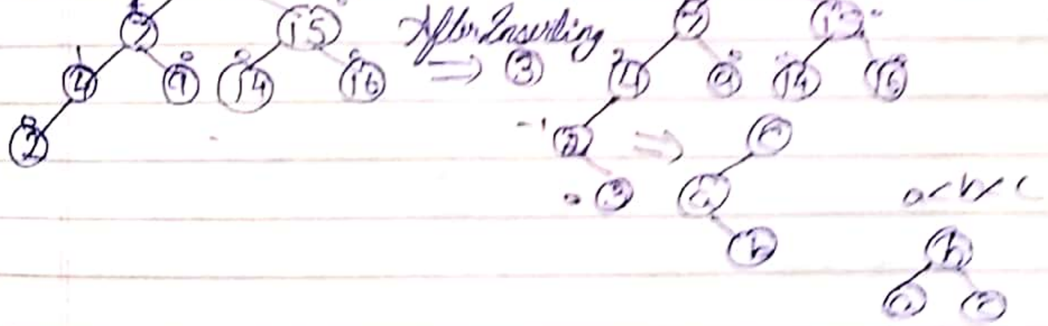
b<a<c



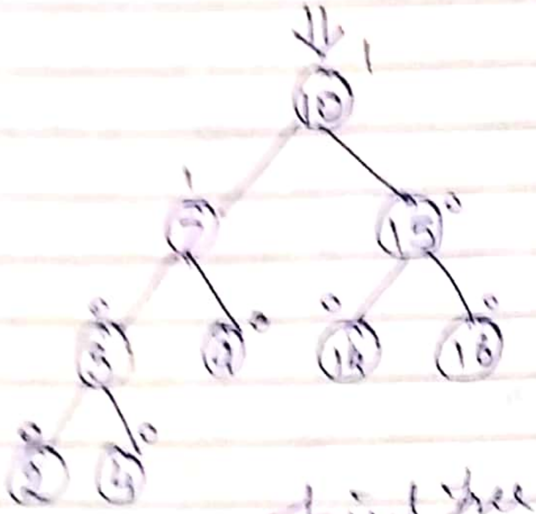


- LR Rotation

The new node is inserted in the right subtree of the left subtree of critical node.



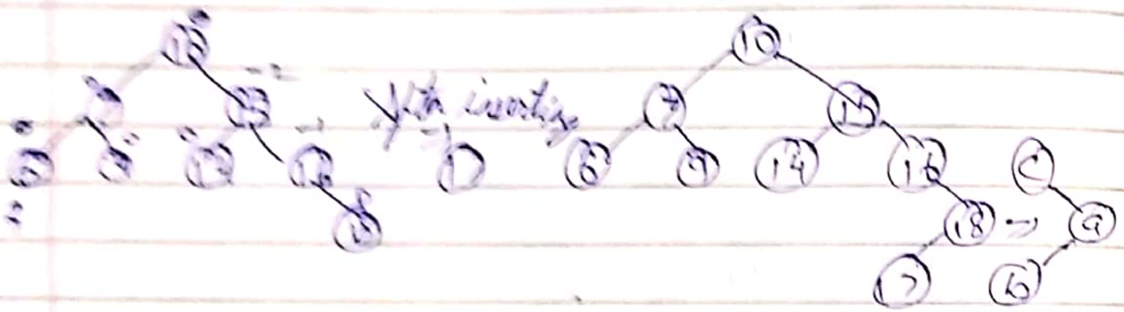
Varun K 2010 13



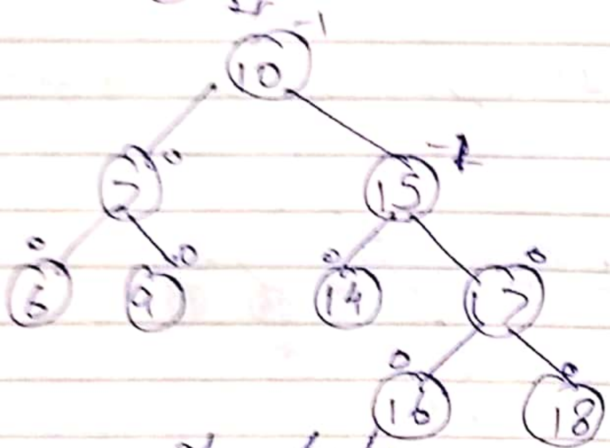
Final tree

- RL Rotation

New node is inserted in the left subtree of the right sub tree to critical node.



c<b<a



Final tree

Varun K A016 A3

Q2. Analysing an algorithm means determining the amount of resources as needed to execute it. Algorithms are generally designed to work with an arbitrary number of inputs, so the efficiency and or complexity of an algorithm is stated in terms of time and space complexity.
The time complexity of an algorithm is basically the running time of a program as a function of the input size. Similarly, the space complexity of an algorithm is the amount of computer memory that is required during the program execution as a function of the input size.
In other words, the number of machine instructions which a program executes is called its time complexity. This number is primarily dependent on size of program's input and algorithm used.
Generally, the space needed by a program depends on the following 2 parts :-
- Fixed Part: It varies from problems to problem. It includes the space needed for storing instruction, constants, variables and structured variables.
- Variable Part: It varies from program to program. It includes the space needed for recursion stack, and for structured variables that are allocated space dynamically during runtime of a program.

Worst Case Running Time Complexity
It denotes the behaviour of an algorithm with respect to the worst possible case of the input instances. The worst case running time of an algrithm is an upper bound on the running time for any input. Therefore, having the knowledge of worst case running time gives us an assurance that the algorithm will never go beyond this time limit.

Varun KA016 A3

Average Case Running Time Complexity
Its an estimate of the running time for an average input. It specifies the expected behaviour of the algorithm when the input is randomly drawn from a given distribution. Thus, it assumes that all inputs of a given size are equally likely.

Amortized Case Running Time Complexity
It refers to the time required to perform a sequence of operations performed. It guarantees the average performance of each operation in the worst case.

Time-Space Trade off
The best algorithm to solve a particular problem at hand is no doubt the one that requires less memory space and takes less time to complete its execution. But practically, designing such an ideal algorithm is not a trivial task. There can be more than one algorithm to solve a particular problem. One may require less memory space while other may require less CPU time to execute. Thus, it is not uncommon to sacrifice one thing for other. Hence, there exist a time space trade off among algorithms.
So, if space is a big constraint, then one might choose a program which takes less space at cost of more CPU time. On the contrary, if time is a major constraint, then one might choose a program which takes minimum time to execute at cost of more space.

Varun K A016 A3

Expressing Time And Space Complexities.
The time and space complexity can be expressed using a function f(n) where n is the input size for a given instance of the problem being solved.
Expressing the complexity is required when
- We want to predict the rate of growth of complexity as the input size of problem increases.
- There are multiple algorithms that find a soln to a given problem and we need to find the algorithm that is most efficient.
The most widely used notation to express this function f(n) is the Big O Notation. It provides the upper bound for the complexity.

Algorithm Efficiency
If a function is linear, the efficiency of that algorithm or running time of that algorithm can be given as number of instruction it contains. However, if an algorithm contains loops, then the efficiency of that algorithm may vary depending on the number of loops and running time of each loop in the algorithm.
Linear Loops
To calculate the efficiency of an algorithm that has a single loop, we need to first determine the number of times the statement in loop will be executed. This is because the number of iteration is directly proportional to the loop factor. Greater the loop factor, more is the number of iterations.

$$f(n) = n/2$$

Logarithmic Loop

The loop-controlling variable is either multiplied or divided during each iteration of the loop. Therefore, putting this analysis in general terms, we can conclude that the efficiency of loops in which iterations divide or multiply the loop-controlling variables can be given as

$$f(n) = \log n$$

Nested Loop

Loops that contain loops are known as nested loops. In order to analyse nested loops, we need to determine the number of iterations each loop completes. The total is then obtained as the product of the number of iteration in the inner loop and the number of iteration in the outer loop. In this case, we analyse the efficiency of the algorithm based on whether it is linear logarathmic, quadratic or dependent quadratic nested loop.

Linear Logarathmic Loop

Consider the following code in which the loop controlling variable of inner loop is multiplied after each iteration. In more general terms, the efficiency of such loop can be given as $f(n) = n\log n$.

Quadratic Loop

The number of iteration in the inner loops is equal to the number of iteration in outer loops. The generalised formula for quadratic loop can be given as $f(n) = n^2$

Dependent Quadratic Loop

The number of iteration in inner loop is dependent to number of iteration in outer loop. In general terms, the inner loop iterates $(n+1)/2$ times. Therefore, the efficiency of such a code can be given as

$$f(n) = n(n+1)/2$$

Scanned with CamScanner