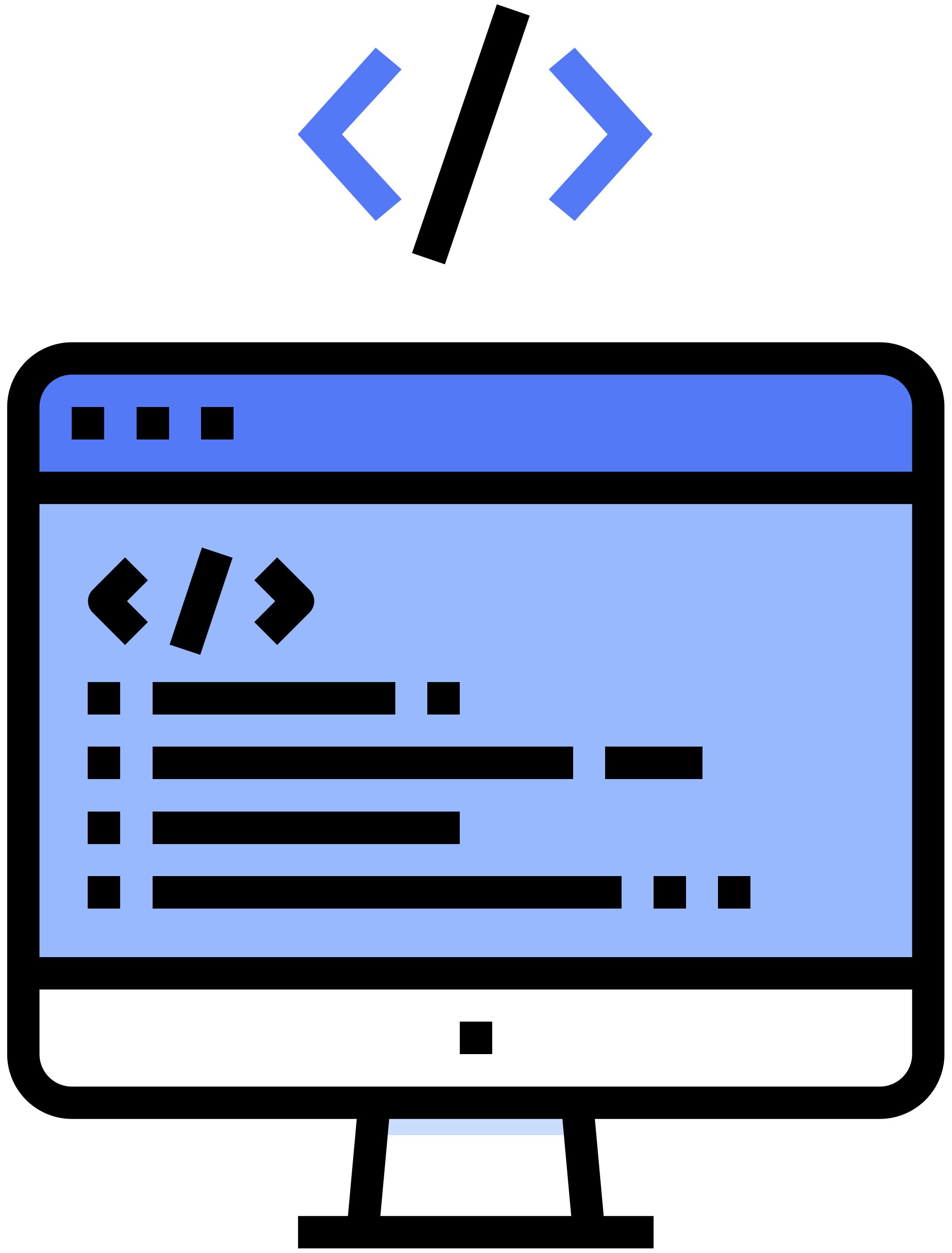
Session 10



INTRO TO ADVANCED DSA

Essential Problem-Solving

Anant Jain

WHY ADVANCED DSA?

Beyond Basic Operations:

- Sometimes simple loops or standard traversals aren't enough.

Optimization Problems:

- Finding the best solution (e.g., shortest path, maximum profit).

Combinatorial Problems:

- Finding all possible ways or a specific configuration (e.g., permutations, valid placements).

Efficiency:

- Solving problems with better time and space complexity, especially for large inputs.

The "Thinking" Process:

These methods teach you structured ways to break down and solve hard problems.

DYNAMIC PROGRAMMING (DP) - THE CORE IDEA

"Don't re-invent the wheel!"

Problem: Many problems have overlapping subproblems – you solve the same smaller problem multiple times.

DP Solution: Solve each subproblem once and store its result. When you encounter the same subproblem again, just look up the stored result.

Analogy: Imagine calculating a large number of Fibonacci numbers. Without DP, you re-calculate fib(3) many times. With DP, you calculate it once, store it, and reuse it.

DYNAMIC PROGRAMMING - KEY PROPERTIES

For a problem to be solvable with Dynamic Programming, it must have:

Optimal Substructure:

- An optimal solution to the problem contains optimal solutions to its subproblems.
- *Example*: The shortest path from A to C through B means the path from A to B must also be the shortest path from A to B.

Overlapping Subproblems:

- The same subproblems are encountered and solved repeatedly during a recursive solution.
- *Example*: Calculating fib(5) needs fib(4) and fib(3). fib(4) needs fib(3) and fib(2). Notice fib(3) is needed twice.

DYNAMIC PROGRAMMING - APPROACHES

V/S

Memoization (Top-Down DP):

- ***"Remembering"***
- Start with the main problem and recursively break it down.
- Store results of subproblems in a table (e.g., array, hash map) as you compute them.
- Before computing, check if the result is already in the table.
- **Think:** Recursive + Caching.

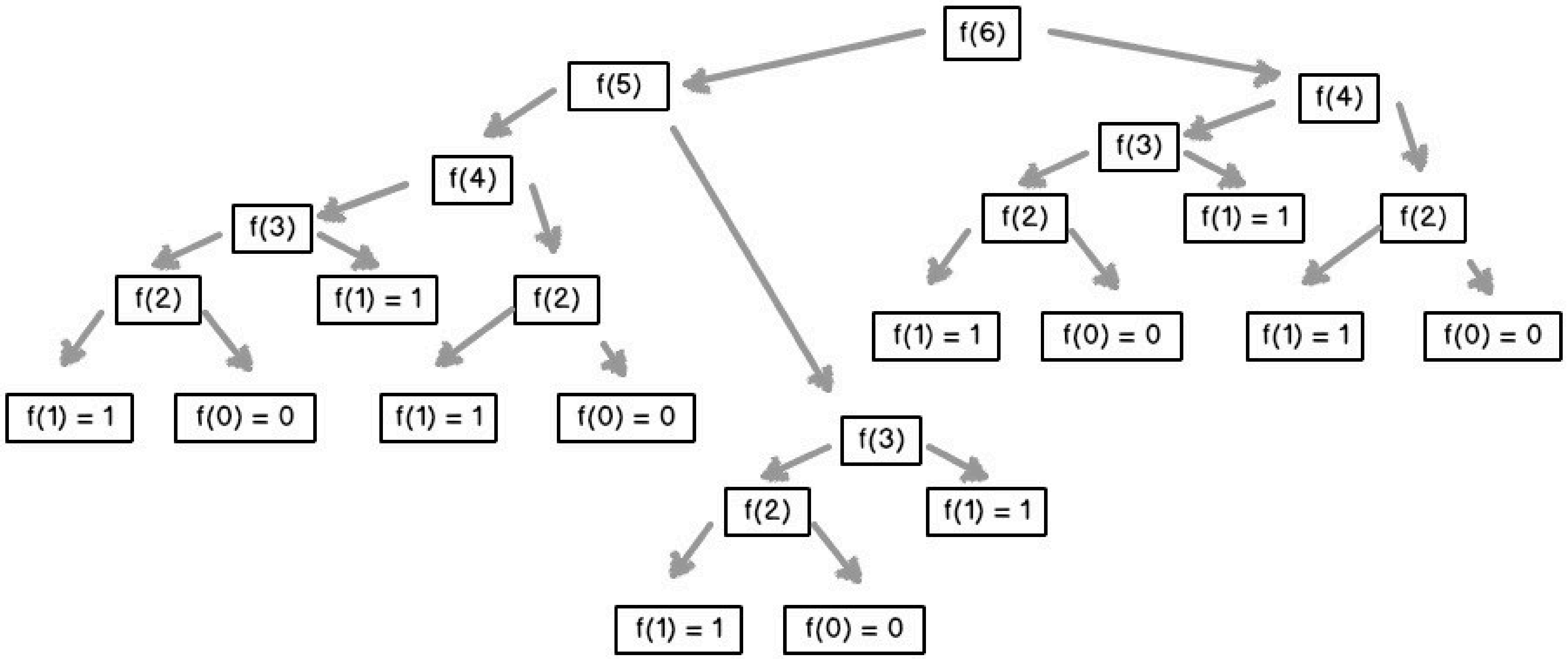
Tabulation (Bottom-Up DP):

- ***"Building Up"***
- Start with the smallest, simplest subproblems (base cases) and solve them.
- Iteratively build up solutions for larger subproblems using the results of smaller ones.
- **Think:** Iterative + Table Filling.

DP EXAMPLE: FIBONACCI NUMBERS (RECURSIVE TREE)

Problem:

Calculate the Nth Fibonacci number: $F(N)=F(N-1)+F(N-2)$, with $F(0)=0, F(1)=1$.



Notice how F(3), F(2), etc., are calculated multiple times!

DP EXAMPLE: FIBONACCI NUMBERS (MEMOIZATION/TABULATION CONCEPT)

TechnoCrats GSV

Memoization (Top-Down):

```
memo = {}

def fib(n):
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    result = fib(n - 1) + fib(n - 2)
    memo[n] = result
    return result
```

Tabulation (Bottom-Up):

```
def fib(N):
    dp = [0] * (N + 1)

    if N == 0:
        return 0
    dp[0] = 0
    dp[1] = 1

    for i in range(2, N + 1):
        dp[i] = dp[i - 1] + dp[i - 2]

    return dp[N]
```

Benefit:
Reduces time complexity from exponential to linear O(N)!

DP - PROBLEM SOLVING TIP

Ask yourself:

1. Can the problem be broken into smaller, similar subproblems?
2. Are these subproblems overlapping (do you solve the same one multiple times)?
3. Does an optimal solution to the large problem depend on optimal solutions to the smaller ones (optimal substructure)?

If YES to all: *DP is likely the way to go!*

Steps:

1. Define the state (parameters that uniquely identify a subproblem).
2. Write the recurrence relation.
3. Identify base cases.
4. Choose Memoization or Tabulation.

GREEDY ALGORITHMS - THE CORE IDEA

*"Take the best option available **RIGHT NOW**."*

Problem

Finding an *optimal solution*.

Greedy Solution

Make a locally optimal choice at each step, hoping that this sequence of local choices leads to a globally optimal solution.

Analogy

You're trying to reach a *mountain peak*. A greedy approach would be to always take the path that *goes steepest upwards* from your current position. **This might lead to the peak, or it might lead to a local maximum!**

GREEDY - WHEN IT WORKS (THE "GREEDY CHOICE PROPERTY")

Crucial Condition: A greedy algorithm only works if the problem exhibits the *"Greedy Choice Property."*

- This means that a globally optimal solution can be reached by making a locally optimal (greedy) choice.
- Once a greedy choice is made, it never needs to be undone.

No Universal Rule:
There's no general way to know if a greedy approach will work without proving the greedy choice property for that specific problem.

Simpler & Faster:
When it works, greedy algorithms are often simpler to implement and more efficient than DP or other approaches.

GREEDY EXAMPLE: ACTIVITY SELECTION PROBLEM

Problem: Given a set of activities, each with a start and finish time, select the maximum number of non-overlapping activities.

Greedy Strategy:

1. Sort activities by their finish times in ascending order.
2. Select the first activity.
3. Then, select the next activity that starts after the previously selected activity finishes.
4. Repeat until no more activities can be selected.

Why it works (Intuition):
Choosing the activity that finishes earliest leaves the maximum time available for subsequent activities.

GREEDY - WHEN IT FAILS: COIN CHANGE PROBLEM

Problem	Given a set of coin denominations and an amount, find the minimum number of coins to make that amount.
Denominations	Standard (1, 5, 10, 25 cents) - Greedy works (always take largest coin less than remaining amount).
Non-Standard Denominations	**(e.g., multiple 1, 3, 4 cents)** • *Amount:* 6 cents • *Greedy:* Take 4, then 1, then 1 (Total 3 coins: 4+1+1) • *Optimal:* Take 3, then 3 (Total 2 coins: 3+3)
Conclusion	Greedy fails here because the locally optimal choice (taking 4) prevents a globally optimal solution. This problem requires DP!

GREEDY
- PROBLEM SOLVING TIP

Ask yourself:

1. Does making the "best" immediate choice at each step seem to lead to the overall best solution?
2. Can you prove that this local choice will never prevent you from reaching the global optimum? (This is the hard part!)

If intuition suggests it:

- Try a greedy approach.

If it seems too complex or local choices might backfire:

- Greedy is probably not the answer. Consider DP or Backtracking.

Test with Counter-Examples:

- Always try to find a case where your greedy strategy fails. If you can't, it might be correct!

BACKTRACKING - THE CORE IDEA

*"Explore all paths, but **be smart about it.**"*

Problem

Finding all (or a specific) solution(s) by trying different combinations or configurations. Often involves constraints.

Backtracking Solution

- Build a solution incrementally, one step at a time.
- At each step, make a choice.
- If the current partial solution violates any constraints or cannot lead to a valid full solution, "backtrack" (undo the last choice) and try a different one.
- If a full, valid solution is found, record it.

Analogy

Navigating a maze. You try a path. If it hits a dead end, you go back to the last junction and try another path.

BACKTRACKING - STATE-SPACE TREE & DFS

State-Space Tree

- Conceptually, every possible sequence of choices forms a "state-space tree."
- Each node represents a partial solution.
- Edges represent choices.

Depth-First Search (DFS) on the Tree:

- Backtracking is essentially a DFS traversal of this state-space tree.
- When a path is found to be invalid or complete, the algorithm "backtracks" up the tree to explore other branches.

Pruning: The "smart" part is pruning branches early if they are guaranteed not to lead to a valid solution.

BACKTRACKING GENERAL APPROACH

(Pseudocode)

```
function solve(current_state):
    // 1. Base Case: Is the current_state a complete solution?
    if current_state is a complete solution:
        add current_state to list_of_solutions
        return

    // 2. Pruning: Is the current_state invalid or impossible to extend?
    if current_state is invalid:
        return

    // 3. Recursive Step: Try all possible choices from current_state
    for each choice in possible_choices_from(current_state):
        // Make the choice (add to current_state)
        apply_choice(current_state, choice)

        // Recurse to explore next step
        solve(current_state)

        // Backtrack: Undo the choice to explore other paths
        undo_choice(current_state, choice)
```

BACKTRACKING EXAMPLE:
N-QUEENS PROBLEM (CONCEPTUAL)

Problem: Place N non-attacking queens on an N x N chessboard.

Approach:

1. Start by placing a queen in the first row.
2. For each column in the current row:
   - Place a queen.
   - **Check for conflicts:** Is it attacked by any previously placed queens (same column, same diagonal)?
   - **If no conflict:** Recursively try to place a queen in the next row.
   - **If conflict OR next row fails:** Backtrack! Remove the queen from the current position and try the next column in the current row.

Visualization: Imagine trying to place queens one by one, and if you hit a wall (conflict), you pull back the last queen and try a different spot.

BACKTRACKING - PROBLEM SOLVING TIP

Ask yourself:

1. Do I need to find all possible combinations/permutations/arrangements that satisfy certain conditions?
2. Can I build the solution step-by-step?
3. Are there clear "invalid" states that I can identify early to stop exploring a path?

If YES:

Backtracking is a strong candidate!

Key:

Define your state, your choices at each step, and your pruning conditions. Remember to undo choices.

BRANCH AND BOUND - THE CORE IDEA

"Backtracking, but Smarter for ***OPTIMIZATION****."*

Problem

Finding the optimal solution (e.g., minimum cost, maximum profit) among many possibilities. [It's an extension of Backtracking]

Branch and Bound Solution

Branching: Same as backtracking – systematically explore the state-space tree.

Bounding: The crucial addition! At each partial solution (node in the tree), calculate a bound on the best possible solution achievable from that node.

- For minimization problems: Calculate a lower bound.
- For maximization problems: Calculate an upper bound.

Pruning: If the bound for a branch indicates that it cannot lead to a better solution than the best solution found so far, then that branch is "pruned" (cut off). You don't explore it further.

BRANCH AND BOUND
- KEY DIFFERENCE FROM BACKTRACKING

Backtracking: Prunes branches if they lead to an invalid solution.

Branch and Bound: Prunes branches if they lead to a solution that is not better than the current best known solution (even if it's valid).

Goal: Backtracking finds all (or a) valid solutions. Branch and Bound finds the optimal valid solution more efficiently by avoiding unnecessary exploration.

Data Structure Aid: Often uses a Priority Queue to explore the "most promising" branches first (though not strictly required for the core concept).

BRANCH AND BOUND EXAMPLE: TRAVELING SALESPERSON PROBLEM (TSP - CONCEPTUAL)

Problem: Find the shortest possible route that visits each city exactly once and returns to the origin city. (A classic NP-hard problem).

Benefit: Dramatically reduces the search space compared to brute-force or pure backtracking for optimization problems.

Branching:

Explore all possible paths (permutations of cities).

Bounding (Conceptual):

- As you build a path (e.g., City A -> City B -> ...), calculate the current path cost.
- Calculate a lower bound for the remaining path: e.g., sum of minimum cost edges from unvisited cities to complete the cycle.
- If (current_path_cost + lower_bound_for_remaining_path) is already greater than the best_solution_found_so_far, then prune this branch. It can't be better!

BRANCH AND BOUND - PROBLEM SOLVING TIP

Ask yourself:

1. Is this an optimization problem (finding min/max value)?
2. Can I define a way to calculate a bound (lower for min, upper for max) for any partial solution?
3. Is pure backtracking too slow because the search space is huge?

If YES:

Branch and Bound might be your answer.

Challenge:

Defining a tight and efficiently computable bound is key to its performance.

KEY TAKEAWAYS

Dynamic Programming:
Solve overlapping subproblems once, store results. Think optimal substructure.

Greedy Algorithms:
Make locally optimal choices. Works only if "greedy choice property" holds.

Backtracking:
Explore all possibilities incrementally, prune invalid paths early. "Try and undo."

Branch & Bound:
Backtracking for optimization. Prune branches that cannot lead to a better solution than the current best, using bounds.

Practice is Key!
Understanding these paradigms comes from applying them to various problems.

BEYOND THIS SESSION: SOME MORE TOPICS

Tries

Hashing

Bit Manipulation

Sliding Window Technique

Two Pointers Technique

Disjoint Set Union (DSU)

Shortest Path Algorithms
1. Bellman-Ford Algorithm,
2. Dijkstra's Algorithm

and more...

THANK YOU FOR LISTENING!

Refrences:

1. Dr. Shweta Ma'am Slides & Notes
2. Data Structures using C (Reema Thareja)