Volume 5: Greedy Algorithms & Problem-Solving Patterns

Table of Contents

Activity Selection, Interval Scheduling, Proof Techniques

Target: Infosys L3 Specialist Programmer
Edition: 2025
Focus: Q2 (Medium Greedy) - 35% weightage in coding round

Preface

Greedy algorithms are **disproportionately important** for Infosys L3:

- Q2 is consistently greedy (activity selection, interval scheduling, jump game variants)
- Faster to code than DP (~20 minutes vs 40+ minutes)
- **Proof of correctness** distinguishes strong candidates

Greedy Strategy: Make locally optimal choice at each step, hoping for global optimum.

Critical Skill: Recognizing when greedy works (exchange argument, staying ahead).

Table of Contents

Part I: Foundations

Chapter 1: Greedy Philosophy

1.1 When Greedy Works

Greedy works when:

1. **Greedy Choice Property**: Locally optimal choice leads to globally optimal solution
2. **Optimal Substructure**: Problem can be broken into subproblems

Example Where Greedy Works:

Coin Change (Standard Denominations): {1, 5, 10, 25} cents

Make 41 cents:

- Greedy: 25 + 10 + 5 + 1 = 4 coins ✓ (optimal)

Example Where Greedy Fails:

Coin Change (Arbitrary Denominations): {1, 3, 4} cents

Make 6 cents:

- Greedy: 4 + 1 + 1 = 3 coins
- Optimal: 3 + 3 = 2 coins ✗ (greedy failed)

Lesson: Greedy needs **proof** (not assumption).

1.2 Proof Techniques

Technique 1: Exchange Argument

Template:

1. Assume optimal solution differs from greedy
2. Show we can exchange a non-greedy choice with greedy choice
3. Prove exchange doesn't worsen solution
4. Contradiction: greedy must be optimal

Example: Activity Selection

Problem: Select maximum non-overlapping activities.

Greedy: Always pick earliest-ending activity.

Proof:

1. Let OPT be optimal solution not following greedy
2. Let a_g = greedy choice (earliest end), a_o = first activity in OPT
3. If $a_g \neq a_o$, replace a_o with a_g in OPT
4. Since a_g ends earlier, it doesn't conflict with remaining activities
5. New solution has same size as OPT → greedy is optimal

Technique 2: Staying Ahead

Template:

1. Show greedy solution "stays ahead" of any other solution at each step
2. At end, greedy is at least as good (usually better)

Example: Fractional Knapsack

Greedy: Always take item with highest value/weight ratio.

Proof: At each step, greedy maximizes value per unit weight, so total value is maximized.

Chapter 2: Pattern 1 - Activity Selection

2.1 Core Problem

Statement: Given start and end times of activities, select maximum non-overlapping activities.

Input:

```
start = [1, 3, 0, 5, 8, 5]
end   = [2, 4, 6, 7, 9, 9]
```

Output: 4 (activities at indices 0, 1, 3, 4)

Greedy Strategy: Sort by end time, always pick earliest-ending activity.

Why? Ending early leaves maximum room for future activities.

2.2 C++ Implementation

```
int activitySelection(vector&lt;int&gt;&amp; start, vector&lt;int&gt;&amp; end) {
    int n = start.size();
    vector&lt;pair&lt;int, int&gt;&gt; activities;

    for (int i = 0; i &lt; n; i++) {
        activities.push_back({end[i], start[i]});
    }

    sort(activities.begin(), activities.end()); // Sort by end time
```

```
    int count = 1;
    int lastEnd = activities[0].first;

    for (int i = 1; i &lt; n; i++) {
        if (activities[i].second &gt;= lastEnd) { // No overlap
            count++;
            lastEnd = activities[i].first;
        }
    }
    return count;
}
```

Time: O(n log n) (sorting dominates)
Space: O(n)

2.3 Python Implementation

```
def activitySelection(start, end):
    activities = sorted(zip(end, start))

    count = 1
    last_end = activities[0][0]

    for e, s in activities[1:]:
        if s &gt;= last_end:
            count += 1
            last_end = e

    return count
```

2.4 Variant: Return Selected Activities

```
vector&lt;int&gt; activitySelection(vector&lt;int&gt;&amp; start, vector&lt;int&gt;&amp; e
    int n = start.size();
    vector&lt;tuple&lt;int, int, int&gt;&gt; activities; // {end, start, index}

    for (int i = 0; i &lt; n; i++) {
        activities.push_back({end[i], start[i], i});
    }

    sort(activities.begin(), activities.end());

    vector&lt;int&gt; selected;
    selected.push_back(get&lt;2&gt;(activities[0]));
    int lastEnd = get&lt;0&gt;(activities[0]);

    for (int i = 1; i &lt; n; i++) {
        if (get&lt;1&gt;(activities[i]) &gt;= lastEnd) {
            selected.push_back(get&lt;2&gt;(activities[i]));
            lastEnd = get&lt;0&gt;(activities[i]);
        }
    }
    return selected;
}
```

Chapter 3: Pattern 2 - Interval Scheduling

3.1 Problem: Minimum Platforms Required (Infosys Favorite)

Statement: Given arrival and departure times of trains, find minimum platforms needed.

Input:

```
arr = [900, 940, 950, 1100, 1500, 1800]
dep = [910, 1200, 1120, 1130, 1900, 2000]
```

Output: 3 (at 950, platforms needed: 900-910, 940-1200, 950-1120)

Greedy Insight: Sort arrivals and departures separately. Use two pointers to simulate timeline.

C++ Solution:

```
int findPlatform(vector&lt;int&gt;&amp; arr, vector&lt;int&gt;&amp; dep) {
    sort(arr.begin(), arr.end());
    sort(dep.begin(), dep.end());

    int platforms = 0, maxPlatforms = 0;
    int i = 0, j = 0, n = arr.size();

    while (i &lt; n &amp;&amp; j &lt; n) {
        if (arr[i] &lt;= dep[j]) {
            platforms++; // Train arrives
            i++;
        } else {
            platforms--; // Train departs
            j++;
        }
        maxPlatforms = max(maxPlatforms, platforms);
    }
    return maxPlatforms;
}
```

Simulation:

```
Timeline: 900  910  940  950  1100  1120  1130  1200  1500  1800  1900  2000
         +1   -1   +1   +1    +1    -1    -1    -1    +1    +1    -1    -1
Platforms: 1    0    1    2     3     2     1     0     1     2     1     0
Max = 3
```

Time: O(n log n), **Space**: O(1)

3.2 Problem: Merge Intervals (Infosys Medium)

Statement: Merge overlapping intervals.

Input: [[1,3], [2,6], [8,10], [15,18]]
Output: [[1,6], [8,10], [15,18]]

Greedy: Sort by start time, merge if current overlaps with last merged.

C++ Solution:

```
vector&lt;vector&lt;int&gt;&gt; merge(vector&lt;vector&lt;int&gt;&gt;&amp; intervals) {
    if (intervals.empty()) return {};

    sort(intervals.begin(), intervals.end());
    vector&lt;vector&lt;int&gt;&gt; merged;

    for (auto&amp; interval : intervals) {
        if (merged.empty() || merged.back()[1] &lt; interval[0]) {
            merged.push_back(interval);
        } else {
            merged.back()[1] = max(merged.back()[1], interval[1]);
        }
    }
    return merged;
}
```

Edge Cases:

- Empty input
- Single interval
- All intervals overlap → one merged interval
- No overlaps → same as input

3.3 Problem: Non-Overlapping Intervals (Remove Minimum)

Statement: Remove minimum intervals to make rest non-overlapping.

Input: [[1,2], [2,3], [3,4], [1,3]]
Output: 1 (remove [1,3])

Greedy: Same as activity selection — select maximum non-overlapping, return total - selected.

C++ Solution:

```
int eraseOverlapIntervals(vector&lt;vector&lt;int&gt;&gt;&amp; intervals) {
    if (intervals.empty()) return 0;

    sort(intervals.begin(), intervals.end(),
         [](auto&amp; a, auto&amp; b) { return a[1] &lt; b[1]; });

    int count = 1;
    int end = intervals[0][1];

    for (int i = 1; i &lt; intervals.size(); i++) {
        if (intervals[i][0] &gt;= end) {
            count++;
            end = intervals[i][1];
        }
    }
    return intervals.size() - count;
}
```

Chapter 4: Pattern 3 - Jump Game Family

4.1 Jump Game I (Can Reach End?)

Statement: Each element = max jump length. Can you reach last index?

Input: [2, 3, 1, 1, 4]
Output: true (jump 0→1→4)

Input: [3, 2, 1, 0, 4]
Output: false (stuck at index 3)

Greedy: Track maximum reachable index.

C++ Solution:

```
bool canJump(vector&lt;int&gt;&amp; nums) {
    int maxReach = 0;

    for (int i = 0; i &lt; nums.size(); i++) {
        if (i &gt; maxReach) return false; // Can't reach i
        maxReach = max(maxReach, i + nums[i]);
        if (maxReach &gt;= nums.size() - 1) return true;
    }
    return true;
}
```

Why Greedy Works: If we can reach index i, we can reach any index ≤ i. So tracking max suffices.

Time: O(n), **Space**: O(1)

4.2 Jump Game II (Minimum Jumps)

Statement: Find minimum jumps to reach end.

Input: [2, 3, 1, 1, 4]
Output: 2 (0→1→4)

Greedy: Use BFS-like levels (jump boundaries).

C++ Solution:

```
int jump(vector&lt;int&gt;&amp; nums) {
    int jumps = 0, currentEnd = 0, farthest = 0;

    for (int i = 0; i &lt; nums.size() - 1; i++) {
        farthest = max(farthest, i + nums[i]);

        if (i == currentEnd) { // End of current level
            jumps++;
            currentEnd = farthest;
        }
    }
    return jumps;
}
```

Intuition: Track "levels" like BFS. When we reach end of current level, increment jumps.

Time: O(n), **Space**: O(1)

Chapter 5: Pattern 4 - Fractional Knapsack

5.1 Problem

Statement: Given weights, values, capacity W. Maximize value (can take fractions of items).

Input:

```
weights = [10, 20, 30]
values  = [60, 100, 120]
W = 50
```

Output: 240 (take full item 3 [120], full item 2 [100], 2/3 of item 1 [20])

Greedy: Always take highest value/weight ratio.

C++ Solution:

```
double fractionalKnapsack(vector&lt;int&gt;&amp; wt, vector&lt;int&gt;&amp; val, int W) {
    int n = wt.size();
    vector&lt;pair&lt;double, int&gt;&gt; items; // {ratio, index}

    for (int i = 0; i &lt; n; i++) {
        items.push_back({(double)val[i] / wt[i], i});
    }

    sort(items.rbegin(), items.rend()); // Descending ratio

    double totalValue = 0;
    int remainingWeight = W;

    for (auto&amp; [ratio, i] : items) {
        if (wt[i] &lt;= remainingWeight) {
            totalValue += val[i];
            remainingWeight -= wt[i];
        } else {
            totalValue += ratio * remainingWeight;
            break;
        }
    }
    return totalValue;
}
```

Time: O(n log n), **Space**: O(n)

Chapter 6: Pattern 5 - Huffman Coding

6.1 Concept

Problem: Assign variable-length codes to characters for minimum total length.

Greedy: Build tree by merging two smallest-frequency nodes repeatedly.

C++ Solution (simplified):

```
struct Node {
    char ch;
    int freq;
    Node *left, *right;
```

```
    Node(char c, int f) : ch(c), freq(f), left(nullptr), right(nullptr) {}
};

struct Compare {
    bool operator()(Node* a, Node* b) {
        return a-&gt;freq &gt; b-&gt;freq; // Min-heap
    }
};

void huffmanCodes(vector&lt;char&gt;&amp; chars, vector&lt;int&gt;&amp; freq) {
    priority_queue&lt;Node*, vector&lt;Node*&gt;, Compare&gt; pq;

    for (int i = 0; i &lt; chars.size(); i++) {
        pq.push(new Node(chars[i], freq[i]));
    }

    while (pq.size() &gt; 1) {
        Node* left = pq.top(); pq.pop();
        Node* right = pq.top(); pq.pop();

        Node* parent = new Node('\0', left-&gt;freq + right-&gt;freq);
        parent-&gt;left = left;
        parent-&gt;right = right;
        pq.push(parent);
    }

    Node* root = pq.top();
    // Traverse tree to generate codes (left=0, right=1)
}
```

Time: O(n log n), **Space**: O(n)

Chapter 7: Pattern 6 - Minimum Spanning Tree

7.1 Kruskal's Algorithm

Problem: Find MST (minimum total weight connecting all vertices).

Greedy: Sort edges by weight, add edge if doesn't create cycle (Union-Find).

C++ Solution:

```
class UnionFind {
    vector&lt;int&gt; parent, rank;
public:
    UnionFind(int n) : parent(n), rank(n, 0) {
        iota(parent.begin(), parent.end(), 0);
    }

    int find(int x) {
        if (parent[x] != x) parent[x] = find(parent[x]);
        return parent[x];
    }

    bool unite(int x, int y) {
        int px = find(x), py = find(y);
        if (px == py) return false;

        if (rank[px] &lt; rank[py]) parent[px] = py;
        else if (rank[px] &gt; rank[py]) parent[py] = px;
        else { parent[py] = px; rank[px]++; }
        return true;
```

```
    }
};

int kruskalMST(int n, vector&lt;vector&lt;int&gt;&gt;&amp; edges) {
    // edges = [[u, v, weight], ...]
    sort(edges.begin(), edges.end(),
         [](auto&amp; a, auto&amp; b) { return a[2] &lt; b[2]; });

    UnionFind uf(n);
    int mstWeight = 0, edgesUsed = 0;

    for (auto&amp; e : edges) {
        if (uf.unite(e[0], e[1])) {
            mstWeight += e[2];
            edgesUsed++;
            if (edgesUsed == n - 1) break; // MST complete
        }
    }
    return mstWeight;
}
```

Time: O(E log E), **Space**: O(V)

Chapter 8: Problem Recognition Framework

8.1 Greedy vs DP Decision Tree

```
Can locally optimal choice lead to global optimum?
├─ YES → Greedy
│   └─ Proof required (exchange argument)
│
└─ NO → Dynamic Programming
    └─ Overlapping subproblems + optimal substructure
```

Greedy Indicators:

- "Maximum/minimum number of..."
- Sorting helps
- Interval/scheduling problems
- Choice doesn't depend on future (only current state)

DP Indicators:

- "Count ways to..."
- "Maximize/minimize with constraints..."
- Need to consider multiple options at each step

8.2 Common Greedy Mistakes

Mistake 1: Assuming Greedy Without Proof

```
// WRONG: Using greedy for 0/1 knapsack (by value/weight ratio)
// Greedy fails for 0/1, works only for fractional

// Example where greedy fails:
// weights = [10, 20, 30], values = [60, 100, 120], W = 50
// Greedy (ratio): Take item 3 (4.0), item 2 (5.0) → 220
```

```
// Optimal: Take items 2, 3 → 220 (same here, but fails in general)

// Correct: Use DP for 0/1 knapsack
```

Mistake 2: Wrong Sorting Criterion

```
// WRONG: Sort activity selection by start time
sort(activities.begin(), activities.end(),
     [](auto&amp; a, auto&amp; b) { return a.start &lt; b.start; });

// Example where it fails:
// Activity 1: [1, 10]
// Activity 2: [2, 3]
// Activity 3: [4, 5]
// Sorting by start chooses [1,10], misses [2,3], [4,5]

// CORRECT: Sort by end time
```

Mistake 3: Not Handling Edge Cases

```
// WRONG: Not checking empty input
int maxActivities(vector&lt;int&gt;&amp; start, vector&lt;int&gt;&amp; end) {
    // Crashes if empty
}

// CORRECT:
int maxActivities(vector&lt;int&gt;&amp; start, vector&lt;int&gt;&amp; end) {
    if (start.empty()) return 0;
    ...
}
```

Chapter 9: Infosys Previous Year Greedy Problems

9.1 Problem Collection (2023-2025)

Problem 1: **Meeting Rooms II**

Given meeting time intervals, find minimum conference rooms required.

Input: [[0,30], [5,10], [15,20]]
Output: 2

Solution: Same as minimum platforms (sort start/end separately).

Problem 2: **Assign Cookies**

Given children greed factors and cookie sizes, maximize satisfied children.

Input: g = [1,2,3], s = [1,1]
Output: 1

Greedy: Sort both, assign smallest cookie satisfying child.

```
int findContentChildren(vector&lt;int&gt;&amp; g, vector&lt;int&gt;&amp; s) {
    sort(g.begin(), g.end());
    sort(s.begin(), s.end());

    int child = 0, cookie = 0;
    while (child &lt; g.size() &amp;&amp; cookie &lt; s.size()) {
```

```
        if (s[cookie] &gt;= g[child]) {
            child++;
        }
        cookie++;
    }
    return child;
}
```

Problem 3: **Gas Station**

Circular route with gas stations. Can you complete circuit?

Input: gas = [1,2,3,4,5], cost = [3,4,5,1,2]
Output: 3 (start at index 3)

Greedy: If total gas ≥ total cost, circuit exists. Find starting point.

```
int canCompleteCircuit(vector&lt;int&gt;&amp; gas, vector&lt;int&gt;&amp; cost) {
    int totalGas = 0, totalCost = 0, tank = 0, start = 0;

    for (int i = 0; i &lt; gas.size(); i++) {
        totalGas += gas[i];
        totalCost += cost[i];
        tank += gas[i] - cost[i];

        if (tank &lt; 0) { // Can't reach i+1 from start
            start = i + 1;
            tank = 0;
        }
    }
    return totalGas &gt;= totalCost ? start : -1;
}
```

Practice Problem Set (40 Greedy Problems)

Easy (15 problems)

1. Assign Cookies
2. Lemonade Change
3. Remove Duplicate Letters
4. Best Time to Buy Sell Stock II
5. Can Place Flowers

Medium (20 problems - Infosys Q2 Level)

6. Jump Game
7. Jump Game II
8. Minimum Platforms
9. Non-Overlapping Intervals
10. Merge Intervals
11. Partition Labels
12. Queue Reconstruction by Height
13. Task Scheduler

14. Boats to Save People
15. Minimum Number of Arrows

Hard (5 problems)

16. Candy
17. Minimum Number of Taps
18. Remove K Digits
19. Create Maximum Number
20. Patching Array

References

[1] CLRS. (2009). *Introduction to Algorithms* (3rd ed.). Chapter 16: Greedy Algorithms.

[2] PrepInsta. (2025). Infosys Greedy Problems. https://prepinsta.com/infosys-sp-and-dse/specialist-programmer/coding-questions/

[3] GeeksforGeeks. (2024). Activity Selection Problem. https://www.geeksforgeeks.org/activity-selection-problem-greedy-algo-1/