Algorithm Note(Masud Rana)

Sorting refers to re-arrangement of a given array or list of elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of elements in the respective data structure.

Selection Sort is a comparison-based sorting algorithm. It sorts an array by repeatedly selecting the smallest (or largest) element from the unsorted portion and swapping it with the first unsorted element.

Time Complexity: $O(n^2)$,as there are two nested loops.
Auxiliary Space: O(1) as the only extra memory used is for temporary variables.

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity are quite high.

Time Complexity: $O(n^2)$
Auxiliary Space: O(1)

Insertion Sort is a comparison-based sorting algorithm that works by iteratively builds the sorted array one element at a time by taking an element from the unsorted portion and placing it in its correct position in the sorted portion.

Algorithm Steps:
- Start with the second element (assuming the first element is already sorted).
- Compare the current element with elements in the sorted portion.
- Shift all elements greater than the current element one position to the right.
- Insert the current element at its correct position.
- Repeat for all elements in the array.

Time Complexity: $O(n^2)$
Auxiliary Space: O(1)

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.

The recurrence relation of merge sort is:
$T(n)=\{\Theta(1)$ if $n=1$
$2T(n/2)+\Theta(n)$ if $n>1$

- T(n) Represents the total time time taken by the algorithm to sort an array of size n.
- 2T(n/2) represents time taken by the algorithm to recursively sort the two halves of the array. Since each half has n/2 elements, we have two recursive calls with input size as (n/2).
- O(n) represents the time taken to merge the two sorted halves.

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

Quick Sort is a divide-and-conquer sorting algorithm that partitions an array into two subarrays based on a pivot element, such that all elements smaller than the pivot go to the left subarray and all larger elements go to the right. The process is recursively applied to the subarrays until the entire array is sorted.

Time Complexity:
Best Case: ($\Omega(n \log n)$), Occurs when the pivot element divides the array into two equal halves.
Worst Case: ($O(n^2)$), Occurs when the smallest or largest element is always chosen as the pivot (e.g., sorted arrays).

Auxiliary Space: O(n), due to recursive call stack

Heap sort is a comparison-based sorting technique based on Binary Heap Data Structure. It can be seen as an optimization over selection sort where we first find the max (or min) element and swap it with the last (or first). We repeat the same process for the remaining elements. In Heap Sort, we use Binary Heap so that we can quickly find and move the max element in O(Log n) instead of O(n) and hence achieve the O(n Log n) time complexity.

Heap Sort is a comparison-based sorting algorithm that uses the properties of a binary heap (a complete binary tree) to sort elements. It repeatedly extracts the maximum (or minimum) element from the heap and rearranges the heap until the entire array is sorted.

Algorithm Steps:

Build a Max-Heap:

- Convert the input array into a max-heap, where the largest element is at the root.

Extract the Maximum Element:

- Swap the root of the heap (largest element) with the last element of the array.
- Reduce the size of the heap and heapify the root to maintain the max-heap property.

Repeat:

- Continue extracting and heapifying until the entire array is sorted.

Properties:

Time Complexity:

Building the heap: O(n).

Sorting: O(nlogn).

Overall: O(nlogn) for all cases (best, worst, average).

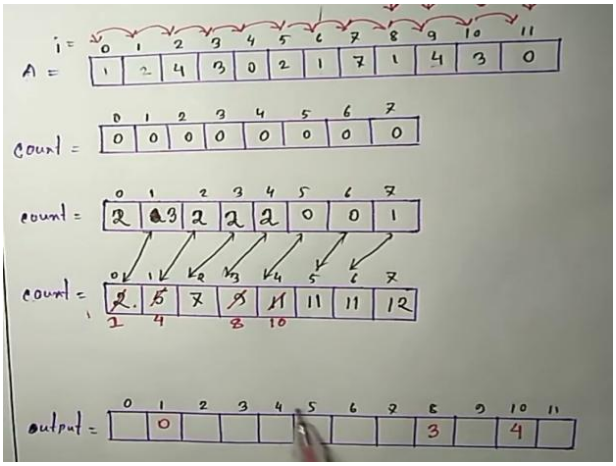
Space Complexity: O(1) (in-place sorting algorithm).

Stability: Not stable (relative order of equal elements may change).

Counting Sort is a non-comparison-based sorting algorithm that sorts elements by counting the frequency of each unique element and using that count to determine their positions in the sorted array.

Steps:

- Find the maximum elements and make a array of their frequency.
- Compute cumulative counts to determine positions. They are the range of each number of following frequency.
- Decrement each cumulative count by 1.
- Traverse from right of original array, go to the count array index of traverse value, place the traverse value in the position of new array at index of the count array value that was decremented before.



Properties:

- Time Complexity: O(n+k), where n is the number of elements and k is the range of values.
- Space Complexity: O(n+k).

Binary Search is an efficient algorithm for finding a target element in a sorted array by repeatedly dividing the search interval in half. If the target element is smaller than the middle element, it narrows the search to the left half; otherwise, it narrows it to the right half.

Time and Space complexity: O(logn)

```cpp
#include <iostream>
using namespace std;

int binarySearch(int arr[], int size, int target) {
    int low = 0, high = size - 1;

    whil(low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == target)
            return mid;
        else if (arr[mid] < target)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1;
}
```

Ternary Search is a divide-and-conquer algorithm that is similar to binary search, but instead of dividing the array into two parts, it divides the array into three parts.

Time coplexity: $O(\log_3 n)$
Space complexity: O(1)

```cpp
#include <iostream>
using namespace std;

int ternarySearch(int l, int r, int key, int ar[]) {
    while (r >= l) {
        int mid1 = l + (r - l) / 3;
        int mid2 = r - (r - l) / 3;
        if (ar[mid1] == key) return mid1;
        if (ar[mid2] == key) return mid2;
        if (key < ar[mid1]) r = mid1 - 1;
        else if (key > ar[mid2]) l = mid2 + 1;
        else { l = mid1 + 1; r = mid2 - 1; }
    }
    return -1;
}

int main() {
    int l = 0, r = 9, p, key;
    int ar[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    key = 5;
    p = ternarySearch(l, r, key, ar);
    cout << "Index of " << key << " is " << p << endl;
    key = 50;
    p = ternarySearch(l, r, key, ar);
    cout << "Index of " << key << " is " << p;
}
```

The term **k-way search** or **multi-way search** refers to a search algorithm where the search space is divided into k distinct parts, instead of just 2 (like binary search) or 3 (like ternary search).

If we declare any variable that auto initial zero but in main function it is garbage value.

Two-pointer algorithm is a technique commonly used to solve problems involving arrays or lists, where two pointers (or indices) are used to traverse the data structure in a linear fashion. It is often used to solve problems related to searching, sorting, and finding pairs or subarrays with specific properties.

It use two point, one from first and another from second.

- Find the closest pair from two sorted arrays.
- Find the pair in array whose sum is closest to x.
- Find all triplets with zero sum.
- Find a triplet that sum to a given value.
- Find a triplet such that sum of two equals to third element.
- Find four elements that sum to a given value.
- Trapping Rain Water.

Binary Exponentiation is an efficient algorithm for calculating a^b in O(logb) time. It works by breaking down the exponent b using its binary representation, allowing us to compute powers by repeated squaring and multiplication.

```
#include <bits/stdc++.h>
using namespace std;

long long power(long long a, long long b) {
   long long result = 1;
   while(b) {
      if (b & 1) {
      result = result * a;
       b --;
      }else{
      a = a * a;
      b >>= 1;
      }
   }
   return result;
}

int main() {
```

Big Mod

```
int func(int base, int power, int mod)
{
    int ans = 1;
    while(power)
    {
        if(power%2==1)
        {
          ans = (ans*base)%mod;
          power--;
        }
        else
        {
          base = (base*base)%mod;
          power = power/2;
        }
    }
    return ans;
}
```

$(a \cdot b)\% m = [(a \% m) \cdot (b \% m)] \% m$

This means we can break the computation of a^b into smaller modular steps to avoid computing the full a^b, which can grow exponentially large.

Matrix exponentiation computes the power of a matrix efficiently like binary exponentiation and use **Identity** matrix with **O(m*n.logn)** time. Where m*n is matrix multiplication and logn is number of multiplication.

Matrix Multiply

```
// Function to multiply two matrices a (p x q) and b (q x r)
    for (int i = 0; i < p; i++) {  // Loop over the rows of matrix a
        for (int j = 0; j < r; j++) {  // Loop over the columns of matrix b
            for (int k = 0; k < q; k++) {  // Loop over the columns of matrix a / rows of matrix b
                result[i][j] += a[i][k] * b[k][j];  // Multiply and add to result
            }
        }
    }
```

- **vector<int> v(5, 10);** creates a vector of size 5 with all elements initialized to 10. The vector v looks like this: {10, 10, 10, 10, 10}.

- **vector<vector<long long>> C(3, vector<long long>(2, 10));** fixed row and column.

C = [[10, 10],
[10, 10],
[10, 10]]

- **vector<long long> vec(5);** this is 1D vector of 5 size.
- **vector<vector<long long>> matrix(n);** Fixed Row Length and Unlimited Column Length

Breadth First Search (BFS) is a fundamental graph traversal algorithm that begins with a node, then first traverses all its adjacent. Once all adjacent are visited, then their adjacent are traversed. It use queuq for structure.

- BFS explores all nodes level by level, making it ideal for problems that require the shortest path or level-wise traversal.
- BFS finds the shortest path between two nodes in an unweighted graph.
- Determines connected components in an undirected graph.
- Cycle Detection in Undirected Graph.
- Minimum Spanning Tree (MST).

Time Complexity: O(V+E), where V is the number of nodes and E is the number of edges.
Auxiliary Space: O(V)

Depth-First Search (DFS) is an algorithm used to traverse or search through graphs and trees. It starts at a given node and explores as far as possible along each branch before backtracking. DFS uses a stack to keep track of visited nodes.

- DFS dives deeper into one branch before backtracking, making it suitable for problems involving backtracking, connectivity, or cycles.
- DFS is used in Directed Acyclic Graphs (DAGs) to produce a valid topological order.
- Can detect cycles in directed graphs using back edges during traversal.
- DFS can identify all connected components in undirected graphs.
- Useful in problems that involve exploring all paths, such as solving mazes.
- DFS is the backbone of backtracking algorithms like Sudoku solving, N-Queens, and combinatorial problems.

Time Complexity: O(V+E), where V is the number of nodes and E is the number of edges.
Auxiliary Space: O(V)

Dijkstra's algorithm is a greedy algorithm used to find the shortest path from a single source node to all other nodes in a weighted graph with non-negative weights.

- Create a distance array to store the shortest distance from the source to each node, initializing all distances to infinity (INF) except the source (set to 0).
- Use a priority queue (min-heap) to keep track of nodes with their current shortest distances.
- Extract the node with the smallest distance from the queue (initially the source).
- For each unvisited neighbor of the current node, calculate the new distance as:
  new_distance=current_distance+edge_weight
- If the new distance is smaller than the recorded distance in the distance array, update it and push the neighbor into the priority queue.
- Continue until the queue is empty or all nodes are processed.

- The distance array will contain the shortest distances from the source to all other nodes.

Time Complexity: O(E+VlogV)
Auxiliary Space: O(V)

Floyd-Warshall algorithm is a dynamic programming approach used to find the shortest paths between all pairs of vertices in a weighted graph. It handles both directed and undirected graphs and supports graphs with negative edge weights, but no negative weight cycles.

- Create a distance matrix dist of size V×V (where V is the number of vertices).
- Set dist[i][j] to the weight of the edge between i and j, or INF if no direct edge exists, and dist[i][i] = 0 for all i.
- For each vertex k, treat k as an intermediate node and update the shortest paths:
  dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])
- This checks whether the path from ii to j via k is shorter than the current known path.
- Repeat the above step for all k from 1 to V.
- After processing, dist[i][j] contains the shortest distance between nodes i and j.

Time Complexity: O(V^3)
Auxiliary Space: O(V^2)

Difference:

Aspect	Dijkstra's Algorithm	Floyd-Warshall Algorithm
Purpose	Single-source shortest path.	All-pairs shortest path.
Graph Type	Weighted, non-negative edges.	Weighted, handles negative edges (no cycles).
Complexity	$O(E + V \log V)$	$O(V^3)$
Formula	$\text{dist}[v] = \min(\text{dist}[v], \text{dist}[u] + w)$	$\text{dist}[i][j] = \min(\text{dist}[i][j], \text{dist}[i][k] + \text{dist}[k][j])$
Approach	Greedy algorithm.	Dynamic programming.
Output	Distances from a single source to all nodes.	Shortest paths between all pairs of nodes.
Usage	GPS navigation.	Network analysis, all-pair pathfinding.

Kruskal's Algorithm is a greedy algorithm used to find the Minimum Spanning Tree (MST) of a graph. It works by selecting the edges with the smallest weights while ensuring no cycles are formed, connecting all vertices in the graph.

- Sort all edges in ascending order based on their weights.
- Use a Disjoint Set Union (DSU) or Union-Find structure to keep track of which vertices are in the same component.
- Iterate through the sorted edges and select the smallest edge that does not form a cycle. This is checked using the DSU.
- Add edges to the MST until it contains V−1 edges, where V is the number of vertices.
- The MST includes all selected edges and their total weight.

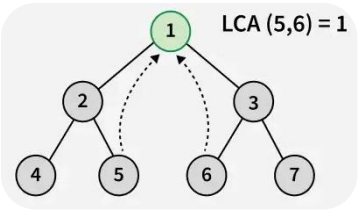
Time Complexity:
O(ElogE) or O(ElogV)
Auxiliary Space: O(V + E)

Prim's algorithm is a greedy algorithm used to find the Minimum Spanning Tree (MST) of a graph. It starts with a single vertex and incrementally grows the MST by adding the smallest edge that connects an unvisited vertex to the tree.

- Determine an arbitrary vertex as the starting vertex of the MST.
- Follow steps 3 to 5 till there are vertices that are not included in the MST
- Find edges connecting any tree vertex with the fringe vertices.
- Find the minimum among these edges.
- Add the chosen edge to the MST if it does not form any cycle.
- Return the MST and exit.

Time Complexity: O(E*log(E)) where E is the number of edges
Auxiliary Space: O(V^2) where V is the number of vertex

Lowest Common Ancestor (LCA) of two nodes in a tree is defined as the most recent common ancestor node shared by two given nodes.



- Applied to find path distance between two path.

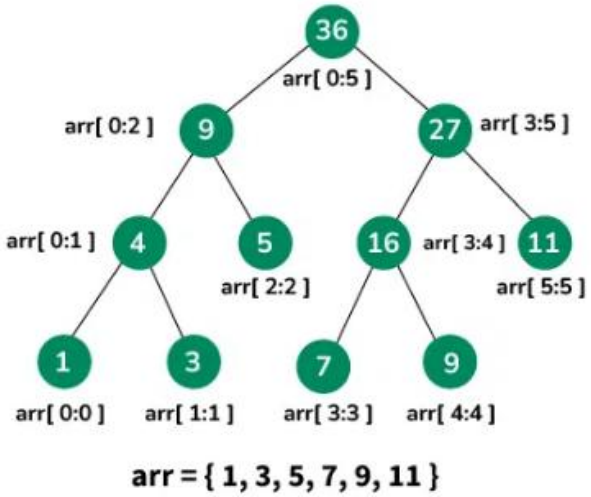
Disjoint Set Union (DSU), also known as Union-Find, is a data structure that efficiently supports two main operations:
- Union: Merges two sets into a single set.
- Find: Determines which set a particular element is in.

The primary use of DSU is in problems involving dynamic connectivity, such as determining whether two elements are in the same set or in graph algorithms like Kruskal's algorithm for finding the Minimum Spanning Tree (MST).

Topological Sort is an ordering of vertices in a directed graph such that for every directed edge u→v, vertex u comes before v. It is only possible for Directed Acyclic Graphs (DAGs). Common methods for performing topological sorting are Kahn's Algorithm (BFS approach) and DFS-based Algorithm. The output of topological sort is a linear ordering of vertices that respects the dependencies among the vertices in the graph.
It is used in tasks like scheduling, build systems, and dependency resolution.

Segment Tree:

Segment Tree is an efficient data structure for range queries and updates on arrays. It divides the array into segments recursively, enabling fast operations like sum, minimum, or maximum over a range with O(logn) complexity, making it highly useful in algorithm optimization.



The array is divided into halves, and O(n) nodes are computed, the build time complexity is O(n).
Query: Only relevant nodes are traversed along the tree height. The maximum traversal is O(logn), so query time complexity is O(logn).
Updates propagate from the affected leaf to the root, traversing O(logn) nodes.

Space Complexity: For unbalanced tree the approximate space is O(4n)=O(n).
Auxiliary Space: While performing recursive operations, the recursion stack depth is at most O(logn).

Binary Indexed Tree:

A Binary Indexed Tree (BIT), also known as Fenwick Tree, is a data structure that supports efficient methods for querying prefix sums and updating elements in an array.

The update operation in BIT requires traversing the tree from the affected index upwards to the root, this is proportional to the number of bits in the index, so time complexity is O(logn).

For a query to calculate the prefix sum, the algorithm traverses the tree from the leaf node to the root, this is proportional to the number of bits in the index, so time complexity is O(logn).

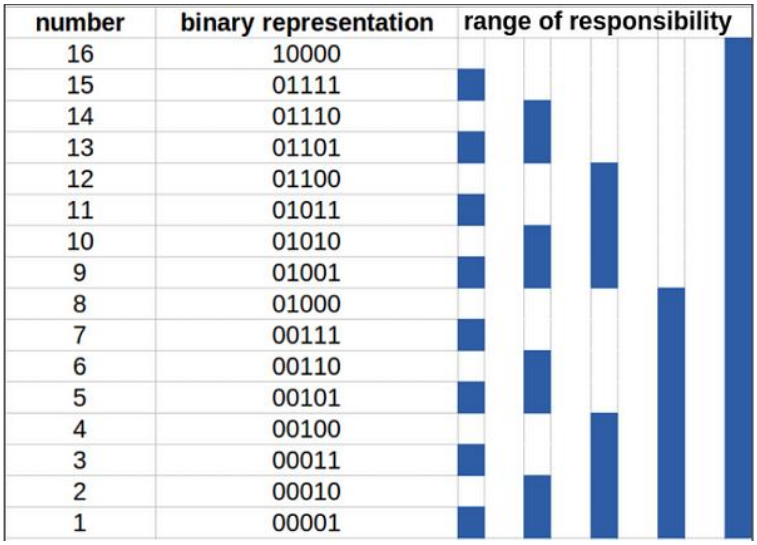
First, we create a BIT array of size n. This step takes O(n) time because we are simply allocating space for the array, initializing all values to 0. Then The number of steps in an update is proportional to the number of bits in the index i, which is O(logn).
BIT Tree built time is O(nlogn).

Complexity:

	Binary Indexed Tree (BIT)	Segment Tree	Prefix Sum Array
Time Complexity (Update)	O(log n)	O(log n)	O(n) (rebuild)
Time Complexity (Query)	O(log n)	O(log n)	O(1)
Space Complexity	O(n)	O(n) (can go up to 4n)	O(n)
Ease of Implementation	Easier	More complex	Very easy
Best for Dynamic Updates	Efficient	Less efficient	Not suited for updates

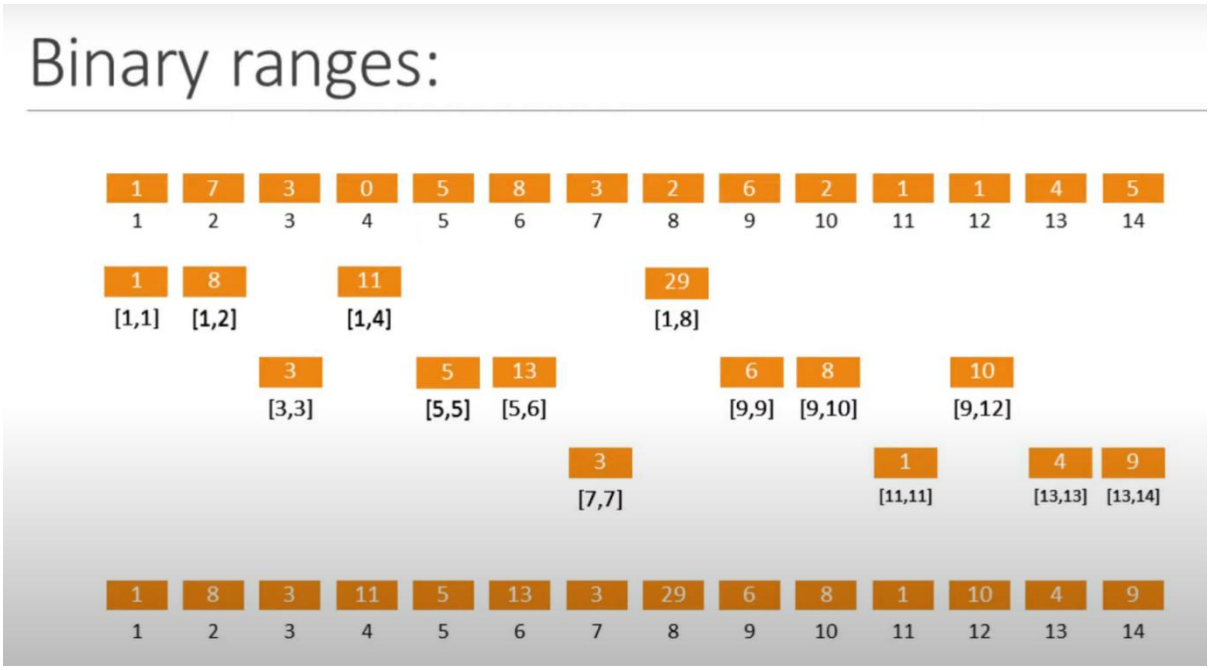
For queries any range from start to n, let n = 13. then
13 - 1 = 12 (12+1,13) -> (13,13)
12 - 8 = 8 (8+1,12) -> (9,12)
8 - 8 = 0 (0+1,80) -> (1,8)

First find 13 then 9-12 then 1-8. In binary 13=1101, now first in 13 index then flip right bit will make 110 that indicate 10 index then flip right bit 1000 that indicate 8 index. So sum 8+10+13 index this sum will provide all sum from 0 to 13.



number	binary representation
16	10000
15	01111
14	01110
13	01101
12	01100
11	01011
10	01010
9	01001
8	01000
7	00111
6	00110
5	00101
4	00100
3	00011
2	00010
1	00001

If first right bit is 1 then it will store 1 number. The right set bit without 1st bit from right side is responsible to store the number times value of equivalent value for the bit. The 2nd bit will say from which index this value is start to store. All rest bit isn't responsible.



We can make BIT like first 1,2,4,8,16 where 1 store 1 value, 2 store 2 value from own index to previous index, 4 store 4 value from own index to previous index, 8 store 8 values from own index to previous index, then empty space with be fill using the same procedure like 1,2,4,8. if there have no space for value jump to next space for next fillup.

Square Root Decomposition (SQD) is an algorithmic technique used to solve range queries efficiently, such as sum, minimum, or maximum, on an array. It divides the array into blocks of size √N and processes each block in O(√N) time. This is ideal for scenarios where there are multiple queries and where updates are rare or non-existent.

Time Complexity:
Preprocessing: O(N)
Query: O(√N)
Update: O(√N)
Space Complexity:
O(NO+√N) where N is the size of the array.

String Matching(KMP):

The Knuth-Morris-Pratt (KMP) algorithm is a string-searching algorithm which is used to find a pattern within large texts efficiently.

It preprocesses the pattern string and creates an array called the LPS array which indicates how much of the pattern can be reused after a mismatch.

- LPS is the Longest Proper Prefix which is also a Suffix.

Two exaple are given with LPS table or failure table.

a	a	b	a	a	b	a	a	a
0	1	0	1	2	3	4	5	2

A	B	C	A	B	D	A	B	C	A	B	C	A	B	D
0	0	0	1	2	0	1	2	3	4	5	3	4	5	6

ABCAB, ABCAB for LPS value 5.

- We will make matching from start using i & j where i will left and j will forward, if a matching is not occurs then I will back 1 length and checked its LPS value to index of LPS table, if match then add its index+1 or not match in the same way make 1 length back, do it until a match occurs or reach to 0 index.
- Any value of LPS table indicate that from start a string of this LPS value length prefix and a LPS value length of this LPS value from the index of suffix is match.

Time Complexity: O(n + m), where n is the length of the text and m is the length of the pattern. This is because creating the LPS (Longest Prefix Suffix) array takes O(m) time, and the search through the text takes O(n) time.
Auxiliary Space: O(m), as we need to store the LPS array of size m.

Rabin-Karp Algorithm:

The Rabin-Karp algorithm is a string matching algorithm that uses hashing to find patterns within a text efficiently.

Hash function for any string s;

$$\text{Hash}(s) = s_0 \cdot B^{m-1} + s_1 \cdot B^{m-2} + \ \ldots\ . + s_{m-1} \cdot B^0$$

Given s_i is the ASCII value of the ith character of string s.

- We can imagine Base-B as a value that is equal to or greater than the length of string or best choice is to select prime number.
- To overcome overflow of hash value we can modulus this hash value with a number M, this M also should prime and very large so that less probability of collision.
- After matching hash value we also need to check if all character of pattern is equal to text.

To avoid O(n*m) complexity when we pass each substring to hash function we need **rolling hash**.

H_i is the hash value of the string of length m starting at the i-th index of s. Then we can write:

$$H_i = s_i \cdot B^{m-1} + s_{i+1} \cdot B^{m-2} + \ldots + s_{i+m-1} \cdot B^0$$

Rolling hash or General formula is:

$$H_i = (H_{i-1} - S_{i-1} \cdot B^{m-1}) \cdot B + s_{i+m-1}$$

Here H_{i-1} is previous value, $S_{i-1} \cdot B^{m-1}$ is first value and s_{i+m-1} is the last value.

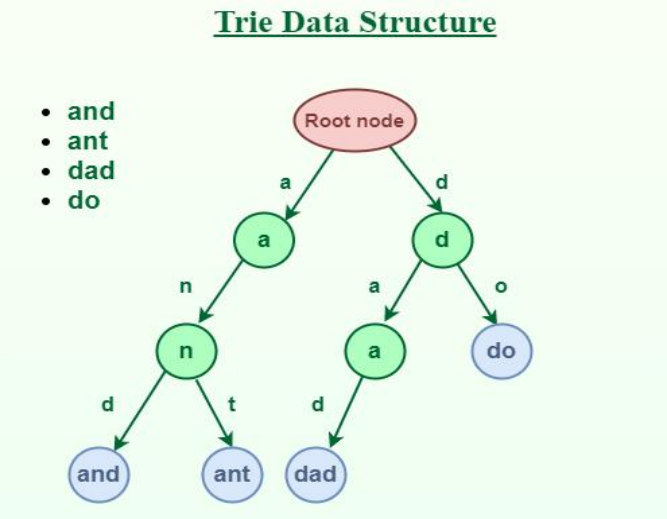
Now using this formula, the hash value of each substring of m length can be easily found in O(n).

Trie Data Structure:

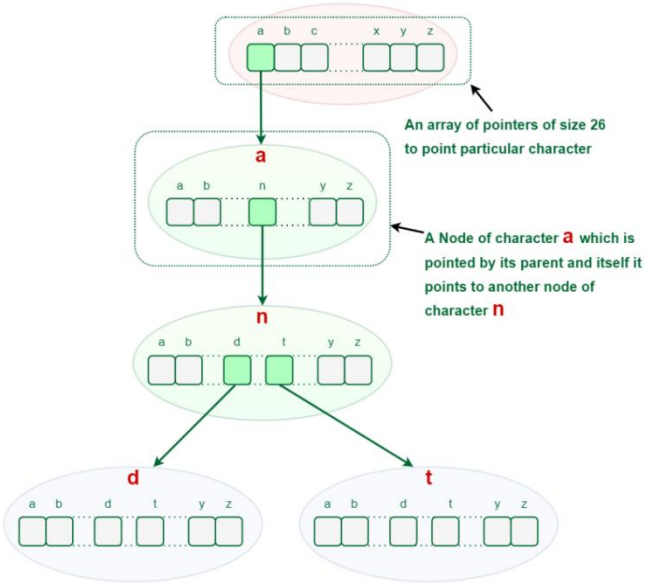
A Trie (prefix tree) is a tree-like data structure that is used to store a dynamic set of strings, where each node represents a prefix of strings.
It is particularly useful for tasks related to string search and manipulation, such as autocomplete(prefix search), prefix matching, Spell Checking, and dictionary implementations.

Properties of Trie:

- Each node can have up to **alphabet_size** children.
- Strings are represented as paths from the root node to leaf nodes.
- A special marker at the end of a string indicates the end of a valid word or key.
- Trie is derived from **reTRIEval**, which means finding something.
- Trie data structure follows a property that If two strings have a common prefix then they will have the same ancestor in the trie. This particular property allows to find all words with a given prefix.
- The main disadvantage of the trie is that it takes a lot of memory to store all the strings. For each node, we have too many node pointers which are equal to the no of characters in the worst case.



After storing the word “and” and “ant” the Trie will look like this:



Time & Space complexity:

Operation	Time Complexity
Insertion	O(n) Here n is the length of string to be searched
Searching	O(n)
Deletion	O(n)

For n keys, each with an average length of L, the maximum number of nodes is proportional to O(n×L).

Dynamic Programming(Divide & Concure Method)

The essence of dynamic programming is to avoid repeated calculation.

Application:
Bellman–Ford Shortest Path, Floyd Warshall, Matrix Chain Multiplication, Knapsack Problem, Longest Common Subsequence, Edit Distance, Coin Change, Longest Increasing Subsequence.

Top Down Method:

Write recursive solution, then save repeated states in a lookup table. It is called dynamic programming with **memoization**. That's read "memoization" (like we are writing in a memo pad) not memorization. Solves the problem recursively by breaking it into smaller subproblems.

Memoization is an optimization technique used in dynamic programming to improve the efficiency of recursive algorithms by storing the results of previously computed subproblems. It ensures that the same computation is not repeated multiple times, reducing redundant calculations and saving time.
Overlap sub problem is repeated sub problem again and again.

- Using look up table we can do Fibonacci calculation in O(n) where it was $O(2^n)$ in normal recursive call.
- This approach is called top-down, as we can call the function with a query value and the calculation starts going from the top (queried value) down to the bottom (base cases of the recursion), and makes shortcuts via memoization on the way.
- It starts from the end(base case) and works backward.
- Recursive, memory-intensive, solves only needed subproblems.
- O(n) time and O(n)+O(n) = O(n) space complexity.

```
//Top down approach
int f(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return f(n - 1) + f(n - 2);
}
```

Bottom Up Method:

We start at the bottom (base cases of the recursion) the smallest subproblems and gradually build up to the final solution. We only use recursive formula on table entries and do not make recursive calls. Solves the problem iteratively by building solutions from smaller subproblems.

- It uses the **tabulation** technique to implement the dynamic programming approach. It solves the same kind of problems but it removes the recursion. If we remove the recursion, there is no stack overflow issue and no overhead of the recursive functions. In this tabulation technique, we solve the problems and store the results in a matrix.
- Iterative, avoids recursion, systematically solves all subproblems.
- O(n) time and O(n) space complexity.

```
//Bottom Up Approach
const int MAXN = 100;
int fib[MAXN];

int f(int n) {
    fib[0] = 0;
    fib[1] = 1;
    for (int i = 2; i <= n; i++) fib[i] = fib[i - 1] + fib[i - 2];

    return fib[n];
}
```

DP step to solve problem:

- Problem Understanding.
- Define Subproblems.
- Recurrence Relation.
- Choose DP Approach (Top-Down or Bottom-Up).
- Base Case.
- Solve Subproblems.
- Build the Final Solution.

Subset generate using -> top down(recursion)
-0> bottor up

Meet in the middle:

It is a powerful technique in algorithm design, particularly useful for solving problems involving combinatorics, search, or optimization when the problem size is too large for brute force to be feasible. Like divide and conquer it splits the problem into two, solves them individually and then merge them.

Steps of the Algorithm:

- **Divide the Input**: Split the problem into two halves.
- **Generate All Possible States**: For both halves, generate all possible states or combinations.
- **Store One Side Efficiently**: Store the results of one half (e.g., in a hash table or sorted array).
- **Search or Match**: For each result of the other half, efficiently search for a match or desired condition in the precomputed results of the first half.
- **Combine Results**: Combine the two halves to find the solution.

Time Complexity

- Generating subset sums for each half: $O(2^{n/2})$
- Sorting one half: $O(2^{n/2}\log(2^{n/2}))$
- Searching for complementary sums: $O(2^{n/2}\log(2^{n/2}))$
- Overall Time Complexity: $O(2^{n/2}\log(2^{n/2}))$

Which is much better than $O(2^n)$ of brute force.

A convex polygon is a polygon in which all interior angles are less than 180 degrees. A convex hull is the smallest convex polygon that contains a given set of points.

Graham scan algorithm is a simple and efficient algorithm for computing the convex hull of a set of points. It works by iteratively adding points to the convex hull until all points have been added.

The algorithm first finds the bottom-most point P^0. If there are multiple points with the same Y coordinate, the one with the smaller X coordinate is considered. This step takes O(N) time.

Next, all the other points are sorted by polar angle in clockwise order. If the polar angle between two or more points is the same, the tie(multiple point is a single distance) should be broken by distance from P^0, in increasing order.

Poler angle for a point = arctan(y/x)
Then we iterate through each point one by one, and make sure that the current point and the two before it make a clockwise turn, otherwise the previous point is discarded, since it would make a non-convex shape. Checking for clockwise or anticlockwise nature can be done by checking the orientation.

We use a stack to store the points, and once we reach the original point P^0, the algorithm is done and we return the stack containing all the points of the convex hull in clockwise order.

Time Complexity: O(nLogn), where n be the number of input points.
The first step (finding the bottom-most point) takes O(n) time. The second step (sorting points) takes O(nLogn) time. The third step takes O(n) time. In the third step, every element is pushed and popped at most one time. So the sixth step to process points one by one takes O(n) time, assuming that the stack operations take O(1) time. Overall complexity is O(n) + O(nLogn) + O(n) + O(n) which is O(nLogn).
Auxiliary Space: O(n), as explicit stack is used, since no extra space has been taken.

Jarvis March algorithm (also called the Gift Wrapping algorithm) is a method for finding the convex hull of a set of points . It works by "wrapping" the points of the convex hull like a gift, starting from an extreme point and iteratively selecting the next point on the hull by comparing angles.

Oriantation are produced by using cross product. The cross product tells us the direction of rotation (left or right) as we move through the points. Positive, negative, or zero determines counterclockwise, clockwise, or collinear arrangement, respectively.

Comparison between Graham Scan and Jarvis March

Aspect	Graham Scan	Jarvis March
Approach	Sorting-based + orientation test	Iterative (Greedy approach)
Time Complexity	O(n log n)	O(nh), worst-case O(n^2)
Space Complexity	O(n)	O(n)
Sorting	Requires sorting by polar angle	No sorting required
Handling Collinear	Skips collinear points	Includes collinear points, selects farthest
Best Use Case	Large datasets, efficiency important	Small datasets or fewer points in hull
Implementation Ease	More complex due to sorting	Easier to implement and understand

Two line segments(Intersection):

The theory behind the Line Segment Intersection problem is based on orientation tests and bounding box checks.

p1(x1, y1) to q1(x2, y2)
p2(x3, y3) to q2(x4, y4)

Orientation Function:
Compute the orientation of the triplet of points (p, q, r):

$$\text{Orientation} = (q_y - p_y) \times (r_x - q_x) - (q_x - p_x) \times (r_y - q_y)$$

o1 = orientation(p1, q1, p2)
o2 = orientation(p1, q1, q2)
o3 = orientation(p2, q2, p1)
o4 = orientation(p2, q2, q1)

If $o1 \neq o2$ and $o3 \neq o4$, then the segments intersect.