Complete Greedy Algorithms Guide

1. Activity Selection (Interval Scheduling)

Objective

Select the maximum number of non-overlapping activities from a given set of activities, each having start and finish times.

What is this?

Given activities with start and finish times, we need to choose the largest subset such that no two activities overlap in time. An activity that finishes at time t can be followed by an activity that starts at time t or later.

Why use Greedy?

The greedy approach works because choosing the activity that finishes earliest leaves the maximum room for future selections. This can be proven using an exchange argument - any optimal solution can be transformed to include our greedy choice without making it worse.

Full Problem Statement

Activity Selection Problem

You are a manager of a conference hall. You have received n requests for organizing events. Each event has a start time and an end time. You can only organize one event at a time, and events cannot overlap. Find the maximum number of events you can organize.

Input:

- First line: n (number of events)
- Next n lines: start_time end_time for each event

Example:

```
5
1 3
2 4
3 5
0 7
```

```
5 7
```

Solution Procedure

1. **Sort intervals by increasing finish time** - This ensures we always pick activities that end earliest
2. **Initialize** `last_end = -∞` to track when last selected activity ends
3. **Iterate through sorted intervals:**
    - If current activity's start time ≥ last_end, select it
    - Update last_end to current activity's end time
4. **Count selected activities** as the answer

Solution Technique (Greedy Choice Property)

The greedy choice is to always pick the activity that finishes earliest among remaining activities. This works because:

- **Greedy Choice Property:** There exists an optimal solution that includes the earliest-finishing activity
- **Optimal Substructure:** After picking an activity, the remaining problem is independent
- **Exchange Argument:** Any optimal solution can be modified to use our greedy choice without losing optimality

Time Complexity: O(n log n) due to sorting

Dry Run

Input: [(1,3), (2,4), (3,5), (0,7), (5,7)]

Step 1: Sort by end time: [(1,3), (2,4), (3,5), (5,7), (0,7)]

Step 2: Process each interval:

- last_end = -∞
- (1,3): start=1 ≥ -∞ → SELECT, last_end = 3
- (2,4): start=2 < 3 → SKIP
- (3,5): start=3 ≥ 3 → SELECT, last_end = 5

- (5,7): start=5 ≥ 5 → SELECT, last_end = 7
- (0,7): start=0 < 7 → SKIP

Result: Selected [(1,3), (3,5), (5,7)] = 3 activities

2. Fractional Knapsack

Objective

Maximize the total value of items that can be put into a knapsack of given capacity, where items can be broken into fractions.

What is this?

Given items with values and weights, and a knapsack with limited capacity, fill the knapsack to maximize total value. Unlike 0/1 knapsack, we can take fractions of items.

Why use Greedy?

For fractional knapsack, the greedy approach of taking items in order of value/weight ratio is optimal. This doesn't work for 0/1 knapsack where items must be taken whole.

Full Problem Statement

Fractional Knapsack Problem

You are a thief robbing a store. You have a knapsack that can carry a maximum weight W. There are n items in the store, each with a value and weight. You can take fractions of items. Maximize the total value you can steal.

Input:

- First line: n (items), W (capacity)
- Next n lines: value weight for each item

Example:

```
3 50
60 10
100 20
```

```
120 30
```

Solution Procedure

1. **Calculate value/weight ratio** for each item
2. **Sort items by ratio in descending order**
3. **Initialize** current_weight = 0, total_value = 0
4. **For each item in sorted order:**
    - If item fits completely, take it all
    - If item doesn't fit completely, take fraction that fits
    - Update current_weight and total_value
5. **Return total_value**

Solution Technique (Ratio-based Greedy)

The greedy choice is to always take the item with highest value/weight ratio first. This works because:

- **Greedy Choice Property:** Taking the highest ratio item first is always part of an optimal solution
- **Optimal Substructure:** After taking an item (or fraction), remaining problem is independent
- **Mathematical Proof:** Any deviation from this order can be improved by swapping items

Time Complexity: O(n log n) due to sorting

Dry Run

Input: Items [(60,10), (100,20), (120,30)], W=50

Step 1: Calculate ratios: [6, 5, 4]

Step 2: Sort by ratio: [(60,10), (100,20), (120,30)]

Step 3: Process items:

- W_remaining = 50, total_value = 0
- Item (60,10): Take fully → W_remaining = 40, total_value = 60

- Item (100,20): Take fully → W_remaining = 20, total_value = 160
- Item (120,30): Take 20/30 fraction → W_remaining = 0, total_value = 160 + (20/30)*120 = 240

Result: Maximum value = 240

3. Canonical Coin Change

Objective

Find the minimum number of coins needed to make change for a given amount using a canonical coin system (like US coins).

What is this?

Given coin denominations and an amount, find the minimum coins needed. The greedy approach only works for "canonical" coin systems where taking the largest coin first is always optimal.

Why use Greedy?

For canonical systems (like 1,5,10,25), the greedy approach is optimal. However, for non-canonical systems (like 1,3,4), greedy may fail and dynamic programming is needed.

Full Problem Statement

Coin Change Problem (Canonical Systems Only)

You are a cashier making change. Given coin denominations and an amount to make change for, find the minimum number of coins needed. Assume the coin system is canonical (greedy approach works).

Input:

- First line: n (number of denominations)
- Second line: denomination values
- Third line: amount to make change for

Example:

```
4
25 10 5 1
93
```

Solution Procedure

1. **Sort denominations in descending order**
2. **Initialize** coins_used = 0
3. **For each denomination from largest to smallest:**
    - Use as many coins of this denomination as possible
    - Update remaining amount and coins_used
4. **If amount becomes 0, return coins_used, else impossible**

Solution Technique (Largest-first Greedy)

The greedy choice is to always use the largest denomination possible. This works for canonical systems because:

- **Canonical Property:** The coin system is designed so that greedy approach is optimal
- **Greedy Choice Property:** Using largest coin first never prevents an optimal solution
- **Counterexample for non-canonical:** For coins [1,3,4] and amount 6, greedy gives [4,1,1] = 3 coins, but optimal is [3,3] = 2 coins

Time Complexity: O(d) where d is number of denominations

Dry Run

Input: Denominations [25,10,5,1], Amount = 93

Step 1: Process largest to smallest:

- amount = 93, coins = 0
- Use 25: 93/25 = 3 coins, remainder = 93-75 = 18, coins = 3
- Use 10: 18/10 = 1 coin, remainder = 18-10 = 8, coins = 4
- Use 5: 8/5 = 1 coin, remainder = 8-5 = 3, coins = 5
- Use 1: 3/1 = 3 coins, remainder = 3-3 = 0, coins = 8

Result: 8 coins total (3×25 + 1×10 + 1×5 + 3×1)

4. Connect Ropes (Minimum Total Cost)

Objective

Connect n ropes with minimum cost, where the cost of connecting two ropes is the sum of their lengths.

What is this?

Given lengths of n ropes, connect them all into one rope. Each connection operation takes two ropes and creates one rope of combined length, costing the sum of the two lengths. Minimize total cost.

Why use Greedy?

Always connecting the two shortest ropes first minimizes the total cost. This is because shorter ropes will be involved in more operations, so keeping them small reduces overall cost.

Full Problem Statement

Connect Ropes Problem

You have n ropes of different lengths. You need to connect them all into one rope. The cost of connecting two ropes is equal to the sum of their lengths. Find the minimum cost to connect all ropes.

Input:

- First line: n (number of ropes)
- Second line: lengths of n ropes

Example:

```
4
4 3 2 6
```

Solution Procedure

1. **Create a min-heap** with all rope lengths

2. **While heap has more than one element:**
    - Extract two smallest lengths
    - Add their sum to total cost
    - Insert the combined length back into heap
3. **Return total cost**

Solution Technique (Optimal Merge Pattern)

The greedy choice is to always merge the two smallest ropes. This works because:

- **Greedy Choice Property:** The two smallest ropes should be merged together in some optimal solution
- **Optimal Substructure:** After merging, the remaining problem is independent
- **Mathematical Proof:** Any other merging order will result in higher or equal cost

Time Complexity: O(n log n) due to heap operations

Dry Run

Input: [4, 3, 2, 6]

Step 1: Create min-heap: [2, 3, 4, 6]

Step 2: Process merges:

- cost = 0, heap = [2, 3, 4, 6]
- Merge 2+3=5, cost = 5, heap = [4, 5, 6]
- Merge 4+5=9, cost = 5+9=14, heap = [6, 9]
- Merge 6+9=15, cost = 14+15=29, heap = [15]

Result: Minimum cost = 29

5. Boats to Save People (Two-pointer Pairing)

Objective

Minimize the number of boats needed to rescue all people, where each boat can carry at most 2 people and has a weight limit.

What is this?

Given weights of people and boat weight limit, find minimum boats needed. Each boat carries at most 2 people, and total weight must not exceed limit.

Why use Greedy?

The heaviest person must go in some boat. Pairing them with the lightest person who fits maximizes efficiency and never hurts future pairings.

Full Problem Statement

Boats to Save People Problem

There are n people waiting to be rescued by boats. Each boat can carry at most 2 people and has a weight limit. Find the minimum number of boats needed to save everyone.

Input:

- First line: n (people), limit (weight limit per boat)
- Second line: weights of n people

Example:

```
4 3
3 2 2 1
```

Solution Procedure

1. **Sort people by weight in ascending order**
2. **Initialize** two pointers: left=0 (lightest), right=n-1 (heaviest)
3. **Initialize** boats = 0
4. **While left ≤ right:**
    - If weight[left] + weight[right] ≤ limit, move both pointers inward

    - Else, move only right pointer inward (send heaviest alone)
    - Increment boats count
5. **Return boats**

Solution Technique (Two-pointer Greedy)

The greedy choice is to try pairing the heaviest remaining person with the lightest. This works because:

- **Greedy Choice Property:** The heaviest person must go in some boat, pairing with lightest maximizes boat utilization
- **Optimal Substructure:** After sending a boat, remaining problem is independent
- **Pairing Principle:** If heaviest can't pair with lightest, it can't pair with anyone heavier

Time Complexity: O(n log n) due to sorting

Dry Run

Input: weights = [3, 2, 2, 1], limit = 3

Step 1: Sort: [1, 2, 2, 3]

Step 2: Two pointers approach:

- boats = 0, left = 0, right = 3
- weights[0] + weights[3] = 1 + 3 = 4 > 3 → send person 3 alone, boats = 1, right = 2
- weights[0] + weights[2] = 1 + 2 = 3 ≤ 3 → send both, boats = 2, left = 1, right = 1
- left == right → send person at index 1 alone, boats = 3

Result: 3 boats needed

6. Gas Station Tour Start

Objective

Find a starting gas station that allows completing a circular tour, visiting all stations exactly once.

What is this?

Given gas available at each station and gas needed to reach the next station, find a starting position that allows completing the full circular tour.

Why use Greedy?

The greedy approach uses the insight that if we fail to complete the tour starting from position i, then no position between i and the failure point can be a valid start.

Full Problem Statement

Gas Station Problem

There are n gas stations arranged in a circle. You have a car with unlimited fuel capacity. At station i, you can get gas[i] amount of fuel, and it costs cost[i] fuel to travel to the next station. Find a starting station that allows you to complete the circular tour, or return -1 if impossible.

Input:

- First line: n (number of stations)
- Second line: gas available at each station
- Third line: cost to reach next station from each station

Example:

```
5
1 2 3 4 5
3 4 5 1 2
```

Solution Procedure

1. **Check feasibility:** If sum(gas) < sum(cost), return -1
2. **Initialize** start = 0, tank = 0
3. **For each station i from 0 to n-1:**
    - Add gas[i] - cost[i] to tank
    - If tank < 0, set start = i+1 and reset tank = 0

4. **Return start**

Solution Technique (Greedy Reset)

The greedy insight is that if we can't reach station j starting from station i, then no station between i and j-1 can be a valid start. This works because:

- **Greedy Choice Property:** When we fail, we reset to the next possible start
- **Elimination Principle:** Failure from i to j eliminates all starts from i to j-1
- **Global Feasibility:** If total gas ≥ total cost, some start must exist

Time Complexity: O(n) single pass

Dry Run

Input: gas = [1,2,3,4,5], cost = [3,4,5,1,2]

Step 1: Check feasibility: sum(gas) = 15, sum(cost) = 15 → feasible

Step 2: Process each station:

- start = 0, tank = 0
- Station 0: tank = 0 + 1 - 3 = -2 < 0 → start = 1, tank = 0
- Station 1: tank = 0 + 2 - 4 = -2 < 0 → start = 2, tank = 0
- Station 2: tank = 0 + 3 - 5 = -2 < 0 → start = 3, tank = 0
- Station 3: tank = 0 + 4 - 1 = 3 ≥ 0 → continue
- Station 4: tank = 3 + 5 - 2 = 6 ≥ 0 → continue

Result: Start from station 3

7. Jump Game I & II

Objective

Jump Game I: Determine if you can reach the last index of an array where each element represents maximum jump length.
Jump Game II: Find minimum jumps needed to reach the last index.

What is this?

Given an array where each element represents the maximum jump length from that position, solve reachability (I) or minimum jumps (II) to reach the end.

Why use Greedy?

For Jump Game I, we greedily track the farthest reachable position. For Jump Game II, we use level-by-level expansion to minimize jumps.

Full Problem Statement

Jump Game Problem

You are standing at the first index of an array. Each element represents your maximum jump length at that position.

- **Part I:** Determine if you can reach the last index
- **Part II:** Find the minimum number of jumps to reach the last index

Input:

- First line: n (array length)
- Second line: maximum jump lengths

Example:

```
5
2 3 1 1 4
```

Solution Procedure

Jump Game I (Reachability)

1. **Initialize** farthest = 0

2. **For each position i from 0 to n-1:**
    - If i > farthest, return false (unreachable)
    - Update farthest = max(farthest, i + nums[i])
3. **Return true** (last index is reachable)

Jump Game II (Minimum Jumps)

1. **Initialize** jumps = 0, current_end = 0, farthest = 0
2. **For each position i from 0 to n-2:**
    - Update farthest = max(farthest, i + nums[i])
    - If i == current_end (reached end of current level):
        - Increment jumps
        - Set current_end = farthest (expand to next level)
3. **Return jumps**

Solution Technique (Greedy Range Expansion)

Jump Game I: Greedily track maximum reachable position

Jump Game II: Use BFS-like level expansion, but greedily expand each level to its maximum reach

Time Complexity: O(n) for both

Dry Run

Input: [2, 3, 1, 1, 4]

Jump Game I:

- i=0: farthest = max(0, 0+2) = 2
- i=1: farthest = max(2, 1+3) = 4
- i=2: farthest = max(4, 2+1) = 4
- i=3: farthest = max(4, 3+1) = 4
- i=4: farthest = max(4, 4+4) = 8
- Result: True (can reach index 4)

Jump Game II:

- jumps = 0, current_end = 0, farthest = 0
- i=0: farthest = max(0, 0+2) = 2, i==current_end → jumps = 1, current_end = 2
- i=1: farthest = max(2, 1+3) = 4
- i=2: farthest = max(4, 2+1) = 4, i==current_end → jumps = 2, current_end = 4
- Result: 2 jumps minimum

8. Minimum Platforms

Objective

Find the minimum number of railway platforms required so that no train waits, given arrival and departure times.

What is this?

Given arrival and departure times of trains, find minimum platforms needed so all trains can be accommodated without any waiting.

Why use Greedy?

We can use a sweep line algorithm to track platform occupancy. At any time, the number of trains present equals arrivals so far minus departures so far.

Full Problem Statement

Railway Platform Problem

A railway station needs to accommodate trains with given arrival and departure times. Find the minimum number of platforms required so that no train has to wait.

Input:

- First line: n (number of trains)
- Second line: arrival times of n trains

- Third line: departure times of n trains

Example:

```
6
900 940 950 1100 1500 1800
910 1200 1120 1130 1900 2000
```

Solution Procedure

1. **Sort arrival times and departure times separately**
2. **Initialize** platforms_needed = 0, max_platforms = 0
3. **Initialize** two pointers: i = 0 (arrivals), j = 0 (departures)
4. **While both pointers are valid:**
    - If arrival[i] ≤ departure[j]:
        - A train arrives before next departure → increment platforms_needed, advance i
    - Else:
        - A train departs → decrement platforms_needed, advance j
    - Update max_platforms = max(max_platforms, platforms_needed)
5. **Return max_platforms**

Solution Technique (Sweep Line Algorithm)

The greedy approach processes events (arrivals/departures) in chronological order:

- **Greedy Choice Property:** Process earliest event first (arrival or departure)
- **Platform Tracking:** Current platforms needed = arrivals processed - departures processed
- **Optimal Substructure:** Maximum over all time points gives the answer

Time Complexity: O(n log n) due to sorting

Dry Run

Input: arrivals = [900,940,950,1100,1500,1800], departures = [910,1200,1120,1130,1900,2000]

Step 1: Sort separately:

- arrivals: [900,940,950,1100,1500,1800]
- departures: [910,1120,1130,1200,1900,2000]

Step 2: Sweep through events:

- platforms = 0, max_platforms = 0, i = 0, j = 0
- 900 ≤ 910: arrival → platforms = 1, max_platforms = 1, i = 1
- 940 ≤ 910: false → departure → platforms = 0, j = 1
- 940 ≤ 1120: arrival → platforms = 1, i = 2
- 950 ≤ 1120: arrival → platforms = 2, max_platforms = 2, i = 3
- 1100 ≤ 1120: arrival → platforms = 3, max_platforms = 3, i = 4
- 1500 ≤ 1120: false → departure → platforms = 2, j = 2
- Continue...

Result: 3 platforms needed

9. Partition Labels

Objective

Partition a string into as many parts as possible so that each letter appears in at most one part.

What is this?

Given a string, split it into maximum number of substrings such that each character appears in at most one substring.

Why use Greedy?

We can greedily extend each partition to include the last occurrence of every character seen so far, then cut when we reach that boundary.

Full Problem Statement

String Partition Problem

Given a string, partition it into as many parts as possible so that each letter appears in at most one part. Return the sizes of these parts.

Input:

- String s

Example:

```
ababcbacadefegdehijhklij
```

Solution Procedure

1. **Precompute last occurrence** index for each character
2. **Initialize** start = 0, end = 0
3. **For each character at position i:**
    - Update end = max(end, last_occurrence[character])
    - If i == end (reached the boundary):
        - Record partition size = end - start + 1
        - Set start = i + 1 for next partition
4. **Return partition sizes**

Solution Technique (Greedy Boundary Expansion)

The greedy choice is to extend current partition to include last occurrence of every character seen:

- **Greedy Choice Property:** Cut partition as early as possible while maintaining constraint
- **Boundary Principle:** Must include last occurrence of every character in current partition
- **Optimal Substructure:** After cutting, remaining problem is independent

Time Complexity: O(n) single pass after preprocessing

Dry Run

Input: "ababcbacadefegdehijhklij"

Step 1: Compute last occurrences:

- a:8, b:5, c:7, d:14, e:15, f:11, g:13, h:19, i:22, j:23, k:20, l:21

Step 2: Process characters:

- start = 0, end = 0
- i=0 'a': end = max(0,8) = 8
- i=1 'b': end = max(8,5) = 8
- i=2 'a': end = max(8,8) = 8
- i=3 'b': end = max(8,5) = 8
- i=4 'c': end = max(8,7) = 8
- i=5 'b': end = max(8,5) = 8
- i=6 'a': end = max(8,8) = 8
- i=7 'c': end = max(8,7) = 8
- i=8 'a': end = max(8,8) = 8, i==end → partition size = 9, start = 9
- Continue for remaining partitions...

Result: Partition sizes = [9, 7, 8]

10. Lemonade Change

Objective

Determine if you can provide correct change to all customers in a lemonade stand, given the constraint of available bill denominations.

What is this?

Customers buy lemonade for $5 and may pay with $5, $10, or $20 bills. You must give correct change and determine if this is possible for all customers.

Why use Greedy?

Always give change using the largest possible bills to preserve flexibility for future customers.

Full Problem Statement

Lemonade Stand Problem

You sell lemonade for $5 each. Customers pay with $5, $10, or $20 bills. You start with no money. For each customer, you must give correct change or determine it's impossible.

Input:

- Array of bills received from customers in order

Example:

```
[5,5,5,10,20]
```

Solution Procedure

1. **Initialize** count of $5 and $10 bills to 0
2. **For each bill received:**
    - If $5: increment count of $5 bills
    - If $10: need $5 change → check if available, give change, update counts
    - If $20: need $15 change → prefer one $10 + one $5, else three $5 bills
    - If cannot give change, return false
3. **Return true if all customers served**

Solution Technique (Greedy Change-Making)

The greedy choice is to give change using largest bills first:

- **Greedy Choice Property:** Using larger bills preserves more flexibility
- **Local Feasibility:** Check if current change is possible with available bills
- **Constraint Maintenance:** Always maintain valid bill counts

Time Complexity: O(n) single pass

Dry Run

Input: [5,5,5,10,20]

Step 1: Process each bill:

- fives = 0, tens = 0
- Bill $5: fives = 1
- Bill $5: fives = 2
- Bill $5: fives = 3
- Bill $10: need $5 change → fives ≥ 1? yes → fives = 2, tens = 1
- Bill $20: need $15 change → tens ≥ 1 and fives ≥ 1? yes → fives = 1, tens = 0

Result: True (all customers can be served)

11. Remove K Digits

Objective

Given a non-negative integer and an integer k, remove k digits from the number to make it as small as possible.

What is this?

Remove exactly k digits from a number to create the lexicographically smallest possible result.

Why use Greedy?

Use a monotonic stack to ensure we always remove the earliest larger digit when we encounter a smaller one, minimizing the final number.

Full Problem Statement

Remove K Digits Problem

Given a non-negative integer represented as a string and an integer k, remove k digits from the number so that the new number is the smallest possible.

Input:

- String num representing the number

- Integer k (digits to remove)

Example:

```
num = "1432219"
k = 3
```

Solution Procedure

1. **Initialize** empty stack for building result
2. **For each digit in the number:**
    - While stack not empty AND top > current digit AND k > 0:
        - Pop from stack (remove larger digit)
        - Decrement k
    - Push current digit to stack
3. **If k > 0 after processing:** remove k digits from end
4. **Remove leading zeros** and return result (return "0" if empty)

Solution Technique (Monotonic Stack Greedy)

The greedy choice is to remove the first larger digit encountered when scanning left to right:

- **Greedy Choice Property:** Removing earlier larger digits gives smaller lexicographic result
- **Monotonic Property:** Maintain non-decreasing stack for optimal result
- **Stack Invariant:** At each step, stack contains best possible prefix

Time Complexity: O(n) single pass

Dry Run

Input: num = "1432219", k = 3

Step 1: Process each digit:

- stack = [], k = 3
- '1': stack = ['1']

- '4': stack = ['1','4']
- '3': '4' > '3' and k > 0 → pop '4', k = 2, stack = ['1'], push '3' → stack = ['1','3']
- '2': '3' > '2' and k > 0 → pop '3', k = 1, stack = ['1'], push '2' → stack = ['1','2']
- '2': stack = ['1','2','2']
- '1': '2' > '1' and k > 0 → pop '2', k = 0, stack = ['1','2'], push '1' → stack = ['1','2','1']
- '9': stack = ['1','2','1','9']

Step 2: k = 0, so no more removals needed

Result: "1219"

12. Egyptian Fractions

Objective

Represent a given fraction as a sum of distinct unit fractions (fractions with numerator 1).

What is this?

Given a fraction a/b, express it as a sum of fractions of the form 1/x where all x values are distinct positive integers.

Why use Greedy?

Always subtract the largest possible unit fraction ($1/\lceil b/a \rceil$) that doesn't exceed the current fraction.

Full Problem Statement

Egyptian Fraction Problem

Given a fraction a/b (where a < b), represent it as a sum of distinct unit fractions.

Input:

- Two integers a and b representing fraction a/b

Example:

a = 6, b = 14

Solution Procedure

1. **While numerator ≠ 0:**
    - Calculate x = ⌈denominator/numerator⌉ (smallest integer such that 1/x ≤ current fraction)
    - Add 1/x to result
    - Subtract 1/x from current fraction: new_num = numerator×x - denominator, new_den = denominator×x
    - Update numerator and denominator
2. **Return list of denominators**

Solution Technique (Greedy Unit Fraction Subtraction)

The greedy choice is to always subtract the largest unit fraction that fits:

- **Greedy Choice Property:** Taking largest possible unit fraction is always part of optimal solution
- **Convergence:** Process always terminates with finite number of unit fractions
- **Distinctness:** Each step produces a different denominator

Time Complexity: Can be exponential in worst case, but typically efficient

Dry Run

Input: a = 6, b = 14

Step 1: Process fraction 6/14:

- x = ⌈14/6⌉ = ⌈2.33⌉ = 3
- Add 1/3 to result
- New fraction: (6×3 - 14)/(14×3) = 4/42 = 2/21

Step 2: Process fraction 2/21:

- x = ⌈21/2⌉ = ⌈10.5⌉ = 11
- Add 1/11 to result

- New fraction: (2×11 - 21)/(21×11) = 1/231

Step 3: Process fraction 1/231:

- Already a unit fraction, add 1/231 to result
- numerator = 0, done

Result: 6/14 = 1/3 + 1/11 + 1/231

13. Assign Cookies (Two Pointer Matching)

Objective

Maximize the number of children that can be satisfied by cookies, where each child has a greed factor and each cookie has a size.

What is this?

Given children with different appetites (greed factors) and cookies with different sizes, assign cookies to children to maximize the number of satisfied children. A child can only be satisfied if the cookie size is at least equal to their greed factor.

Why use Greedy?

The optimal strategy is to assign the smallest possible cookie that can satisfy each child, starting from the child with the smallest appetite. This preserves larger cookies for children with bigger appetites.

Full Problem Statement

Assign Cookies Problem

You are distributing cookies to children. Each child has a greed factor g[i] (minimum cookie size they'll accept), and each cookie has a size s[j]. A child can only be satisfied if they receive a cookie with size >= their greed factor. Find the maximum number of children you can satisfy.

Input:

- First line: n (children), m (cookies)

- Second line: greed factors of n children
- Third line: sizes of m cookies

Example:

```
3 2
1 2 3
1 1
```

Solution Procedure

1. **Sort both arrays** - greed factors in ascending order, cookie sizes in ascending order
2. **Initialize** two pointers: i = 0 (children), j = 0 (cookies)
3. **Initialize** satisfied = 0
4. **While both pointers are within bounds:**
    - If cookie[j] >= greed[i]: assign cookie to child, increment both pointers and satisfied count
    - Else: move to next larger cookie (increment j only)
5. **Return satisfied count**

Solution Technique (Greedy Matching)

The greedy choice is to always try to satisfy the child with smallest appetite using the smallest possible cookie:

- **Greedy Choice Property:** Satisfying smaller appetites first with smaller cookies preserves larger cookies for larger appetites
- **Optimal Substructure:** After satisfying a child, the remaining problem is independent
- **Matching Principle:** No benefit in giving a larger cookie to a child who can be satisfied with a smaller one

Time Complexity: O(n log n + m log m) due to sorting

Dry Run

Input: greed = [1,2,3], cookies = [1,1]

Step 1: Sort both arrays:

- greed = [1,2,3] (already sorted)
- cookies = [1,1] (already sorted)

Step 2: Two pointers matching:

- satisfied = 0, i = 0, j = 0
- cookies[0] = 1 >= greed[0] = 1 → satisfy child 0, satisfied = 1, i = 1, j = 1
- cookies[1] = 1 >= greed[1] = 2? No → j = 2 (out of bounds)
- End of cookies reached

Result: 1 child satisfied

14. Non-overlapping Intervals / Minimum Arrows

Objective

Find the minimum number of arrows needed to burst all balloons, where each balloon is represented by an interval and an arrow can burst all balloons it intersects.

What is this?

Given intervals representing balloons, find the minimum number of arrows such that each balloon is burst by at least one arrow. An arrow shot at position x bursts all balloons whose intervals contain x.

Why use Greedy?

Similar to activity selection, we sort by end points and greedily place arrows to burst as many balloons as possible with each shot.

Full Problem Statement

Minimum Arrows to Burst Balloons

You have balloons at various positions on a 2D plane. Each balloon is represented by an interval [start, end]. An arrow can be shot vertically at any x-coordinate and will burst all balloons whose intervals contain that x-coordinate. Find the minimum number of arrows needed to burst all balloons.

Input:

- First line: n (number of balloons)
- Next n lines: start end for each balloon

Example:

```
4
10 16
2 8
1 6
7 12
```

Solution Procedure

1. **Sort intervals by end points** in ascending order
2. **Initialize** arrows = 0, last_arrow_position = -∞
3. **For each interval:**
    - If interval's start > last_arrow_position: need new arrow
        - Increment arrows count
        - Shoot arrow at interval's end point (last_arrow_position = interval's end)
4. **Return arrows count**

Solution Technique (Greedy Interval Coverage)

The greedy choice is to always shoot the arrow at the end of the earliest-ending interval that hasn't been burst yet:

- **Greedy Choice Property:** Shooting at the rightmost position of current interval maximizes coverage of future intervals
- **Optimal Substructure:** After shooting an arrow, remaining problem is independent
- **Coverage Principle:** An arrow at the end point covers maximum possible overlapping intervals

Time Complexity: O(n log n) due to sorting

Dry Run

Input: [[10,16], [2,8], [1,6], [7,12]]

Step 1: Sort by end points: [[1,6], [2,8], [7,12], [10,16]]

Step 2: Process intervals:

- arrows = 0, last_arrow = -∞
- [1,6]: start = 1 > -∞ → shoot arrow at 6, arrows = 1, last_arrow = 6
- [2,8]: start = 2 <= 6 → already covered by previous arrow
- [7,12]: start = 7 > 6 → shoot arrow at 12, arrows = 2, last_arrow = 12
- [10,16]: start = 10 <= 12 → already covered by previous arrow

Result: 2 arrows needed

15. Single-machine Scheduling (SPT Rule)

Objective

Minimize the total waiting time (or total completion time) by scheduling jobs on a single machine using the Shortest Processing Time first rule.

What is this?

Given jobs with different processing times, determine the order to execute them on a single machine to minimize the sum of completion times (or equivalently, minimize average waiting time).

Why use Greedy?

The SPT (Shortest Processing Time) rule is optimal for minimizing total completion time. Processing shorter jobs first reduces the waiting time for all subsequent jobs.

Full Problem Statement

Single Machine Scheduling Problem

You have n jobs to process on a single machine. Each job has a processing time. Find the order to process jobs to minimize the total completion time (sum of all completion times).

Input:

- First line: n (number of jobs)
- Second line: processing times of n jobs

Example:

```
4
3 1 4 2
```

Solution Procedure

1. **Sort jobs by processing time** in ascending order (SPT rule)
2. **Process jobs in sorted order**
3. **Calculate completion times:**
    - completion_time[i] = completion_time[i-1] + processing_time[i]
4. **Sum all completion times** for the answer

Solution Technique (SPT Rule Optimality)

The greedy choice is to always process the shortest remaining job first:

- **Greedy Choice Property:** SPT rule minimizes total completion time
- **Mathematical Proof:** Any job swap where longer job comes before shorter job increases total completion time
- **Optimal Substructure:** After scheduling first job, remaining problem has same structure

Time Complexity: O(n log n) due to sorting

Dry Run

Input: processing_times = [3, 1, 4, 2]

Step 1: Sort by processing time: [1, 2, 3, 4]

Step 2: Calculate completion times:

- Job with time 1: completion = 1

- Job with time 2: completion = 1 + 2 = 3
- Job with time 3: completion = 3 + 3 = 6
- Job with time 4: completion = 6 + 4 = 10

Step 3: Total completion time = 1 + 3 + 6 + 10 = 20

Result: Minimum total completion time = 20

16. Job Sequencing with Deadlines

Objective

Schedule unit-time jobs with deadlines and profits to maximize total profit, where each job takes exactly one time unit to complete.

What is this?

Given jobs with deadlines and profits, where each job takes one unit of time, schedule jobs to maximize profit while respecting deadlines. Jobs can only be performed before their deadline.

Why use Greedy?

We can use a greedy approach with a min-heap to keep track of scheduled jobs and replace lower-profit jobs when necessary to maximize total profit.

Full Problem Statement

Job Sequencing with Deadlines Problem

You have n jobs, each with a deadline and profit. Each job takes exactly 1 time unit. A job can only be performed before its deadline. Schedule jobs to maximize total profit.

Input:

- First line: n (number of jobs)
- Next n lines: deadline profit for each job

Example:

```
5
2 100
1 19
2 27
1 25
3 15
```

Solution Procedure

1. **Sort jobs by deadline** in ascending order
2. **Use a min-heap** to store profits of scheduled jobs
3. **For each job in sorted order:**
    - Add job's profit to heap
    - If heap size > current job's deadline: remove job with minimum profit from heap
4. **Sum all profits in heap** for maximum profit

Solution Technique (Greedy with Heap Replacement)

The greedy choice is to schedule jobs by deadline and replace lower-profit jobs when capacity is exceeded:

- **Greedy Choice Property:** For each deadline, keep the highest-profit jobs that can be completed by that time
- **Optimal Substructure:** Decisions for earlier deadlines don't affect optimality of later decisions
- **Heap Invariant:** Heap always contains the best jobs that can be completed by current deadline

Time Complexity: O(n log n) for sorting and heap operations

Dry Run

Input: jobs = [(2,100), (1,19), (2,27), (1,25), (3,15)]

Step 1: Sort by deadline: [(1,19), (1,25), (2,100), (2,27), (3,15)]

Step 2: Process with min-heap:

- heap = [], profit = 0
- Job (1,19): heap = [19], deadline = 1, heap_size = 1 <= 1 ✓

- Job (1,25): heap = [19,25], deadline = 1, heap_size = 2 > 1 → remove min(19), heap = [25]
- Job (2,100): heap = [25,100], deadline = 2, heap_size = 2 <= 2 ✓
- Job (2,27): heap = [25,27,100], deadline = 2, heap_size = 3 > 2 → remove min(25), heap = [27,100]
- Job (3,15): heap = [15,27,100], deadline = 3, heap_size = 3 <= 3 ✓

Result: Maximum profit = 15 + 27 + 100 = 142

17. Huffman Coding (Optimal Merge Cost)

Objective

Build an optimal prefix-free binary code that minimizes the expected length of encoded messages, given character frequencies.

What is this?

Given characters and their frequencies, create a binary encoding where no code is a prefix of another, minimizing the average code length (weighted by frequency).

Why use Greedy?

Always merging the two least frequent nodes creates an optimal tree. This minimizes the weighted path length because less frequent characters get longer codes.

Full Problem Statement

Huffman Coding Problem

You need to encode a message containing characters with given frequencies. Create an optimal prefix-free binary code that minimizes the expected length of the encoded message.

Input:

- First line: n (number of characters)
- Next n lines: character frequency

Example:

```
4
a 5
b 9
c 12
d 13
f 16
h 45
```

Solution Procedure

1. **Create a min-heap** with all character frequencies
2. **While heap has more than one node:**
    - Extract two nodes with minimum frequency
    - Create new internal node with frequency = sum of two nodes
    - Add merge cost (sum of frequencies) to total cost
    - Insert new node back into heap
3. **Build codes** by traversing the tree (left=0, right=1)
4. **Calculate weighted path length** for verification

Solution Technique (Optimal Binary Tree Construction)

The greedy choice is to always merge the two least frequent nodes:

- **Greedy Choice Property:** The two least frequent characters should be at the deepest level and be siblings
- **Optimal Substructure:** After merging two nodes, remaining problem has same structure
- **Frequency Principle:** Less frequent characters can afford longer codes

Time Complexity: O(n log n) for heap operations

Dry Run

Input: frequencies = [5, 9, 12, 13, 16, 45]

Step 1: Create min-heap: [5, 9, 12, 13, 16, 45]

Step 2: Build Huffman tree:

- Merge 5+9=14, cost = 14, heap = [12, 13, 14, 16, 45]
- Merge 12+13=25, cost = 14+25=39, heap = [14, 16, 25, 45]
- Merge 14+16=30, cost = 39+30=69, heap = [25, 30, 45]
- Merge 25+30=55, cost = 69+55=124, heap = [45, 55]
- Merge 45+55=100, cost = 124+100=224, heap = [100]

Step 3: Tree structure determines optimal codes

- More frequent characters get shorter codes
- Less frequent characters get longer codes

Result: Optimal prefix-free code with minimum expected length

18. Shortest Processing Time / Shortest Job First

Objective

Minimize average waiting time or total completion time in job scheduling by processing jobs in order of shortest processing time first.

What is this?

A scheduling algorithm that processes jobs in ascending order of their processing/service time to optimize various metrics like average waiting time, total completion time, or average turnaround time.

Why use Greedy?

SJF is provably optimal for minimizing average waiting time. Processing shorter jobs first reduces the waiting time for all subsequent jobs in the queue.

Full Problem Statement

Shortest Job First Scheduling

Given n jobs with their processing times, schedule them to minimize the average waiting time. Assume all jobs arrive at time 0.

Input:

- First line: n (number of jobs)
- Second line: processing times of n jobs

Example:

```
4
6 8 7 3
```

Solution Procedure

1. **Sort jobs by processing time** in ascending order
2. **Calculate waiting times:**
    - First job: waiting_time = 0
    - Subsequent jobs: waiting_time = previous_completion_time
3. **Calculate completion times:** completion_time = waiting_time + processing_time
4. **Calculate average waiting time** and total completion time

Solution Technique (SJF Optimality)

The greedy choice is to always select the job with shortest processing time among remaining jobs:

- **Greedy Choice Property:** SJF minimizes average waiting time
- **Mathematical Proof:** Any swap where longer job precedes shorter job increases total waiting time
- **Optimal Substructure:** After scheduling first job, remaining problem has same structure

Time Complexity: O(n log n) due to sorting

Dry Run

Input: processing_times = [6, 8, 7, 3]

Step 1: Sort by processing time: [3, 6, 7, 8]

Step 2: Calculate scheduling metrics:

- Job 3: waiting = 0, completion = 0+3 = 3

- Job 6: waiting = 3, completion = 3+6 = 9
- Job 7: waiting = 9, completion = 9+7 = 16
- Job 8: waiting = 16, completion = 16+8 = 24

Step 3: Calculate averages:

- Total waiting time = 0 + 3 + 9 + 16 = 28
- Average waiting time = 28/4 = 7
- Total completion time = 3 + 9 + 16 + 24 = 52

Result: Average waiting time = 7, Total completion time = 52

19. Minimum Spanning Tree (Kruskal's Algorithm)

Objective

Find the minimum spanning tree of a weighted undirected graph - a tree that connects all vertices with minimum total edge weight.

What is this?

Given a connected weighted graph, find a subset of edges that connects all vertices without cycles, having minimum total weight. This tree has exactly n-1 edges for n vertices.

Why use Greedy?

Kruskal's algorithm greedily adds the lightest edge that doesn't create a cycle. This works because of the cut property: the minimum weight edge crossing any cut is in some MST.

Full Problem Statement

Minimum Spanning Tree Problem

Given a connected weighted undirected graph, find the minimum spanning tree - a tree that connects all vertices with minimum total edge weight.

Input:

- First line: n (vertices), m (edges)
- Next m lines: u v weight (edge from u to v with given weight)

Example:

```
4 5
0 1 10
0 2 6
0 3 5
1 3 15
2 3 4
```

Solution Procedure

1. **Sort all edges** by weight in ascending order
2. **Initialize Disjoint Set Union (DSU)** for cycle detection
3. **Initialize** MST_weight = 0, edges_added = 0
4. **For each edge in sorted order:**
    - If edge connects different components (no cycle): add to MST
    - Union the components and update MST_weight
    - Stop when edges_added = n-1
5. **Return MST_weight and MST edges**

Solution Technique (Greedy Cut Property)

The greedy choice is to always add the minimum weight edge that doesn't create a cycle:

- **Cut Property:** For any cut in the graph, the minimum weight edge crossing the cut is in some MST
- **Cycle Property:** Adding an edge creates a cycle iff it connects vertices in the same component
- **Optimal Substructure:** After adding an edge to MST, remaining problem has same structure

Time Complexity: O(m log m) for sorting edges

Dry Run

Input: Edges = [(0,1,10), (0,2,6), (0,3,5), (1,3,15), (2,3,4)]

Step 1: Sort by weight: [(2,3,4), (0,3,5), (0,2,6), (0,1,10), (1,3,15)]

Step 2: Process with DSU:

- DSU: {0}, {1}, {2}, {3}, MST_weight = 0, edges = 0
- Edge (2,3,4): different components → add, union(2,3), MST_weight = 4, edges = 1
- Edge (0,3,5): different components → add, union(0,3), MST_weight = 9, edges = 2
- Edge (0,2,6): different components → add, union(0,2), MST_weight = 15, edges = 3
- edges = n-1 = 3, MST complete

Result: MST weight = 15, MST edges = [(2,3,4), (0,3,5), (0,2,6)]

20. Dijkstra's Shortest Path (Non-negative Weights)

Objective

Find the shortest paths from a source vertex to all other vertices in a weighted graph with non-negative edge weights.

What is this?

Given a weighted directed graph with non-negative weights and a source vertex, compute the shortest distance from source to every other vertex.

Why use Greedy?

Dijkstra's algorithm greedily selects the unvisited vertex with minimum tentative distance. This works because with non-negative weights, the shortest path to the nearest unvisited vertex is finalized.

Full Problem Statement

Single Source Shortest Path Problem

Given a weighted directed graph with non-negative edge weights and a source vertex, find the shortest distance from the source to all other vertices.

Input:

- First line: n (vertices), m (edges), s (source vertex)
- Next m lines: u v weight (edge from u to v with given weight)

Example:

```
5 6 0
0 1 2
0 2 4
1 2 1
1 3 7
2 4 3
3 4 2
```

Solution Procedure

1. **Initialize distances:** dist[source] = 0, dist[others] = ∞
2. **Create min-heap** with (distance, vertex) pairs, starting with source
3. **While heap is not empty:**
    - Extract vertex u with minimum distance
    - If distance is outdated (> dist[u]), skip
    - For each neighbor v of u: relax edge (u,v)
        - If dist[u] + weight(u,v) < dist[v]: update dist[v] and add to heap
4. **Return distance array**

Solution Technique (Greedy Label-Setting)

The greedy choice is to always process the unvisited vertex with minimum tentative distance:

- **Greedy Choice Property:** With non-negative weights, the nearest unvisited vertex has its shortest path finalized
- **Label-Setting Property:** Once a vertex is processed, its shortest distance is final
- **Relaxation Principle:** Each edge relaxation potentially improves shortest path estimates

Time Complexity: O((n+m) log n) with binary heap

Dry Run

Input: Graph with edges [(0,1,2), (0,2,4), (1,2,1), (1,3,7), (2,4,3), (3,4,2)], source = 0

Step 1: Initialize: dist = [0, ∞, ∞, ∞, ∞], heap = [(0,0)]

Step 2: Process vertices:

- Extract (0,0): relax edges (0,1) and (0,2)
  - dist[1] = min(∞, 0+2) = 2, heap = [(2,1), (4,2)]
  - dist[2] = min(∞, 0+4) = 4
- Extract (2,1): relax edges (1,2) and (1,3)
  - dist[2] = min(4, 2+1) = 3, heap = [(3,2), (4,2), (9,3)]
  - dist[3] = min(∞, 2+7) = 9
- Extract (3,2): relax edge (2,4)
  - dist[4] = min(∞, 3+3) = 6, heap = [(4,2), (6,4), (9,3)]
- Extract (4,2): outdated, skip
- Extract (6,4): relax no edges (final vertex)
- Extract (9,3): relax edge (3,4)
  - dist[4] = min(6, 9+2) = 6 (no improvement)

Result: Shortest distances = [0, 2, 3, 9, 6]

Summary of Additional Patterns

Matching and Assignment

- **Assign Cookies:** Two-pointer matching after sorting both arrays
- **Bipartite Matching:** Greedy assignment based on compatibility

Interval and Scheduling

- **Non-overlapping Intervals:** Activity selection variant with different optimization target
- **Job Scheduling:** Various objectives (completion time, waiting time, profit) with different greedy criteria

Graph Algorithms

- **MST (Kruskal):** Greedy edge addition with cycle detection using DSU
- **Shortest Paths (Dijkstra):** Greedy vertex selection with distance relaxation

Encoding and Compression

- **Huffman Coding:** Optimal tree construction for prefix-free codes
- **File Merging:** Optimal merge order to minimize total cost

Key Insights:

1. **Sorting enables many greedy approaches** - by processing elements in optimal order
2. **Data structures matter** - heaps, DSU, and priority queues enable efficient greedy choices
3. **Proof techniques** - exchange arguments, cut properties, and optimal substructure justify correctness
4. **Problem variants** - many problems have greedy solutions with slight modifications to the greedy choice

Summary of Core Patterns

1. Sort-then-Pick Pattern

Used in: Activity Selection, Fractional Knapsack, Coin Change

- Sort by appropriate key (end time, ratio, denomination)
- Scan once and make greedy choices

2. Two-Pointer Pattern

Used in: Boats to Save People, Assign Cookies

- Sort first, then use two pointers to find optimal pairings
- Move pointers based on greedy criteria

3. Heap-based Pattern

Used in: Connect Ropes, Job Sequencing, Huffman Coding

- Use priority queue to always access optimal next choice
- Repeatedly extract and process optimal elements

4. Sweep Line Pattern

Used in: Minimum Platforms, Gas Station

- Process events in chronological order
- Maintain running state and track optimal value

5. Stack-based Pattern

Used in: Remove K Digits

- Maintain monotonic stack for optimal lexicographic ordering
- Pop when better choice found

6. Range Extension Pattern

Used in: Jump Games, Partition Labels

- Greedily extend current range to optimal boundary
- Process in levels or segments

Key Success Factors:

1. **Identify the greedy choice** - what local decision to make
2. **Prove greedy choice property** - local optimum leads to global optimum
3. **Verify optimal substructure** - remaining problem is independent
4. **Choose right data structure** - sort, heap, stack, pointers as needed
5. **Handle edge cases** - empty inputs, boundary conditions, impossible cases