Basic

- **What is an algorithm?** It is a sequence of unambiguous instructions for solving a problem ie for obtaining the required output for a legitimate input in a finite amount of time.
- Properties of a Computer Algorithm:
  - **Finiteness**: An algorithm must always terminate after a finite number of steps. It should not run indefinitely.
  - **Definiteness**: Each step of the algorithm must be precisely defined. The instructions should be clear and unambiguous.
  - **Input**: An algorithm should have zero or more inputs. These are the values that are fed into the algorithm before it begins processing.
  - **Output**: An algorithm must produce one or more outputs. These are the results of the algorithm's processing.
  - **Effectiveness**: The steps of the algorithm must be basic enough to be performed, in principle, by a person using paper and pencil. They should be simple and feasible.
- Some important terms:
  - **Precondition**: A precondition is a condition or set of conditions that must be true before a function or a process is executed. It defines the state of the system or the variables before the execution starts. If the precondition is not met, the behavior of the function or process is undefined.
  - **Assertion**: An assertion is a statement that a certain condition is true at a specific point in the execution of a program. Assertions are used as a debugging aid to check for correctness. If an assertion fails (the condition is not true), it typically results in an error or exception, signaling that something went wrong in the program.
  - **Postcondition**: A postcondition is a condition or set of conditions that must be true after a function or process has executed. It defines the expected state of the system or the variables after the execution is completed. Postconditions are used to verify that the function or process has correctly performed its task.
  - **Loop Invariant**:A loop invariant is a condition that holds true before and after each iteration of a loop. Loop invariants are used to reason about the correctness of loops, particularly in the context of program verification and algorithm analysis. They help ensure that the loop is functioning as intended and will lead to the desired postcondition once the loop terminates.
    - Properties of a Loop Invariant:
      - **Initialization**: The invariant must be true before the first iteration of the loop starts.
      - **Maintenance**: If the invariant is true before an iteration of the loop, it must remain true after the iteration.
      - **Termination**: When the loop terminates, the invariant, combined with the loop's exit condition, should provide a useful property that helps prove the correctness of the algorithm.

Algorithm Analysis

The process of comparing two algorithms with respect to time, space, etc.

- Priori analysis:

  - Analyzing before execution, is not dependent on hardware.
  - We count the number of times a line of code executes.
  - Preferred, because it has an uniform value.
  - We use Asymptotic notation, like Big O to denote the time complexity.
- Posterior Analysis:
  - Analyzing after execution, is dependent on hardware.
  - We determine the amount of time an algorithm takes to execute on a particular hardware platform.

Asymptotic Notation

- Also check: Algorithm Analysis
- It is a mathematical way of representing the time complexity.
- Example: Let's take the example of a notebook.
  - Best-case: I find the topic right on the first page, just after opening the notebook.
  - Worst-case: I find the topic on the last page of the notebook.
  - Average-case: I find the topic somewhere in the middle of the notebook, after traversing the pages one by one.

Big-oh (O)

- Worst-case | Upper Bound
- $f(n) = O g(n), f(n) \leq c \cdot g(n)$
  - c is the constant, $c > 0$
  - k is the point where $f(n)$ and $g(n)$ intercept, $k \geq 0$
  - $n \geq k$
- **Example**: $f(n)=2n^2+n$
  - Find the **closest largest** term such that $g(n) \geq f(n)=2n^2+n$. The term is n^2.
  - So, $(2n^2+n) \leq c.g(n^2)$
  - $(2n^2+n) \leq 2n^2$, let c = 2. This is **false**, so increment c by 1.
  - $(2n^2+n) \leq 3n^2$, which is **true**.
  - So, $n \leq n^2$ or $n \geq 1$.
  - This means that for all values of $n \geq 1$ & $c=3$, the condition will hold true.
- Little o: f(n) = O g(n), f(n) < c.g(n)$.

Big-Omega (Ω)

- Best-case | Lower Bound
- $f(n) = Ω\ g(n)$, $f(n) \geq c.g(n)$
- **Example**: $f(n)=2n^2+n$
  - Find the **closest smallest** term such that $g(n) \leq f(n)=2n^2+n$. The term is n^2.
  - So, $(2n^2+n) \geq c.g(n^2)$
  - $(2n^2+n) \geq 2n^2$, let c = 2. which is **true**.
  - So, $n \geq 0$.
  - This means that for all values of $n \geq 0$ & $c=2$, the condition will hold true.
- Little Ω: f(n) = O g(n), f(n) > c.g(n)$.

Theta (θ)

- Average-case | Between Upper & Lower Bound
  - $f(n) = θ g(n)$, $c_1.g(n) \leq f(n) \leq c_2.g(n)$
- **Example**: $f(n)=2n^2+n$
  - Find both the **closest smallest** term and **closest largest** term, for $g(n)$. **Both the terms are same**, n^2.
  - So, $c_1.n^2 \leq (2n^2+n) \leq c_2.n^2$
  - For $c_1 n^2 \leq (2n^2+n)$, c = 2.
  - For $(2n^2+n) \leq c_2.n^2$, c = 3.
  - So, $2n^2 \leq (2n^2+n) \leq 3n^2$
  - This means that between the values $c=2$ & $c=3$ and $g(n)=n^2$, the condition will hold true.

Properties of Asymptotic Notation

Asymptotic Notation	Representation as `f(n)`	Representation as $a & b$	Reflexive	Symmetric	Transitive
Big O (O)	$f(n) \leq c \cdot g(n)$	$a \leq b$	1	0	1
Big Omega (Ω)	$f(n) \geq c \cdot g(n)$	$a \geq b$	1	0	1
Theta (θ)	$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$	$a=b$	1	1	1
Small O (o)	$f(n) < c \cdot g(n)$	$a < b$	0	0	1
Small Omega (Ω)	$f(n) > c \cdot g(n)$	$a > b$	0	0	1

- Reflexive: If $a\ (operator)\ a$ is valid.
- Symmetric: if $a\ (operator)\ b$ is valid, then $b\ (operator)\ a$ should also be valid.
- Transitive: If $a\ (operator)\ b$ is valid and $b\ (operator)\ c$ is valid, then $a\ (operator)\ c$ should also be valid.

Comparison of Time complexities

$c/1 < \log (\log (n)) < \log (n) < n < n \log (n) < n^2 < n^3 < n^k < 2^n < n! < 2^{2^n}$

Time Complexity	Notation	Time complexity, taking $n = 10000$
Constant	$O(c)$ / $O(1)$	1
null	$O(\log (\log (n)))$	1.1461
Logarithmic	$O(\log n)$	14
Linear	$O(n)$	10000
Linearithmic	$O(n \log n)$	132877
Quadratic	$O(n^2)$	100000000

Time Complexity	Notation	Time complexity, taking $n = 10000$
Cubic	$O(n^3)$	1000000000000
Polynomial	$O(n^k)$	null
Exponential	$O(2^n)$	Very large number
Factorial	$O(n!)$	Very large number
Double Exponential	$O(2^{2^n})$	Very large number

Time Complexity Examples

- Example 0:

  ```
  publc int sum(int x, int y) {
      int result = x + y; // i
      return result;      // ii
  }
  ```

  1. 1 unit
  2. 1 unit
  - Time Complexity: $O(1)$
- Example 1:

  ```
  publc int get(int[] arr, int i) {
      return arr[i];      // i
  }
  ```

  1. 1 unit
  - Time Complexity: $O(1)$
- Example 2:

  ```
  public void findSum(int n) {
      int sum = 0;                    // i
      for(int i = 1; i <= n; i++) {
          sum = sum + i;              // ii
      }
      return sum;                     // iii
  }
  ```

  1. 1 unit
  2. n units
  3. 1 unit
  - Time Complexity: $O(n)$

- Example 3:

```
public void print(int n) {
     for(int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            System.out.println("i=" +i+ ", j="+j);  // i
        }
        System.out.println("Enter of inner loop");  // ii
    }
    System.out.println("End of outer loop");        // iii
}
```

1. n^n units
2. n units
3. 1 unit

  - Time Complexity: $O(n^n)$

Common Time Complexities

- Legend:
  - V = Number of vertices
  - E = Number of edges

Algorithm	**Best Case**	**Average Case**	**Worst Case**
Binary Search	$O(1)$	$O(\log n)$	$O(\log n)$
Sequential Search	$O(1)$	$O(n)$	$O(n)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Height of Complete Binary Tree	$O(\log n)$	$O(\log n)$	$O(\log n)$
Insert in Heap	$O(\log n)$	$O(\log n)$	$O(\log n)$
Construct Heap	$O(n)$	$O(n)$	$O(n)$
Delete from Heap	$O(\log n)$	$O(\log n)$	$O(\log n)$
Huffman Coding	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Prims (Matrix)	$O((V + E) \log V)$	$O((V + E) \log V)$	$O((V + E) \log V)$
Prims (Heap)	$O(n^2)$	$O(n^2)$	$O(n^2)$

Algorithm	Best Case	Average Case	Worst Case
Kruskal	$O(E \log V)$	$O(E \log V)$	$O(E \log V)$
BFS	$O(V + E)$	$O(V + E)$	$O(V + E)$
DFS	$O(V + E)$	$O(V + E)$	$O(V + E)$
Floyd Warshall	$O(V^3)$	$O(V^3)$	$O(V^3)$
Dijkstra	$O((V + E) \log V)$	$O((V + E) \log V)$	$O((V + E) \log V)$

- Example 1: $f_1=n^2\ \log n$ vs $f_2=n\ {\log n}^{10}$
  - Method 1: Put a large value of n. Let $n=10^9$
    - $10^9\ x\ 10^9\ x\ \log_{10} 10^9$ vs $10^9\ {(\log_{10} 10^9)}^{10}$
    - $10^9\ x\ 9$ vs 9^{10}
    - $10^9\ x\ 9$ vs $9\ x\ 9^9$
    - 10^9 vs 9^9
    - So, $f_1 > f_2$
  - Method 2: Simplify the equations.
    - $n\ x\ n \log n$ vs $n\ \log n\ x\ \log n^{9}$
    - n vs ${(\log n)}^9$
    - $\log n$ vs $\log (\log n)^9$
    - $\log n$ vs $9\ x\ \log (\log n)$
    - $\log n$ vs $\log (\log n)$ # 9 is constant, can be neglected
    - So, $f_1 > f_2$
- Example 2: $f_1(n) = 2^n$, $f_2(n) = n^{3/2}$, $f_3 (n) = n\ \log_{2} n$, $f_4(n) = n^{\log_{2} n}$
  - (1): $f_2<f_3<f_4<f_1$
  - (2): $f_2<f_1<f_3<f_4$
  - (3): $f_1<f_2<f_3<f_4$
  - **(4)**: $f_3<f_2<f_4<f_1$
  - Method 1: Put a large value of n. Let $n=256$
    - First of all, $2^n = 2^{256}$ will be largest, since it increases `exponentially`. This leaves us with options (2) & (3).
    - $f_1=2^{256}$
    - $f_2=256^{3/2}=2^{8x{3/2}}=2^{12}$
    - $f_3=256\ x \log_{2} 256=2^8\ x\ \log_{2} 2^8=2^8\ x\ 8=2^8$
    - $f_4=256\ x\ 256^{\log_{2} 256}$=$2^8\ x\ 2^{8x8}$=$2^{24}$
    - So, $f_3<f_2<f_4<f_1$
  - Method 2: Simply the equations
    - $f_1=2^n$
    - $f_2=n^{3/2}=n\ x\ n^{1/2}=n \sqrt n$
    - $f_3=n \ log_{2} n$

- In f_2 & f_3, we can remove `n` as it is common between both. So, $f_3<f_2$.
- $f_4=n^{\log_{2} n}=n^k$, which is greater than f_2.
- So, $f_3<f_2<f_4<f_1$

Recurrence Relation

- Example:

```
val start = 0
val end = arr.length-1
function binarySearch(start, end, arr[], target) {
    while(start<=end) {
        val mid = i+j/2
        if(arr[mid] == target) {
            return mid
        } else if(arr[mid] < target) {
            binarySearch(mid+1,end,arr,target)
        } else { // if (arr[mid] > target)
            binarySearch(start,mid-1,arr,target)
        }
    }
}
```

  - Binary Search only works on sorted arrays.
  - Steps, example: ${10,20,30,40,50,60,70}$:
    1. Here, we're first finding the middle index of the array.
    2. We're checking if the middle element is same as the target. If `true`, we're returning the element.
    3. If the middle element is less than the target, the target will be on the right half of the array, ie amongst ${50,60,70}$.
    4. If the middle element is greater than the target, the target will be on the left half of the array, ie amongst ${10,20,30}$.
    5. In essence, we're dividing our problem into half with each iteration, from $n → n/2 → n/4$ and so on. We're only solving one half at a time. This is called recurrence relation.
  - Relation: $T(n/2)+c$
  - We solve recurrence relations using:
    1. Back Substitution Method

Back Substitution Method

- Example 0:

> Recurrence Relation: $T(n)=T(n/2)+c$
> Termination condition: $T(n)=1\ if\ n=1$

  - Step 1: n >> $n/2$.
    - $T(n)=T(n/2)+c$
    - $T(n/2)=T(n/4)+c$

        - $T(n/4)=T(n/8)+c$
    - Step 2: Substitiute $T(n/2)$ in $T(n)$, and so on.
        - $T(n)=[T(n/4)+c]+c$
            - $T(n/4)+2c$

        > $T(n/2^2)+2c$

        - $T(n)=[T(n/8)+c]+2c$
            - $T(n/8)+3c$

        > $T(n/2^3)+3c$

        - After k times, $T(n/2^k)+kc$
    - We need to make $T(n/2^k)=T(1)$. So, let $n/2^k=1$.
        - $n/2^k=1$
        - $n=2^k$
        - $\log n = \log {2^k}$
        - $\log_2 n = k \log_2 2$
        - $log n = k$ or $k = \log n$
    - Substitute the value of `k`:
        - $1+kc$
        - $1+\log n*c$
    - $(\log n)$ is the largest term. So, time complexity: $O(\log n)$

- Example 1:

    > Recurrence Relation: $T(n)={n*T(n-1)}\ if\ n>1$
    > Termination condition: $T(n)=1\ if\ n=1$

    - Step 1: n >> $n-1$
        - $T(n)={n*T(n-1)}$
        - $T(n-1)=(n-1)*T((n-1)-1)=(n-1)*T(n-2)$
        - $T(n-2)=(n-2)*T((n-2)-1)=(n-2)*T(n-3)$
    - Step 2: Substitute `T(n-1)`, etc. with their RHS
        - $T(n)=n*[(n-1)*T(n-2)]$ # Substitute $T(n-1)$ with $(n-1)*T(n-2)$
        - $T(n)=n*(n-1)*[(n-2)T(n-3)] = n(n-1)*(n-2)*T(n-3)$
        - We see a pattern here.
        - So, after `k` iterations, $n*(n-1)*(n-2)*(n-3)...*T(n-k)$
    - Step 3: Try to get $(n-k)=1$, so that the equation can be terminated
        - Equation: $n*(n-1)*(n-2)*(n-3)...*T(n-k)$. If we take $k=(n-1)$,
        - $T(n-(n-1))=T(n-n+1)=T(1)=1$
        - So, $T(n)=n*(n-1)*(n-2)*(n-3)* ... 3*2*1$
    - Step 4: Simplify the equation
        - $T(n)=n*(n-1)*(n-2)*(n-3)* ... 3*2*1$
        - $n*n(n-1/n)*n(n-2/n)*n(n-3/n) ... *n(3/n)*n(2/n)*n(1/n)$
        - $n^{n}*(n-1/n)*(n-2/n)*(n-3/n) ... * (3/n) * (2/n) * (1/n)$
        - So, n^{n} is the most significant term. Time complexity: $O(n!)$, or Factorial.

- Example 2:

> Recurrence Relation: $T(n)=2T{(n/2)}+n$
> Termination condition: $T(n)=1\ if\ n=1$

- Step 1: n >> $n/2$.
  - $T(n)=2T{(n/2)}+n$
  - $T(n/2)=2T{(n/4)}+(n/2)$
  - $T(n/4)=2T{(n/8)}+(n/4)$
- Substitute value of $T(n/2)$ in $T(n)$, and so on.
  - $T(n)=2[{2T{(n/4)}+(n/2)}]+n$
    - $4T(n/4)+2n$

    > $2^2T(n/2^2)+2n$

  - $T(n)=4[2T{(n/8)}+(n/4)]+2n$
    - $8T(n/8)+3n$

    > $2^3T(n/2^3)+3n$

  - After k times, $T(n)=2^kT(n/2^k)+kn$
- We need to make $T(n/2^k)$ 1, using the termination condition, ie $n/2^k=1$ or $n=2^k$.
  - $n=2^k$
  - $\log_2 n = log_2 2^k$
  - $\log_2 n = k \log_2 2$
  - $\log_2 n = k$ or $k=\log_2 n$
- Substitute the value of `k`:
  - $T(n)=n+n \log n$, $n \log n$ is the largest term.
  - So, time complexity = $O(n \log n)$.

- Example 3:

> Recurrence Relation: $T(n-1)+ \log n$
> Termination condition: $T(n)=1\ if\ n=1$

- Step 1: n is decreasing by $(n-1)$.
  - $T(n)=T(n-1)+ \log n$
  - $T(n-1)=T(n-2)+ \log (n-1)$
  - $T(n-2)=T(n-3)+ \log (n-2)$
- Substitute value of $T(n-1)$ in $T(n)$, and so on.
  - $T(n)=[T(n-2)+ \log (n-1)]+ \log n$

    > $T(n-2)+\log(n-1)+\log n$

  - $T(n)=T(n-2)+\log(n-1)+\log n$
    - $[T(n-3)+ \log (n-2)]+\log(n-1)+\log n$

    > $T(n-3)+ \log (n-2)+\log(n-1)+\log n$

  - After k times, $T(n)=T(n-k)+\log (n-(k-1))+\log(n-(k-2))+log(n-(k-3)) + ...\ +log(n-0)$
- We need to make $T(n-k)=T(1)$. So, let $n-k=1$ or $n=k$
- Substitute the value of `k`:
  - $1+\log 1 + \log 2 + \log 3 + ... + \log n$

        - $1+log(1*234...*n)$
        - $1+log(n^n)$
        - $1+n \log n$
    - $(n \log n)$ is the largest term. So, time complexity: $O(n \log n)$

Divide & Conquer

- Divide and Conquer algorithm is a problem-solving strategy that involves breaking down a complex problem into smaller, more manageable parts, solving each part individually, and then combining the solutions to solve the original problem. It is a widely used algorithmic technique in computer science and mathematics.
- Stages of Divide and Conquer Algorithm:
    - 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.
    - 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.
    - 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.

Dynamic Programming

- [vs **Greedy Method**] Dynamic programming guarantees finding the globally optimal solution by considering all possible subproblems and their combinations. On the other hand, the greedy method makes a series of choices, each of which looks best at the moment, but it doesn't always guarantee a globally optimal solution because it doesn't reconsider its choices. Thus, dynamic programming is more reliable for complex problems where the greedy method might fail to find the best solution.
- Dynamic Programming checks all available paths, and chooses the most optimal path. **It always provides the optimal solution.**
- [vs **Divide & Conquer**] Dynamic programming optimizes recursive algorithms by storing the results of subproblems to avoid redundant computations, which significantly reduces the time complexity. Unlike the divide and conquer approach, which solves subproblems independently and may recompute the same subproblems multiple times, dynamic programming ensures that each subproblem is solved only once and then reused, making it more efficient for problems with overlapping subproblems and optimal substructure.

Karatsuba Multiplication Algorithm

- This algorithm helps in multiplying 2 large numbers efficiently.
- Steps (example: $n_1=1234$, $n_2=5678$):
    1. Divide n_1 into $a=12$, $b=34$.
    2. Divide n_2 into $c=56$, $d=78$.
    3. Find $a*c=672$
    4. Find $b*d=2652$
    5. Find $(a+b)(c+d)=6164$
    6. $[5]-[(4)+(3)]=6164-(2652+672)=2840$
    7. $[3]*10^{4}+[4]+[6]*10^2=6720000+2652+284000=7006652$

Strassen's Multiplication of Matrices

- Usually, multiplying 2 2x2 matrices involves 8 recursive calls.
  A (row 1): [a b], B (row 1): [e f], AB (row 1): [ae+bg af+bh]
  A (row 2): [c d], B (row 2): [g h], AB (row 2): [ce+dg cf+dg]
- Using this algorithm, we can reduce the number of system calls to 7.
- This results in a reduction of time complexity from $O(n^3)$ to $O(N^{\log 7}))$.
- Formulae:
    - First, find $p_1 \to p_7$:
        - $p_1=a(f-h)$
        - $p_2=(a+b)h$
        - $p_3=(c+d)e$
        - $p_4=d(g-e)$
        - $p_5=(a+d)(e+h)$
        - $p_6=(b-d)(g+h)$
        - $p_7=(a-c)(e+f)$
    - Next, we derive the final result:
      [a b] [e f] = [$p_5+p_4-p_2+p_6$ p_1+p_2]
      [c d] [g h] = [p_3+p_4 $p_1+p_5-p_3-p_7$]
- Example:
  [1 2] [10 12] = [a b] [e f]
  [3 4] [14 15] = [c d] [g h]
    - $p_1=-3$
    - $p_2=45$
    - $p_3=70$
    - $p_4=16$
    - $p_5=125$
    - $p_6=-58$
    - $p_7=-44$
    - Deriving the final results:
      [38 -42]
      [86 96]