DAA PYQ Final

Illustration of Bounding Function in Backtracking

A bounding function in backtracking is a way to prune the search tree, preventing the exploration of nodes that cannot possibly lead to a solution. It helps optimize the backtracking process by discarding branches that are unlikely to yield an optimal or feasible solution. This is achieved by using a heuristic or a lower/upper bound to estimate the cost or value of a solution at a certain point in the search tree. If a node's bound is not better than the best solution found so far, it is discarded, and the algorithm explores other branches.

e) Examine the shortcomings of the Dijkstra's Algorithm.

❌ **1. Does Not Work with Negative Edge Weights**

- **Why?** Dijkstra assumes that once a node's shortest distance is finalized, it won't change.
- **Problem:** Negative edge weights can lead to shorter paths **after** a node has been finalized.

❌ **2. Inefficient for Large Sparse Graphs (Without Heap)**

- If implemented with an adjacency matrix and no priority queue, its time complexity is **$O(V^2)$**.
- In large sparse graphs, this becomes inefficient.

❌ **3. Cannot Handle Negative Weight Cycles**

- If the graph contains **negative cycles**, Dijkstra doesn't detect them and may return incorrect paths.

❌ **4. Only Works for Single Source**

- Dijkstra solves **single-source shortest path** problems.

❌ **5. Path Reconstruction Requires Extra Effort**

- The basic version only computes distances.
- To get the actual **path**, you must track parent or predecessor nodes.

❌ **6. Greedy Nature Can Mislead in Some Scenarios**

- The algorithm is **greedy**, which means it might make decisions that seem best locally but aren't optimal globally in more complex situations (e.g., time-dependent graphs).

a) What is an algorithm and Describe the characteristics of algorithm.

An **algorithm** is a **finite set of well-defined instructions** used to solve a problem or perform a specific task.

🔍 **Key Characteristics of an Algorithm:**

1. **Input:**
    - Takes zero or more inputs.
2. **Output:**
    - Produces at least one output.
3. **Definiteness:**
    - Each step is clear and unambiguous.
4. **Finiteness:**
    - It must terminate after a finite number of steps.
5. **Effectiveness:**
    - Each step must be basic enough to be performed exactly and in a finite time.

📘 **Example:**

Problem: Find the sum of two numbers.

Algorithm:

1. Start
2. Input two numbers, A and B
3. Add A and B, store in S
4. Output S
5. Stop

Define dynamic programming and explain its significance in algorithm design. How does it differ from other algorithm design techniques?

🔷 **Definition of Dynamic Programming (DP):**

Dynamic Programming is a method used to solve complex problems by **breaking them into smaller overlapping subproblems**, solving each subproblem **only once**, and **storing their results** (usually in a table or array) to **avoid redundant work**.

DP is applicable when a problem has:

1. **Optimal substructure** (solution of the main problem can be constructed from solutions of subproblems)
2. **Overlapping subproblems** (same subproblems are solved multiple times)

📘 **Example: Fibonacci Numbers**

- **Naive recursive** solution:
  Time complexity = **$O(2^n)$** due to repeated calculations.

- **Dynamic Programming** (bottom-up or memoization):
  Time complexity = **O(n)**
  → Because each subproblem is solved only once and stored.

🔍 **Significance of Dynamic Programming in Algorithm Design:**

- **Efficiency:**
  Avoids redundant computations, dramatically reducing time complexity.
- **Deterministic Solutions:**
  Unlike greedy algorithms, DP guarantees an **optimal solution** when applicable.
- **Widely Applicable:**
  Used in:
  - Shortest paths (e.g., Floyd-Warshall)
  - Sequence alignment (bioinformatics)
  - Knapsack problems
  - Matrix chain multiplication
  - Game theory and decision making

🔄 **Comparison with Other Techniques:**

Technique	Key Idea	Reuses Subproblems?	Guarantees Optimal?
Brute Force	Try all possibilities	❌ No	✅ Yes (but inefficient)
Divide & Conquer	Break problem into independent subproblems	❌ No	✅ Sometimes
Greedy	Make locally optimal choice at each step	❌ No	❌ Not always
Dynamic Programming	Store solutions of overlapping subproblems	✅ Yes	✅ Yes

How Can You Prove a Problem to Be NP-Complete?

To prove a problem is NP-Complete, you must follow two main steps. This involves showing that:

🔹 **Step 1: The Problem is in NP**

You must show that:

- Given a **candidate solution**, you can **verify it in polynomial time**.

📌 Example:For the **Subset Sum** problem, if someone gives you a subset, you can easily sum its elements and check if it matches the target — this verification is polynomial-time.

🔹 **Step 2: NP-Hardness — Reduce a Known NP-Complete Problem to Your Problem**

You must:

- Take a **known NP-complete problem**, and
- Show that it can be **polynomially reduced** to your problem.

This means:"If we could solve **your problem** efficiently, then we could solve the known NP-complete problem efficiently too."

This step proves that your problem is **at least as hard** as any problem in NP.

d) What is the difference between polynomial and exponential running time?

Aspect	**Polynomial Running Time**	**Exponential Running Time**
Definition	Time complexity expressed as **n^k** for some constant k (e.g., n, n^2, n^3)	Time complexity expressed as **c^n** where c > 1 (e.g., 2^n, 3^n)
Growth Rate	Grows **slowly** as input size n increases	Grows **very fast** as input size n increases
Examples	O(n), O(n^2), O(n^3)	O(2^n), O(3^n), O(2^{n/2})
Feasibility	Generally **feasible** for reasonably large n	Generally **infeasible** even for moderate n
Algorithm Type	Efficient algorithms (e.g., sorting, searching)	Brute force, exhaustive search algorithms
Impact on Computation	Computation time increases moderately	Computation time becomes impractically large quickly
Use in Practice	Preferred for large-scale problems	Often used in small input cases or theoretical study

e) What is the time complexity of algorithm finding all pair shortest path?

1. Floyd-Warshall Algorithm

- Uses dynamic programming.
- Works for **weighted graphs** (with positive or negative weights, but no negative cycles).
- **Time Complexity:**

$$O(V^3)$$

where V = number of vertices.

- **Space Complexity:**
  $O(V^2)$ for storing the distance matrix.

2. Using Dijkstra's Algorithm for Each Vertex

- Run Dijkstra's algorithm from every vertex.
- If implemented with a priority queue (min-heap) and adjacency list:
  - Dijkstra runs in $O((V + E) \log V)$
- For all vertices:

$$O(V \times (V + E) \log V)$$

- Efficient for **sparse graphs**.

3. Using Bellman-Ford Algorithm for Each Vertex

- Can handle graphs with negative weights (but no negative cycles).
- Bellman-Ford runs in $O(VE)$ per source.
- For all vertices:

$$O(V^2E)$$

b) Which Big-O notation has the worst time complexity?

📉 **Common Big-O Notations (Ordered from Best to Worst):**

1. **O(1)** – Constant time
2. **O(log n)** – Logarithmic time
3. **O(n)** – Linear time
4. **O(n log n)** – Linearithmic time
5. **O(n²)** – Quadratic time
6. **O(n³)** – Cubic time
7. **O(2^n)** – Exponential time
8. **O(n!)** – Factorial time (very slow)

❌ **Worst Time Complexity: O(n!)**

- This occurs in problems where you must **generate all permutations** of n items.
- **Examples:**
  - **Traveling Salesman Problem (TSP)** (brute-force approach)
  - **Solving puzzles using exhaustive search**

🔺 **Even worse than exponential!**

If n = 20, then:

- 2^n = 1,048,576
- n! = 2,432,902,008,176,640,000 ← **much worse!**

f) Explain Graph-coloring problem.

Given an undirected graph G=(V,E)G = (V, E)G=(V,E), assign a color to each vertex in V such that:

- If (u,v)∈E then color(u) ≠ color(v)
- Use the **minimum** number of colors possible.

✅ **Example:**

Consider a triangle graph with 3 vertices (each connected to the other):

- You need **3 different colors** so that no adjacent vertices share the same color.

2m and 4m) **Definition and Significance of Big-O, Omega, and Theta Notations**

🔹 **1. Big-O Notation (O)**

📘 **Definition:**

Big-O represents the **upper bound** of an algorithm's growth rate. It tells us the **maximum time or space** the algorithm could take.

f(n) = O(g(n)) means that **f(n)** grows **no faster** than **g(n)**, up to constant factors.

🎯 **Significance:**

- Describes the **worst-case** performance.
- Ensures that performance will not degrade beyond a certain limit.
- Helps compare different algorithms' scalability.

🧠 **Example:**If an algorithm is **O(n²)**, it means in the worst case, its time grows proportionally to n2n^2n2.

🔹 **2. Omega Notation (Ω)**

📘 **Definition:**

Omega represents the **lower bound** of an algorithm's growth rate. It tells us the **minimum time or space** required.

f(n) = Ω(g(n)) means that **f(n)** grows at **least as fast** as **g(n)**.

🎯 **Significance:**

- Describes the **best-case** performance.
- Helps understand the **fastest** an algorithm could possibly perform.

🧠 **Example:**For linear search, **Ω(1)** implies that in the best case, the item is found in the first position.

🔹 **3. Theta Notation (Θ)**

📘 **Definition:**

Theta provides a **tight bound**, meaning the algorithm grows **exactly** at the rate of a given function (both upper and lower bounds).

f(n) = Θ(g(n)) means that **f(n)** grows at the **same rate** as **g(n)**.

🎯 **Significance:**

- Describes the **average-case** or **exact** performance.
- Gives a precise measure of an algorithm's efficiency.

🧠 **Example:**Merge sort takes **Θ(n log n)** time in all cases, so it's consistently efficient.

State and explain Cook's theorem.
Formally:
"Every problem in the complexity class **NP** can be **polynomial-time reduced** to the **SAT** problem."

This means that:

SAT is the first known NP-Complete problem.

🔍 **What is SAT (Satisfiability Problem)?**

- Given a Boolean formula (e.g., in CNF form), determine if there exists a **truth assignment** to variables that makes the formula **evaluate to true**.

📘 **Explanation of Cook's Theorem:**

1. SAT ∈ NP

- It is easy to **verify** if a given truth assignment satisfies the Boolean formula — in **polynomial time**.

2. SAT is NP-Hard

- Cook showed that for **any problem in NP**, its computation (by a non-deterministic Turing machine) can be **encoded as a SAT instance** in **polynomial time**.
- So, **solving SAT would allow us to solve any NP problem**.

e) Are Merge Sort and Quick Sort stable sorts? Justify your answer

🔸 **1. Merge Sort ✅ Stable**

- **Reason:**
  Merge Sort **compares elements in order** and when two elements are equal, it **chooses the one that appeared first** in the original list (from the left half).

- **Therefore:**
  It **preserves the original relative order** of equal elements.
- **Conclusion:**
  ✅ **Merge Sort is a stable sort.**

🔸 **2. Quick Sort ❌ Not Stable**

- **Reason:**
  Quick Sort may **swap elements that are far apart**, especially during partitioning.
    - It does **not guarantee** the relative order of equal elements.
- **Example:**
  Original:
  [(John, 25), (Jane, 25)]
  After Quick Sort:
  [(Jane, 25), (John, 25)] ← order changed
- **Conclusion:**
  ❌ **Quick Sort is not stable** in its standard form.

Suppose you have an O(n) time algorithm that finds the median of an unsorted array. Now consider a QuickSort implementation where we first find the median using the above algorithm, then use the median as a pivot. What will be the worst-case time complexity of this modificd QuickSort? Justify your answer.

🔹 **Given:**

- We have an **O(n)** time algorithm to find the **median** of an unsorted array.
- We modify QuickSort to:
    - First **find the median** (O(n)),
    - Then **use it as the pivot**.

🔹 **Objective:**

Determine the **worst-case time complexity** of this modified QuickSort.

🔍 **Standard QuickSort Recap:**

- Worst-case occurs when pivot selection is poor (e.g., smallest/largest element).
- Standard worst-case time:O(n2)

🔸 **Modified QuickSort with Median Pivot:**

By choosing the **true median** as the pivot:

- The array is **always perfectly split** into two halves of size ≈ n/2
- So, the recurrence relation becomes:

$$T(n) = T(n/2) + T(n/2) + O(n) = 2T(n/2) + O(n)$$

This recurrence solves to:

T(n)=O(nlogn)

🔹 ✅ **Justification:**

- Finding the median takes **O(n)** time.
- Partitioning takes **O(n)** time.
- The subproblems are **balanced** (each half has ≈ n/2n/2n/2 elements).
- Solving this recurrence gives:

O(nlogn)

✅ **Final Answer:**

The **worst-case time complexity** of the modified QuickSort using the median as pivot is **O(n log n)**.

This improvement makes QuickSort's worst-case as good as its average-case!

a) Providc a scenario where understanding asymptotic notations is crucial for designing and analyzing algorithms effectively.

🔷 **Scenario: Choosing an Algorithm for Sorting Large Data Sets**

Imagine you're building a **real-time dashboard** that processes and displays **live data from millions of users**, and you need to **sort** this incoming data frequently.

You have two algorithm choices:

1. **Algorithm A**: Has time complexity **O(n²)**
2. **Algorithm B**: Has time complexity **O(n log n)**

🔍 **Why Asymptotic Notation Matters:**

Even if **Algorithm A** runs faster for **small inputs** due to a simpler implementation, it becomes **unusable** for large inputs due to the **quadratic growth** in time.

Let's assume n = 1,000,000:

- $O(n^2)$:
  $\approx 10^{12}$ operations — practically **impossible in real time**
- O(n log n):
  $\approx 1,000,000 \times 20 = 20,000,000$ operations — **efficient and feasible**

📌 **Key Takeaway:**

Understanding asymptotic notations allows engineers to:

- **Predict scalability**

- **Choose the right algorithm**
- Avoid writing code that works well in testing but **fails in production**

4Marks

Appraise the importance of using greedy method and relaxing the condition of xi 0 or I to 0 xi 1 while computing optimal solution for 0/1 Knapsack problem using a recursive backtracking algorithm.

🔷 **Background: 0/1 Knapsack Problem**

Given:

- A set of nnn items, each with a **weight** wiw_iwi and **value** viv_ivi.
- A **knapsack capacity** WWW.

Goal:
Maximize the total value such that the total weight doesn't exceed WWW, and each item is either **included (1)** or **excluded (0)**.

This is a **combinatorial optimization problem**, and **backtracking** can be used to explore all valid combinations.

📌 **Part 1: Using the Greedy Method in Backtracking**

🔍 **Why Use the Greedy Method?**

- The **Greedy approach** (selecting items based on highest value/weight ratio) does **not always give optimal results** for the 0/1 Knapsack problem, but it is still useful as a **heuristic**.

✅ **Role in Backtracking:**

- When recursively exploring solutions, the greedy solution can provide an **upper bound** on the best possible solution in a branch.
- This allows us to **prune** branches early in the recursion (via bounding function), reducing computation.

🔹 **Conclusion:**Using greedy estimates improves **efficiency** by guiding the backtracking algorithm to explore **promising branches** first and ignore hopeless ones.

📌 **Part 2: Relaxing the Condition xi∈{0,1}x_i \in \{0,1\}xi∈{0,1} to 0≤xi≤10 \leq x_i \leq 10≤xi≤1**

🔍 **Why Relax the Condition?**

- In 0/1 Knapsack, each item must be fully included or excluded.
- By **relaxing the constraint** to allow **fractional items**, we turn it into the **Fractional Knapsack Problem**, which **can be solved optimally using a greedy algorithm in O(n log n)** time.

✅ **Significance in Backtracking:**

- This relaxed version provides a **tight upper bound** on the value possible from the current point onward.
- It allows the recursive algorithm to **estimate the maximum gain** in each branch, even if full items can't be included.
- If this estimate is less than the best solution found so far, the branch can be **safely pruned**.

🔹 **Conclusion:**

Relaxing the constraint gives a **fast, optimistic bound** used in pruning, improving **search efficiency** without compromising optimality.

🧠 **Overall Importance:**

Technique	**Purpose**	**Benefit**
Greedy method (value/weight)	Provides quick estimate of best gain	Helps in prioritizing and pruning
Relaxing xi∈[0,1]x_i \in [0,1]xi∈[0,1]	Allows use of fractional items for bounding	Tight upper bounds → faster pruning

Q4.Differentiate Divide-and-Conquer and Greedy Method.

Feature	**Divide and Conquer**	**Greedy Method**
Approach	Divides problem into subproblems and solves recursively	Makes locally optimal choices at each step
Problem Solving	Break → Solve → Combine	Build solution step-by-step
Recursion	Uses recursion heavily	Typically iterative
Subproblem Dependency	Subproblems are independent	Subproblems are not always independent
Global Optimality	Ensures optimal solution by solving all parts	May not guarantee globally optimal solution
Examples	Merge Sort, Quick Sort, Binary Search, Strassen's Matrix	Prim's Algorithm, Kruskal's Algorithm, Fractional Knapsack
Complexity	Often **O(n log n)** (e.g., merge sort)	Often **faster**, sometimes **O(n)**
Combine Step	Required to merge sub-solutions	No combine step — solution is built on the go

Define backtracking and explain its significance in algorithm design. How does backtracking differ from other algorithmic techniques?

🔹 Definition of Backtracking:

Backtracking is a general algorithmic technique used to solve constraint satisfaction problems by exploring all possible solutions and abandoning those paths that lead to an invalid or suboptimal solution.

It's a depth-first search approach where decisions are made step-by-step, and if a path fails, the algorithm "backtracks" to try a different choice.

🔧 How Backtracking Works:

1. Choose an option.
2. Recur to see if this leads to a solution.
3. If it doesn't, undo the choice (backtrack) and try another option.

🎯 Significance in Algorithm Design:

- Solves problems with combinatorial complexity where brute-force would be too slow.
- Helps in finding all or best possible solutions.
- Efficient when combined with pruning (i.e., skipping impossible paths early).

📘 Common Problems Solved by Backtracking:

- N-Queens Problem
- Sudoku Solver
- Graph Coloring
- Hamiltonian Cycle
- Subset Sum / Combinatorial Optimization

🔍 Difference from Other Algorithmic Techniques:

Feature	Backtracking	Divide & Conquer	Dynamic Programming	Greedy Method
Exploration	Tries all possible options	Divides into independent subproblems	Stores and reuses solutions	Picks best local solution
Recursive?	Yes (depth-first search)	Yes	Yes	Not necessarily

Feature	Backtracking	Divide & Conquer	Dynamic Programming	Greedy Method
Optimization Type	Optimal / all solutions	Exact solutions	Optimal substructure problems	Often not optimal
Use of Memory	Stack (recursive calls)	Stack + merges	Table (memoization)	Minimal
Efficiency	Slower unless optimized with pruning	Medium	Efficient if overlapping subproblems exist	Very fast (but may not give optimal solution)

What are the characteristics of a good approximation' algorithm?

1. **Performance Guarantee:**
   The algorithm provides a known bound on how close the solution is to the optimal one. This is often expressed as an approximation ratio or factor.
2. **Polynomial Time Complexity:**
   It should run in polynomial time, making it efficient and practical for large inputs.
3. **Simplicity:**
   The algorithm should be relatively simple to understand and implement.
4. **Scalability:**
   It should work well as the size of the input grows, maintaining reasonable accuracy and performance.
5. **Deterministic or Probabilistic Guarantees:**
   It either always guarantees a certain quality of solution or does so with high probability.
6. **Good Practical Performance:**
   Beyond theoretical guarantees, the algorithm should perform well on real-world instances.
7. **Robustness:**
   It should handle a variety of input types and still provide a good approximation.

Implement and demonstrate binary search algorithm to find the position of the element 35 in the array [5, 10, 15, 20, 25, 30, 35, 40, 45, 50].

Given array:

$A = [5, 10, 15, 20, 25, 30, 35, 40, 45, 50]$

Goal:

Find the position of element $x = 35$.

Binary Search Algorithm Steps (Mathematically):

1. Define:
   - $low = 0$ (starting index)
   - $high = n - 1 = 9$ (ending index, where $n = 10$ is the size of array)
2. Calculate the middle index:

$$mid = \left\lfloor \frac{low + high}{2} \right\rfloor$$

3. Compare $A[mid]$ with x:
   - If $A[mid] = x$, return mid as the position of x.
   - If $A[mid] < x$, set $low = mid + 1$ (search right half).
   - If $A[mid] > x$, set $high = mid - 1$ (search left half).
4. Repeat steps 2-3 until $low > high$ (element not found).

Step-by-step for $x = 35$:

Iteration	low	high	mid = floor((low + high)/2)	$A[mid]$	Comparison	Action
1	0	9	floor((0 + 9)/2) = 4	25	25 < 35	$low = mid + 1 = 5$
2	5	9	floor((5 + 9)/2) = 7	40	40 > 35	$high = mid - 1 = 6$
3	5	6	floor((5 + 6)/2) = 5	30	30 < 35	$low = mid + 1 = 6$
4	6	6	floor((6 + 6)/2) = 6	35	35 = 35	Found at index 6

Result:

$\boxed{\text{Element 35 found at index 6}}$

Support the statement that an optimization problem cannot be NP-complete whereas a decision problem can be NP-complete.

To support the statement:

"An optimization problem cannot be NP-complete, whereas a decision problem can be NP-complete,"
we need to understand the **definitions** and **requirements of NP-completeness**.

1. NP-complete problems

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

- It is in **NP** (i.e., a *yes* answer can be verified in polynomial time).
- It is **NP-hard** (i.e., every problem in NP can be reduced to it in polynomial time).

But this definition **only applies to decision problems**.

2. Optimization problem vs. Decision problem

- **Optimization problem**: Asks for the **best solution** (e.g., shortest path, maximum profit). Example: *What is the shortest route that visits every city once?* (Traveling Salesman Problem — Optimization version)
- **Decision problem**: Asks a **yes/no** question about the solution.
  Example: *Is there a route that visits every city once with total distance ≤ k?* (Traveling Salesman Problem — Decision version)

3. Why optimization problems cannot be NP-complete

- **Verification in polynomial time**: To be in NP, a problem must allow verification of a *yes* answer in polynomial time.
- **Optimization problems** do not have a *yes/no* answer; they require finding the best among many possible answers.
- Hence, they **do not fit into the NP framework**, because there's **no single answer to verify** in a yes/no fashion.

4. But decision problems can be NP-complete

- They fit the NP-completeness definition:
  - A "yes" solution can be checked quickly (in polynomial time).
  - Many hard problems reduce to them.

Conclusion

An **optimization problem cannot be NP-complete** because **NP-completeness only applies to decision problems**, which are **yes/no questions**. Optimization problems are often **NP-hard** but not **NP-complete**, because they don't fall within the **NP class** due to their output format

a. Analyse and outline the difference between dynamic programming and backtracking. (6 marks)

Difference Between Dynamic Programming and Backtracking

Aspect	**Dynamic Programming (DP)**	**Backtracking**
Purpose	Solves problems by breaking them into overlapping subproblems and storing results to avoid recomputation.	Explores all possible solutions by trying out all options and discarding invalid ones.
Approach	**Bottom-up** or **top-down with memoization**; builds solutions from smaller subproblems.	**Top-down**; incrementally builds candidates and abandons those that fail constraints.

Aspect	Dynamic Programming (DP)	Backtracking
Optimal Substructure	Requires the problem to have optimal substructure property (optimal solution composed of optimal subsolutions).	Does not necessarily require optimal substructure; explores all paths.
Overlapping Subproblems	Efficiently handles overlapping subproblems by caching intermediate results.	Does not reuse previously computed solutions; may recompute subproblems repeatedly.
Use case	Mainly used for optimization problems (e.g., shortest path, knapsack).	Used for constraint satisfaction problems and combinatorial search (e.g., puzzles, permutations).
Time complexity	Generally polynomial (if the problem has limited subproblems).	Potentially exponential, as it tries all possibilities unless pruned.
Example problems	Fibonacci series, matrix chain multiplication, knapsack problem.	N-Queens, Sudoku solver, generating permutations/combinations.

$$T(n) = T(n-1) + \log n$$

Q7. Solve the recurrence relation $a_n = a_{n-1} + n$ with initial term $a_0 = 4$.

Solve these questions by rivision of discrete mathematics, you can go through the notes of discrete mathematics and rivision the recurrence relation part, not a big deal, you know, you can go through only basics and you will be able to solve these questions.