Complete Guide to Greedy Algorithms

Introduction

Greedy algorithms are a class of algorithms that make the locally optimal choice at each step, hoping to find a global optimum. The fundamental principle is to choose the best option available at each moment without considering future consequences. While greedy algorithms don't always produce the globally optimal solution, they are efficient and work well for many optimization problems.

Key Characteristics:

- Make the best choice at each step
- Never reconsider previous choices
- Often simple and efficient
- Don't always guarantee optimal solutions
- Work well for problems with optimal substructure and greedy choice property

When to Use Greedy Algorithms:

- When local optimal choices lead to global optimal solutions
- For problems with greedy choice property
- When you need fast, approximate solutions
- For optimization problems with specific structures

Easy Problems on Greedy Algorithm

1. Fractional Knapsack

Problem: Maximize value by picking full or fractional items to fill knapsack of capacity W.

Explanation: Greedily pick items with highest value/weight ratio. If can't take whole item, take fraction of it.

Pseudocode:

```
sort items by value/weight descending
total_value = 0

for item in items:
    if W >= item.weight:
        W -= item.weight
        total_value += item.value
    else:
        total_value += item.value * (W / item.weight)
        break

return total_value
```

2. Minimum Rotations for Circular Lock

Problem: Find min total rotations needed to match each digit of a circular lock (0–9).

Explanation: For each digit, find min distance either clockwise or counter-clockwise.

Pseudocode:

```
total_rotations = 0

for i from 0 to n-1:
    diff = abs(current[i] - target[i])
    rotations = min(diff, 10 - diff)
    total_rotations += rotations

return total_rotations
```

3. Max Composite Numbers to Make n

Problem: Find max number of composite numbers (≥4) that sum to n.

Explanation: Greedily use as many 4s as possible, since 4 is smallest composite.

Pseudocode:

```
if n < 4: return -1
if n == 5 or n == 7: return -1

return n // 4
```

4. Smallest Subset with Sum Greater Than Rest

Problem: Find smallest subset whose sum is greater than remaining elements.

Explanation: Greedily take largest elements until sum > rest.

Pseudocode:

```
sort array descending
total = sum(array)
subset_sum = 0

for i in 0 to n-1:
    subset_sum += arr[i]
    if subset_sum > total - subset_sum:
        return i + 1
```

5. Assign Cookies

Problem: Maximize number of children satisfied with cookies ≥ their greed.

Explanation: Greedily assign smallest cookie that satisfies the current child.

Pseudocode:

```
sort greed[]
sort cookies[]
i = j = 0

while i < g.length and j < s.length:
    if s[j] >= g[i]:
        count += 1
        i += 1
    j += 1

return count
```

6. Buy Maximum Stocks

Problem: You can buy at most i stocks on ith day. Maximize total stocks with given amount.

Explanation: Greedily buy cheapest days first.

Pseudocode:

```
for i = 0 to n-1:
    stock_options.append((price[i], i+1))  // day i → max i+1 stocks

sort stock_options by price
total_stocks = 0

for price, limit in stock_options:
    buy = min(limit, amount // price)
    total_stocks += buy
    amount -= buy * price

return total_stocks
```

7. Max Consecutive Diff Sum

Problem: Rearrange array to maximize sum of absolute differences between consecutive elements.

Explanation: Arrange elements alternately: smallest, largest, 2nd smallest, 2nd largest, etc.

Pseudocode:

```
sort array
i = 0, j = n - 1
result = []

while i <= j:
    if i == j:
        result.append(arr[i])
    else:
        result.append(arr[j])
        result.append(arr[i])
    i += 1
    j -= 1

sum = 0
for k = 1 to n-1:
    sum += abs(result[k] - result[k-1])

return sum
```

8. Min and Max Costs to Buy All Candies

Problem: Each time you buy 1 candy, you get up to k free. Find min and max money to buy all.

Explanation: For min, buy cheapest first. For max, buy costliest first.

Pseudocode:

```
sort prices

// Min cost
min_cost = 0
i = 0
while i < n:
    min_cost += prices[i]
    i += 1
    n -= k  // get k free

// Max cost
sort prices descending
max_cost = 0
i = 0
while i < n:
    max_cost += prices[i]
    i += 1
    n -= k

return min_cost, max_cost
```

9. Min Notes with Given Sum

Problem: Find min number of currency notes (1, 2, 5, 10, 20, 50, 100, 500, 1000) to sum to n.

Explanation: Use largest denomination possible at each step.

Pseudocode:

```
denominations = [1000, 500, 100, 50, 20, 10, 5, 2, 1]
notes = []

for coin in denominations:
    while n >= coin:
        n -= coin
        notes.append(coin)

return notes
```

10. Max Equal Sum of Three Stacks

Problem: Make sums of 3 stacks equal by removing top elements.

Explanation: Greedily remove from the tallest (max sum) stack.

Pseudocode:

```
sum1 = sum(stack1), sum2 = sum(stack2), sum3 = sum(stack3)

while not all sums equal:
    max_sum = max(sum1, sum2, sum3)
    if sum1 == max_sum:
        sum1 -= stack1.pop()
    elif sum2 == max_sum:
        sum2 -= stack2.pop()
    else:
        sum3 -= stack3.pop()

return sum1
```

Medium Problems on Greedy Algorithm

1. Activity Selection Problem

Problem: Select the maximum number of activities that don't overlap (each has a start and end time).

Explanation: Always pick the activity that ends earliest. Sort activities by end time and select each activity if it starts after the last selected activity ends.

Pseudocode:

```
sort activities by end time
last_end = -∞
count = 0

for each activity in activities:
    if activity.start >= last_end:
        count += 1
        last_end = activity.end

return count
```

2. Jump Game

Problem: From an array where arr[i] means max jump from i, check if you can reach the end.

Explanation: Always try to jump as far as possible at each step. Track the furthest index reachable at each step.

Pseudocode:

```
maxReach = 0

for i = 0 to n-1:
    if i > maxReach:
        return false
    maxReach = max(maxReach, i + nums[i])

return true
```

3. Job Sequencing Problem

Problem: Schedule jobs (each with deadline and profit) to maximize total profit.

Explanation: Pick jobs with higher profit first. Sort jobs by descending profit and place each job at the latest available slot before its deadline.

Pseudocode:

```
sort jobs by profit descending
initialize slots[] of size max_deadline as empty
total_profit = 0

for each job:
    for d = job.deadline down to 1:
        if slots[d] is empty:
            slots[d] = job
            total_profit += job.profit
            break

return total_profit
```

4. Egyptian Fraction

Problem: Represent a fraction like 4/13 as a sum of distinct unit fractions.

Explanation: Always subtract the largest possible unit fraction less than the current fraction.

Pseudocode:

```
while numerator ≠ 0:
    x = ceil(denominator / numerator)
    print 1/x
    numerator = numerator * x - denominator
    denominator = denominator * x
```

5. Merge Overlapping Intervals

Problem: Merge intervals that overlap.

Explanation: Always extend the current merged interval if the next one overlaps. Sort by start time and merge overlapping intervals as you iterate.

Pseudocode:

```
sort intervals by start time
merged = []

for interval in intervals:
    if merged is empty or merged.last.end < interval.start:
        append interval to merged
    else:
        merged.last.end = max(merged.last.end, interval.end)

return merged
```

6. Min Fibonacci Terms with Sum K

Problem: Represent K as a sum of minimum number of Fibonacci numbers.

Explanation: Subtract the largest Fibonacci number ≤ K. Generate Fibonacci numbers ≤ K and subtract largest possible in each step.

Pseudocode:

```
generate all Fibonacci numbers ≤ K
count = 0

while K > 0:
    choose largest Fib ≤ K → F
    K -= F
    count += 1

return count
```

7. Minimum Platforms

Problem: Given arrival and departure of trains, find min platforms needed at any time.

Explanation: Count overlapping trains at any time. Sort arrivals and departures, use two pointers to simulate timeline.

Pseudocode:

```
sort arrival[], departure[]
i = j = 0
platforms = max_platforms = 0

while i < n and j < n:
   if arrival[i] <= departure[j]:
      platforms += 1
      i += 1
   else:
      platforms -= 1
      j += 1
   max_platforms = max(max_platforms, platforms)

return max_platforms
```

8. Min Cost to Connect N Ropes

Problem: Connect all ropes with minimal cost; cost = sum of lengths being connected.

Explanation: Always connect two smallest ropes first. Use a Min Heap to keep smallest ropes on top.

Pseudocode:

```
push all rope lengths into min-heap
cost = 0

while heap.size > 1:
   a = pop min
   b = pop min
   cost += a + b
   push (a + b)

return cost
```

9. Max Trains on Platform

Problem: Maximize trains stopped on a single platform without overlap.

Explanation: Accept train with earliest departure time (like Activity Selection). Sort by departure time and greedily assign trains.

Pseudocode:

```
sort trains by departure time
last_departure = -1
count = 0

for train in sorted_trains:
    if train.arrival > last_departure:
        count += 1
        last_departure = train.departure

return count
```

10. Partition 1 to N into Two Groups with Min Difference

Problem: Partition set {1,2,...,n} into two groups with minimum difference in their sums.

Explanation: Greedily assign larger numbers to balance group sums. Try to balance sum by alternating placing largest values in groups.

Pseudocode:

```
sum = n * (n + 1) / 2
target = sum // 2
group1, group2 = [], []

for i from n down to 1:
    if target ≥ i:
        add i to group1
        target -= i
    else:
        add i to group2

return group1, group2
```

11. Paper Cut into Min Squares

Problem: Given length × width paper, cut into min number of squares.

Explanation: Always cut the largest square possible. Repeatedly cut a square of side min(length, width).

Pseudocode:

```
count = 0

while length > 0 and width > 0:
    side = min(length, width)
    count += 1
    if length > width:
        length -= side
    else:
        width -= side

return count
```

12. Min Diff Groups of Size Two

Problem: Group array elements in pairs so that the sum of absolute differences is minimized.

Explanation: Always pair the closest elements. Sort array and pair consecutive elements.

Pseudocode:

```
sort array
diff_sum = 0

for i = 0 to n-1 step 2:
    diff_sum += abs(arr[i+1] - arr[i])

return diff_sum
```

13. Max Satisfied Customers

Problem: Given mood of shop owner and customer count each minute, find how many customers can be satisfied using a "grumpy-free" trick for X minutes.

Explanation: Use a sliding window to find max customers that can be recovered using trick. Add all naturally satisfied customers and use sliding window of size X.

Pseudocode:

```
for i in 0 to n-1:
    if grumpy[i] == 0:
        total += customers[i]

max_gain = gain = 0

for i in 0 to X-1:
    if grumpy[i] == 1:
        gain += customers[i]
max_gain = gain

for i in X to n-1:
    if grumpy[i-X] == 1:
        gain -= customers[i-X]
    if grumpy[i] == 1:
        gain += customers[i]
    max_gain = max(max_gain, gain)

return total + max_gain
```

14. Min Initial Vertices to Traverse Matrix with Constraints

Problem: Find minimum starting points in a grid to eventually cover all '1' cells under movement rules.

Explanation: Always start from cells that can't be reached by others. Treat it as a graph problem, find nodes with no incoming edges.

Pseudocode:

```
build graph from matrix edges
calculate in-degrees of all nodes
initial_vertices = []

for node in all nodes:
    if in-degree[node] == 0:
        initial_vertices.append(node)

return initial_vertices
```

15. Largest Palindromic Number by Permuting Digits

Problem: Rearrange digits to form the largest palindrome.

Explanation: Use highest digits in mirrored pairs. Count digits, place pairs from largest to smallest.

Pseudocode:

```
count digits
half = ""

for digit from 9 to 0:
    while count[digit] ≥ 2:
        half += digit
        count[digit] -= 2

middle = ""
for digit from 9 to 0:
    if count[digit] > 0:
        middle = digit
        break

return half + middle + reverse(half)
```

16. Smallest Number with N Digits and Digit Sum S

Problem: Find smallest n-digit number whose digits sum to S.

Explanation: Fill digits from least significant to most, placing largest values last. Start from the rightmost digit.

Pseudocode:

```
if S == 0 and N > 1: return -1
digits = array of N zeros

for i from N-1 down to 0:
    if S > 9:
        digits[i] = 9
        S -= 9
    else:
        digits[i] = S
        S = 0

if digits[0] == 0:
    digits[0] = 1
    find first i > 0 where digits[i] > 0
    digits[i] -= 1

return digits as number
```

17. Lexicographically Largest Subsequence with Every Char at Least K Times

Problem: Pick a subsequence such that every chosen char appears at least K times, and result is lexicographically largest.

Explanation: Greedily select the highest letter if it has enough occurrences remaining. Use a stack and frequency map.

Pseudocode:

```
count all frequencies in input string
stack = []
seen = {}

for i = 0 to len(s)-1:
   c = s[i]
   count[c] -= 1

   if seen[c] ≥ k: continue

   while stack is not empty and c > stack.top and count[stack.top] + seen[stack.top] > k:
      popped = stack.pop()
      seen[popped] -= 1

   stack.push(c)
   seen[c] += 1

return stack as string
```

Hard Problems on Greedy Algorithm

1. Minimize the Max Height Difference (Towers Problem)

Problem: Increase or decrease each tower's height by k exactly once. Minimize the difference between tallest and shortest tower.

Explanation: Sort array. Smallest can be increased, largest can be decreased. Compare min and max after adjustment, check edge cases greedily.

Pseudocode:

```
sort heights
initial_diff = max - min

for i = 1 to n-1:
    if heights[i] - k < 0: continue
    min_elem = min(heights[0] + k, heights[i] - k)
    max_elem = max(heights[i-1] + k, heights[n-1] - k)
    result = min(result, max_elem - min_elem)

return result
```

2. Making All Elements Equal with K Updates

Problem: Make all elements in an array equal using at most k increments (only increase allowed).

Explanation: Greedily pick the most frequent high number. Use a frequency map to calculate cost of converting others to it.

Pseudocode:

```
sort array in descending order
prefix_sum = 0

for i = 1 to n:
    total_ops = i * arr[i] - prefix_sum
    if total_ops <= k:
        return arr[i]
    prefix_sum += arr[i]

return max_element
```

3. Minimize Cash Flow Among Friends

Problem: People owe each other money. Minimize number of transactions to settle all debts.

Explanation: Calculate net amount for each person. Greedily cancel the maximum debt and credit pair.

Pseudocode:

```
calculate net balance of each person
while max_credit != 0 and max_debit != 0:
    min_amount = min(max_credit, -max_debit)
    pay from debtor to creditor
    update balances

repeat until all balances are 0
```

4. Minimum Cost to Cut a Board into Squares

Problem: Cut a board into squares with given vertical and horizontal cut costs.

Explanation: Sort both cuts in descending order. Greedily apply higher cost first while tracking current segments.

Pseudocode:

```
sort vertical[] and horizontal[] descending
v_pieces = 1
h_pieces = 1

while vertical or horizontal not empty:
    if next_vertical ≥ next_horizontal:
        total_cost += next_vertical * h_pieces
        v_pieces += 1
    else:
        total_cost += next_horizontal * v_pieces
        h_pieces += 1

return total_cost
```

5. Minimum Cost to Process M Tasks with Switching Cost

Problem: Each task belongs to a type. Switching types has cost. Find min total cost.

Explanation: Use greedy + memoization. At each step, decide whether to switch or stay.

Pseudocode:

```
initialize dp[i][last_type]
for each task i:
   for each possible type:
      if type == last_type:
         dp[i][type] = min(dp[i][type], dp[i-1][type] + cost[i])
      else:
         dp[i][type] = min(dp[i][type], dp[i-1][last_type] + switch_cost + cost[i])

return min over dp[n-1][all types]
```

6. Minimum Time to Finish All Jobs (with Worker Constraints)

Problem: Assign jobs to workers with constraints such as max time per worker.

Explanation: Use binary search on answer (time limit). Greedily assign jobs to workers, and check if possible under the limit.

Pseudocode:

```
binary_search(min_time, max_time):
   mid = (low + high) / 2
   if is_possible(jobs, k_workers, mid):
      high = mid
   else:
      low = mid + 1

return low

is_possible(jobs, k, limit):
   assign jobs greedily or using backtracking
   return true if all jobs assigned within k workers
```

7. Minimum Edges to Reverse for Path from Source to Destination

Problem: Given a directed graph, reverse minimum edges to make path from src to dest.

Explanation: Model as 0-1 BFS. Normal edge has weight 0, reversed edge has weight 1. Use Dijkstra or deque for efficient min reversal count.

Pseudocode:

```
for each edge (u → v):
    add edge u→v with weight 0
    add edge v→u with weight 1

use 0-1 BFS from source:
    push edge with weight 0 to front
    push edge with weight 1 to back
    track min reversal count to reach dest
```

8. Largest Cube by Deleting Minimum Digits

Problem: Delete digits to form largest cube number.

Explanation: Try all cube numbers ≤ 10^9. For each, check if it's a subsequence of the original number.

Pseudocode:

```
for cube = largest to smallest:
    if is_subsequence(cube_str, input_str):
        return cube

return -1

is_subsequence(small, big):
    use two pointers to check if small in big
```

9. Rearrange Characters So No Two Adjacent Are Same

Problem: Rearrange string so adjacent characters are not same.

Explanation: Use Max Heap (priority queue) of char frequencies. Greedily place the highest frequency char, then the next.

Pseudocode:

```
build max_heap of (frequency, character)
prev = (0, '')  // previous char used

while heap not empty:
    freq, char = heap.pop()
    result += char
    if prev.freq > 0:
        heap.push(prev)
    prev = (freq-1, char)

if result length == input length:
    return result
else:
    return "Not possible"
```

10. Rearrange String with D Distance Between Same Characters

Problem: Rearrange string so same characters are at least d apart.

Explanation: Similar to above, but maintain a wait queue for d distance. Use max heap and a cooldown queue.

Pseudocode:

```
build max_heap of (frequency, character)
wait_queue = empty queue
time = 0
result = ""

while heap or wait_queue:
    if heap not empty:
        freq, char = heap.pop()
        result += char
        freq -= 1
        wait_queue.push((char, freq, time + d))

    if wait_queue.front().ready_time == time:
        if wait_queue.front().freq > 0:
            heap.push(wait_queue.pop())

    time += 1

if result.length == input.length:
    return result
else:
    return "Not possible"
```

Summary

Greedy algorithms provide efficient solutions to many optimization problems by making locally optimal choices. The key to success with greedy algorithms is identifying problems where:

1. **Greedy Choice Property**: A locally optimal choice leads to a globally optimal solution
2. **Optimal Substructure**: The problem can be broken down into subproblems
3. **No Future Dependencies**: Current choices don't depend on future states

Common Patterns:

- Sorting and selecting optimal elements
- Using priority queues/heaps for dynamic selection
- Interval-based problems with sorting by start/end times
- Resource allocation and scheduling problems
- Graph problems with edge selection

Time Complexity: Most greedy algorithms run in O(n log n) time due to sorting, with some running in O(n) time.

Space Complexity: Usually O(1) to O(n) depending on auxiliary data structures needed.

Remember that while greedy algorithms are efficient and intuitive, they don't always produce optimal solutions. Always verify that the greedy choice property holds for your specific problem!