7) Activity Selection Problem

- Greedy choice: always pick the next activity that finishes first (earliest end time) that doesn't overlap with what we've picked. Then repeat this process until no more activities remain.
- Why it works: this choice leaves the most time for future picks, so we can fit more activities in without breaking their activity.

Steps:
1. Sort activities based on finish time.
2. Pick the first activity (earliest finish).
3. Skip any activity that overlaps with the last one picked.
4. Repeat: keep picking the next activity that doesn't overlap with the last one picked.

```
def ActivitySelection(activities):
    sort activities by finish time
    selected = [activities[0]]
    last_finish = activities[0].finish
    for activity in activities[1:]:
        if activity.start >= last_finish:
            selected.append(activity)
            last_finish = activity.finish
    return selected
```

8) Job Sequencing Problem (weighted job sequencing)

- Job sequencing is about picking jobs to maximize profit while respecting deadlines.
- Sort jobs by profit (highest first), then try to place each job in the latest possible deadline slot.

Steps:
1. Sort jobs by profit (descending).
2. Create a slot array for each deadline.
3. Loop through the sorted jobs:
4. Try to place each job in the latest available slot.

```
def JobSequencing(jobs):
    jobs.sort(key=lambda j: j.profit, reverse=True)
    max_deadline = max(j.deadline for j in jobs)
    slots = [None] * (max_deadline + 1)
    for job in jobs:
        for d in range(min(max_deadline, job.deadline), 0, -1):
            if slots[d] is None:
                slots[d] = job
                break
    return [s for s in slots if s]
```

9) Huffman Coding

- is a lossless data compression technique.
- assigns shorter codes to more frequent characters and longer codes to less frequent characters.

Steps:
1. Count the frequency of each character in the input.
2. Build a binary tree:
   - Start with leaf nodes for each character.
   - Combine the two nodes with the lowest frequency into a new parent node.
   - Repeat until there's only one node left (the root).
3. Assign binary codes:
   - Traverse the tree: left = 0, right = 1.
4. Encode the input using the generated codes.

```
def HuffmanCoding(text):
    freq = {}
    for char in text:
        freq[char] = freq.get(char, 0) + 1
    heap = [HuffmanNode(char, f) for char, f in freq.items()]
    heapify(heap)
    while len(heap) > 1:
        left = heappop(heap)
        right = heappop(heap)
        merged = HuffmanNode(None, left.freq + right.freq)
        merged.left = left
        merged.right = right
        heappush(heap, merged)
    root = heap[0]
    codes = {}
    def traverse(node, prefix):
        if node.char:
            codes[node.char] = prefix or '0'
        else:
            traverse(node.left, prefix + '0')
            traverse(node.right, prefix + '1')
    traverse(root, '')
    return codes
```

Algorithm

1. Count each character and build a min-heap of leaf nodes.
2. While heap.size() > 1:
   - Extract two nodes with the lowest frequency.
   - Create a new internal node with these as children and with frequency = sum of the two.
   - Insert the new node back into the heap.
3. The last node left is the root of the Huffman tree.
4. Traverse tree to assign codes: left = 0, right = 1.

```
Class HuffNode:
    char, freq, left, right
    def __init__(self, char, freq):
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None
    def __lt__(self, other):
        return self.freq < other.freq
def build_huffman_tree(freq_dict):
    heap = [HuffNode(char, f) for char, f in freq_dict.items()]
    heapify(heap)
    while len(heap) > 1:
        left = heappop(heap)
        right = heappop(heap)
        merged = HuffNode(None, left.freq + right.freq)
        merged.left = left
        merged.right = right
        heappush(heap, merged)
    return heap[0]
def generate_codes(root, prefix="", codes={}):
    if root.char is not None:
        codes[root.char] = prefix or '0'
    else:
        generate_codes(root.left, prefix + '0', codes)
        generate_codes(root.right, prefix + '1', codes)
    return codes
```

10) Huffman Decoding

- Decode the binary data using the Huffman tree built during encoding.
- Start at the root and follow the bits (0 = left, 1 = right) until you reach a leaf node, then output the character and restart from the root.

```
def HuffmanDecoding(encoded, root):
    result = []
    node = root
    for bit in encoded:
        if bit == '0':
            node = node.left
        else:
            node = node.right
        if node.char is not None:
            result.append(node.char)
            node = root
    return ''.join(result)
```

11) Minimum Coins Problem

- Greedy choice: pick the largest coin first, then repeat until you've made the total.

```
def MinCoins(coins, amount):
    coins.sort(reverse=True)
    count = 0
    for c in coins:
        while amount >= c:
            amount -= c
            count += 1
    return count if amount == 0 else -1
```

12) Fractional Knapsack Problem

- Greedy choice: pick items with the highest value-to-weight ratio first, and take as much of each item as possible until the knapsack is full.

```
def FractionalKnapsack(items, capacity):
    items.sort(key=lambda x: x[1]/x[0], reverse=True)
    total = 0.0
    for w, v in items:
        if capacity <= 0:
            break
        take = min(w, capacity)
        total += v * (take / w)
        capacity -= take
    return round(total, 2)
```

13) Activity Selection Problem

- Sort activities by finish time.
- Pick the first activity.
- Skip any activity that overlaps with the last one picked.
- Repeat until no more activities remain.

```
def ActivitySelection(activities):
    activities.sort(key=lambda a: a.finish)
    selected = [activities[0]]
    last = activities[0].finish
    for a in activities[1:]:
        if a.start >= last:
            selected.append(a)
            last = a.finish
    return selected
```

14) Job Sequencing Problem

- Sort jobs by profit (highest first).
- Find the latest available slot before the deadline.
- Place the job in that slot.

```
def JobSequencing(jobs):
    jobs.sort(key=lambda j: j.profit, reverse=True)
    max_deadline = max(j.deadline for j in jobs)
    slots = [None] * (max_deadline + 1)
    for job in jobs:
        for d in range(min(max_deadline, job.deadline), 0, -1):
            if slots[d] is None:
                slots[d] = job
                break
    return [s for s in slots if s]
```

15) Huffman Coding

- Count character frequencies.
- Build a binary tree by combining the two lowest frequency nodes.
- Assign codes: left = 0, right = 1.
- Encode the input using the codes.

```
def HuffmanCoding(text):
    freq = {}
    for char in text:
        freq[char] = freq.get(char, 0) + 1
    heap = [HuffmanNode(char, f) for char, f in freq.items()]
    heapify(heap)
    while len(heap) > 1:
        left = heappop(heap)
        right = heappop(heap)
        merged = HuffmanNode(None, left.freq + right.freq)
        merged.left = left
        merged.right = right
        heappush(heap, merged)
    root = heap[0]
    codes = {}
    def traverse(node, prefix):
        if node.char:
            codes[node.char] = prefix or '0'
        else:
            traverse(node.left, prefix + '0')
            traverse(node.right, prefix + '1')
    traverse(root, '')
    return codes
```

16) Huffman Decoding

- Decode the binary data using the Huffman tree built during encoding.
- Start at the root and follow the bits (0 = left, 1 = right) until you reach a leaf node, then output the character and restart from the root.

```
def HuffmanDecoding(encoded, root):
    result = []
    node = root
    for bit in encoded:
        if bit == '0':
            node = node.left
        else:
            node = node.right
        if node.char is not None:
            result.append(node.char)
            node = root
    return ''.join(result)
```

17) Minimum Coins Problem

- Greedy choice: pick the largest coin first, then repeat until you've made the total.

```
def MinCoins(coins, amount):
    coins.sort(reverse=True)
    count = 0
    for c in coins:
        while amount >= c:
            amount -= c
            count += 1
    return count if amount == 0 else -1
```

18) Fractional Knapsack Problem

- Greedy choice: pick items with the highest value-to-weight ratio first, and take as much of each item as possible until the knapsack is full.

```
def FractionalKnapsack(items, capacity):
    items.sort(key=lambda x: x[1]/x[0], reverse=True)
    total = 0.0
    for w, v in items:
        if capacity <= 0:
            break
        take = min(w, capacity)
        total += v * (take / w)
        capacity -= take
    return round(total, 2)
```

19) Minimum Array Sum after K negative sign flips

- Greedy + min-heap approach: flip smallest elements first to minimize the array sum.
- Use a min-heap to quickly find the smallest elements.
- Flip negative numbers first, then if flips remain, flip the smallest element.

Time Complexity: $O(n \log n + k \log n)$

```
def minSumAfterKNegFlips(nums, k):
    heapify(nums)
    while k > 0 and nums[0] < 0:
        nums[0] = -nums[0]
        heapreplace(nums, nums[0])
        k -= 1
    if k % 2 == 1:
        nums[0] = -nums[0]
    return sum(nums)
```

20) Minimum Coins Problem

- Greedy choice: pick the largest coin first, then repeat until you've made the total.

```
def MinCoins(coins, amount):
    coins.sort(reverse=True)
    count = 0
    for c in coins:
        while amount >= c:
            amount -= c
            count += 1
    return count if amount == 0 else -1
```

21) Fractional Knapsack Problem

- Greedy choice: pick items with the highest value-to-weight ratio first, and take as much of each item as possible until the knapsack is full.

```
def FractionalKnapsack(items, capacity):
    items.sort(key=lambda x: x[1]/x[0], reverse=True)
    total = 0.0
    for w, v in items:
        if capacity <= 0:
            break
        take = min(w, capacity)
        total += v * (take / w)
        capacity -= take
    return round(total, 2)
```

22) Greedy Algo to find Maximum Number of Tasks

- Sort tasks by finish time.
- Pick the first task.
- Skip any task that overlaps with the last one picked.
- Repeat until no more tasks remain.

```
def MaxTasks(tasks):
    tasks.sort(key=lambda t: t[1])
    count = 0
    last_finish = -1
    for start, finish in tasks:
        if start >= last_finish:
            count += 1
            last_finish = finish
    return count
```

23) Max Profit Job Scheduling

- Sort jobs by profit (highest first).
- Find the latest available slot before the deadline.
- Place the job in that slot.

```
def JobSequencing(jobs):
    jobs.sort(key=lambda j: j.profit, reverse=True)
    max_deadline = max(j.deadline for j in jobs)
    slots = [None] * (max_deadline + 1)
    for job in jobs:
        for d in range(min(max_deadline, job.deadline), 0, -1):
            if slots[d] is None:
                slots[d] = job
                break
    return [s for s in slots if s]
```

24) Max Number of platforms required for a railway/bus station

Algorithm: Take arrival and departure times, and sort them separately. Iterate through both arrays, keeping track of how many trains are at the station at any time. Increment when a train arrives, decrement when a train departs. The maximum value reached during this process is the minimum number of platforms needed.

Time Complexity: $O(n \log n)$

Space Complexity: $O(n)$

- Idea is to consider all events in sorted order.
- Traverse through the sorted array and keep track of the number of trains present at any point in time.
- Keep a count variable that stores the current number of platforms needed.
- Update the result whenever the count exceeds the previous maximum.

```
def findPlatform(arr, dep):
    arr.sort()
    dep.sort()
    plat_needed = 1
    result = 1
    i = 1
    j = 0
    while (i < len(arr) and j < len(dep)):
        if (arr[i] <= dep[j]):
            plat_needed += 1
            i += 1
            if (plat_needed > result):
                result = plat_needed
        else:
            plat_needed -= 1
            j += 1
    return result
```

25) Max Profit Job Scheduling

- Sort jobs by profit (highest first).
- Find the latest available slot before the deadline.
- Place the job in that slot.

```
def JobSequencing(jobs):
    jobs.sort(key=lambda j: j.profit, reverse=True)
    max_deadline = max(j.deadline for j in jobs)
    slots = [None] * (max_deadline + 1)
    for job in jobs:
        for d in range(min(max_deadline, job.deadline), 0, -1):
            if slots[d] is None:
                slots[d] = job
                break
    return [s for s in slots if s]
```

26) Coin Problem (DP)

- Use a DP array to track the minimum coins needed for each amount from 0 to the target.
- Initialize dp[0] = 0 (0 coins for amount 0).
- For each amount, try each coin and update dp[amount] = min(dp[amount], dp[amount - coin] + 1).
- Return dp[target] if it's not infinity, otherwise return -1 (impossible).

```
def CoinChange(coins, amount):
    dp = [float('inf')] * (amount + 1)
    dp[0] = 0
    for coin in coins:
        for i in range(coin, amount + 1):
            dp[i] = min(dp[i], dp[i - coin] + 1)
    return dp[amount] if dp[amount] != float('inf') else -1
```

27) Activity Selection Problem

- Sort activities by finish time.
- Pick the first activity.
- Skip any activity that overlaps with the last one picked.
- Repeat until no more activities remain.

```
def ActivitySelection(activities):
    activities.sort(key=lambda a: a.finish)
    selected = [activities[0]]
    last = activities[0].finish
    for a in activities[1:]:
        if a.start >= last:
            selected.append(a)
            last = a.finish
    return selected
```

28) Job Sequencing Problem

- Sort jobs by profit (highest first).
- Find the latest available slot before the deadline.
- Place the job in that slot.

```
def JobSequencing(jobs):
    jobs.sort(key=lambda j: j.profit, reverse=True)
    max_deadline = max(j.deadline for j in jobs)
    slots = [None] * (max_deadline + 1)
    for job in jobs:
        for d in range(min(max_deadline, job.deadline), 0, -1):
            if slots[d] is None:
                slots[d] = job
                break
    return [s for s in slots if s]
```

29) Huffman Coding

- Count character frequencies.
- Build a binary tree by combining the two lowest frequency nodes.
- Assign codes: left = 0, right = 1.
- Encode the input using the codes.

```
def HuffmanCoding(text):
    freq = {}
    for char in text:
        freq[char] = freq.get(char, 0) + 1
    heap = [HuffmanNode(char, f) for char, f in freq.items()]
    heapify(heap)
    while len(heap) > 1:
        left = heappop(heap)
        right = heappop(heap)
        merged = HuffmanNode(None, left.freq + right.freq)
        merged.left = left
        merged.right = right
        heappush(heap, merged)
    root = heap[0]
    codes = {}
    def traverse(node, prefix):
        if node.char:
            codes[node.char] = prefix or '0'
        else:
            traverse(node.left, prefix + '0')
            traverse(node.right, prefix + '1')
    traverse(root, '')
    return codes
```

30) Huffman Decoding

- Decode the binary data using the Huffman tree built during encoding.
- Start at the root and follow the bits (0 = left, 1 = right) until you reach a leaf node, then output the character and restart from the root.

```
def HuffmanDecoding(encoded, root):
    result = []
    node = root
    for bit in encoded:
        if bit == '0':
            node = node.left
        else:
            node = node.right
        if node.char is not None:
            result.append(node.char)
            node = root
    return ''.join(result)
```

31) Greedy Approach, Algorithm for N queens Problem

- Not a true greedy approach, but uses greedy choices within a backtracking framework.
- Place queens row by row, choosing the first safe position in each row.
- Backtrack when no safe position exists.
- Time: $O(n!)$ worst case, but much better in practice due to pruning.
- Space: $O(n)$ for the board and recursion stack.

```
def solve(n):
    board = [-1] * n
    def is_safe(row, col):
        for r in range(row):
            if board[r] == col or abs(board[r] - col) == abs(r - row):
            return False
        return True
    def place(row):
        if row == n:
            return True
        for col in range(n):
            if is_safe(row, col):
                board[row] = col
                if place(row + 1):
                    return True
                board[row] = -1
        return False
    place(0)
    return board
```

32) Greedy Algorithm

- Sort the activities by finish time.
- Pick the first activity.
- Skip any activity that overlaps with the last one picked.
- Repeat until no more activities remain.

```
def ActivitySelection(activities):
    activities.sort(key=lambda a: a.finish)
    selected = [activities[0]]
    last = activities[0].finish
    for a in activities[1:]:
        if a.start >= last:
            selected.append(a)
            last = a.finish
    return selected
```

33) Coin Problem

- Greedy choice: pick the largest coin first, then repeat until you've made the total.

```
def MinCoins(coins, amount):
    coins.sort(reverse=True)
    count = 0
    for c in coins:
        while amount >= c:
            amount -= c
            count += 1
    return count if amount == 0 else -1
```

34) Longest Increasing Subsequence (LIS)

- Use binary search (bisect_left) to find the position where each element should be placed.
- Maintain a tail array where tail[i] is the smallest ending element for an increasing subsequence of length i+1.
- Time: $O(n \log n)$
- Space: $O(n)$

```
def lis(nums):
    tail = []
    for num in nums:
        pos = bisect_left(tail, num)
        if pos == len(tail):
            tail.append(num)
        else:
            tail[pos] = num
    return len(tail)
```

35) Minimum Coins Problem

- Greedy choice: pick the largest coin first, then repeat until you've made the total.

```
def MinCoins(coins, amount):
    coins.sort(reverse=True)
    count = 0
    for c in coins:
        while amount >= c:
            amount -= c
            count += 1
    return count if amount == 0 else -1
```

36) Fractional Knapsack Problem

- Greedy choice: pick items with the highest value-to-weight ratio first, and take as much of each item as possible until the knapsack is full.

```
def FractionalKnapsack(items, capacity):
    items.sort(key=lambda x: x[1]/x[0], reverse=True)
    total = 0.0
    for w, v in items:
        if capacity <= 0:
            break
        take = min(w, capacity)
        total += v * (take / w)
        capacity -= take
    return round(total, 2)
```

37) Activity Selection Problem

- Sort activities by finish time.
- Pick the first activity.
- Skip any activity that overlaps with the last one picked.
- Repeat until no more activities remain.

```
def ActivitySelection(activities):
    activities.sort(key=lambda a: a.finish)
    selected = [activities[0]]
    last = activities[0].finish
    for a in activities[1:]:
        if a.start >= last:
            selected.append(a)
            last = a.finish
    return selected
```

38) Job Sequencing Problem

- Sort jobs by profit (highest first).
- Find the latest available slot before the deadline.
- Place the job in that slot.

```
def JobSequencing(jobs):
    jobs.sort(key=lambda j: j.profit, reverse=True)
    max_deadline = max(j.deadline for j in jobs)
    slots = [None] * (max_deadline + 1)
    for job in jobs:
        for d in range(min(max_deadline, job.deadline), 0, -1):
            if slots[d] is None:
                slots[d] = job
                break
    return [s for s in slots if s]
```

39) Huffman Coding

- Count character frequencies.
- Build a binary tree by combining the two lowest frequency nodes.
- Assign codes: left = 0, right = 1.
- Encode the input using the codes.

```
def HuffmanCoding(text):
    freq = {}
    for char in text:
        freq[char] = freq.get(char, 0) + 1
    heap = [HuffmanNode(char, f) for char, f in freq.items()]
    heapify(heap)
    while len(heap) > 1:
        left = heappop(heap)
        right = heappop(heap)
        merged = HuffmanNode(None, left.freq + right.freq)
        merged.left = left
        merged.right = right
        heappush(heap, merged)
    root = heap[0]
    codes = {}
    def traverse(node, prefix):
        if node.char:
            codes[node.char] = prefix or '0'
        else:
            traverse(node.left, prefix + '0')
            traverse(node.right, prefix + '1')
    traverse(root, '')
    return codes
```

40) Huffman Decoding

- Decode the binary data using the Huffman tree built during encoding.
- Start at the root and follow the bits (0 = left, 1 = right) until you reach a leaf node, then output the character and restart from the root.

```
def HuffmanDecoding(encoded, root):
    result = []
    node = root
    for bit in encoded:
        if bit == '0':
            node = node.left
        else:
            node = node.right
        if node.char is not None:
            result.append(node.char)
            node = root
    return ''.join(result)
```

41) Greedy Algorithm

- Sort the activities by finish time.
- Pick the first activity.
- Skip any activity that overlaps with the last one picked.
- Repeat until no more activities remain.

```
def ActivitySelection(activities):
    activities.sort(key=lambda a: a.finish)
    selected = [activities[0]]
    last = activities[0].finish
    for a in activities[1:]:
        if a.start >= last:
            selected.append(a)
            last = a.finish
    return selected
```

42) Coin Problem

- Greedy choice: pick the largest coin first, then repeat until you've made the total.

```
def MinCoins(coins, amount):
    coins.sort(reverse=True)
    count = 0
    for c in coins:
        while amount >= c:
            amount -= c
            count += 1
    return count if amount == 0 else -1
```

43) Fractional Knapsack Problem

- Greedy choice: pick items with the highest value-to-weight ratio first, and take as much of each item as possible until the knapsack is full.

```
def FractionalKnapsack(items, capacity):
    items.sort(key=lambda x: x[1]/x[0], reverse=True)
    total = 0.0
    for w, v in items:
        if capacity <= 0:
            break
        take = min(w, capacity)
        total += v * (take / w)
        capacity -= take
    return round(total, 2)
```

44) Activity Selection Problem

- Sort activities by finish time.
- Pick the first activity.
- Skip any activity that overlaps with the last one picked.
- Repeat until no more activities remain.

```
def ActivitySelection(activities):
    activities.sort(key=lambda a: a.finish)
    selected = [activities[0]]
    last = activities[0].finish
    for a in activities[1:]:
        if a.start >= last:
            selected.append(a)
            last = a.finish
    return selected
```

45) Job Sequencing Problem

- Sort jobs by profit (highest first).
- Find the latest available slot before the deadline.
- Place the job in that slot.

```
def JobSequencing(jobs):
    jobs.sort(key=lambda j: j.profit, reverse=True)
    max_deadline = max(j.deadline for j in jobs)
    slots = [None] * (max_deadline + 1)
    for job in jobs:
        for d in range(min(max_deadline, job.deadline), 0, -1):
            if slots[d] is None:
                slots[d] = job
                break
    return [s for s in slots if s]
```

46) Huffman Coding

- Count character frequencies.
- Build a binary tree by combining the two lowest frequency nodes.
- Assign codes: left = 0, right = 1.
- Encode the input using the codes.

```
def HuffmanCoding(text):
    freq = {}
    for char in text:
        freq[char] = freq.get(char, 0) + 1
    heap = [HuffmanNode(char, f) for char, f in freq.items()]
    heapify(heap)
    while len(heap) > 1:
        left = heappop(heap)
        right = heappop(heap)
        merged = HuffmanNode(None, left.freq + right.freq)
        merged.left = left
        merged.right = right
        heappush(heap, merged)
    root = heap[0]
    codes = {}
    def traverse(node, prefix):
        if node.char:
            codes[node.char] = prefix or '0'
        else:
            traverse(node.left, prefix + '0')
            traverse(node.right, prefix + '1')
    traverse(root, '')
    return codes
```

47) Huffman Decoding

- Decode the binary data using the Huffman tree built during encoding.
- Start at the root and follow the bits (0 = left, 1 = right) until you reach a leaf node, then output the character and restart from the root.

```
def HuffmanDecoding(encoded, root):
    result = []
    node = root
    for bit in encoded:
        if bit == '0':
            node = node.left
        else:
            node = node.right
        if node.char is not None:
            result.append(node.char)
            node = root
    return ''.join(result)
```

48) Greedy Approach, Algorithm for N queens Problem

- Not a true greedy approach, but uses greedy choices within a backtracking framework.
- Place queens row by row, choosing the first safe position in each row.
- Backtrack when no safe position exists.
- Time: $O(n!)$ worst case, but much better in practice due to pruning.
- Space: $O(n)$ for the board and recursion stack.

```
def solve(n):
    board = [-1] * n
    def is_safe(row, col):
        for r in range(row):
            if board[r] == col or abs(board[r] - col) == abs(r - row):
            return False
        return True
    def place(row):
        if row == n:
            return True
        for col in range(n):
            if is_safe(row, col):
                board[row] = col
                if place(row + 1):
                    return True
                board[row] = -1
        return False
    place(0)
    return board
```

49) Greedy Approach, Algorithm for N queens Problem

- Not a true greedy approach, but uses greedy choices within a backtracking framework.
- Place queens row by row, choosing the first safe position in each row.
- Backtrack when no safe position exists.
- Time: $O(n!)$ worst case, but much better in practice due to pruning.
- Space: $O(n)$ for the board and recursion stack.

```
def solve(n):
    board = [-1] * n
    def is_safe(row, col):
        for r in range(row):
            if board[r] == col or abs(board[r] - col) == abs(r - row):
            return False
        return True
    def place(row):
        if row == n:
            return True
        for col in range(n):
            if is_safe(row, col):
                board[row] = col
                if place(row + 1):
                    return True
                board[row] = -1
        return False
    place(0)
    return board
```

50) Greedy Algorithm

- Sort the activities by finish time.
- Pick the first activity.
- Skip any activity that overlaps with the last one picked.
- Repeat until no more activities remain.

```
def ActivitySelection(activities):
    activities.sort(key=lambda a: a.finish)
    selected = [activities[0]]
    last = activities[0].finish
    for a in activities[1:]:
        if a.start >= last:
            selected.append(a)
            last = a.finish
    return selected
```

51) Coin Problem

- Greedy choice: pick the largest coin first, then repeat until you've made the total.

```
def MinCoins(coins, amount):
    coins.sort(reverse=True)
    count = 0
    for c in coins:
        while amount >= c:
            amount -= c
            count += 1
    return count if amount == 0 else -1
```

52) Fractional Knapsack Problem

- Greedy choice: pick items with the highest value-to-weight ratio first, and take as much of each item as possible until the knapsack is full.

```
def FractionalKnapsack(items, capacity):
    items.sort(key=lambda x: x[1]/x[0], reverse=True)
    total = 0.0
    for w, v in items:
        if capacity <= 0:
            break
        take = min(w, capacity)
        total += v * (take / w)
        capacity -= take
    return round(total, 2)
```

53) Activity Selection Problem

- Sort the activities by finish time.
- Pick the first activity.
- Skip any activity that overlaps with the last one picked.
- Repeat until no more activities remain.

```
def ActivitySelection(activities):
    activities.sort(key=lambda a: a.finish)
    selected = [activities[0]]
    last = activities[0].finish
    for a in activities[1:]:
        if a.start >= last:
            selected.append(a)
            last = a.finish
    return selected
```

54) Job Sequencing Problem

- Sort jobs by profit (highest first).
- Find the latest available slot before the deadline.
- Place the job in that slot.

```
def JobSequencing(jobs):
    jobs.sort(key=lambda j: j.profit, reverse=True)
    max_deadline = max(j.deadline for j in jobs)
    slots = [None] * (max_deadline + 1)
    for job in jobs:
        for d in range(min(max_deadline, job.deadline), 0, -1):
            if slots[d] is None:
                slots[d] = job
                break
    return [s for s in slots if s]
```

55) Huffman Coding

- Count character frequencies.
- Build a binary tree by combining the two lowest frequency nodes.
- Assign codes: left = 0, right = 1.
- Encode the input using the codes.

```
def HuffmanCoding(text):
    freq = {}
    for char in text:
        freq[char] = freq.get(char, 0) + 1
    heap = [HuffmanNode(char, f) for char, f in freq.items()]
    heapify(heap)
    while len(heap) > 1:
        left = heappop(heap)
        right = heappop(heap)
        merged = HuffmanNode(None, left.freq + right.freq)
        merged.left = left
        merged.right = right
        heappush(heap, merged)
    root = heap[0]
    codes = {}
    def traverse(node, prefix):
        if node.char:
            codes[node.char] = prefix or '0'
        else:
            traverse(node.left, prefix + '0')
            traverse(node.right, prefix + '1')
    traverse(root, '')
    return codes
```

56) Huffman Decoding

- Decode the binary data using the Huffman tree built during encoding.
- Start at the root and follow the bits (0 = left, 1 = right) until you reach a leaf node, then output the character and restart from the root.

```
def HuffmanDecoding(encoded, root):
    result = []
    node = root
    for bit in encoded:
        if bit == '0':
            node = node.left
        else:
            node = node.right
        if node.char is not None:
            result.append(node.char)
            node = root
    return ''.join(result)
```