**Unit - 1**

Algorithms Analysis: Algorithms and structured programming. Analyzing algorithms, asymptotic behavior of an algorithm, recurrence relation, Order notations, time and space complexities, average and worst case analysis, lower and upper bounds.

**Why the Analysis of Algorithm 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:**

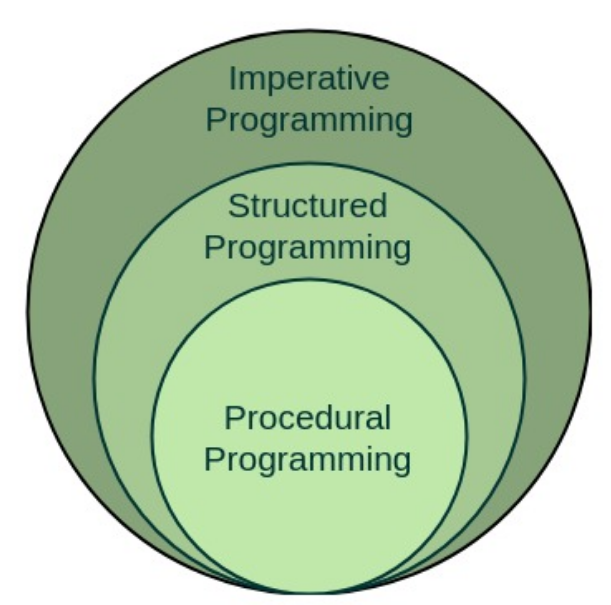
1. Best case
2. Worst case
3. Average case

* **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 case, 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

**Structured Programming**

Structured Programming can be defined as a programming approach in which the program is made as a single structure. It means that the code will execute instruction by instruction one after the other. It doesn’t support the possibility of jumping from one instruction to some other with the help of any statement like GOTO, etc. Therefore, the instructions in this approach will be executed in a serial and structured manner. The languages that support Structured programming approach are : C,C++,Java,C#,etc.



**Asymptotic Behaviour of an algorithm**

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.

**Asymptotic Notations:**

* Asymptotic Notations are mathematical tools used to analyze the performance of algorithms by understanding how their efficiency changes as the input size varies.
* By using asymptotic notations, such as Big O, Big Omega, and Big Theta, we can categorize algorithms based on their worst-case, best-case, or average-case time or space complexities, providing valuable insights into their efficiency.

***There are mainly three asymptotic notations:***

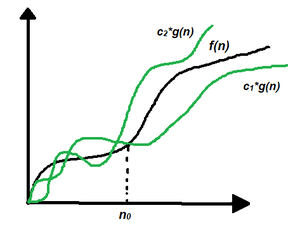
1. *Big-O Notation (O-notation)*
2. *Omega Notation (Ω-notation)*
3. *Theta Notation (Θ-notation)*

**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.*

*.Theta (Average Case) You add the running times for each possible input combination and take the average in the average case.*

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 **3n3 + 6n2 + 6000 = Θ(n3)**, the dropping lower order terms is always fine because there will always be a number(n) after which Θ(n3) has higher values thanΘ(n2) 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( n2)***

***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.*

*.It is the most widely used notation for Asymptotic analysis.  
.It specifies the upper bound of a function.  
.The maximum time required by an algorithm or the worst-case time complexity.  
.It returns the highest possible output value(big-O) for a given input.  
.Big-O(Worst Case) It is defined as the condition that allows an algorithm to complete statement execution in the longest amount of time possible.*

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.**

BigO

**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](http://www.geeksforgeeks.org/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(n2).   
**Note**: O(n2) also covers linear time.

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

* The worst-case time complexity of Insertion Sort is Θ(n2).
* The best case time complexity of Insertion Sort is Θ(n).

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

**Examples :**

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

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

**3. Omega Notation (Ω-Notation):**

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

**The execution time serves as a 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

BigOmega

**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.***

**Recurrence Relation**

*A****recurrence relation****is a mathematical expression that defines a sequence in terms of its previous terms. In the context of algorithmic analysis, it is often used to model the time complexity of recursive algorithms.*

*General form of a****Recurrence Relation:****an=f(an−1,an−2,….,an−k)an​=f(an−1​,an−2​,….,an−k​)  
where****f****is a function that defines the relationship between the current term and the previous terms.*

Some of the common uses of **Recurrence Relations** are:

* Time Complexity Analysis
* Generalizing Divide and Conquer Algorithms
* Analyzing Recursive Algorithms
* Defining State and Transitions for Dynamic Programming.

**There are mainly three ways of solving recurrences:**

1. Substitution Method
2. Recurrence Tree Method
3. Master Method

**Types of Recurrence Relations:**

1. Linear Recurrence Relations
2. Divide and Conquer Recurrences
3. Substitution Recurrences
4. Homogeneous Recurrences
5. Non-Homogeneous Recurrences

**Time and Space Complexity**

**Time Complexity:**The time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input. Note that the time to run is a function of the length of the input and not the actual execution time of the machine on which the algorithm is running on.

**Definition**–

The valid algorithm takes a finite amount of time for execution. The time required by the algorithm to solve given problem is called ***time complexity*** of the algorithm. Time complexity is very useful measure in algorithm analysis.

 It is the time needed for the completion of an algorithm. To estimate the time complexity, we need to consider the cost of each fundamental instruction and the number of times the instruction is executed.

**Space Complexity:**

**Definition –**

Problem-solving using computer requires memory to hold temporary data or final result while the program is in execution. The amount of memory required by the algorithm to solve given problem is called **space complexity** of the algorithm.

The space complexity of an algorithm quantifies the amount of space taken by an algorithm to run as a function of the length of the input. Consider an example: Suppose a problem to find the [frequency of array elements](https://www.geeksforgeeks.org/counting-frequencies-of-array-elements/).

It is the amount of memory needed for the completion of an algorithm.

To estimate the memory requirement we need to focus on two parts:

**(1) A fixed part:** It is independent of the input size. It includes memory for instructions (code), constants, variables, etc.

**(2) A variable part:**It is dependent on the input size. It includes memory for recursion stack, referenced variables, etc.

**Lower Bound**

Lower Bound Theory Concept is based upon the calculation of minimum time that is required to execute an algorithm is known as a lower bound theory or Base Bound Theory.

Let L(n) be the running time of an algorithm A(say), then g(n) is the **Lower Bound** of A if there exist two constants C and N such that L(n) >= C\*g(n) for n > N. Lower bound of an algorithm is shown by the asymptotic notation called [Big Omega](https://www.geeksforgeeks.org/analysis-of-algorithms-set-3asymptotic-notations/) (or just Omega).

**Upper Bound**

Upper Bound Theory Concept is based upon the calculation of maximum time that is required to execute an algorithm is known as a upper bound theory.

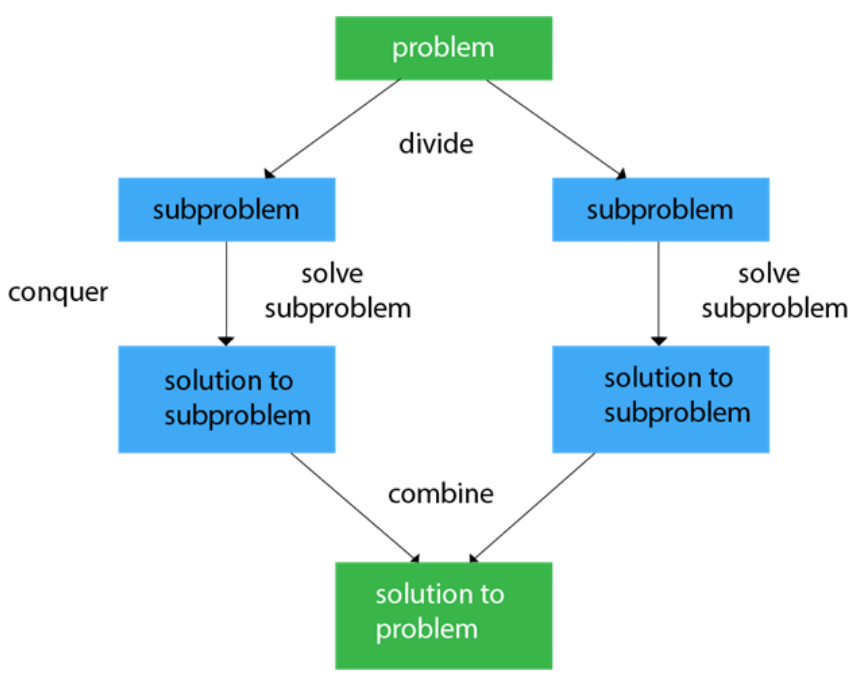
Let U(n) be the running time of an algorithm A(say), then g(n) is the **Upper Bound** of A if there exist two constants C and N such that U(n) <= C\*g(n) for n > N. Upper bound of an algorithm is shown by the asymptotic notation called [Big Oh(O)](https://www.geeksforgeeks.org/analysis-algorithms-big-o-analysis/) (or just Oh).

**Unit - 2**

Algorithm design strategies: Divide and conquer (Merge sort, Quick sort, matrix multiplication),Greedy method (knapsack problem, minimum spanning trees). Basic search & Traversal Techniques (Breadth first and Depth first traversals of Graphs).

**Divide and Conquer**

***Divide and Conquer Algorithm****involves breaking a larger problem into smaller subproblems, solving them independently, and then combining their solutions to solve the original problem. The basic idea is to recursively divide the problem into smaller subproblems until they become simple enough to be solved directly. Once the solutions to the subproblems are obtained, they are then combined to produce the overall solution.*



**1. Divide:**

* Break down the original problem into smaller subproblems.
* Each subproblem should represent a part of the overall problem.
* The goal is to divide the problem until no further division is possible.

**2. Conquer:**

* Solve each of the smaller subproblems individually.
* If a subproblem is small enough (often referred to as the “base case”), we solve it directly without further recursion.
* The goal is to find solutions for these subproblems independently.

**3. Merge:**

* Combine the sub-problems to get the final solution of the whole problem.
* Once the smaller subproblems are solved, we recursively combine their solutions to get the solution of larger problem.
* The goal is to formulate a solution for the original problem by merging the results from the subproblems.

**Applications of Divide and Conquer Algorithm:**

The following are some standard algorithms that follow Divide and Conquer algorithm:

* [**Quicksort**](https://www.geeksforgeeks.org/quick-sort/)is a sorting algorithm that picks a pivot element and rearranges the array elements so that all elements smaller than the picked pivot element move to the left side of the pivot, and all greater elements move to the right side. Finally, the algorithm recursively sorts the subarrays on the left and right of the pivot element.
* [**Merge Sort**](https://www.geeksforgeeks.org/merge-sort/)is also a sorting algorithm. The algorithm divides the array into two halves, recursively sorts them, and finally merges the two sorted halves.
* [**Closest Pair of Points**](https://www.geeksforgeeks.org/closest-pair-of-points-using-divide-and-conquer-algorithm/)The problem is to find the closest pair of points in a set of points in the x-y plane. The problem can be solved in O(n^2) time by calculating the distances of every pair of points and comparing the distances to find the minimum. The Divide and Conquer algorithm solves the problem in O(N log N) time.
* [**Strassen’s Algorithm**](https://www.geeksforgeeks.org/strassens-matrix-multiplication/)is an efficient algorithm to multiply two matrices. A simple method to multiply two matrices needs 3 nested loops and is O(n^3). Strassen’s algorithm multiplies two matrices in O(n^2.8974) time.
* [**Cooley–Tukey Fast Fourier Transform (FFT) algorithm**](http://en.wikipedia.org/wiki/Cooley%E2%80%93Tukey_FFT_algorithm)is the most common algorithm for FFT. It is a divide and conquer algorithm which works in O(N log N) time.
* [**Karatsuba algorithm for fast multiplication**](https://www.geeksforgeeks.org/karatsuba-algorithm-for-fast-multiplication-using-divide-and-conquer-algorithm/)does the multiplication of two binary strings in O(n1.59) where n is the length of binary string.

**Advantages of Divide and Conquer Algorithm:**

* **Solving difficult problems:**Divide and conquer technique is a tool for solving difficult problems conceptually. e.g. Tower of Hanoi puzzle. It requires a way of breaking the problem into sub-problems, and solving all of them as an individual cases and then combining sub- problems to the original problem.
* **Algorithm efficiency:**The divide-and-conquer algorithm often helps in the discovery of efficient algorithms. It is the key to algorithms like Quick Sort and Merge Sort, and fast Fourier transforms.
* **Parallelism:**Normally Divide and Conquer algorithms are used in multi-processor machines having shared-memory systems where the communication of data between processors does not need to be planned in advance, because distinct sub-problems can be executed on different processors.
* **Memory access:**These algorithms naturally make an efficient use of memory caches. Since the subproblems are small enough to be solved in cache without using the main memory that is slower one. Any algorithm that uses cache efficiently is called cache oblivious.

**Disadvantages of Divide and Conquer Algorithm:**

* **Overhead:**The process of dividing the problem into subproblems and then combining the solutions can require additional time and resources. This overhead can be significant for problems that are already relatively small or that have a simple solution.
* **Complexity:**Dividing a problem into smaller subproblems can increase the complexity of the overall solution. This is particularly true when the subproblems are interdependent and must be solved in a specific order.
* **Difficulty of implementation:**Some problems are difficult to divide into smaller subproblems or require a complex algorithm to do so. In these cases, it can be challenging to implement a divide and conquer solution.
* **Memory limitations:**When working with large data sets, the memory requirements for storing the intermediate results of the subproblems can become a limiting factor.

**Merge Sort**

**Merge sort**is a sorting algorithm that follows the **divide-and-conquer**approach. It works by recursively dividing the input array into smaller subarrays and sorting those subarrays then merging them back together to obtain the sorted array.

In simple terms, we can say that the process of **merge sort**is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.

**How does Merge Sort work?**

Merge sort is a popular sorting algorithm known for its efficiency and stability. It follows the **divide-and-conquer**approach to sort a given array of elements.

Here’s a step-by-step explanation of how merge sort works:

1. **Divide:**Divide the list or array recursively into two halves until it can no more be divided.
2. **Conquer:**Each subarray is sorted individually using the merge sort algorithm.
3. **Merge:**The sorted subarrays are merged back together in sorted order. The process continues until all elements from both subarrays have been merged.

**Algorithm**

MERGE\_SORT(arr, beg, end)

**if** beg < end

set mid = (beg + end)/2

MERGE\_SORT(arr, beg, mid)

MERGE\_SORT(arr, mid + 1, end)

MERGE (arr, beg, mid, end)

end of **if**

END MERGE\_SORT

**Complexity Analysis of Merge Sort:**

**Time Complexity:**

|  |  |
| --- | --- |
| **Case** | **Time Complexity** |
| Best Case | O(n\*logn) |
| Average Case | O(n\*logn) |
| Worst Case | O(n\*logn) |

* **Best Case:**O(n log n), When the array is already sorted or nearly sorted.
* **Average Case:**O(n log n), When the array is randomly ordered.
* **Worst Case:**O(n log n), When the array is sorted in reverse order.

**Space Complexity:**O(n), Additional space is required for the temporary array used during merging.

**Applications of Merge Sort:**

* Sorting large datasets
* External sorting (when the dataset is too large to fit in memory)
* Inversion counting (counting the number of inversions in an array)
* Finding the median of an array

**Advantages of Merge Sort:**

* **Stability**: Merge sort is a stable sorting algorithm, which means it maintains the relative order of equal elements in the input array.
* **Guaranteed worst-case performance:**Merge sort has a worst-case time complexity of **O(N logN)**, which means it performs well even on large datasets.
* **Simple to implement:**The divide-and-conquer approach is straightforward.

**Disadvantages of Merge Sort:**

* **Space complexity:**Merge sort requires additional memory to store the merged sub-arrays during the sorting process.
* **Not in-place:**Merge sort is not an in-place sorting algorithm, which means it requires additional memory to store the sorted data. This can be a disadvantage in applications where memory usage is a concern.

**Quick Sort**

QuickSort is a sorting algorithm based on the [Divide and Conquer algorithm](https://www.geeksforgeeks.org/divide-and-conquer-algorithm-introduction/)that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

**How does QuickSort work?**

The key process in quickSort is a partition() . The target of partitions is to place the pivot (any element can be chosen to be a pivot) at its correct position in the sorted array and put all smaller elements to the left of the pivot, and all greater elements to the right of the pivot.*Partition is done recursively on each side of the pivot after the pivot is placed in its correct position and this finally sorts the array.*

**Choice of Pivot:**

There are many different choices for picking pivots.

* Always pick the first element as a pivot.
* Always pick the last element as a pivot.
* Pick a random element as a pivot.
* Pick the middle as the pivot.

**Partition Algorithm:**

*The logic is simple, we start from the leftmost element and keep track of the index of smaller (or equal) elements as****i****. While traversing, if we find a smaller element, we swap the current element with arr[i]. Otherwise, we ignore the current element.*

**Time Complexity:**

|  |  |
| --- | --- |
| ***Case*** | ***Time Complexity*** |
| ***Best Case*** | *O(n\*logn)* |
| ***Average Case*** | *O(n\*logn)* |
| ***Worst Case*** | *O(n2)* |

The space complexity of quicksort is **O(n\*logn)**.

**Advantages of Quick Sort:**

* It is a divide-and-conquer algorithm that makes it easier to solve problems.
* It is efficient on large data sets.
* It has a low overhead, as it only requires a small amount of memory to function.

**Disadvantages of Quick Sort:**

* It has a worst-case time complexity of O(N^2), which occurs when the pivot is chosen poorly.
* It is not a good choice for small data sets.
* It is not a stable sort, meaning that if two elements have the same key, their relative order will not be preserved in the sorted output in case of quick sort, because here we are swapping elements according to the pivot’s position (without considering their original positions).

**Matrix Multiplication**