Infosys SP/DSE Complete Greedy Algorithm Mastery

Table of Contents

Volume D: Pure, Orthogonal & Relaxed Greedy - Theory to Expert

Target: Infosys L3/SP/DSE Question 2 (Medium Greedy)
Edition: 2025
Weightage: 35% of Coding Round
Difficulty: Codeforces 2200-2400

Preface

Question 2 in Infosys SP/DSE is **consistently greedy**. This volume provides:

- **Pure Greedy Algorithms**: Classic greedy (activity selection, fractional knapsack)
- **Orthogonal Greedy**: Multiple independent greedy choices
- **Relaxed Greedy**: Greedy with constraints/modifications

Critical: 60 minutes for Q2 is tight. Master templates for instant recognition.

Table of Contents

PART I: PURE GREEDY ALGORITHMS

Chapter 1: Activity Selection (Foundation)

1.1 Problem Statement

Classic Version: Select maximum number of non-overlapping activities.

Input:

- `start[]`: Start times of activities
- `end[]`: End times of activities

Output: Maximum activities that can be performed.

1.2 Greedy Strategy

Key Insight: Always select activity with **earliest end time**.

Proof by Exchange Argument:

Suppose optimal solution O includes activity A instead of greedy choice G, where `end[G] &lt; end[A]`.

Claim: We can replace A with G in O without reducing count.

Proof:

1. G ends before A
2. Any activity after A can also be scheduled after G
3. Replacing A with G maintains feasibility
4. Contradiction → Greedy is optimal

1.3 C++ Implementation

```
#include &lt;bits/stdc++.h&gt;
using namespace std;

struct Activity {
    int start, end, index;
};

bool compareByEnd(Activity a, Activity b) {
    return a.end &lt; b.end;
}

int maxActivities(vector&lt;int&gt;&amp; start, vector&lt;int&gt;&amp; end) {
    int n = start.size();
    vector&lt;Activity&gt; activities(n);

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

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

    int count = 1;
    int lastEnd = activities[0].end;

    for (int i = 1; i &lt; n; i++) {
        if (activities[i].start &gt;= lastEnd) {
            count++;
            lastEnd = activities[i].end;
        }
    }

    return count;
}

int main() {
    vector&lt;int&gt; start = {1, 3, 0, 5, 8, 5};
    vector&lt;int&gt; end = {2, 4, 6, 7, 9, 9};

    cout &lt;&lt; "Max activities: " &lt;&lt; maxActivities(start, end) &lt;&lt; endl;
    // Output: 4 (activities at indices 0, 1, 3, 4)

    return 0;
}
```

Time Complexity: O(n log n) — dominated by sorting
Space Complexity: O(n) — storing activities

1.4 Python Implementation

```
def max_activities(start, end):
    """
    Select maximum non-overlapping activities.

    Args:
        start: List of start times
        end: List of end times

    Returns:
        Maximum number of activities
    """
    # Create list of (end, start, index) tuples
    activities = sorted(zip(end, start, range(len(start))))

    count = 1
    last_end = activities[0][0]
    for end_time, start_time, idx in activities[1:]:
        if start_time &gt;= last_end:
            count += 1
            last_end = end_time

    return count

# Test<a></a>
start = [1, 3, 0, 5, 8, 5]
end = [2, 4, 6, 7, 9, 9]
print(f"Max activities: {max_activities(start, end)}")
# Output: 4<a></a>
```

1.5 Variations

Variation 1: Return Selected Activities

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

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

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

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

    for (int i = 1; i &lt; n; i++) {
        if (activities[i].start &gt;= lastEnd) {
            selected.push_back(activities[i].index);
            lastEnd = activities[i].end;
        }
    }

    return selected;
}
```

Variation 2: Weighted Activity Selection

Problem: Each activity has a weight/value. Maximize total value.

Approach: **NOT GREEDY** — requires Dynamic Programming.

```
// This is DP, not greedy
int maxWeightedActivities(vector&lt;int&gt;&amp; start, vector&lt;int&gt;&amp; end,
                          vector&lt;int&gt;&amp; weight) {
    int n = start.size();

    // Sort by end time
    vector&lt;tuple&lt;int, int, int, int&gt;&gt; activities; // {end, start, weight, index}
    for (int i = 0; i &lt; n; i++) {
        activities.push_back({end[i], start[i], weight[i], i});
    }
    sort(activities.begin(), activities.end());

    // DP: dp[i] = max weight using first i activities
    vector&lt;int&gt; dp(n + 1, 0);

    for (int i = 1; i &lt;= n; i++) {
        auto [e, s, w, idx] = activities[i-1];
        // Option 1: Don't include this activity
        dp[i] = dp[i-1];

        // Option 2: Include this activity
        // Find last compatible activity (binary search)
        int compatible = 0;
        for (int j = i - 1; j &gt;= 1; j--) {
            if (get&lt;0&gt;(activities[j-1]) &lt;= s) {
                compatible = j;
                break;
            }
        }

        dp[i] = max(dp[i], dp[compatible] + w);
    }

    return dp[n];
}
```

Time Complexity: $O(n^2)$ or $O(n \log n)$ with binary search

Variation 3: Minimum Removals

Problem: Minimum activities to remove to make rest non-overlapping.

Answer: Total activities - Maximum non-overlapping = Removals

```
int minRemove(vector&lt;int&gt;&amp; start, vector&lt;int&gt;&amp; end) {
    int n = start.size();
    int maxActivities = maxActivities(start, end);
    return n - maxActivities;
}
```

1.6 LeetCode Practice Problems

Problem	Difficulty	Pattern
LC 435: Non-Overlapping Intervals	Medium	Activity selection variant
LC 452: Minimum Arrows	Medium	Interval scheduling
LC 646: Maximum Chain Length	Medium	Activity selection
LC 1353: Maximum Events	Medium	Priority queue + greedy

Chapter 2: Fractional Knapsack

2.1 Problem Statement

Given n items with weights and values, and a knapsack capacity W:

- Can take **fractions** of items
- Maximize total value

Input:

- `weights[]`: Weight of each item
- `values[]`: Value of each item
- `W`: Knapsack capacity

Output: Maximum value achievable

2.2 Greedy Strategy

Key Insight: Always take items with **highest value-to-weight ratio** first.

Algorithm:

1. Compute `ratio[i] = value[i] / weight[i]` for each item
2. Sort items by ratio (descending)
3. Greedily take items until capacity full

2.3 C++ Implementation

```
#include &lt;bits/stdc++.h&gt;
using namespace std;

struct Item {
    int value, weight;
    double ratio;

    Item(int v, int w) : value(v), weight(w) {
        ratio = (double)value / weight;
    }
};

double fractionalKnapsack(vector&lt;int&gt;&amp; values, vector&lt;int&gt;&amp; weights, int W) {
    int n = values.size();
    vector&lt;Item&gt; items;

    for (int i = 0; i &lt; n; i++) {
        items.push_back(Item(values[i], weights[i]));
    }

    // Sort by ratio (descending)
    sort(items.begin(), items.end(),
         [](Item&amp; a, Item&amp; b) { return a.ratio &gt; b.ratio; });

    double totalValue = 0.0;
    int remainingCapacity = W;

    for (auto&amp; item : items) {
        if (remainingCapacity &gt;= item.weight) {
            // Take full item
            totalValue += item.value;
            remainingCapacity -= item.weight;
        } else {
            // Take fraction
            totalValue += item.ratio * remainingCapacity;
            break;
        }
    }

    return totalValue;
}

int main() {
    vector&lt;int&gt; values = {60, 100, 120};
    vector&lt;int&gt; weights = {10, 20, 30};
    int W = 50;

    cout &lt;&lt; "Max value: " &lt;&lt; fractionalKnapsack(values, weights, W) &lt;&lt; endl;
    // Output: 240 (take all item 3, all item 2, 2/3 of item 1)

    return 0;
}
```

Time Complexity: O(n log n)
Space Complexity: O(n)

2.4 Python Implementation

```
def fractional_knapsack(values, weights, W):
    """
    Fractional knapsack with greedy approach.

    Returns:
        Maximum value achievable
    """
    # Create list of (ratio, value, weight) tuples
    items = [(v/w, v, w) for v, w in zip(values, weights)]

    # Sort by ratio (descending)
    items.sort(reverse=True)

    total_value = 0.0
    remaining = W

    for ratio, value, weight in items:
        if remaining &gt;= weight:
            total_value += value
            remaining -= weight
        else:
            total_value += ratio * remaining
            break

    return total_value

# Test<a></a>
values = [60, 100, 120]
weights = [10, 20, 30]
W = 50
print(f"Max value: {fractional_knapsack(values, weights, W)}")
# Output: 240.0<a></a>
```

2.5 Why 0/1 Knapsack Needs DP

Critical Difference:

- **Fractional**: Can take fractions → Greedy works
- **0/1**: Must take whole item → Greedy FAILS

Counterexample (0/1 Knapsack):

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

Greedy (by ratio): Take item 3 (ratio=4), item 2 (ratio=5)
Value = 120 + 100 = 220

Optimal (DP): Take item 2 and item 3
Value = 100 + 120 = 220

Wait, same! Let's try another:

weights = [10, 20, 30]
values = [60, 100, 120]
W = 40

Greedy: Take item 3 (30kg, v=120), item 1 (10kg, v=60)
Value = 180
```

```
Optimal: Take item 2 (20kg, v=100) + item 1 (10kg, v=60) + ??
Actually can't fit item 3.
Value = 160

Hmm, greedy won here. Better example:

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

Greedy (by ratio):
- Item 2: ratio = 6
- Item 3: ratio = 4
- Item 1: ratio = 6
Take items 1 and 2: value = 180

Optimal: Items 2 and 3: value = 240

Greedy FAILS for 0/1!
```

Chapter 3: Jump Game Variants (Infosys Favorite)

3.1 Jump Game I (Can Reach End?)

Problem: Array where `nums[i]` = max jump from index i. Can you reach last index?

Input: `nums = [2, 3, 1, 1, 4]`
Output: `true` (0→1→4)

Greedy Approach

Insight: Track maximum reachable index. If current index > max reachable, return false.

```
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;
}
```

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

3.2 Jump Game II (Minimum Jumps)

Problem: Find minimum jumps to reach end.

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

BFS-Like Greedy

Insight: Track "levels" like BFS. Each jump creates a new level.

```
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;
}
```

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

3.3 Jump Game III (Reach Zero)

Problem: Start at index `start`. Jump left or right by `nums[i]` steps. Can you reach index with value 0?

Input: `nums = [4,2,3,0,3,1,2], start = 5`
Output: `true` (5→4→1→2→0)

BFS Approach

```
bool canReach(vector&lt;int&gt;&amp; nums, int start) {
    int n = nums.size();
    queue&lt;int&gt; q;
    vector&lt;bool&gt; visited(n, false);

    q.push(start);
    visited[start] = true;

    while (!q.empty()) {
        int pos = q.front();
        q.pop();

        if (nums[pos] == 0) return true;

        // Jump left
        int left = pos - nums[pos];
        if (left &gt;= 0 &amp;&amp; !visited[left]) {
            visited[left] = true;
            q.push(left);
        }

        // Jump right
        int right = pos + nums[pos];
        if (right &lt; n &amp;&amp; !visited[right]) {
            visited[right] = true;
            q.push(right);
        }
    }

    return false;
}
```

Chapter 4: Infosys-Specific Greedy Problems

4.1 Monster Defeat (RPG Game)

Full solution in Volume A

Summary:

- **Greedy**: Sort monsters by power (ascending)
- **Proof**: Exchange argument
- **Time**: O(n log n)

4.2 Maximum Vacation Days

Full solution in Volume B

Summary:

- **Greedy**: Sliding window on sorted obligations
- **Find**: Longest gap with ≤K obligations
- **Time**: O(M log M)

4.3 Binary String Ugliness

Full solution in Volume A

Summary:

- **Greedy Decision**: Swap vs flip based on cost
- **Strategy**: Minimize most significant bits first
- **Time**: O(n)

Practice Problem Set (40 Greedy Problems)

Easy (15 problems)

1. ✓ LC 455 - Assign Cookies
2. ✓ LC 860 - Lemonade Change
3. ✓ LC 1005 - Maximize Sum After K Negations
4. ✓ CF 1141A - Game 23
5. ✓ CF 1196A - Three Piles of Hay

... [15 total]

Medium (20 problems)

16. ✓ LC 435 - Non-Overlapping Intervals
17. ✓ LC 452 - Minimum Arrows
18. ✓ LC 55 - Jump Game
19. ✓ LC 45 - Jump Game II
20. ✓ LC 56 - Merge Intervals

... [20 total]

Hard (5 problems)

36. ✓ LC 135 - Candy
37. ✓ LC 502 - IPO
38. ✓ LC 630 - Course Schedule III
39. ✓ LC 1326 - Minimum Taps
40. ✓ CF 1462E - Close Tuples

References

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

[2] PrepInsta. (2025). Infosys SP Greedy Problems.

[3] LeetCode Greedy Tag Problems (2025)