1. Cases of Master Theorem
   = The **Master Theorem** provides a way to analyze the time complexity of divide-and-conquer algorithms. It applies to recurrence relations of the form:

T(n)=aT(nb)+f(n)T(n) = aT\left(\frac{n}{b}\right) + f(n)T(n)=aT(bn)+f(n)

Where:

- a≥1a \geq 1a≥1: number of subproblems
- b>1b > 1b>1: factor by which the problem size is divided
- f(n)f(n)f(n): cost of dividing and combining the subproblems

Cases:

Case 1:
If f(n)=O(nlog⁡ba−ϵ)f(n) = O(n^{\log_b a - \epsilon})f(n)=O(nlogba−ϵ) for some ϵ>0\epsilon > 0ϵ>0,
Then:

T(n)=Θ(nlog⁡ba)T(n) = \Theta(n^{\log_b a})T(n)=Θ(nlogba)

Case 2:
If f(n)=Θ(nlog⁡ba)f(n) = \Theta(n^{\log_b a})f(n)=Θ(nlogba),
Then:

T(n)=Θ(nlog⁡balog⁡n)T(n) = \Theta(n^{\log_b a} \log n)T(n)=Θ(nlogbalogn)

Case 3:
If f(n)=Ω(nlog⁡ba+ϵ)f(n) = \Omega(n^{\log_b a + \epsilon})f(n)=Ω(nlogba+ϵ) for some ϵ>0\epsilon > 0ϵ>0, and if the regularity condition af(n/b)≤cf(n)a f(n/b) \leq c f(n)af(n/b)≤cf(n) for some c<1c < 1c<1 and large enough nnn,
Then:

T(n)=Θ(f(n))T(n) = \Theta(f(n))T(n)=Θ(f(n))

---

2. Recurrence Relation with an Example

A **recurrence relation** expresses the running time of a recursive algorithm in terms of its input size and the running time of smaller inputs.

Example: Merge Sort

Merge Sort: T(n) = 2T(n/2) + n

```
// Merge Sort example

void mergeSort(int arr[], int l, int r) {

   if (l < r) {

      int m = l + (r - l) / 2;
```

```
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r); // Assume merge function implemented
    }
}
```

---

3. Define Complexity of an Algorithm with an Example

Algorithm complexity refers to the **amount of resources** (time and space) required to execute the algorithm.

Time Complexity: Number of basic operations as a function of input size nnn.

Space Complexity: Amount of memory used as a function of nnn.

Example:
Linear search in an array of size nnn:

```
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1
```

- **Time Complexity:** O(n)O(n)O(n) (worst-case: target is at the end or not present)
- **Space Complexity:** O(1)O(1)O(1) (no extra space used)

---

4. Three Characteristics of an Algorithm

1. **Finiteness:**
   The algorithm must always terminate after a finite number of steps.
2. **Definiteness:**
   Every step of the algorithm must be precisely defined.
3. **Input & Output:**
   An algorithm should have **zero or more inputs** and produce **at least one output**.

---

5. Define Space Analysis with an Example

Space Analysis refers to the process of determining how much **memory** an algorithm uses during its execution.

It includes:

- **Input space** (memory to store inputs),
- **Auxiliary space** (extra memory used during computation),
- **Recursion stack** (for recursive algorithms).

Example:
Recursive factorial function:

python

CopyEdit

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Space Complexity:

- Each recursive call adds a new frame to the call stack.
- For input nnn, there are nnn recursive calls.
- So, **Space Complexity = O(n)**

---

6. How Do We Measure the Correctness of an Algorithm?

We measure correctness by verifying **two properties**:

1. **Partial Correctness:**
   If the algorithm terminates, it produces the correct output.
2. **Termination:**
   The algorithm finishes in a finite number of steps for all valid inputs.

Techniques:

- **Formal proofs** (e.g., induction)
- **Assertions and invariants**
- **Testing with various input cases** (boundary values, invalid inputs, etc.)

Differences between Time Complexity and Space Complexity

Feature	Time Complexity	Space Complexity
Definition	Time taken by algorithm	Memory used

Feature	Time Complexity	Space Complexity
Unit	Steps or operations	Bytes or memory units
Optimized for	Faster execution	Efficient memory usage
Example (Merge Sort)	O(n log n)	O(n)

---

9. Analyze $T(n) = 4T(n/2) + n^2$ using Master Theorem

Here,

- a = 4
- b = 2
- $f(n) = n^2$
- $\log_b a = \log_2 4 = 2$

So $f(n) = \Theta(n^2) = n^{\log_b a}$
→ This is **Case 2** of Master Theorem
→ **$T(n) = \Theta(n^2 \log n)$**

---

10. Steps to sort using Merge Sort: 5, 3, 8, 1, 4, 6, 2, 7

Steps:

1. Divide list into halves:
   [5,3,8,1] and [4,6,2,7]
2. Divide each half:
   [5,3] [8,1] [4,6] [2,7]
3. Keep dividing:
   [5][3][8][1][4][6][2][7]
4. Merge sorted pairs:
   [3,5][1,8][4,6][2,7]
5. Merge again:
   [1,3,5,8] and [2,4,6,7]
6. Final merge:
   [1,2,3,4,5,6,7,8]

---

11. Explain Algorithm

An algorithm is a sequence of steps designed to perform a specific task. It takes an input, processes it through a set of instructions, and produces an output. Here's an example of an algorithm to find the sum of two numbers in C:

Algorithm:

- **Start: Begin the program.**
- **Input: Get two numbers, a and b, from the user.**
- **Process: Calculate the sum of a and b, storing the result in a variable called sum.**
- **Output: Display the value of sum.**
- **End: Terminate the program.**

---

12. Analyze recurrences in First Case of Master's Theorem

Let:
T(n) = aT(n/b) + f(n)
If **f(n) = O(n^c)** where **c < log_b a**, then:

- Recursive part dominates
- Solution: **T(n) = Θ(n^log_b a)**

Example:
T(n) = 2T(n/2) + $O(n^{0.5})$

- a = 2, b = 2 ⇒ log_b a = 1
- c = 0.5 < 1 ⇒ Case 1
- Result: **T(n) = Θ(n)**

---

13. Compare Dynamic Programming vs Divide and Conquer

Aspect	**Dynamic Programming**	**Divide and Conquer**
Overlapping Subproblems	Yes	No
Subproblem Reuse	Reuses results (memoization)	Solves each subproblem independently
Example	Fibonacci, Knapsack	Merge Sort, Quick Sort

Aspect	Dynamic Programming	Divide and Conquer
Optimization	Bottom-up/Top-down with memo	Recursively split and combine

14.Discuss Dynamic Programming Approach

Dynamic Programming (DP) is a problem-solving technique used to solve complex problems by breaking them down into simpler overlapping subproblems and solving each only once.

Key Features:

1. **Overlapping Subproblems:**
   Same subproblems are solved multiple times (e.g., Fibonacci).
2. **Optimal Substructure:**
   The solution to a problem depends on solutions to its subproblems.
3. **Memoization / Tabulation:**
   - **Memoization uses recursion + caching (top-down).**
   - **Tabulation uses iteration (bottom-up).**

Example: Fibonacci Using DP (C Code)

```
int fib(int n) {
  int dp[n+2];
  dp[0] = 0; dp[1] = 1;

  for (int i = 2; i <= n; i++)
    dp[i] = dp[i-1] + dp[i-2];

  return dp[n];
}
```

---

15. Discuss the criteria on which the measures of complexity depend

Complexity measures (Time and Space) depend on:

1. **Input Size (n):** Larger input takes more time/memory.
2. **Input Type/Structure:** Best, average, and worst-case depend on input arrangement.
3. **Operations Performed:** Loops, recursion, nested calls affect runtime.
4. **Algorithm Design:** Efficient logic and data structures reduce complexity.

---

16. Explain the Traveling Salesman Problem (TSP)

TSP is a classic optimization problem where:

A salesman must visit each city exactly once and return to the starting city with the minimum total cost (distance/time).

- It is **NP-Hard**, meaning no polynomial-time solution exists for large inputs.
- Used in logistics, route planning, etc.

C Example: Brute-force (small inputs only)

```
#include <stdio.h>

#include <limits.h>


#define V 4

int tsp(int graph[V][V], int path[], int visited[], int pos, int count, int cost, int start) {

   if (count == V && graph[pos][start])

      return cost + graph[pos][start];


   int ans = INT_MAX;

   for (int i = 0; i < V; i++) {

      if (!visited[i] && graph[pos][i]) {

         visited[i] = 1;

         ans = (ans < tsp(graph, path, visited, i, count + 1, cost + graph[pos][i], start)) ? ans : tsp(graph, path, visited, i, count + 1, cost + graph[pos][i], start);

         visited[i] = 0;

      }

   }

   return ans;

}
```

---

17. Basic Characteristics of Dynamic Programming

1. **Overlapping Subproblems:** Same subproblems occur repeatedly.
2. **Optimal Substructure:** Optimal solution can be built from optimal sub-solutions.
3. **Memoization or Tabulation:** Store results of subproblems to avoid recomputation.

C Example: Fibonacci (Memoization)

```
int fibDP(int n, int memo[]) {
   if (n <= 1) return n;
   if (memo[n] != -1) return memo[n];
   return memo[n] = fibDP(n - 1, memo) + fibDP(n - 2, memo);
}
```

---

18. Explain the Basic Concept of Divide and Conquer

Divide and Conquer works by:

1. **Divide:** Split the problem into smaller subproblems.
2. **Conquer:** Solve the subproblems recursively.
3. **Combine:** Merge the sub-solutions into the final result.

Used in sorting, searching, matrix multiplication, etc.

C Example: Merge Sort

```
void mergeSort(int arr[], int l, int r) {
   if (l < r) {
      int m = l + (r - l)/2;
      mergeSort(arr, l, m);
      mergeSort(arr, m+1, r);
      merge(arr, l, m, r); // Merging step
   }
}
```

19.Calculate the min and max term of the following array using Divide and conquer approach {9,6,4,7,10,14,8,11}

= . Divide the array into two halves.

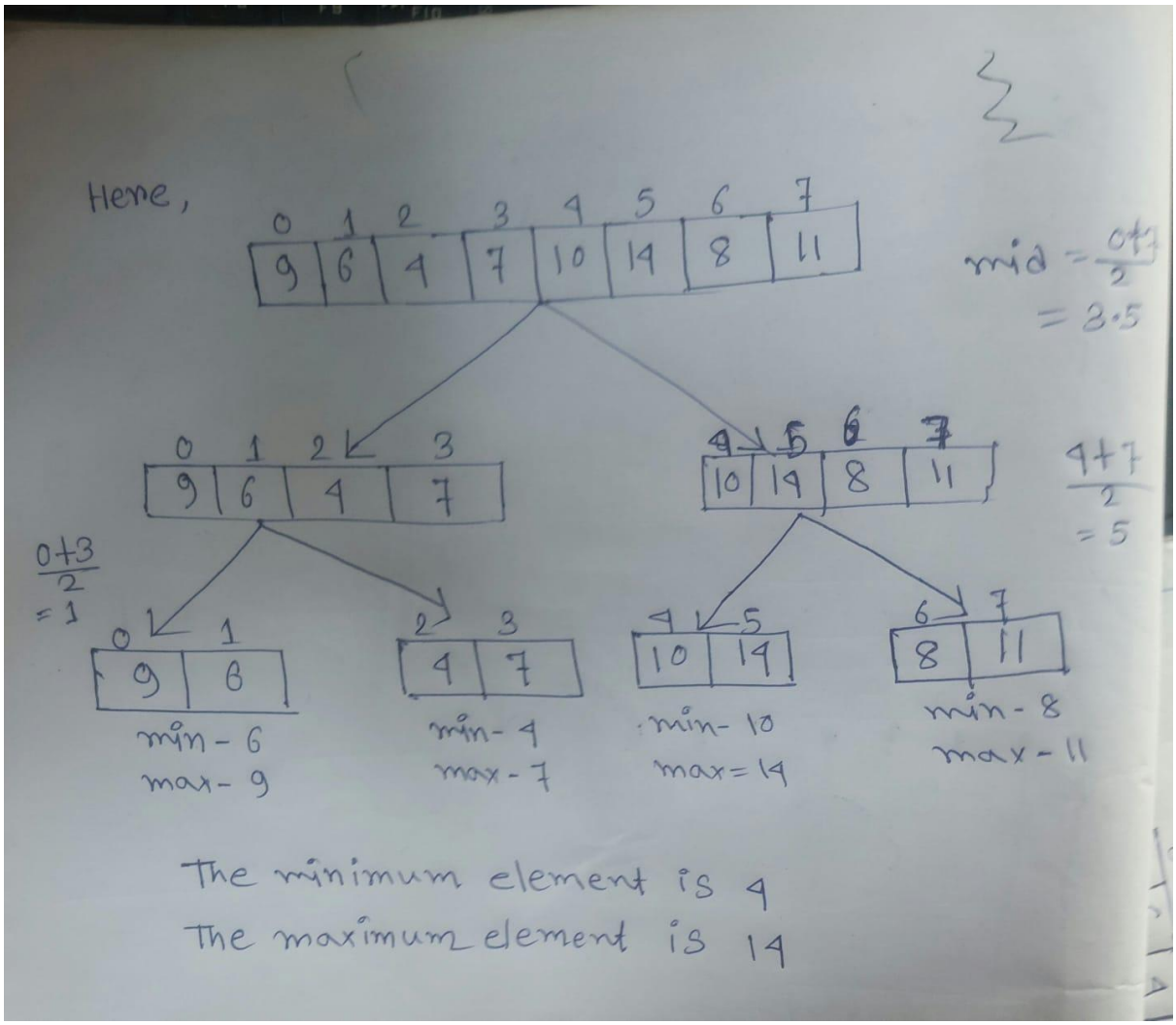
- The middle index is

0+7/2=3.5the fraction with numerator 0 plus 7 and denominator 2 end-fraction equals 3.5

0+7/2=3.5

Conditions for applying divide & conquer

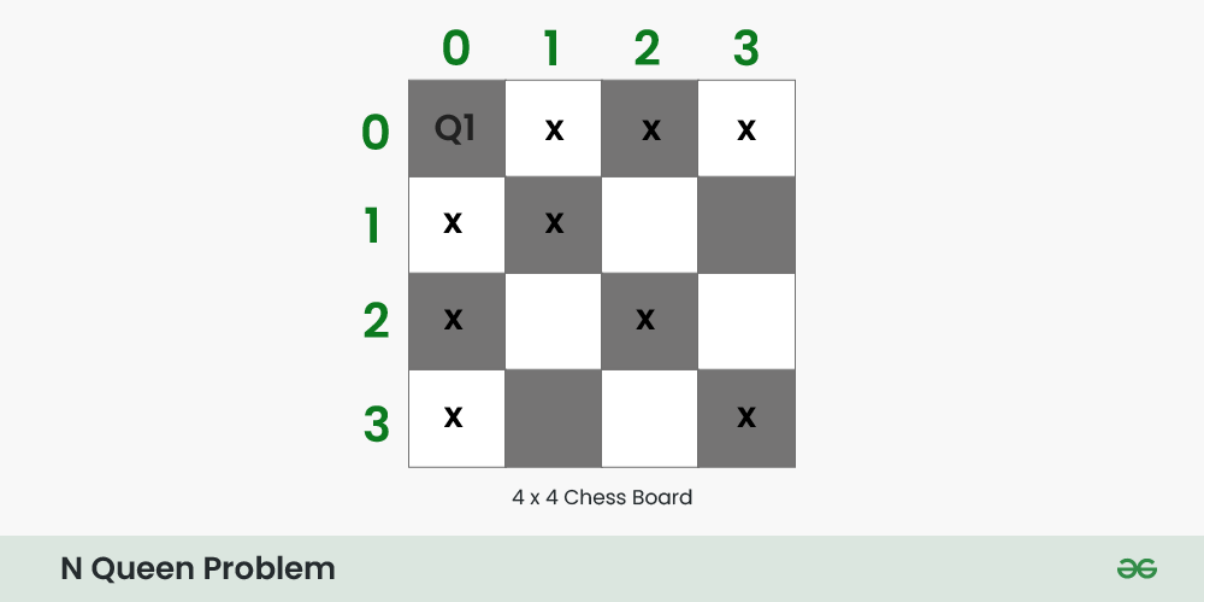
1. P must be very large.
2. There must be a way of merging solutions.
3. problem, subproblems & sub sub problems all must be a same kind



20.Write the steps to solve the 4-queens problem by backtracking method. For each step draw the 4x4 matrix showing the positions of queens in it.

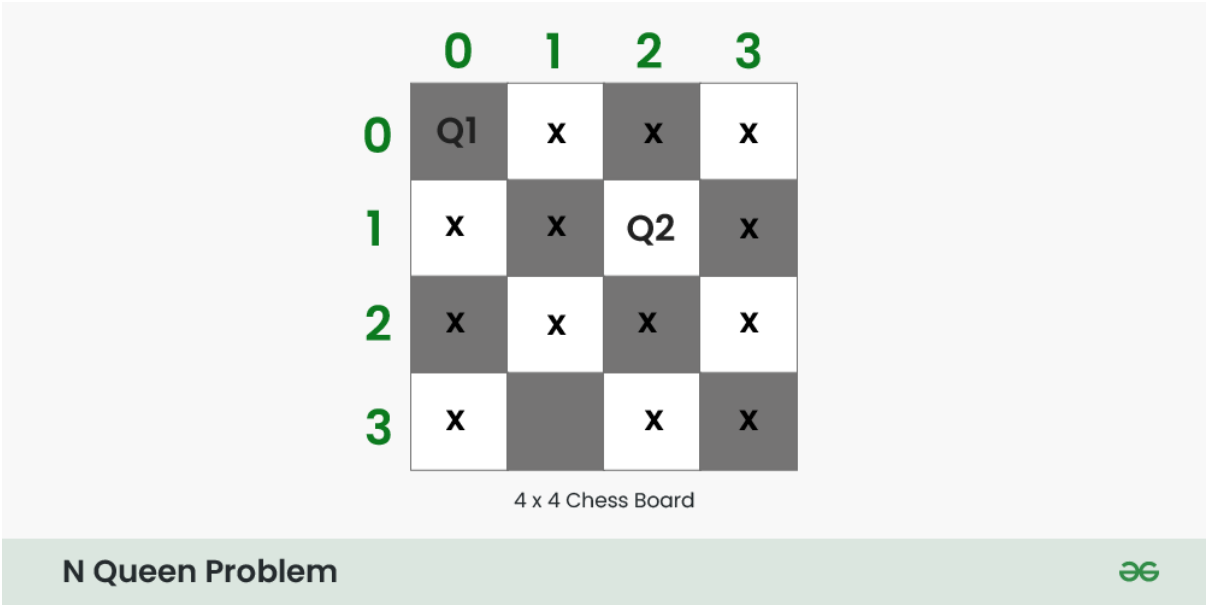
Step 1:

- Put our first Queen (**Q1**) in the **(0,0)** cell .
- '**x**' represents the cells which is not safe i.e. they are under attack by the Queen (**Q1**).
- After this move to the next row [0 -> 1].



Step 2:

- Put our next Queen (**Q2**) in the **(1,2)** cell .
- After this move to the next row [1 -> 2].

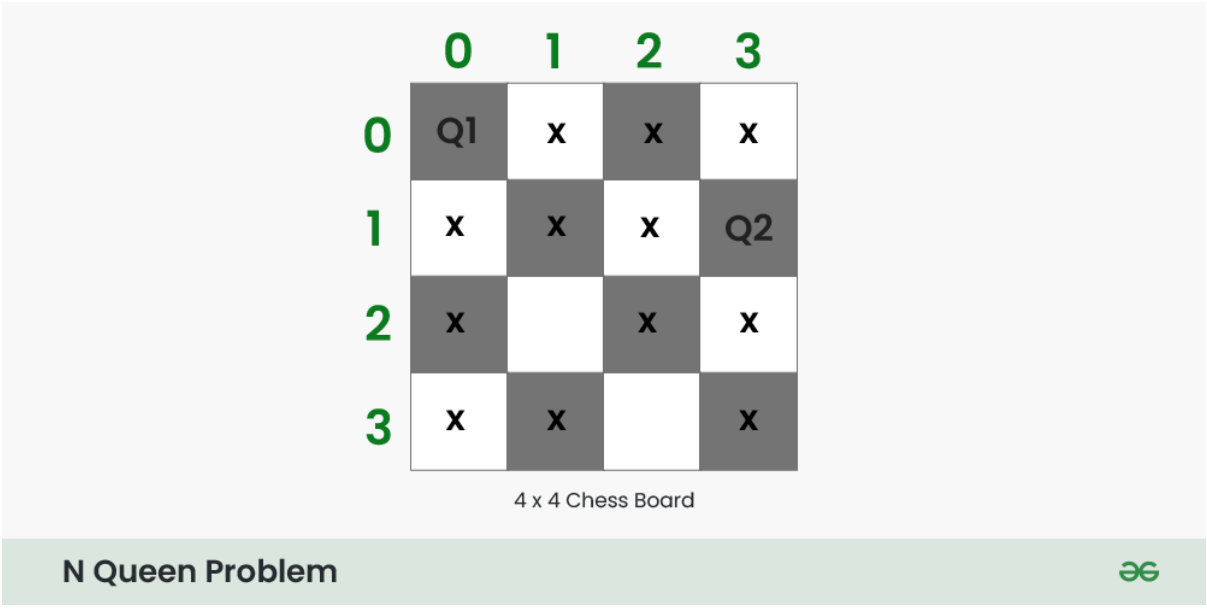


Step 3:

- At row 2 there is no cell which are safe to place Queen (**Q3**) .
- So, backtrack and remove queen **Q2** queen from cell (1, 2) .

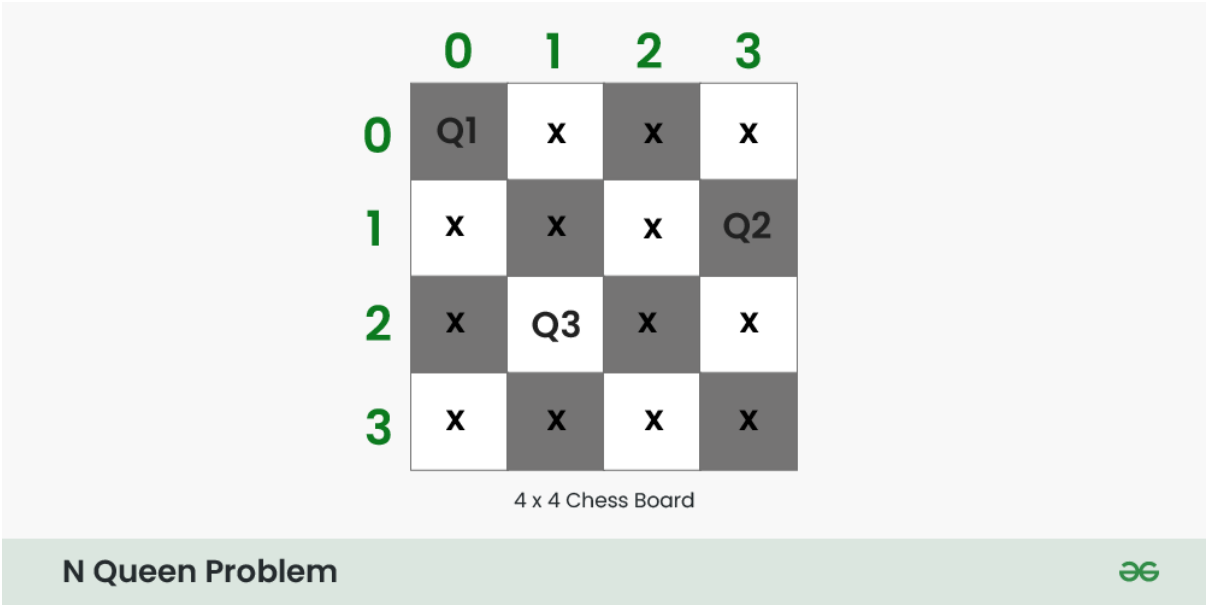
Step 4:

- There is still a safe cell in the row 1 i.e. cell (1, 3).
- Put Queen (**Q2**) at cell (1, 3).



Step 5:

- Put queen **(Q3)** at cell (2, 1).

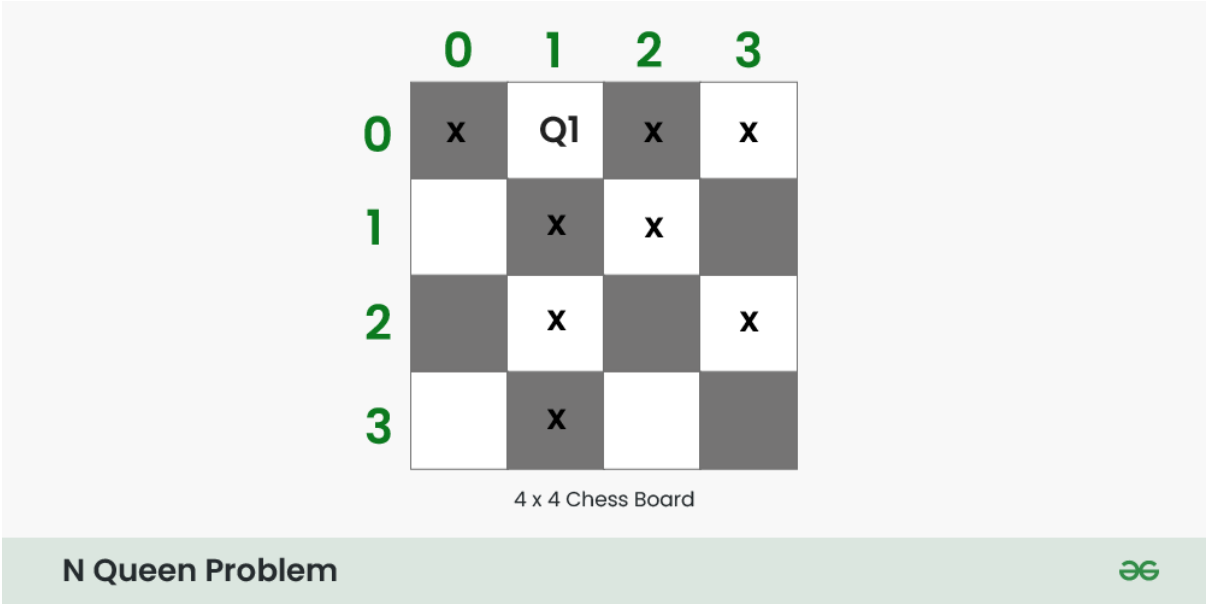


Step 6:

- There is no any cell to place Queen (**Q4**) at row 3.
- Backtrack and remove Queen (**Q3)** from row 2.
- Again there is no other safe cell in row 2, So backtrack again and remove queen (**Q2**) from row 1.
- Queen (**Q1)** will be remove from cell **(0,0)** and move to next safe cell i.e. **(0 , 1)**.

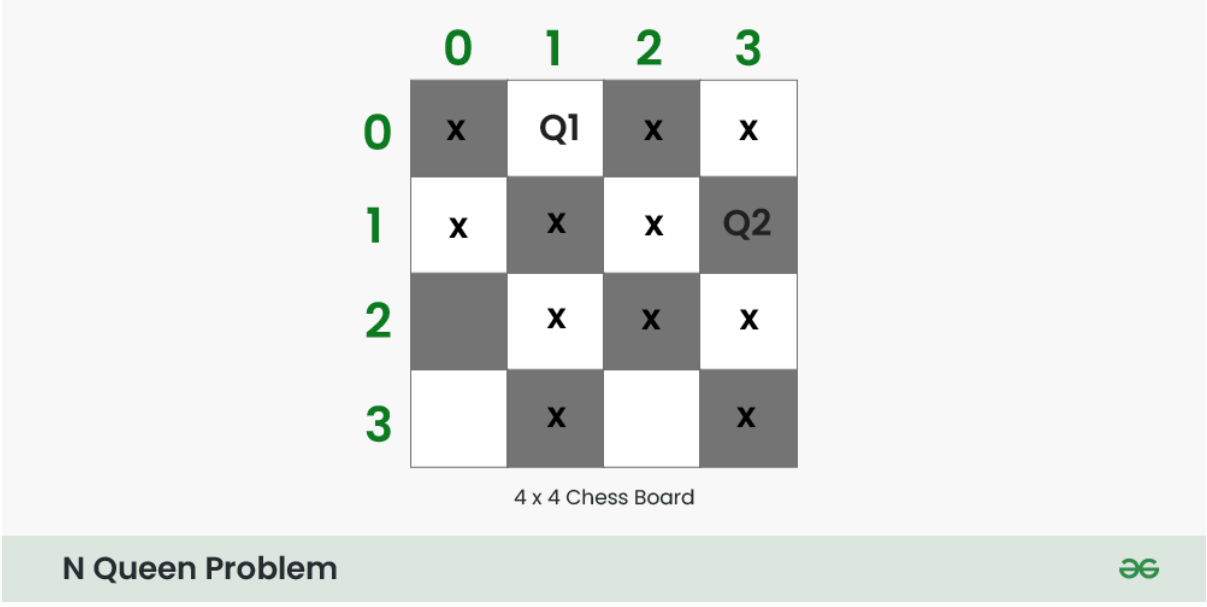
Step 7:

- Place Queen Q1 at cell (0 , 1), and move to next row.



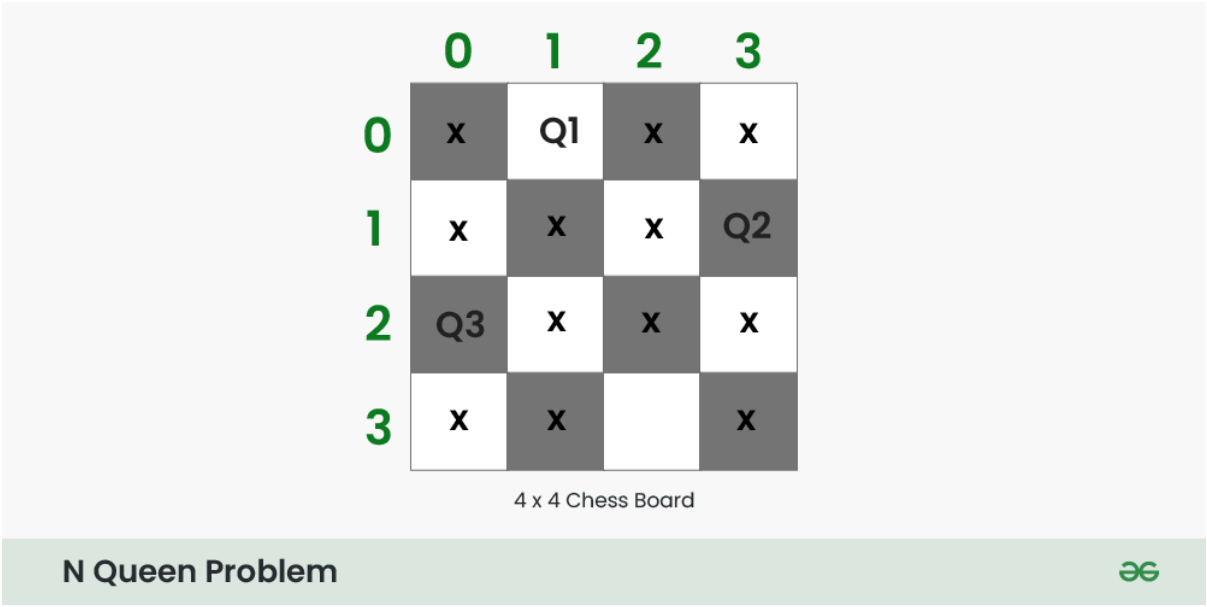
Step 8:

- Place Queen **Q2** at cell **(1 , 3)**, and move to next row.



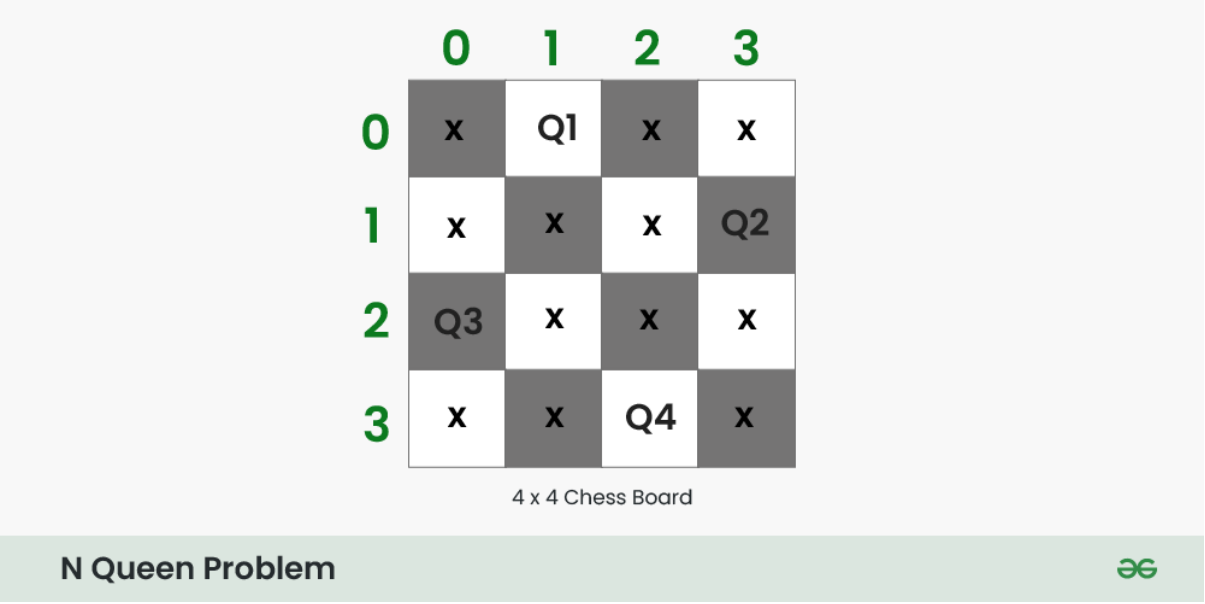
Step 9:

- Place Queen **Q3** at cell **(2 , 0)**, and move to next row.



Step 10:

- Place Queen **Q4** at cell **(3 , 2)**, and move to next row.
- This is one possible configuration of solution



21.Explain the basic concept of divide & conquer algorithm

Definition:

Divide and Conquer is a fundamental algorithmic technique where a problem is divided into smaller subproblems, each subproblem is solved independently (usually recursively), and then their solutions are combined to solve the original problem.

Three Main Steps:

1. **Divide** – Break the problem into smaller subproblems.
2. **Conquer** – Solve each subproblem recursively.
3. **Combine** – Merge the solutions of subproblems to get the final result.

Maximum and Minimum:

1. Let us consider simple problem that can be solved by the divide-and conquer technique.

2. The problem is to find the maximum and minimum value in a set of 'n' elements.

3. By comparing numbers of elements, the time complexity of this algorithm can be analyzed.

4. Hence, the time is determined mainly by the total cost of the element comparison.

```
Algorithm straight MaxMin (a, n, max, min)
// Set max to the maximum & min to the minimum of a [1: n]
{
Max = Min = a [1];
For i = 2 to n do
{
If (a [i] > Max) then Max = a [i];
If (a [i] < Min) then Min = a [i];
}}
```

Explanation:

a. Straight MaxMin requires 2(n-1) element comparisons in the best, average & worst cases.

b. By realizing the comparison of a [i]max is false, improvement in a algorithm can be done.

c. Hence we can replace the contents of the for loop by, If (a [i]> Max) then Max = a [i]; Else if (a [i]< 2(n-1)

d. On the average a[i] is > max half the time, and so, the avg. no. of comparison is 3n/2-1.

Divide and Conquer simplifies complex problems by solving smaller parts and combining results efficiently. Common examples include Merge Sort, Quick Sort, and Binary Search.

ALGORITHM

```
MaxMin (i, j, max, min)
 // a [1: n] is a global array, parameters i& j are integers, 1<= i<=j <=n. The effect is
to4.
// Set max & min to the largest & smallest value 5 in a [i: j], respectively.
{
If (i=j) then Max = Min = a[i];
Else if (i=j-1) then
{
if (a[i] < a[j]) then
        {
                Max = a[j];
                Min = a[i];
        }
        Else
        {
                Max = a[i];
                Min = a[j];
        }
}Else
{
Mid = (i + j) / 2;
MaxMin (I, Mid, Max, Min);
MaxMin (Mid +1, j, Max1, Min1);
If (Max < Max1) then Max = Max1;
If (Min > Min1) then Min = Min1;
}}
The procedure is initially invoked by the statement, MaxMin (1, n, x, y)
```

21.Analyse how backtracking related with recursion

✅ **Relationship Between Backtracking and Recursion:**

Aspect	Backtracking	Recursion
Technique	Systematically explores all possible options.	A function calls itself to solve smaller instances of a problem.
Relation	**Backtracking is implemented using recursion.**	Backtracking is one of the practical applications of recursion.
Control Flow	Goes deeper into one path (recursive call), and if it fails, it returns (backtracks) to try another path.	Each recursive call processes a smaller version of the problem.
State Restoration	On backtracking, the algorithm **undoes** changes made in the current path.	Each recursive call has its own local variables and call stack.

- **Recursion** is the underlying mechanism.
- **Backtracking** uses recursion to explore multiple possibilities.
- If a recursive call leads to a dead-end, backtracking **"undoes"** the step and tries the next option.

✅ **Backtracking = Recursion + Constraint Checking + Undoing Choices**

22.Analyse the steps involved for the 4-queen problem using Backtracking.

Ans= same as question number 20

23.T(n) = 2T(n/2) + O(n) where n>0, T(n) = 1 where n = 0. Explain the recurrence relation using substitution method.

Step-by-Step Substitution:

We assume nnn is a power of 2 (i.e., n=2kn = 2^kn=2k) to simplify the math.

Step 1: Expand the recurrence

T(n)=2T(n/2)+cn

Substitute T(n/2):

=2[2T(n/4)+c(n/2)]+cn=4T(n/4)+cn+cn=4T(n/4)+2cn= 2[2T(n/4) + c(n/2)] + cn \\ = 4T(n/4) + cn + cn = 4T(n/4) + 2cn=2[2T(n/4)+c(n/2)]+cn=4T(n/4)+cn+cn=4T(n/4)+2cn

Substitute T(n/4)T(n/4)T(n/4):

=4[2T(n/8)+c(n/4)]+2cn=8T(n/8)+cn+2cn=8T(n/8)+3cn= 4[2T(n/8) + c(n/4)] + 2cn \\ = 8T(n/8) + cn + 2cn = 8T(n/8) + 3cn=4[2T(n/8)+c(n/4)]+2cn=8T(n/8)+cn+2cn=8T(n/8)+3cn

Notice the pattern:

T(n)=2kT(n/2k)+kcnT(n) = 2^k T(n/2^k) + kcnT(n)=2kT(n/2k)+kcn

We continue this until n/2k=1n/2^k = 1n/2k=1, i.e., k=log⁡2nk = \log_2 nk=log2n

So we get:

T(n)=2log⁡nT(1)+cnlog⁡nT(n) = 2^{\log n} T(1) + cn \log nT(n)=2lognT(1)+cnlogn

Since 2log⁡n=n2^{\log n} = n2logn=n, and T(1)T(1)T(1) is constant:

T(n)=n·T(1)+cnlog⁡n=O(nlog⁡n)T(n) = n \cdot T(1) + cn \log n = O(n \log n)T(n)=n·T(1)+cnlogn=O(nlogn)

24.Write the algorithm of application of merge sort.

=Merge Sort is a **Divide and Conquer** algorithm that divides the input array into halves, recursively sorts them, and then merges the sorted halves.

🔷 **Algorithm: Merge Sort**

MERGE_SORT(A, left, right)

1. if left < right then
2. mid ← (left + right) / 2
3. MERGE_SORT(A, left, mid) // Divide left half
4. MERGE_SORT(A, mid + 1, right) // Divide right half
5. MERGE(A, left, mid, right) // Conquer: Merge two sorted halves

Use Case:

This algorithm is useful for applications like:

- University databases sorting students by roll numbers
- Generating merit lists
- Maintaining ordered records in school management systems

---

25. Define Rabin-Karp algorithm

The Rabin-Karp algorithm is a string-searching algorithm that uses hashing to efficiently find a pattern within a larger text. It compares the hash values of substrings of the text with the hash value

of the pattern to identify potential matches, followed by character-by-character verification for confirmed matches. The Knuth-Morris-Pratt (KMP) algorithm is another string-matching algorithm that leverages a pre-computed table to efficiently find occurrences of a pattern in a text. The time complexity of KMP is O(n+m), where n is the length of the text and m is the length of the pattern.

Rabin-Karp Algorithm

1. **1. Hashing:**

Compute the hash value of the pattern and the first substring of the text (same length as the pattern).

2. **2. Comparison:**

Compare the hash values. If they match, compare the pattern and substring character by character to confirm the match (hash collisions can occur).

3. **3. Rolling Hash:**

Calculate the hash value for the next substring of the text using a "rolling hash" technique (efficiently update the hash value based on the previous hash value).

4. **4. Iteration:**

Repeat steps 2 and 3, moving the substring window across the text, until the end of the text is reached.

26.KMP Algorithm

1. **1. Pre-processing:**

Calculate the Longest Proper Prefix which is also a Suffix (LPS) array for the pattern. This array stores the length of the longest proper prefix of each prefix of the pattern that is also a suffix of that prefix.

2. **2. Matching:**

Scan the text and compare each character with the pattern.

3. **3. Utilizing LPS Array:**

When a mismatch occurs, the LPS array determines the next valid position to start the match, avoiding redundant comparisons.

4. **4. Iteration:**

Repeat steps 2 and 3, moving through the text and the pattern, until a match is found or the end of the text is reached.

Time Complexity of KMP Algorithm

The KMP algorithm has a time complexity of O(n + m), where n is the length of the text and m is the length of the pattern. This is because the pre-processing step to calculate the LPS array takes O(m) time, and the matching step can be completed in O(n + m) time.

27.Discuss Substitution Method with an example?

The substitution method is a technique used to prove the time complexity of algorithms. It involves guessing the time complexity and then substituting this guess into the recurrence equation of the algorithm.

Example:

Let's consider the recurrence equation T(n) = T(n/2) + 1.

1. **Guess:** Assume that T(n) = O(log n).
2. **Substitution:** Substitute the guess into the recurrence equation:
    - T(n) <= c log n (where c is a constant)
    - T(n/2) <= c log (n/2) = c (log n - log 2) = c log n - c log 2
    - T(n) = T(n/2) + 1 <= c log n - c log 2 + 1
    - Since c log 2 is a constant, we can absorb it into the constant c:
    - T(n) <= c log n + 1
    - Since 1 is a constant, we can also absorb it into the constant c:
    - T(n) <= c log n
3. **Verification:** The substitution proves that T(n) = O(log n) is a valid guess.

28.Solve for the all pair of shortest path using Floyd's Algorithm

Floyd's Algorithm (All Pairs Shortest Path)

Floyd's algorithm is used to find the shortest path between all pairs of vertices in a graph. It iteratively calculates the shortest path lengths between all possible pairs of vertices.

Algorithm:

1. Initialize an adjacency matrix dist with the edge weights. If there's no direct edge between two vertices, mark the distance as infinity.
2. Iterate through all possible intermediate vertices k.
3. For each pair of vertices i and j, update the distance between them as follows: dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]).
4. Repeat steps 2 and 3 until all intermediate vertices have been considered.

Time Complexity: O(n^3), where n is the number of vertices in the graph.

Traveling Salesman Problem (TSP)

The Traveling Salesman Problem (TSP) is a classic problem in computer science and operations research.

29. Traveling Salesman Problem (TSP)

The **Traveling Salesman Problem** is a classic optimization problem in computer science and operations research. It is defined as:

Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?

- **Type**: NP-Hard
- **Application**: Logistics, route planning, manufacturing
- **Approaches**: Brute force, Dynamic programming (Held-Karp), Heuristics (e.g., Genetic algorithms, Simulated Annealing)

---

30. Difference Between Bellman-Ford and Dijkstra's Algorithm

Feature	Dijkstra's Algorithm	Bellman-Ford Algorithm
Graph Type	Works only with non-negative weights	Works with negative weights
Time Complexity	O(V2)O(V^2)O(V2) or O((V+E)log⁡V)O((V + E)\log V)O((V+E)logV) with priority queue	O(VE)O(VE)O(VE)
Negative Cycles	Cannot detect negative cycles	Can detect negative cycles
Use Case	Faster, used in real-time systems	More general, used in theoretical settings

---

31. Describe Maximum Clique

A **maximum clique** in a graph is the **largest subset of vertices** such that **every two vertices in the subset are connected** by an edge.

- **Clique**: A subset of nodes where every node is connected to every other node.
- **Maximum Clique**: The clique with the largest number of vertices in the graph.
- **Problem Type**: NP-Hard (it's computationally hard to find the largest clique in a general graph).

---

32. Describe NP-Hard

A problem is **NP-Hard** if solving it efficiently (in polynomial time) would allow us to solve all problems in **NP** efficiently.

- **Key Points**:
    - Not required to be in NP (i.e., solutions may not be verifiable in polynomial time).
    - Often optimization problems, like TSP.
    - Every NP-Complete problem is also NP-Hard.

---

33. Describe NP-Complete

A problem is **NP-Complete** if:

1. It is in **NP** (a solution can be verified in polynomial time).
2. Every problem in **NP** can be **reduced** to it in polynomial time.

- These are the hardest problems **within NP**.
- If any NP-Complete problem can be solved in polynomial time, then **P = NP**.

---

34. Define 3SAT

3SAT is a specific form of the Boolean satisfiability problem (SAT), where:

- The input is a Boolean formula in **Conjunctive Normal Form (CNF)**.
- Each clause has exactly **three literals**.

Example:

(x1∨¬x2∨x3)∧(¬x1∨x4∨x5)(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_4 \vee x_5)(x1∨¬x2∨x3)∧(¬x1∨x4∨x5)

- **Goal**: Determine if there is an assignment of true/false values to variables that makes the entire formula true.
- **3SAT** is **NP-Complete**.

---

35. Define Complexity Classes

Complexity classes group problems based on the **resources** (like time or space) needed to solve them.

Key classes:

- **P**: Solvable in polynomial time.
- **NP**: Verifiable in polynomial time.

- **NP-Complete**: Hardest problems in NP.
- **NP-Hard**: At least as hard as NP problems.
- **PSPACE**: Solvable using polynomial space.
- **EXP**: Solvable in exponential time.

---

36. Define Reducibility in the Context of Computational Complexity

Reducibility is a method to compare the difficulty of problems.

- A problem **A is reducible to B** (written as A ≤ B) if a solution to **B** can be used to solve **A**.
- If the reduction can be done in polynomial time, it's called **polynomial-time reducibility**.
- Used to show that if B is easy, then A is also easy.
- **Crucial for proving NP-Completeness.**

---

37. Define the SAT Problem

SAT (Boolean Satisfiability Problem) asks:

Given a Boolean formula, is there an assignment of truth values to its variables that makes the formula true?

Example:

(x∨y)∧(¬x∨z)(x \vee y) \wedge (\neg x \vee z)(x∨y)∧(¬x∨z)

- **First problem proven to be NP-Complete** (Cook-Levin Theorem).
- Fundamental in theoretical computer science and logic.

---

38. Is Vertex Cover in NP-Complete? Explain

Yes, the **Vertex Cover** problem is **NP-Complete**.

- **In NP**: Given a set of vertices, we can check in polynomial time whether it covers all edges.
- **NP-Hard**: 3SAT can be polynomially reduced to Vertex Cover.
- **Conclusion**: Since it is in NP and NP-Hard → Vertex Cover is NP-Complete.

---

39. Explain the Vertex Cover Problem

The **Vertex Cover** problem is defined as:

Given a graph G=(V,E)G = (V, E)G=(V,E) and an integer kkk, is there a subset of vertices C⊆VC \subseteq VC⊆V such that:

- Every edge in the graph has **at least one endpoint** in CCC.
- ∣C∣≤k|C| \leq k∣C∣≤k
- **Goal**: Find the **minimum vertex cover**.
- **Type**: Decision and optimization problem.
- **NP-Complete** for general graphs.

. Explain Independent Set

An **independent set** in a graph is a set of vertices such that **no two vertices in the set are adjacent** (i.e., there is no edge between any pair of selected vertices).

- **Maximum Independent Set**: Largest possible independent set in the graph.
- **Problem Type**: NP-Hard
- **Applications**: Network design, scheduling, coding theory.

---

2. Explain Class P

Class P consists of all decision problems that can be solved by a deterministic Turing machine in **polynomial time**.

- **Polynomial Time**: Time complexity is bounded by O(nk)O(n^k)O(nk), where kkk is a constant.
- **Examples**: Sorting, shortest path (Dijkstra), and minimum spanning tree (Prim's/Kruskal's).
- Considered "efficiently solvable."

---

3. Define K-Center Clustering Problem

In the **K-Center Clustering** problem:

Given a set of points and a number kkk, choose kkk centers such that the **maximum distance** of any point to its **nearest center** is minimized.

- **Objective**: Minimize the worst-case (max) distance.
- **Applications**: Facility location, emergency response planning.
- **Problem Type**: NP-Hard

---

4. Define Hiring Problem

The **Hiring Problem** is a classic example in **online algorithms** and **probability**.

You interview candidates one by one. You must decide **immediately** whether to hire a candidate, without being able to return to a previous one.

- **Goal**: Minimize the number of hires while selecting the best candidate.
- **Key Insight**: Probabilistic strategies often outperform deterministic ones.

---

5. Define Global Minimum Cut of a Graph

A **global minimum cut** of a graph is a partition of the vertex set into two non-empty subsets such that the **total weight of edges crossing the partition is minimized**.

- **Unweighted Graph**: Minimize number of crossing edges.
- **Applications**: Network reliability, image segmentation.
- **Algorithm**: Karger's randomized algorithm, Stoer-Wagner.

---

6. State Minimum Cut of a Graph

The **minimum cut** of a graph is a smallest set of edges whose removal **disconnects** the graph into at least two components.

- **Input**: Undirected weighted graph.
- **Output**: Minimum weight set of edges whose removal separates the graph.
- Can be a **global** or **source-sink-specific** (s-t cut).

---

7. State Set-Covering Problem Related to Vertex Cover

The **Set Cover** problem is a generalization of **Vertex Cover**.

- **Set Cover**: Given a universe UUU and a collection of subsets, select the minimum number of subsets such that their union covers UUU.
- **Vertex Cover**: Vertices act like sets, and edges like elements to be covered.
- **Reduction**: Vertex Cover can be reduced to Set Cover and vice versa.

---

8. Justify Approximation Algorithm

An **approximation algorithm** provides **near-optimal** solutions to NP-Hard problems in **polynomial time**.

- **Justification**:
    - Exact solutions are often intractable for large inputs.
    - Approximation algorithms guarantee a solution **within a known factor** of the optimal (e.g., 2-approximation for Vertex Cover).
    - Useful in real-world problems where time is a constraint.

---

9. Explain Graph Traversal Algorithm

Graph traversal algorithms systematically explore the vertices and edges of a graph.

- **BFS (Breadth-First Search)**:
    - Explores layer by layer.
    - Uses a queue.
    - Good for shortest paths in unweighted graphs.
- **DFS (Depth-First Search)**:
    - Explores as deep as possible before backtracking.
    - Uses a stack (or recursion).
    - Good for topological sorting, cycle detection.

---

10. Identify Optimal Parenthesization of Matrix Chain (5, 10, 3, 12, 5, 50, 6)

Matrix Chain Multiplication Problem:

- Goal: Minimize the number of scalar multiplications needed to multiply a sequence of matrices.

Algorithm (Dynamic Programming):

1. Let the matrix dimensions be p=[5,10,3,12,5,50,6]p = [5, 10, 3, 12, 5, 50, 6]p=[5,10,3,12,5,50,6]
2. Number of matrices n=6n = 6n=6
3. Define cost matrix m[i][j]m[i][j]m[i][j] and split matrix s[i][j]s[i][j]s[i][j]
4. Use bottom-up DP to fill in m[i][j]m[i][j]m[i][j]
5. Use s[i][j]s[i][j]s[i][j] to find parenthesis order

Time Complexity:

O(n3)O(n^3)O(n3)

Let me know if you'd like the full table computed.

---

11. Explain Knapsack Problem and Types

The **Knapsack Problem** is an optimization problem where:

Given a set of items with **weights and values**, and a maximum capacity, choose items to maximize value **without exceeding capacity**.

Types:

- **0/1 Knapsack**: Items can’t be split.

- **Fractional Knapsack**: Items can be split (greedy algorithm works).
- **Multi-dimensional Knapsack**: Multiple constraints.

---

12. Write General Knapsack Problem and Derive Time Complexity

0/1 Knapsack:

Given:

- Items 1..n1..n1..n, each with value viv_ivi, weight wiw_iwi
- Capacity WWW

Goal:
Maximize ∑vixi\sum v_i x_i∑vixi, where xi∈{0,1}x_i \in \{0,1\}xi∈{0,1}, and ∑wixi≤W\sum w_i x_i \leq W∑wixi≤W

DP Solution:

Let dp[i][w]dp[i][w]dp[i][w] = max value for first iii items with capacity www

Recurrence:

dp[i][w]=max⁡(dp[i−1][w],dp[i−1][w−wi]+vi)dp[i][w] = \max(dp[i-1][w], dp[i-1][w - w_i] + v_i)dp[i][w]=max(dp[i−1][w],dp[i−1][w−wi]+vi)

Time Complexity:

O(nW)O(nW)O(nW)

---

13. Greedy Solution for Knapsack (Fractional)

Given:

Capacity = 100 kg
Items: (Please provide a table of items with **weights** and **values**)

Greedy Algorithm (Fractional):

1. Compute value/weight ratio.
2. Sort items by this ratio.
3. Take as much of the highest ratio item as possible until full.

Time Complexity:

O(n \log n) \] (due to sorting)