Module-1: Introduction to Algorithms

1.1 Algorithm:

Abu Jafar Mohammed Ibn Musa Al khowarizmi wrote a text book entitled "Algorithmi de numero indorum" from which the term "Algorithmi" has led to the term Algorithm. It is an effective way of finding a solution to a problem is by using an algorithm. Thus, the solution will be written as a sequence of instructions conveying the problem.

Definition:

> **Algorithm** is a step-by-step procedure to solve a computational problem in a finite amount of time.
>
> OR
>
> An Algorithm is a step-by-step plan for a computational procedure that possibly begins with an input and yields an output value in a finite number of steps in order to solve a particular problem.

As an illustration, let's consider the task of preparing a cup of tea. The algorithm involves the following steps:

1. Pour milk and water into the kettle.
2. Boil the mixture.
3. Add tea leaves and sugar.
4. Stir, filter and serve the tea in a cup.

In general, to complete a task an algorithm can be defined as a set of steps, with the level of precision required to enable execution by a computer. However, achieving this level of precision can be challenging for machines, as evidenced by the difficulty in specifying exact quantities of water, milk, etc., in the aforementioned algorithm for tea-making.

Such algorithms are designed to operate on computational devices or computers. For instance, smartphones have algorithms such as GPS and Google Hangouts. The GPS employs a shortest path algorithm, while online shopping relies on cryptography, specifically utilizing the RSA algorithm.

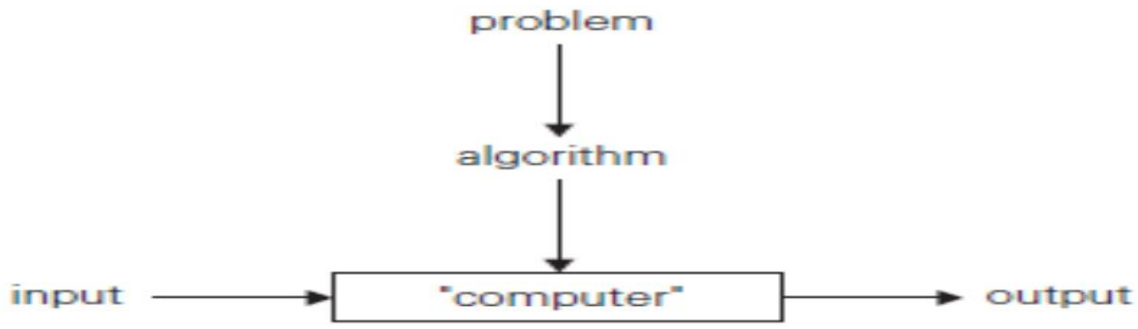
Definition-1 of Algorithm:

An algorithm is a finite set of instructions crafted to accomplish a specific task. Essential criteria for algorithms include:

- Input: Accepts zero or more externally supplied quantities.
- Output: Generates at least one quantity.
- Definiteness: Each instruction is clear and unambiguous.
- Finiteness: The algorithm concludes after a finite number of steps.
- Effectiveness: Every instruction must be sufficiently basic and feasible.

Definition-2 of Algorithm:

- An algorithm is a sequence of unambiguous instructions formulated to solve a problem, ensuring the generation of a required output for any legitimate input within a finite timeframe.
- Algorithms that are both definite and effective are also denoted as computational procedures.
- A program serves as the implementation of an algorithm in a programming language.



Problem Solving Algorithm Steps:

1. Define the problem.
2. Design or specify the algorithm.
3. Analyze the algorithm.
4. Implement the solution.
5. Perform testing.
6. Maintenance

Step 1: Define the problem

- Determine the task to be completed. Example: Calculating average of grades of a student.

Step 2: Design or specify the algorithm

- Describe the solution using natural language, pseudo-code, diagrams, etc.

Step 3: Analyze the algorithm

- Assess the algorithm's space complexity (required space) and time complexity (execution time).

Computer Algorithm

- An algorithm is a procedure, consisting of a finite set of well-defined instructions, designed to achieve specific tasks. It initiates from an initial state and concludes in a defined end-state.
- Computational complexity and efficient implementation depend on suitable data structures.

Steps 4, 5, 6: Implement the solution, Perform testing, Maintenance:

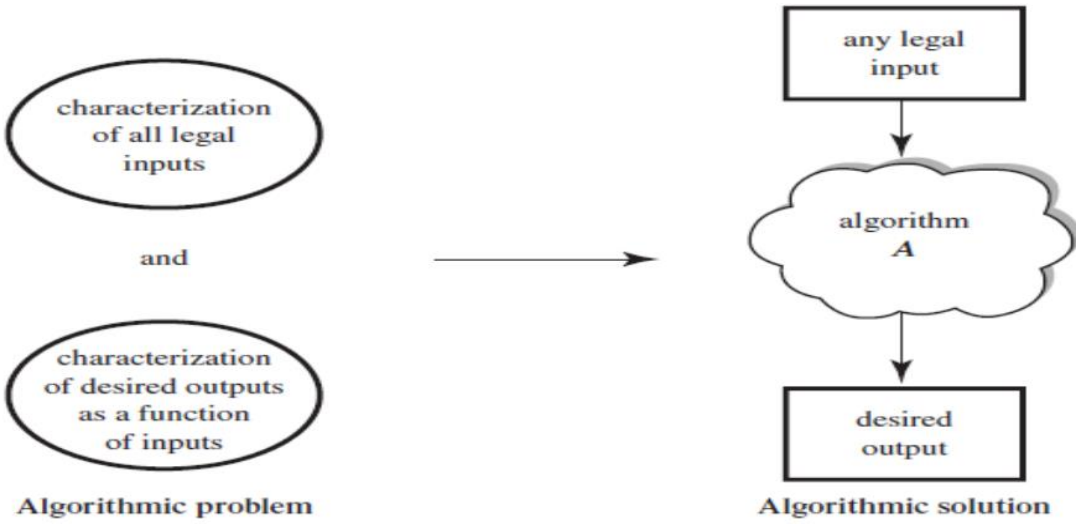
- Choose a programming language (e.g., C, C++, Lisp, Java, Perl, Prolog, assembly).
- Write clean, well-documented code.

Testing:

- Rigorously test the code.
- Integrate user feedback, fix bugs, and ensure compatibility across different versions.

Maintenance:

- Release updates and address bugs.
- The algorithmic problem handles illegal inputs, while the algorithm itself manages special classes of unusual or undesirable inputs.



Four Distinct Areas of Algorithmic Study:

1. **Algorithm Devising Techniques:**
    - Focus on developing algorithms using techniques like Divide & Conquer, Branch and Bound, and Dynamic Programming

2. **Algorithm Validation:**
    - Ensure correctness by checking if the algorithm produces accurate results for all possible legal inputs.
        - First Phase: Algorithm validation.
        - Second Phase: Convert Algorithm to Program, also known as Program Proving or Program Verification.
            - Solutions can be stated in two forms:
                - First Form: Program annotated with assertions about input and output variables using predicate calculus.
                - Second Form: Specification.
3. **Algorithm Analysis:**
    - Assess the computing time and storage requirements of an algorithm.
        - Performance analysis, or the Analysis of Algorithms, involves determining the efficiency of an algorithm.

        1. The process of executing a correct program on datasets while measuring the time and space needed to compute the results.

4. **Program Testing:**
    1. Consists of two phases:
        1. **Debugging:**
            1. The process of executing programs on sample datasets to identify and correct faulty results.
        2. **Profiling or Performance Measurement:**

1.2 Algorithm pseudocode conventions

Algorithms can be depicted in both Text and Graphic modes:

- The graphical representation is known as a Flowchart.
- Text mode is typically represented in proximity to high-level languages like C and Pascal, often referred to as Pseudocode.
- Pseudocode is a high-level description of an algorithm:
    - It is more structured than plain English.
    - It is less detailed than an actual program.
    - It is a preferred notation for describing algorithms.
    - It helps conceal program design intricacies.

Example:

To determine the maximum element within an array.

```
Function arrayMax(A, n)
Input: Array A of n integers
Output: Maximum element of A
currentMax ← A[0]
for i ← 1 to n - 1 do
 if A[i] > currentMax then
 currentMax ← A[i]
return currentMax
```

This algorithm takes an array A of n integers as input and outputs the maximum element of the array. It initializes a variable currentMax with the first element of the array and then iterates through the remaining elements, updating currentMax if a larger element is encountered. Finally, the algorithm returns the calculated maximum element.

Control Flow:

- if ... then ... [else ...]
- while ... do ...
- repeat ... until ...
- for ... do ...
- Indentation replaces braces

Method Declaration:

- Algorithm method (arg [, arg...])

Input and Output:

- Input …
- Output …

Method Call:

- var.method (arg [, arg,…])

Return Value:

- return expression

Expressions:

- Assignment (equivalent to =)
- Equality testing (equivalent to ==)
- n^2 (Superscripts and other mathematical formatting allowed)

1.3 Performance analysis

Criteria for Evaluating Algorithms:

1. Direct Relationship to Performance Criteria:
   - Computing time and storage requirements.
2. Performance Evaluation Phases:
   - Performance evaluation involves two major phases:
     - A priori estimates.
     - A posteriori testing.
   - These phases are commonly referred to as performance analysis and performance measurement, respectively.
3. Space Complexity:
   - The space complexity of an algorithm is the amount of memory it requires to run to completion.
4. Time Complexity:
   - The time complexity of an algorithm is the amount of computer time it needs to run to completion.

1.3.1 Space Complexity Example

```
Algorithm abc(a, b, c) {
   return a + b++ * c + (a + b - c) / (a + b) + 4.0;
}
```

The space required by each of these algorithms can be analyzed as the sum of distinct components:

1. **Fixed Part:**
   - Independent of input and output characteristics.
   - Includes instruction space, space for code, simple variables, fixed-size component variables (aggregate), space for constants, etc.

1. **Variable Part:**
    - Depends on the specific problem instance being solved.
    - Comprises space needed by component variables whose size varies based on the problem instance, space for referenced variables (based on instance characteristics), and recursion stack space.

The space requirement $S(p)$ of any algorithm p can be expressed as:

$S(p)=c+Sp$(Instance characteristics) where 'c' is a constant.

Example 2: Algorithm for calculating the sum of elements in an array:

```
Algorithm sum(a, n)
{
   s = 0.0;
   for i = 1 to n do
      s = s + a[i];
   return s;
}
```

- Problem instances are characterized by 'n,' the number of elements to be summed.
- Space needed by 'n' is one word, being of type integer.
- Space needed by 'a' is at least 'n' words, considering 'a' as an array of floating-point numbers.
- Hence, $Ssum(n)\geq(n+s)$ where [n for a[], one each for n,i,a & s].

1.3.2 Time Complexity

- Time $T(p)$ taken by a program P is the sum of compile time and run time (execution time).
- Compile time is independent of instance characteristics.
- Run time is denoted by tp(instance characteristics).

Determining Step Count:

1. **Algorithmic Representation:**
    - Introduce a variable 'count' into the program to increment with an initial value of 0.
    - Increment 'count' with the appropriate amount for each statement in the program.
    - Each invocation of the algorithm executes a total of $2n+3$ steps.

2. Table Method:

- Determine the number of steps per execution (s/e) for each statement.
- Identify the total number of times (frequency) each statement is executed.
- Combining these quantities yields the total step count for the entire algorithm.

Statement	*Steps per execution*	*Frequency*	*Total*
1. Algorithm Sum(a,n) 2.{ 3. S=0.0; 4. for I=1 to n do 5. s=s+a[I]; 6. return s; 7. }	0 0 1 1 1 1 0	- - 1 n+1 n 1 -	0 0 1 n+1 n 1 0
Total			2n+3

How to analyze an algorithm?

Let's create an algorithm for Insertion Sort, a process for sorting a sequence of numbers. The pseudo-code for the algorithm is provided below:

Pseudo-code for Insertion Algorithm:

Identify each line of the pseudo-code with symbols such as C1, C2, and so on.

PSeudocode for Insertion Algorithm	**Line Identification**
for j=2 to A length	C1
key=A[j]	C2
//Insert A[j] into sorted Array A[1.....j-1]	C3
i=j-1	C4
while i>0 & A[j]>key	C5
A[i+1]=A[i]	C6
i=i-1	C7
A[i+1]=key	C8

Let, *Ci* represents the algorithms *i*th cost. Note that comment lines contribute zero cost, *C*3=0.

Cost	**No. Of times Executed**
C1	N
C2	n-1
C3=0	n-1
C4	n-1
C5	$\sum_{j=2}^{n-1} t_j$
C6	$\sum_{j=2}^{n} t_j - 1$
C7	$\sum_{j=2}^{n} t_j - 1$
C8	n-1

The execution time of the algorithm is:

The time complexity of the algorithm is expressed as:

$$T(n) = C1n + C2(n-1) + O(n-1) + C4(n-1) + C5 + C6 + C7 + C8(n-1)$$

Best case:

In the best-case scenario, when the array is sorted and all tj values are set to 1, the time complexity T(n) simplifies to (C1 + C2 + C4 + C5 + C8) n - (C2 + C4 + C5 + C8), taking the form of an linear function an + b, indicating linear growth.

Worst case:

For the worst-case scenario, where the array is reverse sorted and tj = j, the time complexity T(n) takes the form of an2 + bn + c, representing a quadratic function. This implies that in the worst case, the insertion set grows quadratically with n.

Importance of Worst-Case Running Time: Analyzing the worst-case running time is crucial because it provides a guaranteed upper bound on the running time for any input. In many algorithms, the worst case is a common occurrence, especially when dealing with scenarios like searching for an absent item. Analyzing the average case is often not preferred because it tends to be as bad as the worst case.

Order of Growth:

The order of growth is determined by the highest degree term in the formula for running time, excluding lower-order terms and ignoring constant coefficients. In the example of insertion sort, the worst-case running time is expressed as an2 + bn + c. By dropping lower-order terms and ignoring constant coefficients, the result is n2. It's important to note that we say the running time is Θ(n2) to convey that it grows like n2, but it doesn't necessarily equal n2.

Comparing Algorithm Efficiency:

In the realm of algorithm analysis, we typically deem one algorithm more efficient than another if its worst-case running time exhibits a smaller order of growth.

Algorithm Complexity: The complexity of an algorithm, denoted by the function f(n), encapsulates the running time and/or storage space requirements relative to the input data size 'n.' In most cases, the storage space needed by an algorithm is a straightforward multiple of the data size. Here, complexity specifically refers to the running time of the algorithm.

Types of Complexity Functions: The function f(n) representing the running time of an algorithm depends not only on the input data size 'n' but also on the specific data itself. Key complexity functions include:

1. Best Case: The minimum possible value of f(n), known as the best case.
2. Average Case: The expected value of f(n).
3. Worst Case: The maximum value of f(n) for any key possible input.

1.3.3 Asymptotic Notation

Asymptotic notation serves as a formal way to discuss and classify functions in performance analysis. Commonly used notations include:

1. Big–OH (O)
2. Big–OMEGA (Ω)
3. Big–THETA (Θ)
4. Little–OH (o)

Asymptotic Analysis of Algorithms: The approach to algorithm analysis relies on asymptotic complexity measures. Instead of counting the exact number of steps in a program, the focus is on how that number

grows concerning the input size. This provides a measure applicable across different operating systems, compilers, and CPUs. Asymptotic complexity is expressed using big-O notation, describing the function's characteristics in the limit. It allows for the comparison of function sizes using notations such as O (≤), Ω (≥), Θ (=), o (<), and ω (>).

Time complexity	**Name**	**Example**
O(1)	Constant	Adding an element to the front of a linked list
O(logn)	Logarithmic	Finding an element in a sorted array
O (n)	Linear	Finding an element in an unsorted array
O(nlog n)	Linear	Logarithmic Sorting n items by 'divide-and-conquer'- Mergesort
$O(n^2)$	Quadratic	Shortest path between two nodes in a graph
$O(n^3)$	Cubic	Matrix Multiplication
$O(2^n)$	Exponential	The Towers of Hanoi problem

Big O Notation: In the context of Big O notation, the function f(n) is denoted as O(g(n)) if and only if there exist positive constants c and n_o such that $f(n) \leq c * g(n)$ for all n, where $n \geq n_o$.

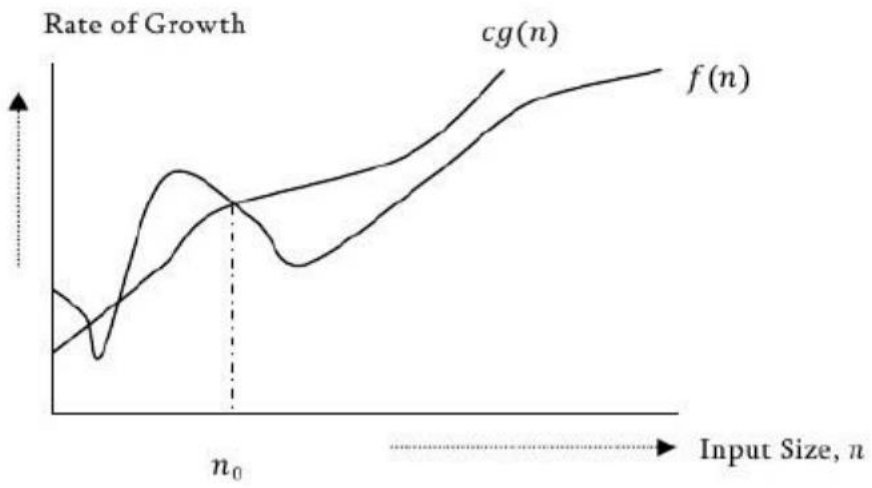
Omega Notation: For Omega notation, the function f(n) is represented as Ω(g(n)) if there exist positive constants c and n_o such that $f(n) \geq c * g(n)$ for all n, where $n \geq n_o$.

Theta Notation: In Theta notation, the function f(n) is expressed as Θ(g(n)) if and only if there exist positive constants c_1, c_2, and n_o such that $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$ for all n, where $n \geq n_o$.

Understanding Big-O Notation: Big O notation provides a tight upper bound for a given function. It is typically denoted as f(n) = O(g(n)), signifying that, at larger values of n, the upper bound of f(n) is g(n). For instance, if the algorithm's function is $f(n) = n^4 + 100n^2 + 10n + 50$, then n^4 serves as g(n), indicating the maximum rate of growth for f(n) at larger values of n.

The O-notation is defined as O(g(n)) = {f(n): there exist positive constants c and n_o such that $0 \leq f(n) \leq c * g(n)$ for all $n \geq n_o$}. Here, g(n) serves as an asymptotic tight upper bound for f(n). The primary objective is to identify a rate of growth, g(n), that surpasses the given algorithm's rate of growth, f(n).

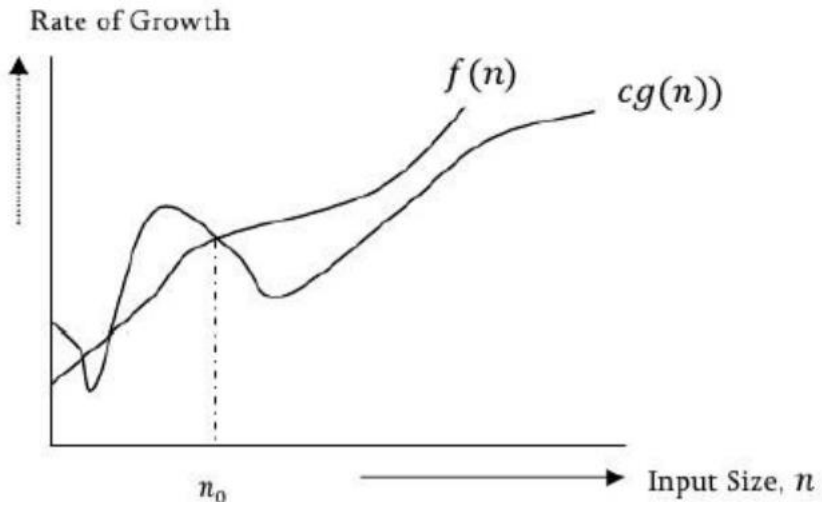
Consideration of Lower Values of n: In general, lower values of n are not taken into account, signifying that the rate of growth at lower values of n is not considered important. The point n_o marks the threshold from which the rate of growth for a given algorithm is considered. Below n_o, the rate of growth may exhibit variations.



Consideration of Algorithm Analysis: It is essential to analyze algorithms primarily at larger values of n. Below the threshold denoted as n_o, there is no concern for the rates of growth.

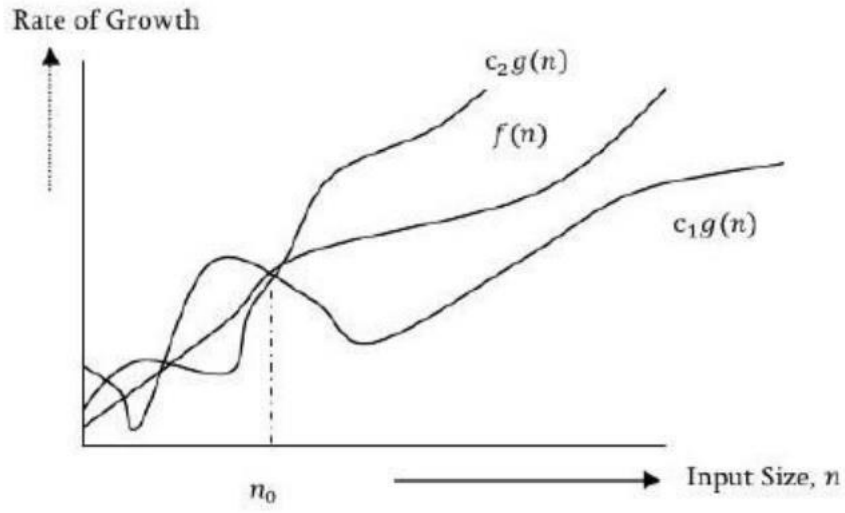
Omega Notation (Ω): Similar to the preceding discussion, Omega notation provides a tighter lower bound for a given algorithm, represented as f(n) = Ω(g(n)). This implies that at larger values of n, the algorithm's tighter lower bound is g(n). For instance, if $f(n) = 100n^2 + 10n + 50$, g(n) is denoted as $\Omega(n^2)$.

Definition of Ω Notation: The Ω notation is defined as Ω(g(n)) = {f(n): there exist positive constants c and n_o such that $0 \leq c * g(n) \leq f(n)$ for all $n \geq n_o$}. Here, g(n) serves as an asymptotic lower bound for f(n). The set Ω(g(n)) encompasses functions with a smaller or equal order of growth compared to f(n).



Theta Notation (Θ): The Θ notation serves to determine whether the upper and lower bounds of a given function are identical. It effectively encapsulates the average running time of an algorithm within the bounds of both the lower and upper limits. If the upper bound (O) and lower bound (Ω) yield identical results, then the Θ notation will also exhibit the same rate of growth. Consider the example of f(n) = 10n + n. In this case, its tight upper bound, g(n), is denoted as O(n), and the rate of growth in the best case is g(n) = Ω(n). With the rates of growth being the same for both best and worst cases, the average case follows suit.

Non-Matching Bounds: Conversely, if the bounds for O and Ω are not identical for a given function (algorithm), the rate of growth in the Θ case may not align.



Definition of Θ Notation: The Θ notation is defined as Θ(g(n)) = {f(n): there exist positive constants C1, C2, and n_o such that 0 <= C1 * g(n) <= f(n) <= C2 * g(n) for all n ≥ n_o}. Here, g(n) is an asymptotic tight bound for f(n). The set Θ(g(n)) comprises functions with the same order of growth as g(n).

Key Points: In algorithm analysis, we strive to provide upper bounds (O), lower bounds (Ω), and average running times (Θ) for best, worst, and average cases. However, it is important to note that obtaining upper bounds (O), lower bounds (Ω), and average running times (Θ) for a given function (algorithm) may not always be feasible.

For instance, when discussing the best case of an algorithm, efforts are made to present upper bounds (O) and lower bounds (Ω) along with average running time (Θ). In subsequent chapters, the focus is generally on upper bounds (O), as knowledge of the lower bound (Ω) is often not practically significant. The Θ notation is employed when upper bound (O) and lower bound (Ω) coincide.

Little Oh Notation: The little oh notation, denoted as "o," is defined as follows: Let f(n) and g(n) be non-negative functions such that f(n) = o(g(n)), indicating that f(n) is considered little oh of g(n). Specifically, f(n) = o(g(n)) if and only if f'(n) = o(g(n)) and f(n) is not Θ(g(n)).

1.3 Amortized analysis

Amortized analysis involves averaging the time required for a sequence of data structure operations across all performed operations. This analysis allows us to demonstrate that the average cost of an operation remains small when considering the entire sequence, even if individual operations within that sequence might be resource-intensive. Notably, amortized analysis focuses on worst-case scenarios and differs from average-case analysis, as it does not involve probability considerations.

Three primary techniques employed in amortized analysis are as follows:

Aggregate Analysis:

In this approach, an upper bound T(n) on the total cost of a sequence of n operations is determined.
The average cost per operation is calculated as T(n)/n, and this average cost is considered as the amortized cost for each operation.

Accounting Method:

When dealing with multiple types of operations, each operation type may have a distinct amortized cost.

The accounting method involves overcharging certain operations early in the sequence, accumulating this overcharge as "prepaid credit" on specific objects in the data structure.
Later in the sequence, this credit is used to offset operations that are charged less than their actual cost.

Potential Method:

The potential method manages the credit as the "potential energy" of the entire data structure, rather than associating it with individual objects within the structure.
Similar to the accounting method, the potential method determines the amortized cost of each operation and may involve overcharging operations early on to compensate for undercharges later in the sequence.

1.4 Recurrence

A recurrence is an equation or inequality that characterizes a function by expressing it in relation to its values on smaller inputs. Solving a recurrence relation entails deriving a function defined on the natural numbers that meets the conditions specified by the recurrence. For instance, the recurrence describing the Worst Case Running Time T(n) of the MERGE SORT procedure is as follows:

$$\begin{cases} \theta(1) & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + \theta(n) & \text{if } n > 1 \end{cases}$$

Example:

Consider solving the equation using the Substitution Method:

$T(n) = TDAA \text{ Recurrence Relation} + n$

The objective is to demonstrate that it is asymptotically bounded by $O(\log n)$.

Solution:

To show that $T(n) = O(\log n)$, we need to prove that for some constant $c, T(n) \leq c \log n$. Substituting this into the given recurrence equation:

$T(n) \leq c \log DAA \text{ Recurrence Relation} + 1$

$\leq c \log DAA \text{ Recurrence Relation} + 1 = c \log n - c \log 2 + 1$

$\leq c \log n \text{ for } c \geq 1$

Thus, $T(n) = O(\log n)$.

There exist three methods for resolving recurrences:
Substitution Method
Recursion Tree Method
Master Method

1. **Substitution Method:** The Substitution Method involves two primary steps: a. Guessing the solution. b. Employing mathematical induction to determine the boundary condition and validating the correctness of the guess.

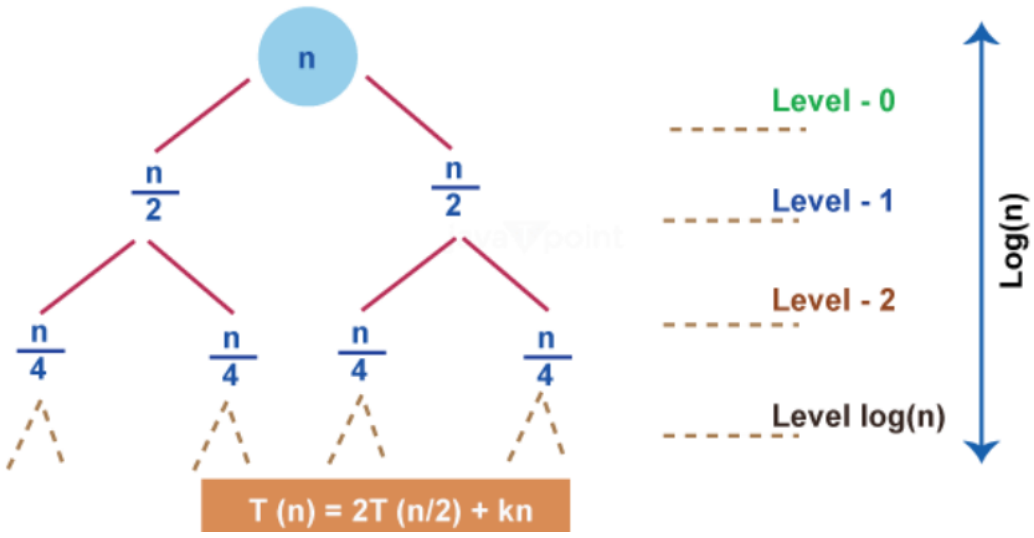
1. **Recursion Tree Method**

   Recurrence relations, such as T(N) = T(N/2) + N or those previously discussed in the types of recursion section, are addressed using the recursion tree approach. These relations often employ a divide and conquer strategy to tackle problems.

Integrating solutions to smaller subproblems generated when breaking down a larger problem into more manageable components takes time. For example, in the recurrence relation T(N) = 2 * T(N/2) + O(N) for Merge Sort, the time required to combine answers from two subproblems with a combined size of T(N/2) is denoted as O(N), a representation valid at the implementation level.

Consider the recurrence relation for binary search, T(N) = T(N/2) + 1, where each iteration halves the search space. The +1 is added to account for the constant time operation involved in determining the outcome before exiting the function.

Another noteworthy recurrence relation is T(n) = 2T(n/2) + Kn, where Kn represents the time needed to combine answers from n/2-dimensional subproblems. Let's visualize the recursion tree for this particular recurrence relation.



Several conclusions can be drawn from an examination of the recursion tree presented:

1. The significance of a node's value is solely determined by the magnitude of the problem at each level. The issue size is n at level 0, n/2 at level 1, n/2 at level 2, and so forth.
2. Generally, the height of the tree is defined as log(n), where n denotes the size of the problem. The height of the recursion tree corresponds to the number of levels in the tree. This correlation holds because recurrence relations employ a divide-and-conquer strategy, and transitioning from an issue size of n to a problem size of 1 necessitates log(n) steps. For example, consider the value N = 16. If we can divide N by 2 at each step, the number of steps required to reach N = 1 is calculated by log(16) base 2, yielding 4 steps.
3. At each level, the second term in the recurrence is considered as the root. Despite the term "tree" in the strategy's name, one need not be an expert on trees to comprehend it.

3. **Master Method:** The Master Method is a valuable tool for solving recurrence relations of the form $T(n)=a\cdot T(bn)+f(n)$, where $a\geq 1$, $b\geq 1$ are constants, and $f(n)$ is a function. The Master Method can be understood as follows:
   Consider a recurrence relation $T(n)=a\cdot T(bn)+f(n)$, and interpret it in the analysis of a recursive algorithm, where the constants and functions carry specific significance:
   - n represents the size of the problem.
   - a is the number of subproblems in the recursion.
   - bn indicates the size of each subproblem (assuming all subproblems are essentially of the same size).
   - $f(n)$ corresponds to the sum of the work done outside the recursive calls, encompassing the effort of dividing the problem and combining the solutions to the subproblems.

   Given that it is not always possible to bound the function according to specific requirements, three cases are established to determine the type of bound applicable to the function.

For Numerical examples, solved in a class. Find the below links

Master's theorem
https://www.youtube.com/watch?v=OynWkEj0S-s