Greedy Algorithm

What Is the Greedy Algorithm?

- **Basic concept**: A greedy algorithm builds a solution iteratively by choosing the "best" available option at each step, without reconsidering previous choices.
- **Why "greedy"?** Because at each step it grabs the locally optimal choice, hoping these local choices lead to a global optimum.
- **When it works**: Only when the problem has both the **greedy-choice property** (local optimum leads to global optimum) and **optimal substructure** (optimal solution of the problem contains optimal solutions of subproblems).

Real-World Use Cases

1. **Coin Change (making change with fewest coins)**
2. **Activity Selection (maximizing number of non-overlapping intervals)**
3. **Fractional Knapsack (maximizing value with divisible items)**
4. **Huffman Coding (minimum-redundancy prefix codes)**
5. **Minimum Spanning Tree (e.g., Kruskal's algorithm)**
6. **Job Scheduling on a Single Machine (minimize lateness)**

Greedy Strategy: Step by Step

1. **Define choices.** What are the units you'll pick? (e.g., coins, intervals, items.)
2. **Determine a criterion.** How do you rank choices? (e.g., largest coin, earliest finish time, highest value-weight ratio.)
3. **Sort or select.** Arrange your choices by that criterion.
4. **Iterate & pick.** Repeatedly take the best remaining choice that's still feasible.
5. **Stop.** When no more choices fit or you've reached the goal.

Variants of Greedy Algorithms

- **Standard Greedy**: Single pass, like activity selection.
- **Fractional Greedy**: Allows partial picks (fractional knapsack).
- **Multi-criteria Greedy**: Choose by one criterion, break ties by another (e.g., earliest deadline, then shortest duration).
- **Online Greedy**: Make choices as data arrives, without knowing future data (e.g., caching, CPU scheduling).

Python Code Examples

Coin Change (Standard Greedy)

Problem Statement:

You are given an amount and a list of coin denominations. Implement a function that returns the minimum number of coins (using the greedy approach) needed to make up the amount, assuming the coin system is canonical (i.e., greedy yields an optimal result). If exact change cannot be made, raise an error.

Constraints:

- **amount is a non-negative integer.**
- **coins is a list of positive integers.**
- **You can use each coin denomination an unlimited number of times.**

Function Signature:

def greedy_coin_change(amount: int, coins: List[int]) -> List[int]

🧪 Test Cases:

```
# Test Case 1

coins = [1, 5, 10, 25]

print(greedy_coin_change(63, coins))

# Expected Output: [25, 25, 10, 1, 1, 1]
# Test Case 2

coins = [1, 3, 4]

print(greedy_coin_change(6, coins))

# Note: Greedy gives [4,1,1], but optimal is [3,3]

# Output: [4,1,1] (non-optimal due to non-canonical system)


# Test Case 3 (Edge Case)

coins = [2, 5]

try:

   print(greedy_coin_change(3, coins))

except ValueError as e:

   print(e)

# Expected Output: Cannot make exact change with given coins.
```

Activity Selection (Interval Scheduling)

📝 **Problem Statement:**

Given a list of activities with their start and end times, select the maximum number of non-overlapping activities. Only one activity can be performed at a time.

Constraints:

- **Each activity is represented as a tuple (start_time, end_time).**
- **Activities are inclusive of start time and exclusive of end time.**

Function Signature:

def select_activities(activities: List[Tuple[int, int]]) -> List[Tuple[int, int]]

🧪 **Test Cases:**

```
# Test Case 1

acts = [(1,4), (3,5), (0,6), (5,7), (3,9), (8,9)]

print(select_activities(acts))

# Expected Output: [(1, 4), (5, 7), (8, 9)]


# Test Case 2

acts = [(10, 20), (20, 30), (5, 15), (30, 40)]

print(select_activities(acts))

# Expected Output: [(5, 15), (20, 30), (30, 40)]


# Test Case 3 (No overlap at all)

acts = [(1,2), (3,4), (5,6)]

print(select_activities(acts))

# Expected Output: [(1,2), (3,4), (5,6)]
```

Fractional Knapsack

📝 **Problem Statement:**

Given a set of items, each with a value and weight, and a knapsack with limited capacity, find the maximum total value achievable. You can take fractions of items if needed.

Constraints:

- Items: List of tuples (value, weight).
- Knapsack capacity is a positive float or integer.
- Fractions of items can be taken (unlike 0/1 Knapsack).

✅ **Function Signature:**

def fractional_knapsack(capacity: float, items: List[Tuple[float, float]]) -> Tuple[float, List[Tuple[float, float, float]]]

🧪 **Test Cases:**

# Test Case 1

items = [(60,10), (100,20), (120,30)]

capacity = 50

print(fractional_knapsack(capacity, items))

# Expected Output: (240.0, [(60,10,1.0), (100,20,1.0), (120,30,0.6667)])

# Test Case 2

items = [(500, 30), (400, 20), (200, 10)]

capacity = 40

print(fractional_knapsack(capacity, items))

# Expected Output: (900.0, [(500, 30, 1.0), (400, 20, 0.5)])

# Test Case 3 (Edge Case: capacity = 0)

```
items = [(100, 10), (50, 5)]

capacity = 0

print(fractional_knapsack(capacity, items))

# Expected Output: (0.0, [])
```

Coin Change (Standard Greedy)

```
def greedy_coin_change(amount, coins):
    """
    Return a list of coins (from largest to smallest) that sum to amount,
    using the fewest coins possible (if the coin system is canonical).
    """
    coins.sort(reverse=True)  # largest-first
    result = []
    for coin in coins:
        while amount >= coin:
            amount -= coin
            result.append(coin)
    if amount != 0:
        raise ValueError("Cannot make exact change with given coins.")
    return result


# Example
coins = [1, 5, 10, 25]
print(greedy_coin_change(63, coins))  # [25, 25, 10, 1, 1, 1]
```

5.2 Activity Selection (Interval Scheduling)

```
def select_activities(activities):
    """
    activities: list of (start, finish) tuples.
    Returns the max set of non-overlapping activities.
    """
    # sort by finish time
    activities.sort(key=lambda x: x[1])
    selected = []
    last_finish = 0
    for start, finish in activities:
        if start >= last_finish:
            selected.append((start, finish))
            last_finish = finish
    return selected

# Example
acts = [(1,4), (3,5), (0,6), (5,7), (3,9), (8,9)]
print(select_activities(acts))  # [(1,4),(5,7),(8,9)]
```

5.3 Fractional Knapsack (Allowing Fractions)

```
def fractional_knapsack(capacity, items):
    """
    items: list of (value, weight).
    Returns max total value (floats allowed) and fractions picked.
    """
    # compute value/weight ratio and sort descending
    items = [(v, w, v/w) for v, w in items]
    items.sort(key=lambda x: x[2], reverse=True)

    total_value = 0.0
    fractions = []
    for value, weight, ratio in items:
        if capacity == 0:
            break
        take = min(weight, capacity)
        total_value += take * ratio
        fractions.append((value, weight, take / weight))
        capacity -= take
    return total_value, fractions


# Example
items = [(60,10), (100,20), (120,30)]
capacity = 50
print(fractional_knapsack(capacity, items))
# (240.0, [(60,10,1.0), (100,20,1.0), (120,30,0.6667)])
```

Best Practices & Performance Tips

- **Verify greedy-choice property**: Prove that picking the local optimum won't ruin the global optimum.
- **Sort once**: Most greedy solutions spend O(n log n) sorting, then a single O(n) pass.
- **Use efficient data structures**: Heaps (priority queues) when you need dynamic selection, e.g., Huffman coding or event simulation.
- **Validate on edge cases**: Empty input, single item, no possible solution.
- **Combine with other paradigms**: Sometimes greedy + DP or greedy + backtracking solves near-optimal or constrained problems.

Common Mistakes to Avoid

1. **Assuming greedy always works**: Some coin systems (e.g., coins [1, 3, 4]) break the "fewest coins" greedy choice.
2. **Forgetting to sort**: Picking in arbitrary order leads to wrong answers.
3. **Not handling ties**: Tie-breaking criteria can affect correctness.
4. **Ignoring feasibility checks**: Always verify that adding a choice keeps your partial solution valid.
5. **Off-by-one in loops**: Especially when updating capacities, times, or indices.

Time & Space Complexity

- **Time**
    - Sorting step: O(n log n)
    - Single pass: O(n)
    - **Total**: O(n log n)
- **Space**
    - Usually O(n) for storing input and result
    - Can be O(1) extra if you modify in place and stream output