CORE DSA APPROACHES FOR SOLVING PROBLEMS:

1. 🪓 Brute Force — *Try Everything*

🧠 Concept

Brute force solves the problem by checking every possible solution. It’s exhaustive and guarantees correctness but is often inefficient for large inputs.

📚 Deep Theory

This method often uses:

- Nested loops
- Complete recursion
- Full state exploration (e.g., subsets, permutations)

Common in:

- Basic array problems
- First draft of problem-solving

🧪 Analogy

Like guessing a 4-digit pin by trying 0000, 0001...9999.

💡 Code Example: All pairs with sum = target

Cpp:

```cpp
void printPairs(vector<int>& arr, int target) {

    for(int i = 0; i < arr.size(); ++i)
        for(int j = i + 1; j < arr.size(); ++j)
            if(arr[i] + arr[j] == target)
                cout << arr[i] << " + " << arr[j] << "\n";
}
```

⏱️ Complexity

Time: $O(n^2)$

Space: $O(1)$

2. ⚡ Greedy — *Pick the Best Now*

🧠 Concept

Make the best decision at every step, hoping it leads to a globally optimal solution. It works when the problem exhibits a greedy-choice property.

📚 Deep Theory

It typically involves:

- Sorting based on a condition
- Selecting options based on maximum or minimum criteria
- No backtracking

🧪 Analogy

Like picking the tallest ladder rung you can reach, step by step.

💡 Code Example: Activity selection

Cpp:

```cpp
int maxActivities(vector<pair<int,int>>& meetings) {
    sort(meetings.begin(), meetings.end(), [](auto& a, auto& b) {
        return a.second < b.second;
    });

    int count = 1, end = meetings[0].second;
    for(int i = 1; i < meetings.size(); ++i)
        if(meetings[i].first > end) {
            ++count;
            end = meetings[i].second;
        }
    return count;
}
```

⏱️ Complexity

Time: O(n log n)

Space: O(1)

3. 🔀 Divide and Conquer — *Break, Solve, Combine*

🧠 Concept

Split the problem, solve each half recursively, then combine the results.

📚 Deep Theory

Applied in:

- Sorting (Merge Sort, Quick Sort)
- Searching (Binary Search)
- Complex recursive strategies

🧪 Analogy

Like breaking a chocolate bar into squares to eat one at a time.

💡 Code Example: Merge Sort

Cpp:
```cpp
void merge(vector<int>& arr, int l, int m, int r) {
    vector<int> temp;
    int i = l, j = m+1;
    while(i <= m && j <= r)
        temp.push_back(arr[i] <= arr[j] ? arr[i++] :
arr[j++]);
    while(i <= m) temp.push_back(arr[i++]);
    while(j <= r) temp.push_back(arr[j++]);

    for(int k = l; k <= r; ++k)
        arr[k] = temp[k - l];
}

void mergeSort(vector<int>& arr, int l, int r) {
    if(l >= r) return;
    int m = (l + r)/2;
    mergeSort(arr, l, m);
    mergeSort(arr, m+1, r);
    merge(arr, l, m, r);
}
```

⏱️ Complexity

Time: O(n log n)

Space: O(n)

4. 🔁 Recursion & Backtracking — *Explore and Undo*

🧠 Concept

Recursion solves problems by breaking them into smaller instances of the same problem. Backtracking builds decision trees and rolls back if a branch fails.

📚 Deep Theory

Backtracking checks all options but prunes invalid paths. Useful in:

- Constraint satisfaction (Sudoku, N-Queens)
- Combinatorics (subsets, permutations)

🧪 Analogy

Like undoing moves in a chess game when the path looks bad.

💡 Code Example: Generate subsets

Cpp:
```cpp
void generate(int i, vector<int>& nums, vector<int>& temp,
vector<vector<int>>& res) {
    if(i == nums.size()) {
        res.push_back(temp);
        return;
    }
    generate(i + 1, nums, temp, res);
    temp.push_back(nums[i]);
    generate(i + 1, nums, temp, res);
    temp.pop_back(); // backtrack
}
```

⏱️ Complexity

Time: $O(2^n)$

Space: O(n) recursion

5. 🧠 Dynamic Programming — *Solve Once, Reuse Always*

🧠 Concept

DP is used when a problem has overlapping subproblems and optimal substructure. Solutions are stored to avoid repeated work.

📚 Deep Theory

Two styles:

- Top-down: recursion + memoization
- Bottom-up: tabulation

Common problems:

- Fibonacci, Knapsack, LCS, LIS, Grid paths

🧪 Analogy

Remembering travel routes instead of checking the map each time.

💡 Code Example: Climb Stairs

Cpp:
```cpp
int climbStairs(int n) {
    if(n <= 1) return 1;
    int a = 1, b = 1;
    for(int i = 2; i <= n; ++i) {
        int temp = a + b;
        a = b;
        b = temp;
    }
    return b;
}
```

⏱️ Complexity

Time: O(n)

Space: O(1)

6. 👣 Two Pointers / Sliding Window — *Scan Efficiently*

🧠 Concept

Two pointers move across an array or string to form a range. Useful in optimizing nested loops.

📚 Deep Theory

Two Pointers:

- Best on sorted data Sliding Window:
- Track contiguous subarrays that meet a condition

🧪 Analogy

Reading a paragraph with two fingers tracking the start and end.

💡 Code Example: Longest substring without repeats

Cpp:
```
int lengthOfLongestSubstring(string s) {
    unordered_set<char> seen;
    int left = 0, maxLen = 0;
    for(int right = 0; right < s.size(); ++right) {
        while(seen.count(s[right]))
            seen.erase(s[left++]);
        seen.insert(s[right]);
        maxLen = max(maxLen, right - left + 1);
    }
    return maxLen;
}
```

⏱️ Complexity

Time: O(n)

Space: O(k)

7. 📉 Binary Search — *Divide the Range*

🧠 Concept

Search for a value or answer by repeatedly halving the range. Input must be sorted or follow a monotonic logic.

📚 Deep Theory

Applications:

- Index-based search
- Decision problems ("minimum time", "maximum distance")

🧪 Analogy

Guessing a number by asking "higher or lower?" after each guess.

💡 Code Example: Sqrt of n

Cpp:
```cpp
int sqrt(int x) {
    int low = 0, high = x, ans = 0;
    while(low <= high) {
        int mid = (low + high)/2;
        if(1LL * mid * mid <= x) {
            ans = mid;
            low = mid + 1;
        } else high = mid - 1;
    }
    return ans;
}
```

⏱️ Complexity

Time: O(log n)

Space: O(1)

8. 🧊 Hashing — *Quick Lookup and Count*

🧠 Concept

Store data using hash tables to achieve fast access, insertion, and deletion.

📚 Deep Theory

Hashing is used for:

- Frequency counting
- Duplicate detection
- Grouping (e.g. Anagrams)

🧪 Analogy

Jumping directly to a page in a dictionary using alphabetical index.

💡 Code Example: Two Sum

Cpp:
```
vector<int> twoSum(vector<int>& nums, int target) {
    unordered_map<int, int> map;
    for(int i = 0; i < nums.size(); ++i) {
        int complement = target - nums[i];
        if(map.count(complement))
            return {map[complement], i};
        map[nums[i]] = i;
    }
    return {};
}
```

⏱️ Complexity

Time: O(n)

Space: O(n)

9. 🌐 Graph Algorithms — *Follow Connections*

🧠 Concept

Graph theory represents relationships between objects. Graph algorithms traverse, search, and analyze these relationships.

📚 Deep Theory

Graph can be:

- Directed/Undirected
- Weighted/Unweighted

Core algorithms:

- BFS, DFS (Traversal)
- Dijkstra (Shortest Path)
- Kruskal/Prim (MST)
- Union-Find (Connected Components)
- Topological Sort (DAG ordering)

🧪 Analogy

Like navigating a subway map with various routes

💡 Code Example: BFS Traversal (Undirected Graph)

cpp
```
#include <iostream>
#include <vector>
#include <queue>

void bfs(int start, const std::vector<std::vector<int>>& adj,
std::vector<bool>& visited) {
    std::queue<int> q;
    q.push(start);
    visited[start] = true;

    while(!q.empty()) {
        int node = q.front();
        q.pop();
        std::cout << node << " ";

        for(int neighbor : adj[node]) {
            if(!visited[neighbor]) {
                visited[neighbor] = true;
                q.push(neighbor);
            }
        }
    }
}
```

✅ Usage Example

cpp
```cpp
int main() {
    int n = 6; // Number of nodes
    std::vector<std::vector<int>> adj(n);

    // Sample edges for undirected graph
    adj[0] = {1, 2};
    adj[1] = {0, 3};
    adj[2] = {0, 4};
    adj[3] = {1, 5};
    adj[4] = {2};
    adj[5] = {3};

    std::vector<bool> visited(n, false);
    bfs(0, adj, visited); // Output: 0 1 2 3 4 5

    return 0;
}
```

⏱️ Time & Space Complexity

Metric	Value
Time Complexity	O(V + E)
Space Complexity	O(V) (for visited + queue)