CSE2221 : Design and Analysis of Algorithm

Made by Md. Mehedi Hasan Rafy

Topics:

Sorting:

1. Merge Sort
2. Quick Sort
3. Counting Sort

Searching:

1. Binary Search
2. Ternary Search
3. Two Pointer

Number Theory:

1. Binary exponentiation
2. Matrix exponentiation
3. Inverse binary expopentiation

Graph:

1. DFS
2. Iterative depending DFS
3. BFS
4. Dijkstra
5. Prim’s algorithm
6. Krushkal’s algorithm
7. A Star search
8. Lowest common ancestor
9. Disjoint set union (DSU)
10. Warshall algorithm
11. Topological Sort

String:

1. KMP
2. Robin Karp
3. Trie

DP:

1. Subset Generate
2. Top Down
3. Bottom Up
4. Meet in the middle

Data Structure:

1. Prefix sum
2. Difference array
3. Segment tree
4. Segment tree lazy
5. Binary Index Tree
6. Sqrt decomposition

Geometry:

1. Graham’s Scan
2. Jarvis March
3. Line Segment Intersection

Previous Year Questions on topics

Intoduction

1. ***Define Algorithm and efficiency of algorithm.*****

Solution:

Algorithm

An algorithm is a finite sequence of well-defined instructions to solve a problem or perform a computation. It must be unambiguous, correct, and terminate in a finite number of steps.

Efficiency of an Algorithm

The efficiency of an algorithm measures its resource utilization, primarily in terms of time complexity and space complexity.

1. ***Time Complexity:*** Measures the number of basic operations as a function of input size n. Expressed using Big-O notation (e.g., O(n), O(logn), O(n^2)). For example, Merge Sort runs in O(nlogn) time.
2. ***Space Complexity:*** Measures the amount of memory required as a function of input size n. Includes both input storage and auxiliary space. For example, Merge Sort requires O(n) extra space, while Quick Sort requires O(logn) in-place.

2. ***What are the steps required for algorithm design?***

Solution:

Steps in Algorithm Design

1. ***Problem Definition :*** Clearly define the input, output, and constraints of the problem. For example, finding the shortest path in a graph.
2. ***Analysis of Requirements :*** Identify performance constraints (time, space, hardware limitations).
3. ***Choosing the Appropriate Data Structure :*** Select a data structure that optimizes efficiency. For example, use a heap for Dijkstra's algorithm instead of an array.
4. ***Designing the Algorithm :*** Use a suitable design paradigm for example, divide and conquer or greedy or dynamic programming etc.
5. ***Analyzing the Algorithm :*** Determine time and space complexity using asymptotic analysis. Consider best, worst, and average-case scenarios.
6. ***Implementing the Algorithm :*** Convert the algorithm into a programming language. Ensure correctness with boundary and edge case testing.

7. ***Optimization and Refinement :*** Improve efficiency by reducing complexity (e.g., optimizing loops, using memoization). For example, using binary search (O(logn)) instead of linear search (O(n)).
8. ***Testing and Debugging :*** Validate with different test cases, including worst-case inputs. Debug logical errors and optimize further if necessary.

3. How to optimize an algorithm? Illustrate with an example.

Solution:

<u>Optimization of algorithm</u>

To optimize an algorithm, we systematically reduce its time and space complexity using different techniques. Below, we illustrate this with the Fibonacci number problem.

1. *<u>Brute Force Approach – Recursion</u>*

The naive recursive approach directly implements the Fibonacci recurrence relation:

$$F(n)=F(n-1)+F(n-2)$$

- ***Time Complexity:*** O(2^n) (exponential due to repeated computations for example, F(5) computes F(3) twice and (2) thrice.).
- ***Space Complexity:*** O(n) (due to recursion stack).
- ***Optimization Needed:*** Avoid redundant calculations.

2. *<u>Dynamic Programming – Memoization</u>*

We store previously computed values in an array to avoid redundant recursion.

- ***Time Complexity:*** O(n) (each Fibonacci number is computed once).
- ***Space Complexity:*** O(n) (due to recursion and memo dictionary).
- ***Optimization Needed:*** Reduce space usage.

3. *<u>Space Optimized Dynamic Programming</u>*

Instead of storing all Fibonacci numbers, we only keep the last two.

- ***Time Complexity:*** O(n) (single loop).
- ***Space Complexity:*** O(1) (only two variables used).
- ***Optimization Needed:*** Reduce time complexity.

Each step optimized the previous approach, demonstrating how algorithmic improvements lead to more scalable solutions. The final method is the most efficient, making it suitable for large inputs.

4. ***Explain the relation between data structure and algorithm.***

Solution:

Relationship between data structure and algorithm

- A data structure is like a storage system (array, list, tree), and an algorithm is like a method (sorting, searching) used to retrieve or manipulate the stored data.
- Algorithms operate on data stored in data structures to perform computations efficiently. For example, Sorting algorithms (Merge Sort, Quick Sort) manipulate arrays or linked lists.
- The efficiency of an algorithm depends on the data structure used. For example, searching in a sorted array using Binary Search (O(logn)) is faster than Linear Search in an unsorted array (O(n)).
- The design and implementation of algorithms depend on the characteristics of data structures. For example, Dijkstra's Algorithm uses a Priority Queue (Min-Heap) to extract the minimum-cost node efficiently.

Analysis of Complexity

1. ***What is meant by best case, worst case, and average case complexity?***

Solution:

1. ***Best Case Complexity***

   The minimum time taken by an algorithm for an input of size n. It occurs when the input is ideal for the algorithm. For example, In *Linear Search*, if the element is found at the first position, the time complexity is Ω(1).

2. ***Worst Case Complexity***

   The maximum time taken by an algorithm for an input of size n. It occurs in the least favorable scenario. For example, in *Linear Search*, if the element is found at the last position, the time complexity is O(n).

3. **Average Case Complexity**

   The expected time taken by an algorithm for random inputs of size n. It is calculated by averaging the running times over all possible inputs. For example: In *Linear Search*, the average number of comparisons is approximately Θ(n).

2. ***What is space complexity?***

Solution:

<u>Space Complexity</u>

Space complexity of an algorithm is the total amount of memory required by the algorithm in terms of input size n. Space complexity includes the memory used for, variables and data structures, recursive function call stacks and dynamically allocated memory.

For example,

- ***Constant space O(1) :*** An algorithm that swaps two variables using a temporary variable uses constant space.
- ***Linear Space O(n) :*** An algorithm that creates an array of size 'n' to store data has a space complexity of O(n).
- ***Quadratic Space O(n^2) :*** An algorithm that creates a 2D array of size 'n x n' has a space complexity of O(n^2).

3. ***What is the smallest value of n such that an algorithm whose running time is $100n^2$ runs faster than an algorithm whose running time is 2^n on the same machine.*****

Solution:

We need to find the smallest n such that:

$$100n^2 < 2^n$$

We evaluate both functions iteratively for small n and find the required value of n.

n	$100n^2$	2^n	$Condition\, 100n^2 < 2^n\,?$
1	100	2	No
2	400	4	No
5	2500	32	No
10	10000	1024	No
14	19600	16384	No
15	22500	32768	Yes

So, the value of n is 15.

4. ***Deduce the time complexity of the following function.***

```
void recursive(int n, int m, int o){
    if (n <= 0){
        printf("%d, %d\n", m, o);
    } else{
        recursive(n-1, m+1, o);
        recursive(n-1, m, o+1);
    }
}
```

Solution:

<u>Time Complexity</u>

The function recursive(n, m, o) follows this recurrence relation $T(n)=T(n-1)+T(n-1)+O(1)$

Which simplifies to: $T(n)=2T(n-1)+O(1)$

Expanding for a few terms:

$$\begin{aligned} T(n) &= 2T(n-1)+O(1) \\ &= 2(2T(n-2)+O(1))+O(1) \\ &= 4T(n-2)+2O(1)+O(1) \\ &= 8T(n-3)+4O(1)+2O(1)+O(1) \end{aligned}$$

After k expansions, $T(n)=2^kT(n-k)+\sum_{i=0}^{k-1}2^iO(1)$

When $k=n$, we reach the base case $T(0)=O(1)$, $T(n)=2^nT(0)+O(2^n)$

Since T(0) is a constant, we get the final time complexity: $T(n)=O(2^n)$

5. ***Deduce the time complexity of the following function.***

```
int recursive(int n){
        for (i = 0; i < n; i += 2){
            printf("%d", i);
        }
    if (n <= 0) return 1;
    else return 1 + recursive(n-5);
}
```

Solution:

<u>Time Complexity</u>

This function makes one recursive call with $n-5$. The loop runs O(n) times. Thus the recurrence relation is of the form: $T(n)=T(n-5)+O(n)$

Expanding the recurrence:

$$\begin{aligned} T(n) &= T(n-5)+O(n) \\ &= T(n-10)+O(n-5)+O(n) \\ &= T(n-15)+O(n-10)+O(n-5)+O(n) \end{aligned}$$

Continuing this process, the recursion stops when n≤0, which happens after $n/5$ recursive calls.

Summing up the work done at each level:

$$O(n)+O(n-5)+O(n-10)+\cdots+O(5)+O(0)$$

This forms an arithmetic series: $O(n)\cdot O(n/5)=O(n^2)$

Thus, the time complexity of the function is: $T(n)=O(n^2)$

6. ***Calculate the time complexity of the following function.******

```
Void f(int n) {
    if (n == 1) return;
    f(n – 1);
}
```

Solution:

The function makes one recursive call with $n-1$. The recursion stops when $n=1$.

Thus, the recurrence relation is: $T(n)=T(n-1)+O(1)$

Expanding it recursively:

$$\begin{aligned} T(n) &= T(n-1)+O(1) \\ &= T(n-2)+O(1)+O(1) \\ &= T(n-3)+O(1)+O(1)+O(1) \end{aligned}$$

After $n-1$ expansions, we reach the base case $T(1)=O(1)$:

$$T(n)=O(1)+O(1)+O(1)+\cdots+O(1)n-1\,terms$$

hence, $T(n)=O(n)$

7. ***Calculate the time complexity of the following function.******

```
Void g(int n) {
    if (n == 1) return;
    g(n – 1);
    g(n – 1);
}
```

Solution:

The function makes two recursive calls, each reducing n by 1. The recursion stops when $n=1$.

This gives the recurrence relation: $T(n)=2T(n-1)+O(1)$

Expanding the recurrence:

$$\begin{aligned} T(n) &= 2T(n-1)+O(1) \\ &= 2(2T(n-2)+O(1))+O(1) \\ &= 4T(n-2)+2O(1)+O(1) \\ &= 8T(n-3)+4O(1)+2O(1)+O(1) \end{aligned}$$

After k expansions, $T(n)=2^kT(n-k)+\sum_{i=0}^{k-1}2^iO(1)$

When $k=n$, we reach the base case $T(0)=O(1)$, $T(n)=2^nT(0)+O(2^n)$

Since T(0) is a constant, we get the final time complexity: $T(n)=O(2^n)$

<u>Sorting</u>

1. ***When does the worst case of quick sort algorithm occur? Explain.***

Solution:

The worst case of Quick Sort occurs when the pivot selection consistently results in the most unbalanced partitions. As a result, time complexity becomes O(n) at the worst case. This happens in the following cases:

1. ***<u>Already Sorted or Reverse Sorted Array</u> :*** If the pivot is always the smallest or largest element, one partition contains n−1 elements while the other contains 0. This leads to high recursion depth and degrades Quick Sort to O(n^2) time complexity.
2. ***<u>Choosing the First or Last Element as Pivot in a Sorted Array</u> :*** If Quick Sort always picks the first or last element as the pivot in a sorted or reverse-sorted array, it results in highly unbalanced partitions. In this case, Quick Sort degrades to O(n^2) time complexity.
3. ***<u>All Elements are Identical</u>*** **:** If all elements are the same and a naive pivot selection strategy is used, each partitioning step does not divide the array effectively.

2. ***Derive the time complexity of quick sort algorithm for worst case.***

Solution:

If the pivot is always the smallest or largest element, one partition contains $n-1$ elements, and the other contains 0.

Hence, the recurrence relation is: $T(n)=T(n-1)+O(n)$

where O(n) accounts for partitioning.

Expanding the recurrence:

$$\begin{aligned} T(n) &= T(n-1)+O(n) \\ &= T(n-2)+O(n-1)+O(n) \\ &= T(n-3)+O(n-2)+O(n-1)+O(n) \end{aligned}$$

Continuing until $n=1$: $T(n)=O(1)+O(2)+O(3)+\cdots+O(n)$

This forms a summation: $T(n)=O(1+2+3+\cdots+n)$

Using the formula for the sum of the first n natural numbers: $T(n)=O(\frac{n(n+1)}{2})=O(n^2)$

3. ***Find out the time complexity of quick sort algorithm.***

Solution:

1. ***Best case***

   The pivot divides the array into two equal halves at every step.

   The recurrence relation: $T(n)=2T(n/2)+O(n)$

   where O(n) is the time for partitioning.

   Using ***Master Theorem,***

   $T(n)=aT(n/b)+f(n)$,
   where $a=2$, $b=2$, $f(n)=O(n)$.
   Since $f(n)=O(n)$ matches $O(n^{\log_2 2})=O(n)$,

   So, time complexity is $T(n)=O(nlogn)$ at best case.

2. ***Average case***

   The pivot divides the array into uneven but not worst partitions, say 3n/4 and n/4.

   The recurrence: $T(n)=T(3n/4)+T(n/4)+O(n)$

   Using recurrence tree expansion, it results in: $T(n)=O(nlogn)$

3. ***Worst case***

*****Discussed above*****

4. ***Is it possible to have O(N + n) time complexity for sorting some numbers? If possible, write down the procedure.***

Solution:

Yes, it is possible to achieve $O(N+n)$ time complexity for sorting certain numbers using Counting Sort.

Procedure

Counting Sort is a non-comparison-based sorting algorithm that sorts in O(N+n) time, where**:**

- N is the range of numbers (maximum value in input).
- n is the number of elements to sort.

Steps:

1. Determine the maximum number N in the input.
2. Initialize an array count of size N+1 with zeros.
3. Traverse the input array and store the count of each element in count.
4. Iterate through count, placing elements back into the output array in sorted order.

Time Complexity Analysis:

- Counting occurrences: $O(n)$
- Creating the count array: $O(N)$
- Generating the sorted output: $O(N)$

Thus, the total complexity is $O(N+n)$.

5. ***Write down the insertion sort algorithm.***

Solution:

Algorithm

```
1. For i=1 to n−1:
   1.1 Set key = A[i].
   1.2 Set j = i - 1.
   1.3 While j≥0 and A[j]>key:
           Move A[j] to A[j+1].
           Decrement j (j = j - 1).
   1.4 Place key in A[j+1].
2. End
```

6. ***How can you find out the time complexity of merge sort algorithm?***

Solution:

Merge Sort works by:

1. **Dividing** the array into two halves → O(1)
2. **Recursively sorting** each half → 2T(n/2)
3. **Merging** the two sorted halves → O(n)

Thus, the recurrence relation is: $T(n)=2T(n/2)+O(n)$

Expanding the recurrence,

$$\begin{aligned} T(n) &= 2T(n/2)+n \\ &= 2(2T(n/4)+n/2)+n \\ &= 4T(n/4)+2(n/2)+n \\ &= 8T(n/8)+4(n/4)+2(n/2)+n \end{aligned}$$

After $\log_2 n$ levels, the size of the subproblem becomes T(1), which takes constant time.

At each level, the total work done is O(n), and there are $O(logn)$ levels.

Thus, the total time complexity is: $T(n)=O(nlogn)$

7. ***Sort the numbers {2, 5, 3, 0, 2, 3, 0, 3, 6} using counting sort algorithm including all steps of sorting.*****

Solution:

Lets say the given array, A = {2, 5, 3, 0, 2, 3, 0, 3, 6}.
Here, max(A) = 6.

So, we create a count array of size 6 + 1 = 7 and store the frequency of the corresponding numbers.

index	0	1	2	3	4	5	6
count	2	0	2	3	0	1	1

Now, the prefix sum of the count array would be,

index	0	1	2	3	4	5	6
prefix	2	2	4	7	7	8	9

Right shifting the elements by 1 we get,

index	0	1	2	3	4	5	6
prefix	0	2	2	4	7	7	8

Now, we have to place the elements in the output array, to do this we traverse the array A in reverse and place elements in their correct position from the prefix array.

Elements	***Position***	***Output array***								
6	8									6
3	4					3				6
0	0	0				3				6
3	5	0				3	3			6
2	2	0		2		3	3			6
0	1	0	0	2		3	3			6
3	6	0	0	2		3	3	3		6
5	7	0	0	2		3	3	3	5	6
2	3	0	0	2	2	3	3	3	5	6

So the sorted array New = {0, 0, 2, 2, 3, 3, 3, 5, 6}.

Searching

1. ***An integer number is given. Your task is to write a program to calculate the square root of the number using binary search algorithm.***

Solution:

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

int sqrt(int num) {
    if (num == 0 || num == 1) return num;
    int low = 1, high = num, ans = 0;
    while (low <= high) {
        long long mid = low + (high - low) / 2, sq = mid * mid;
        if (sq == num) return mid;
        else if (sq < num) {
            ans = mid;
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return ans;
}

int main() {
    int num;
    cout << "Enter a number: ";
    cin >> num;

    cout << "Square root (floor value): " << sqrt(num) << endl;
    return 0;
}
```

Number Theory

1. How can you calculate the greatest common divisor (GCD) of two non-negative integers?

Solution:

We use Euclid's Algorithm to find the gcd of two non-negative integers.

The Euclidean algorithm is based on the property:

$$GCD(a,b)=GCD(b,a \bmod b)$$

until $b=0$, at which point a is the GCD.

Steps:

1. If $b=0$, return a as GCD.
2. Otherwise, compute $GCD(b,a \bmod b)$.

Example: Find GCD(48,18).

- 48 mod 18 = 12, so compute GCD(18, 12).
- 18 mod 12 = 6, so compute GCD(12, 6).
- 12 mod 6 = 0, so GCD = 6.

2. ***How can you calculate the trailing zeros of factorial n?*****

Solution:

We can calculate trailing zeros in $n!$ using the formula below:

$$Trailing\ Zeros = \lfloor \frac{n}{5} \rfloor + \lfloor \frac{n}{5^2} \rfloor + \lfloor \frac{n}{5^3} \rfloor + \dots$$

until $5^k > n$.

We can further improve this formula as this is a geometric progression.

First term = $\frac{1}{5}$

general ratio = $\frac{\frac{1}{5^2}}{\frac{1}{5}} = \frac{1}{5}$

hence, trailing zeros = $\lfloor \frac{n \times (5^k - 1)}{4 \times 5^k} \rfloor$

where $k = \lfloor \log_5 n \rfloor$.

3. ***How can you calculate a^n (for large a and n) using only $O(\log n)$ multiplications instead of O(n) multiplications?***

Solution:

We can use the Binary Exponentiation algorithm to find a^n in $O(\log n)$ time complexity.

<u>Algorithm</u>

The idea is based on the property:

- $a^n = a^{(\frac{n}{2})2}$ if n is **even**
- $a^n = a^{(\frac{(n-1)}{2})2}$ if n is **odd**

<u>Steps</u>

1. Initialize **result** = 1.
2. While $n > 0$:
   - If n is odd, multiply **result** by a.
   - Square a and halve n.
3. Return **result**.

Example: Compute 3^{13}

1. 13 is odd → $result = 3$, $a = 3^2 = 9$, $n = 6$.
2. 6 is even → $a = 9^2 = 81$, $n = 3$.

3. 3 is odd → $result = 3 \times 81 = 243$, $a = 81^2 = 6561$, $n = 1$.
4. 1 is odd → $result = 243 \times 6561 = 1594323$, $n = 0$.

Final answer: $3^{13} = 1594323$.

4. Why do you need to calculate modular inverse?

Solution:

The **modular inverse** of a modulo m is an integer x such that: $a \cdot x \equiv 1 (mod\, m)$

It is useful in various computational problems, including:

1. ***Division in Modular Arithmetic***

   Since division is not directly defined in modular arithmetic, we use the modular inverse:

   $$\frac{a}{b} mod\, m = a \cdot b^{-1} mod\, m$$

   where b^{-1} is the modular inverse of $b\, mod\, m$.

2. ***Fermat's Little Theorem***

   If m is prime, the modular inverse is:

   $$a^{-1} \equiv a^{m-2} mod\, m$$

   This is widely used in modular exponentiation.

3. ***Solving Linear Congruences***

   For equations like $ax \equiv b (mod\, m)$, the inverse helps compute x .

4. ***Chinese Remainder Theorem (CRT)***

   Modular inverse is used in CRT to solve systems of congruences efficiently.

5. ***Combinatorics (Modular Binomial Coefficients)***

   Computing combinations efficiently under a modulus:

   $$C(n,k) \equiv \frac{n!}{k!(n-k)!} mod\, m$$

   requires modular inverses of factorials.

5. How can you calculate number of digits of factorial n?

Solution:

So we have to find number of digits in $n!$. We can get digits by this formula:

$$digit = \lfloor \log_{10} n! \rfloor + 1$$

Which further simplifies to,

$$digit = \lfloor \sum_{i=1}^{n} \log_{10} i \rfloor + 1$$

So, we will initialize a variable $digit = 1$. Then, run a loop from 1 to n, and each time calculate $\lfloor \log_{10} i \rfloor$ and add it with digit and eventually getting the required number of digit in $n!$.

String

1. ***Prepare three datasets for the best case, average case, and the worst case of KMP(Knuth Morris Pratt) string matching algorithm. Explain why the datasets are belonging to the best case, average case, and the worst case.***

Solution:

The Knuth-Morris-Pratt (KMP) algorithm preprocesses the pattern using a prefix function to allow efficient string matching, achieving O(n + m) complexity in the worst case. Below are three datasets demonstrating its best case, average case, and worst case performance.

1. <u>Best Case Dataset</u>

Text: "abcdefghijklmnopqrstuvwxyz"
Pattern: "zyx"

Why is this the Best Case?

- The pattern does not appear in the text at all.
- The prefix function contains only zeros because no proper prefixes match proper suffixes.
- The algorithm performs a simple left-to-right scan, checking each character only once, leading to O(n) time complexity.

2. <u>Average Case Dataset</u>

Text: "abcabcabcabcabcabcabcabc"
Pattern: "abcabc"

Why is this the Average Case?

- The pattern appears multiple times in the text, but not in a fully overlapping manner.
- The prefix function allows partial matches to be reused instead of starting over from index 0, improving efficiency.
- The algorithm runs in O(n) on average due to effective skipping of redundant comparisons.

3. <u>Worst Case Dataset</u>

Text: "aaaaaaaaaaaaaaaa" (length **n**)
Pattern: "aaaaab" (length **m**)

Why is this the Worst Case?

- The text consists of repeated occurrences of "a", while the pattern has a single differing character ("b") at the end.
- The prefix function builds a high failure function because most of the pattern is repeated, forcing frequent backtracking.
- The algorithm spends a significant amount of time attempting matches before realizing they fail at the last character, leading to O(n + m) worst-case complexity.

2. ***Implement the Rabin-Karp algorithm and mention which problems are solved using this algorithm?***

Solution:

<u>Implementation of Rabin-Karp Algorithm</u>

```
RabinKarp(string T, string P, int p) {
    n = T.length();
    m = P.length();
    d = 256;
    h = 1;
    hashPattern = 0, hashText = 0;

    for (i, m - 1)
        h = (h * d) % p;

    for (i, m) {
        hashPattern = (d * hashPattern + P[i]) % p;
        hashText = (d * hashText + T[i]) % p;
    }

    for (i, n - m) {
        if (hashPattern == hashText) {
            bool match = true;
            for (j, m) {
                if (T[i + j] != P[j]) {
                    match = false;
                    break;
                }
            }
            if (match)
                return i;
        }

        if (i < n - m) {
            hashText = (d * (hashText - T[i] * h) + T[i + m]) % p;
            if (hashText < 0)
                hashText += p;
        }
    }

    return -1;
}
```

Problems Solved Using Rabin-Karp Algorithm

1. ***String Matching*** – Efficiently finds a substring within a text.
2. ***Plagiarism Detection*** – Detects copied sections in documents.
3. ***DNA Sequence Matching*** – Finds a specific gene sequence.
4. ***Duplicate File Detection*** – Uses hashing to compare document contents.

3. ***Suppose you have a text AABAACAADAABAABA and a pattern ABA. Find out that how many times this pattern exists in this text. Show all the steps.***

Solution:

Finding Pattern "ABA" in Text "AABAACAADAABAABA" Using KMP Algorithm. We will apply the Knuth-Morris-Pratt (KMP) Algorithm to find occurrences of "ABA" in "AABAACAADAABAABA".

Step 1: Construct the Longest Prefix Suffix (LPS) Array

Pattern: "ABA"

- Length of pattern (m = 3)
- Compute LPS values:

Index	***Pattern Prefix,*** P_i	***LPS Value***
0	A	0
1	AB	0
2	ABA	1

Thus, LPS = [0, 0, 1].

Step 2: Search for the Pattern in the Text

- ***Text:*** "AABAACAADAABAABA" (length n = 15)
- ***Pattern:*** "ABA" (length m = 3)

Text Index (i)	**Pattern Index (j)**	**Text Char (T[i])**	**Pattern Char (P[j])**	**Comparison**	**Match?**	**Action**
0	0	A	A	A == A		Move i++, j++
1	1	A	B	A ≠ B		Set j = LPS[0] = 0
1	0	A	A	A == A		Move i++, j++
2	1	B	B	B == B		Move i++, j++
3	2	A	A	A == A		Move i++, j++
j == m						Pattern found at **i - j = 1**
4	1	A	B	A ≠ B		Set j = LPS[0] = 0
4	0	A	A	A == A		Move i++, j++
5	1	C	B	C ≠ B		Set j = LPS[0] = 0
5	0	C	A	C ≠ A		Move i++
6	0	A	A	A == A		Move i++, j++
						Set j = LPS[2] = 1

7	1	A	B	A ≠ B	Set j = LPS[0] = 0
7	0	A	A	A == A	Move i++, j++
8	1	D	B	D ≠ B	Set j = LPS[0] = 0
8	0	D	A	D ≠ A	Move i++
9	0	A	A	A == A	Move i++, j++
10	1	A	B	A ≠ B	Set j = LPS[0] = 0
10	0	A	A	A == A	Move i++, j++
11	1	B	B	B == B	Move i++, j++
12	2	A	A	A == A	Move i++, j++
j == m					Pattern found at **i - j = 10**
					Set j = LPS[2] = 1
13	1	A	B	A ≠ B	Set j = LPS[0] = 0
13	0	A	A	A == A	Move i++, j++
14	1	B	B	B == B	Move i++, j++
15	2	A	A	A == A	Move i++, j++
j == m					Pattern found at **i - j = 12**
					Set j = LPS[2] = 1
16					End of text reached

Step 3: Count Matches

Pattern "ABA" appears at indices 3, 10, and 12.

4. ***Calculate the time complexity of KMP algorithm.***

Solution:

The time complexity of the Knuth-Morris-Pratt (KMP) algorithm consists of two main parts:

1. Preprocessing the Pattern (LPS Array Computation)

- The Longest Prefix Suffix (LPS) array is computed in O(m) time.
- This is because each character of the pattern is processed at most once.

2. Searching for the Pattern in the Text

- The pattern is matched against the text in O(n) time.
- Even though we may backtrack in the pattern using the LPS array, we never revisit characters in the text.
- Hence, every character in the text is processed at most once.

Overall Time Complexity

Since both steps run in linear time, the total complexity is:

$$O(m)+O(n)=O(n+m)$$

5. ***One million citizens are living in the city. The length of their name is maximum 20 letters. There is a phonebook having phone numbers of all the citizens. Your task is to find the name of a citizen in the phonebook. If it exists, print YES, otherwise NO. Which data structure is best for storing all the names and for searching the names and why?***

Solution:

A Trie (prefix tree) is the best data structure for storing and searching names in this scenario.

Why Trie?

1. ***Efficient Searching:***
   - Searching for a name in a Trie takes O(L) time, where L is the length of the name (at most 20 in this case).
   - This is significantly faster than a linear search O(N) or even a balanced BST O(log N).
2. ***Efficient Storage:***
   - A Trie stores names in a compressed hierarchical structure, reducing redundancy in prefix storage.
   - Unlike a hash table, it does not require extra space for handling collisions.
3. ***Avoids Hash Collisions:***
   - Using a Hash Table (O(1) search on average) may seem faster, but hash collisions and rehashing increase worst-case complexity to O(L).
   - A Trie guarantees worst-case O(L) time consistently, making it more reliable.
4. ***Prefix-Based Searches:***
   - If partial name lookups (e.g., auto-complete, suggestions) are required, a Trie is inherently better.
   - Hash tables and BSTs are not efficient for prefix-based searches.

Dynamic Programming

1. ***What does dynamic programming mean? When we can apply dynamic programming algorithm to solve a problem.***

Solution:

Dynamic Programming

Dynamic Programming (DP) is an optimization technique used to solve problems by breaking them down into overlapping subproblems and solving each subproblem only once, storing its result to avoid redundant computations.

Conditions for Applying Dynamic Programming

A problem can be solved using DP if it satisfies the following properties:

1. ***Optimal Substructure:***

- A problem exhibits optimal substructure if its optimal solution can be constructed from the optimal solutions of its subproblems.
- ***Example:*** Shortest path problems (e.g., Bellman-Ford Algorithm).

2. ***Overlapping Subproblems:***

   - A problem has overlapping subproblems if the same subproblems are solved multiple times.
   - ***Example:*** Fibonacci sequence, where `F(n) = F(n-1) + F(n-2)`, leading to repeated calculations in a naïve recursive approach.

If a problem satisfies both properties, DP can be applied for an efficient solution.

2. ***During calculation of Fibonacci series recursively the same function is called several times. How can you reduce the calling time?*****

Solution:
To reduce the calling time in the recursive calculation of the Fibonacci series, Memoization and Tabulation techniques of Dynamic Programming can be used.

1. Memoization (Top-Down Approach)

- Store results of previously computed subproblems in a table (array or hash map).
- When a function call is repeated, return the stored value instead of recomputing it.
- Time Complexity: O(n), as each Fibonacci number is computed only once.
- Space Complexity: O(n), due to recursion stack and storage.

2. Tabulation (Bottom-Up Approach)

- Solve smaller subproblems first and store results in an array.
- Use previously computed values to build up the final result iteratively.
- Time Complexity: O(n), as each Fibonacci number is computed once.
- Space Complexity: O(n) (or O(1) with space optimization by storing only the last two computed values).

3. Space Optimization

- Instead of storing all Fibonacci numbers, keep only the last two values.
- Reduces space complexity to O(1) while maintaining O(n) time complexity.

These techniques eliminate redundant recursive calls, improving efficiency.

3. ***Prepare three datasets for the best case, average case, and the worst case of Meet in the middle algorithm. Explain why the datasets are belonging to the best case, average case, and the worst case.***

Solution:
Dataset 1: Best Case

- ***Input Array***: [5,8,12,15,18,20,25,30]
- ***Target Sum***: 5

Explanation:

- The target sum equals the first element (or a singleton subset) in the first half of the array.
- When the algorithm generates the subset sums for the first half, it immediately finds 5 without needing to combine many elements.
- This leads to early termination and minimal exploration of the exponential search space.

Dataset 2: Average Case

- ***Input Array****:* [3,34,4,12,5,2,7,8]
- ***Target Sum****:* 9

Explanation:

- The target is not directly equal to any single element; it requires combining two or more elements (for instance, 3+4+2=9).
- Both halves contribute nontrivial subset sums.
- On average, the algorithm must search a moderate portion of the 2n/2 possibilities from each half before a valid pair of subset sums is found, representing average-case behavior.

Dataset 3: Worst Case

- ***Input Array****:* [10,10,10,10,10,10,10,10]
- ***Target Sum****:* 35

Explanation:

- Every element is identical, so all subset sums in each half are multiples of 10.
- Since 35 is not a multiple of 10, no combination of numbers can exactly sum to 35.
- The algorithm cannot prune or short-circuit the search because every generated subset sum is "ambiguous" with respect to the target; it must enumerate nearly all 2n/2 possible sums from each half and then check all possible pairings before concluding that no solution exists.
- This exhaustive exploration represents the worst-case scenario.

4. ***How can you find out the time complexity of dynamic programming problem?***

Solution:
The time complexity of dynamic programming algorithms by considering:

- **S**: The number of distinct DP states.
- **T**: The average transition work per state.

Thus, the overall time complexity is typically O(S×T), and the space complexity is usually O(S) (with possible optimizations).

Fibonacci Example

- **States**: N (from dp[1] to dp[N])
- **Transition**: Each state depends on 2 states (constant work)
- **Complexity**: O(N)×O(1)=O(N)

5. ***What is backtracking?***

Solution:

Backtracking is a recursive algorithmic technique for solving problems by trying all possible solutions and discarding those that fail constraints. It explores a solution space **depth-first** and backtracks when an invalid state is reached.

Steps:

1. Choose a possible option.
2. Check if it satisfies constraints.
3. If valid, proceed recursively; otherwise, backtrack.
4. Repeat until a solution is found or all options are exhausted.

Time Complexity: Worst-case **O(bd)**, where b is branching factor and d is depth.

6. ***How backtracking can be used to solve the N-queens problem?***

Solution:

Backtracking solves the **N-Queens problem** by placing queens **column by column** and checking for conflicts. If a valid placement is found, it proceeds to the next column; otherwise, it backtracks.

Algorithm:

1. Place a queen in an **empty column**.
2. Check if it conflicts with any previously placed queens (same row or diagonal).
3. If valid, move to the next column.
4. If no valid position exists, **backtrack** and try a different row.
5. Repeat until all queens are placed or all options are exhausted.

Time Complexity: O(N!) due to the factorial growth of possibilities.

7. ***Assume that you 4×4 chessboard. You have to place 4 queens on that board in such a way that they do not attack each other. How can you solve the problem?***

Solution:

The **4×4 N-Queens problem** can be solved using **backtracking** by placing queens **column by column** and backtracking when conflicts arise.

Steps:

1. **Start with the first column.**
2. **Try placing a queen in each row** of the column.
3. **Check for conflicts** with previously placed queens (same row, same diagonal).
4. If valid, **place the queen and move to the next column**.
5. If no valid position exists in a column, **backtrack** to the previous column and try a different row.
6. Repeat until all 4 queens are placed.

Valid Solutions for 4×4 Board:

```
.  Q  .  .        .  .  Q  .
.  .  .  Q        Q  .  .  .
Q  .  .  .        .  .  .  Q
.  .  Q  .        .  Q  .  .
```

8. ***Given a set of non-negative integers {3, 34, 4, 12, 5, 8} and a value sum (9), determine if there is a subset of the given set with sum equal to the given sum. Show all the steps.***

Solution:

The problem is a Subset Sum Problem, solved using Dynamic Programming and the steps are shown with a state table.

Step 1: Define the DP State

Let dp[i][j] be true if there is a subset from the first i elements that sums to j, otherwise false.

Step 2: Recurrence Relation

- ***Base Case:*** $dp[0][0]=true$
- If an element is not included: $dp[i][j]=dp[i-1][j]$
- If an element is included: $dp[i][j]=dp[i-1][j] \; || \; dp[i-1][j-arr[i-1]]$

Step 3: DP Table Initialization

Given set: {3, 34, 4, 12, 5, 8}
Sorted set: {3, 4, 5, 8, 12, 34}

Target sum: 9
Number of elements: 6

i \ j	0	1	2	3	4	5	6	7	8	9
0	T	F	F	F	F	F	F	F	F	F
3	T	F	F	T	F	F	F	F	F	F
4	T	F	F	T	T	F	F	T	F	F
5	T	F	F	T	T	T	F	T	T	T
8	T	F	F	T	T	T	F	T	T	T
12	T	F	F	T	T	T	F	T	T	T
34	T	F	F	T	T	T	F	T	T	T

From the DP table we can conclude there is a subset of the given set with sum equal to the given sum.

9. ***Discuss two efficient algorithms for finding subset sum.***

Solution:

Two Efficient Algorithms for Finding Subset Sum

1. Meet-in-the-Middle Approach

- ***Use Case:*** n is large (typically n <= 40)
- ***Idea:*** Split the set into two halves and compute all subset sums for each half. Then using the first half combine sum with the second half and find subset sums equal to some given sum x.
- ***Steps:***
    1. Divide the set into two halves.
    2. Generate all possible subset sums for both halves.
    3. Sort one half and use binary search or two-pointer technique to check if any sum from one half pairs with a sum from the other half to reach the target.

2. Dynamic Programming

- ***Use Case:*** n is small to moderate and sum is not too large.
- ***Idea:*** Use a boolean DP table dp[i][j] where dp[i][j] is true if a subset of the first i elements sums to j.
- ***Steps:***
    1. ***Base Case:*** dp[0][0]=true (empty subset sums to 0).
    2. ***Transition:***
        - If an element is not included: dp[i][j]=dp[i−1][j].
        - If included: dp[i][j]=dp[i−1][j] ∨ dp[i−1][j−arr[i−1]] (if j≥arr[i−1]).
    3. Fill the DP table up to dp[n][sum].

10. ***Determine the time complexity of the two efficient algorithms?***

Solution:

1. Meet-in-the-Middle

- The problem is split into two halves, each of size n/2.
- We compute all subset sums for both halves, taking $O(2^{n/2})$.
- Sorting takes $O(2^{n/2}\log 2^{n/2})=O(2^{n/2}n)$.
- Searching via binary search takes $O(n)$.

Recurrence Relation: $T(n)=2T(n/2)+O(2^{n/2}n)$

Applying Master Theorem:

- $a=2, b=2, f(n)=O(2^{n/2}n)$.
- Compare $f(n)$ with $O(n^{\log_2 2})=O(n)$.
- Since $f(n)=O(2^{n/2}n)$ grows faster than O(n).

So, time complexity is $O(2^{n/2})$

2. Dynamic Programming

- The problem size reduces by 1 in each step.
- Each subproblem requires checking previous states, leading to: $T(n)=T(n-1)+O(s)$

Using recursion expansion: $T(n)=T(n-2)+O(s)+O(s)$

Expanding fully for n terms we have: $T(n)=O(n\times s)$

So, time complexity: $O(n\times s)$.

Geometry

1. ***What is Convex Hull? Explain the Graham's scan algorithm for finding convex hull with suitable example.*****

Solution:

Convex hull

The convex hull of a set of points is the smallest convex polygon that encloses all the given points. Mathematically, a convex hull is the smallest convex set containing all given points.

Graham's Scan Algorithm for Convex Hull

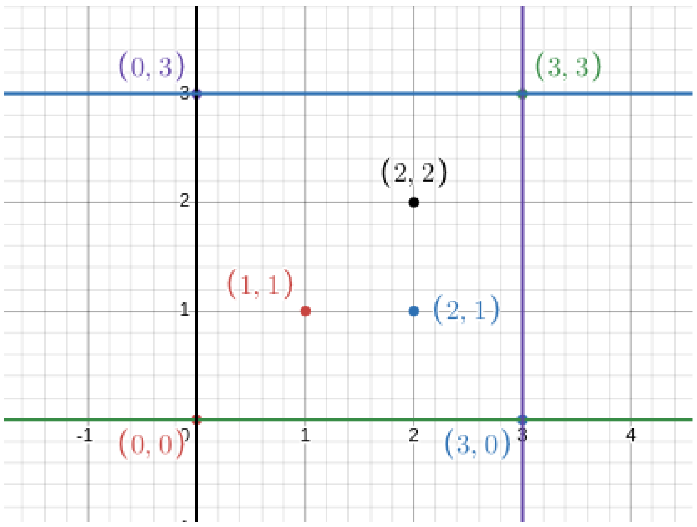
Graham's scan is an efficient algorithm to find the convex hull of a set of n points in $O(nlogn)$ time.

Steps of the Algorithm

1. ***Find the Point with the Lowest Y-Coordinate***
   - If multiple points have the same y-coordinate, choose the leftmost one.
   - Let this point be the starting point P_0 .
2. ***Sort the Points by Polar Angle with Respect to*** P_0
   - Compute the polar angle (counterclockwise) of each point relative to P_0 .
   - Sort the points in increasing order of this angle.
3. ***Scan and Construct the Convex Hull***
   - Initialize an empty stack.
   - Push the first three sorted points onto the stack.
   - For each remaining point P_i :
     - While the turn formed by the top two points on the stack and P_i is ***not counterclockwise***, pop the stack.
     - Push P_i onto the stack.
   - The points in the stack form the **convex hull**.

Example:

Given Points: (0, 3), (2, 2), (1, 1), (2, 1), (3, 0), (0, 0), (3, 3)



Step 1: Find the Lowest Point

The lowest y-coordinate is **(0,0)**, so it becomes P_0 .

Step 2: Sort by Polar Angle

Sorting the remaining points based on their polar angle from P_0 , we get:

(3, 0), (2, 1), (1, 1), (2, 2), (3, 3), (0, 3)

Step 3: Construct the Convex Hull

- Start with **(0,0), (3,0), (2,1)**.
- Process each point, ensuring left turns.
- The final convex hull is **(0, 0), (3, 0), (3, 3), (0, 3)**.

2. *Show the time complexity of Jarvis March algorithm.*

Solution:

Jarvis March algorithm is used to find the convex hull of a set of n points. It works by repeatedly selecting the next hull point by checking all other points.

<u>Time Complexity Analysis</u>

- The algorithm runs in $O(n \times h)$ time, where:
  - n = Total number of points.
  - h = Number of points in the convex hull.
- ***Explanation:***
  - For each hull point, the algorithm scans all n points to find the next point on the hull.
  - This process is repeated h times, once for each hull point.
  - Each iteration takes O(n) time, leading to O(n×h) complexity.

Best, Worst, and Average Case Complexity

Case	*Condition*	*Complexity*
Best Case	All points are collinear, h = 2.	$O(n)$
Worst Case	All points outside hull, h = n	O(n^2)
Average Case	Few points on hull, h < n	$O(n \times h)$

3. ***How can we check whether or not two line segments are intersecting or not? Explain with example.***

Solution:

Orientation Method (Using Cross Product)

The orientation of three points **(p,q,r)** is found using the determinant:

$$val=(q_y-p_y)\times(r_x-q_x)-(q_x-p_x)\times(r_y-q_y)$$

Value	*Orientation*
$val>0$	Counterclockwise
$val=0$	Colinear
$val<0$	Clockwise

For segments AB and CD, compute:

1. orient(A,B,C)
2. orient(A,B,D)
3. orient(C,D,A)
4. orient(C,D,B)

Intersection Conditions

- ***General Case (Proper Intersection):***
  AB and CD intersect if

  $orient(A,B,C)=orient(A,B,D)$ and $orient(C,D,A)=orient(C,D,B)$

- ***Special Case (Collinear Segments Overlapping):***
  If $orient=0$, check if the segments overlap using bounding box conditions:

  $max(A_x,B_x)\geq min(C_x,D_x)$ and $max(A_y,B_y)\geq min(C_y,D_y)$

Example

A(1,1), B(4,4) and C(1,4), D(4,1)

1. orient(A,B,C)= Clockwise.
2. orient(A,B,D)= Counterclockwise.
3. orient(C,D,A)= Counterclockwise.
4. orient(C,D,B)= Clockwise.

Since orientations are opposite, segments intersect.

4. ***Given a set of line segments. Develop an algorithm for determining whether any two or three line segments intersect at a single point. In addition to this, assume that input segment could be vertical.***

Solution:

Algorithm for Detecting Intersections Among Line Segments

Given a set of line segments, the goal is to determine whether any **two** or **three** segments intersect at a single point, considering vertical segments as well.

Step 1: Defining Intersection

- Two segments (p_1, q_1) and (p_2, q_2) intersect if they share a common point.
- Three segments intersect at a single point if they all meet at the same (x, y) coordinate.

Step 2: Approach - Sweep Line Algorithm

The Sweep Line Algorithm efficiently finds intersections in $O((n+k)\log n)$, where n is the number of segments and k is the number of intersections.

Algorithm

1. ***Sort Events*:**
    - Create events for all segment endpoints:
        - Left endpoint → Insert segment into active set.
        - Right endpoint → Remove segment from active set.
    - If a vertical segment exists, consider it in a special case.
    - Sort events by increasing x-coordinate. If x-coordinates are the same, process upper endpoint first.
2. ***Process Events with a Balanced BST (Active Set):***
    - Maintain active segments using a Balanced BST ordered by y-coordinate.
    - When inserting a segment, check for intersections with its immediate neighbors in the BST.
3. ***Handling Intersections:***
    - If an intersection is detected, store the intersection point and check if a third segment also passes through this point.
    - If three segments intersect at the same point, return true.
4. ***Output Results:***
    - If an intersection of two or three segments is found, return the intersection point.

Special Case: Vertical Segments

- A vertical segment x=c intersects a non-vertical segment (p_1, q_1) if the y-coordinate of the intersection is within the range of $p_{1.}y$ and $q_{1.}y$.
- Maintain a separate set of vertical segments to check for these cases efficiently.

Miscellaneous

1. ***Is STL vector dynamic? Justify your answer.***

Solution:

Yes, STL ***vector*** is dynamic in C++. Because__

- ***Automatic Resizing:*** vector dynamically allocates memory as elements are added. If the current capacity is exceeded, a new larger memory block is allocated, and existing elements are copied.
- ***Variable Capacity:*** Unlike arrays (fixed-size), vector grows automatically. The capacity typically increases exponentially (2× growth factor in many implementations).
- ***Dynamic Operations Supported:*** Supports insertion, deletion, resizing without manual memory handling. push_back() dynamically increases size.

2. ***How can you implement priority queue efficiently.***

Solution:

A priority queue can be efficiently implemented using a ***binary heap*** (min-heap or max-heap).

1. ***Insertion****:* O(logn)
2. ***Extracting min/max****:* O(logn)
3. ***Peek (get min/max)****:* O(1).

Implementation of priority queue

- ***Using STL***
  - ***Max-Heap:*** (default): priority_queue<int>
  - ***Min-Heap****:* priority_queue<int, vector<int>, greater<int>>

- ***Using Binary Heap***
  - Store elements in an ***array***.
  - Use ***heapify-up*** for insertion and ***heapify-down*** for deletion.

3. ***What is Greedy algorithm. What kind of problems can be solved using greedy algorithm?***

Solution:

A Greedy Algorithm is an approach where, at each step, the locally optimal choice is made with the hope that this leads to a globally optimal solution.

Problems solved using greedy algorithm

1. ***Optimization Problems:***

    1. Fractional Knapsack Problems.
    2. Huffman Coding.
    3. Dijkstra Algorithm.
2. ***Scheduling Problems:***
    1. Activity Selection Problems.
    2. Job Sequencing Problems.
3. ***Graph Problems:***
    1. Prim’s Algorithm.
    2. Kruskal’s Algorithm.

Graph

- **DSU**

1. There is an image of N×M pixels. Originally all are white, but then a few black pixels are drawn. You want to determine the size of each white connected component in the final image. You have three algorithms DFS, BFS, DSU. Which one is best algorithm for solving the problem and why? **

- **A* Search**

1. How to calculate the heuristic value used in A* search algorithm.
2. Write down the A* search algorithm.

- **Tree**

1. What is the purpose of merge sort tree?
2. Write the Kruskal’s algorithm for Minimum Spanning Tree.
3. What are Huffman trees? Explain how to construct Huffman Trees. Construct Huffman Tree for the following dataset where alphabets represent the data and numbers represent the corresponding frequency:

data	a	b	c	d	e	f
frequency	5	9	12	13	16	45

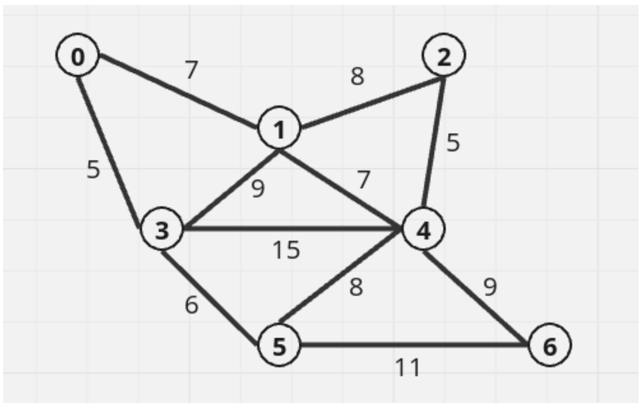
4. Why Huffman coding algorithm is used?
5. Assume that the following table states the character and its frequency in a text document, respectively:

character	a	b	c	d	e	f

frequency	16	12	13	45	9	5

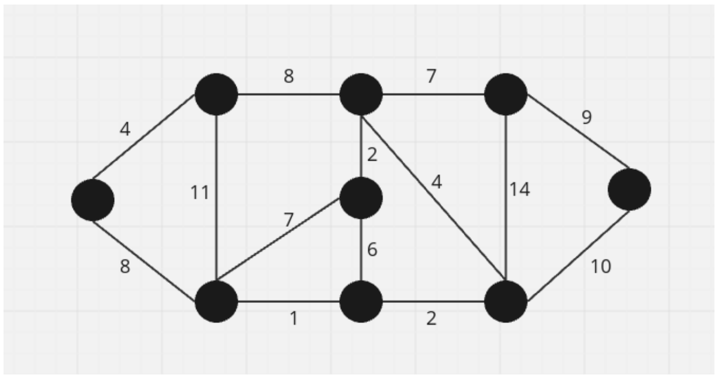
Can you generate the Huffman Tree(with step by step explanation) for this graph symbolic data?

6. Compute the Minimum Spanning Tree and its cost for the following graph using Krushkal's algorithm. Indicate each step clearly.



◆ **<u>Dijsktra Algorithm</u>**

1. How do you optimize the dijsktra algorithm?
2. Find a shortest path from 0 to 4 in the following graph. Show all steps. Priority queue and set can be used for implementing the Dijkstra's shortest path algorithm. Which one is better and why?

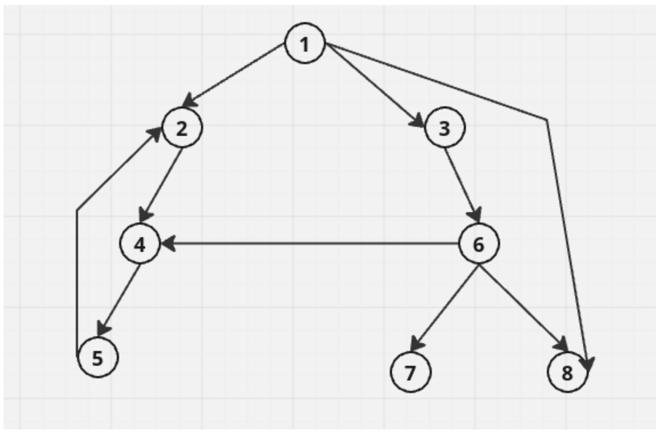


3. Why does Dijkstra's algorithm not solve the single source shortest-path problem on a weighted directed graph having negative weight edges? Explain with an example.

◆ **<u>Graph Related</u>**

1. How can you create graph using STL?**
2. How can you represent a graph?
3. How to check if a graph is connected or not?
4. Given an undirected graph, how to check if there is a cycle in the graph?

5. An unweighted and undirected graph and two nodes in a graph are given. How can you find the minimum distance between the nodes? Is it possible to find the minimum distance using DFS algorithm?
6. Write the algorithm for DFS and analyze its complexity.
7. What is a Hamiltonian Cycle? Explain how to find Hamiltonian path and cycle using backtracking algorithm.
8. What is an articulation edge? how can you find an articulation edge?
9. Define Implicit graphs with examples.**
10. Given an nxn grid whose each square is either black (1) or white (0), develop an algorithm to calculate the number of subgrids whose all corners are black.
11. Write down the BFS algorithm.
12. A graph is Bipartite if its nodes can be colored using two colors so that there are no adjacent nodes with the same color. How can you check if a graph is bipartite or not?
13. A graph is given. There are some cycles in the graph. Write down the algorithm to count the number of cycles.
14. How to detect a cross edge in a directed graph. Cross edge can be defined as an edge which connects two nodes such that they do not have any ancestor and a descendant relationship between them. Edge from node 5 to 4 is a cross edge.



Data Structure

- **Prefix Sum**

1. Give an example of cumulative sum.
2. Which method is faster for finding cumulative sum and why?

- **Difference Array**

1. You have an array of length L, each element initially has the color 0. You have to repaint the subarray [l, r] with the color c for each query [l, r, c]. At the end you want to find the final color of each cell. You can assume that you know all the queries in advance, i.e. the task is offline. Write down a solution for this problem.
2. There are two operations find() and union() in disjoint set union (DSU) data structure. How can you optimize these two operations in terms of time complexity?

- **Binary Indexed Tree (BIT)**

1. Why do you need to start an array from index 1 in Binary Indexed Tree (BIT)? Explain if it is started from 0 index.
2. How many nodes do you need to traverse for each update in a Binary Indexed Tree (BIT)?**
3. Why do you need to find parent in a BIT? How can you find parent in a Binary Indexed Tree (BIT)?**
4. How can you traverse the nodes for updating the nodes in a Binary Indexed Tree (BIT)?
5. Some integers numbers are given. You need to solve various range-based queries such as finding the sum of elements within a range and update a number 500 times. How can you store the numbers in memory so that you can perform the operations efficiently?
6. An array A = {6, 8, 5, 25, 16, 18, 36, 24, 9, 11, 7, 28, 37, 22, 40, 13, 35, 14, 20, 2} is given. Draw a Binary Indexed Tree(BIT) and insert the numbers in the tree. Add 5 to the 7th node and update all other necessary nodes.

- **Binary Search Tree (BST)**

1. How can you update a range of nodes in a segment tree with the same time complexity as taken for updating a single node?
2. Define Binary Search Tree with example. Write down the properties of Binary Search Tree.

- **Segment tree**

1. How can you build a segment tree for an input array {1, 3, 5, 7, 9, 11, 13}?

Solution:

- **Square Root Decomposition**

1. ***Why do you need to compress a path in square root decomposition algorithm? How can you compress a path in square root decomposition algorithm?***

Solution:

Path compression in square root decomposition is used to speed up operations by reducing the number of steps needed to traverse or update data.

Specifically, in range query problems like sum, minimum, or GCD over a segment of an array, path compression helps in optimizing updates and queries.

Without compression, each query may take O(√N) time, but with compression, it can be improved to O(1) in some cases.

Path compression is mainly used in two ways:

Flattening Updates within Blocks

- Normally, when updating an element, we might traverse the entire block to recalculate values.
- ***Compression trick:*** Store aggregated values at the block level so that updates affect only a single block rather than the entire array.

Lazy Propagation in Updates

- Instead of updating each element in a block, we store a lazy update marker.
- When a query is performed, the block is fully updated only when accessed (lazy evaluation).
- This reduces redundant updates and improves efficiency.