DEVIN MAYA 1761702 7/17/24

CSE 102 Homework 3

Ex.1 (Playing with $O, o, \omega, \Omega, \Theta$) In this exercise, I want you to recall and use the basic definitions we used for asymptotic notation.

1. Prove the following: $f(n) = \Omega(g(n))$ if and only if $g(n) = O(f(n))$. To do this you need to use the definitions for O, Ω.
2. Prove that $\sqrt{n + 399} = \Theta(\sqrt{n})$.
3. Justify: True or False: If we know $f(n) = \Omega(g(n))$ and $h(n) = O(g(n))$, then we can safely conclude that $h(n) = o(f(n))$. (to justify you need to compare the relevant definitions or find counterexamples).
4. Justify: True or False: If we know $f(n) = o(g(n))$ and $h(n) = \Theta(g(n))$, then we can safely conclude that $f(n) = O(h(n))$. (to justify you need to compare the relevant definitions or find counterexamples).
5. Let $g(n) = n$ and $f(n) = (1+\cos n \cdot \sin^{2024}(n-\frac{\pi}{2}))\cdot n$. A student claims that $f(n) = \Omega(g(n))$. Is this True or False? Justify your answer in either case.
6. Justify: True or False: If we know $f(n) = o(g(n))$ and $h(n) = O(g(n))$, then we can safely conclude that $f(n) = O(h(n))$. (to justify you need to compare the relevant definitions or find counterexamples).
7. Let $g(n) = n$ and $f(n) = (2 + \sin n \cdot \cos^{2024} n) \cdot n$. Prove that $g(n) = \Theta(f(n))$.
8. Prove or disprove: If $f(n) = \Theta(g(n))$, then $f(n)^2 = \Theta(g(n)^2)$. Recall the definition of big-Theta. Be careful with your calculations.
9. Prove or disprove: If $f(n) = \Theta(g(n))$, then $2^{f(n)} = \Theta(2^{g(n)})$. Recall the definition of big-Theta. Be careful with your calculations.
10. Prove or disprove: If $f(n) = O(g(n))$, then $2^{f(n)} = O(2^{g(n)})$. Recall the definition of big-O. Be careful with your calculations.

Solution.

1. Show that $f(n) = \Omega(g(n))$ if and only if $g(n) = O(f(n))$. **Proof:** ($\Rightarrow$) Assume $f(n) = \Omega(g(n))$. Then, by definition, $\exists c > 0$ and $n_0 > 0$ such that $f(n) \geq cg(n)$ for all $n \geq n_0$. Equivalently, $g(n) \leq \frac{1}{c} f(n)$ for all $n \geq n_0$, which is precisely the definition of $g(n) = O(f(n))$.

   ($\Leftarrow$) Now suppose $g(n) = O(f(n))$. Then, $\exists c > 0$ and $n_0 > 0$ such that $g(n) \leq cf(n)$

for all $n \geq n_0$. This implies $f(n) \geq \frac{1}{c}g(n)$ for all $n \geq n_0$, which matches the definition of $f(n) = \Omega(g(n))$.

2. Demonstrate that $\sqrt{n+399} = \Theta(\sqrt{n})$.

   Proof: To prove this, we must find constants $c_1, c_2 > 0$ and $n_0 > 0$ such that $c_1\sqrt{n} \leq \sqrt{n+399} \leq c_2\sqrt{n}$ for all $n \geq n_0$.

   For the lower bound, observe that $\sqrt{n+399} \geq \sqrt{n}$ for all $n > 0$. Thus, we can choose $c_1 = 1$.

   For the upper bound, note that $\sqrt{n+399} \leq \sqrt{n+n} = \sqrt{2n}$ for all $n > 399$. Hence, $\sqrt{n+399} \leq \sqrt{2}\sqrt{n}$, so we can set $c_2 = \sqrt{2}$.

   In summary, $\sqrt{n} \leq \sqrt{n+399} \leq \sqrt{2}\sqrt{n}$ for all $n > 399$, confirming that $\sqrt{n+399} = \Theta(\sqrt{n})$.

3. Assess the following claim: If $f(n) = \Omega(g(n))$ and $h(n) = O(g(n))$, then it necessarily follows that $h(n) = o(f(n))$.

   False. Consider the counter-example: $f(n) = n$, $g(n) = n$, and $h(n) = n$. Here, $f(n) = \Omega(g(n))$ and $h(n) = O(g(n))$ hold, but $h(n) \neq o(f(n))$.

4. Evaluate the validity of the following statement: If $f(n) = o(g(n))$ and $h(n) = \Theta(g(n))$, then we can confidently deduce that $f(n) = O(h(n))$.

   True. Given $f(n) = o(g(n))$, for any $c > 0$, $\exists n_0$ such that $f(n) < cg(n)$ for all $n \geq n_0$. From $h(n) = \Theta(g(n))$, we know $\exists c_1, c_2 > 0$ and n_1 such that $c_1 g(n) \leq h(n) \leq c_2 g(n)$ for all $n \geq n_1$. Let $c' = \frac{c}{c_1}$. Then, for $n \geq \max(n_0, n_1)$, we have $f(n) < cg(n) \leq c'h(n)$. Thus, $f(n) = O(h(n))$.

5. Consider $g(n) = n$ and $f(n) = (1 + \cos(n) \cdot \sin^{2024}(n - \frac{\pi}{2})) \cdot n$. A student asserts that $f(n) = \Omega(g(n))$. Is this assertion valid? Justify your response.

   True. Observe that $-1 \leq \cos n \leq 1$ and $-1 \leq \sin(n - \frac{\pi}{2}) \leq 1$ for all n. Consequently, $0 \leq 1 + \cos n \cdot \sin^{2024}(n - \frac{\pi}{2}) \leq 2$ for all n. It follows that $0 \leq f(n) \leq 2n$ for all n. Since $f(n) \geq 0 \cdot n$ for all n, we can infer that $f(n) = \Omega(g(n))$.

6. Determine the correctness of the following claim: If $f(n) = o(g(n))$ and $h(n) = O(g(n))$, then it necessarily holds that $f(n) = O(h(n))$.

   False. A counter-example suffices: Let $f(n) = \log n$, $g(n) = n$, and $h(n) = \sqrt{n}$. In this case, $f(n) = o(g(n))$ and $h(n) = O(g(n))$ are true, but $f(n) \neq O(h(n))$.

7. Let $g(n) = n$ and $f(n) = (2 + \sin(n) \cdot \cos^{2024}(n)) \cdot n$. Show that $g(n) = \Theta(f(n))$.

Proof: Our goal is to find constants $c_1, c_2 > 0$ and $n_0 > 0$ such that $c_1 n \leq (2 + \sin n \cdot \cos^{2024} n) \cdot n \leq c_2 n$ for all $n \geq n_0$. Since $-1 \leq \sin n \leq 1$ and $-1 \leq \cos n \leq 1$ for all n, we have $1 \leq 2 + \sin n \cdot \cos^{2024} n \leq 3$ for all n. As a result, $n \leq f(n) \leq 3n$ for all $n > 0$. Choosing $c_1 = 1$ and $c_2 = 3$ satisfies the definition of $g(n) = \Theta(f(n))$.

8. Establish the validity of the following proposition: If $f(n) = \Theta(g(n))$, then $f(n)^2 = \Theta(g(n)^2)$.

   Proof: Assume $f(n) = \Theta(g(n))$. Then, $\exists c_1, c_2 > 0$ and n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$. By squaring both sides (noting that $g(n)$ is non-negative for sufficiently large n since $f(n)$ is), we obtain: $c_1^2 g(n)^2 \leq f(n)^2 \leq c_2^2 g(n)^2$ for all $n \geq n_0$. This aligns with the definition of $f(n)^2 = \Theta(g(n)^2)$. Hence, the proposition holds.

9. Assess the correctness of the following claim: If $f(n) = \Theta(g(n))$, then $2^{f(n)} = \Theta(2^{g(n)})$.

   False. A counter-example disproves the claim: Consider $f(n) = 2n$ and $g(n) = n$. Here, $f(n) = \Theta(g(n))$ is true, but $2^{f(n)} = 2^{2n} = (2^n)^2 \neq \Theta(2^n) = \Theta(2^{g(n)})$.

10. Determine the validity of the following statement: If $f(n) = O(g(n))$, then $2^{f(n)} = O(2^{g(n)})$.

    False. A counter-example suffices: Let $f(n) = n$ and $g(n) = n + 1$. While $f(n) = O(g(n))$ holds, $2^{f(n)} = 2^n \neq O(2^{n+1}) = O(2^{g(n)})$.

Ex.2 (Sorting Tortillas) In the Famous Chef's Restaurant, they make (circular) Tortillas of all different sizes for the demanding customers. The Chef faces the following problem and needs your help. In front of you is a stack of n tortillas of different sizes. You want to sort the tortillas so that smaller tortillas are on top of larger tortillas. Unfortunately, you are not allowed to touch any food, but you have a spatula with which you can perform a "flip" operation: insert the spatula under the top k tortillas, for any integer k you choose between 1 and n, and flip them all over. See Figure 1.

Describe an algorithm to sort an arbitrary stack of n tortillas using $O(n)$ flip operations with the spatula. Follow-up question: Exactly how many flips does your algorithm perform in the worst case? (Hint: This problem has nothing to do with the Tower of Hanoi.)

Solution.

Exercise 2: Sorting Tortillas

Problem Description

In the Famous Chef's Restaurant, there is a stack of n tortillas of different sizes. The objective is to sort the tortillas so that smaller tortillas are on top of larger ones. The only permitted operation is a "flip": inserting a spatula under the top k tortillas, for any integer k between 1 and n, and flipping them all over.

Algorithm Description

We can sort the tortillas using a modified selection sort algorithm:

1. Begin with the unsorted stack of n tortillas.
2. For i from n down to 2:
   (a) Identify the position j of the largest tortilla among the top i tortillas.
   (b) If $j \neq i$ (i.e., the largest tortilla is not already at the bottom of the unsorted portion):
      - If $j \neq 1$, execute flip(j) to bring the largest tortilla to the top.
      - Execute flip(i) to move the largest tortilla to its correct position at the bottom of the unsorted portion.

Proof of Correctness

In each iteration, we position the largest remaining tortilla in its correct spot at the bottom of the unsorted portion. After $n - 1$ iterations, all tortillas except the smallest one will be in their correct positions, with the smallest on top.

Time Complexity Analysis

- Locating the largest tortilla: $O(n)$ per iteration
- At most 2 flips per iteration: $O(1)$ per iteration
- Total iterations: $n - 1$

Thus, the total time complexity is $O(n^2)$ comparisons and $O(n)$ flips.

Number of Flips in Worst Case

In the worst case, we execute 2 flips in each of the $n-1$ iterations:

- One flip to bring the largest tortilla to the top (if it's not already there)
- One flip to move it to its correct position

Therefore, the maximum number of flips is $2(n-1) = 2n-2$.

Optimality

This algorithm is optimal in terms of the number of flips, as we require at least $n-1$ flips to ensure each tortilla (except the smallest) is in its correct position, and in the worst case, we may need to flip twice for each.

Example

Consider a stack of 6 tortillas with sizes [4, 2, 6, 1, 5, 3]:

1. Find the largest tortilla among the top 6: it is 6 at position 3. Flip the top 3 tortillas: [6, 2, 4, 1, 5, 3]. Flip the top 6 tortillas to move 6 to the bottom: [3, 5, 1, 4, 2, 6].
2. Find the largest tortilla among the top 5: it is 5 at position 2. Flip the top 2 tortillas: [5, 3, 1, 4, 2, 6]. Flip the top 5 tortillas to move 5 to the correct position: [2, 4, 1, 3, 5, 6].
3. Find the largest tortilla among the top 4: it is 4 at position 2. Flip the top 2 tortillas: [4, 2, 1, 3, 5, 6]. Flip the top 4 tortillas to move 4 to the correct position: [3, 1, 2, 4, 5, 6].
4. Find the largest tortilla among the top 3: it is 3 at position 1. Flip the top 3 tortillas to move 3 to the correct position: [2, 1, 3, 4, 5, 6].
5. Find the largest tortilla among the top 2: it is 2 at position 1. Flip the top 2 tortillas: [1, 2, 3, 4, 5, 6].
6. The stack is now sorted.

This example used 7 flips.

Conclusion

This algorithm successfully sorts a stack of n tortillas using $O(n)$ flip operations, with a worst-case performance of exactly $2n - 2$ flips. It is optimal in terms of the number of flips required.

Ex.3 (Walking in Manhattan) Manhattan, New York, NY is a very organized city so walking in the streets is easy. We can think of the road network as an n-by-n square board. Imagine that your initial position is $(0, 0)$ and you want to reach your destination at position (n, n). You are allowed only to **move right or up** and you cannot leave the board. Moreover at all times, your x coordinate should be larger or equal to the y coordinate (you **never cross the diagonal**). Design a dynamic programming algorithm that computes the number of possible ways to reach your destination.

For example, for n = 4, in Figure 2 you can see two possible paths (green and blue lines, with orange we have the diagonal that we cannot cross). In total there are 14 ways (count them and convince yourself).

Solution.

Exercise 3: Walking in Manhattan

Problem Description

We start at position (0,0) on an n-by-n square board representing Manhattan's road network. The goal is to reach position (n, n) under the following constraints:

- We can only move right or up.
- We must stay on or above the diagonal (i.e., $x \geq y$ at all times).
- We cannot leave the board.

The task is to design a dynamic programming algorithm that computes the number of possible ways to reach the destination.

Dynamic Programming Solution

We define a 2D array dp where $dp[i][j]$ represents the number of ways to reach the point (i, j) from $(0, 0)$ without crossing the diagonal.

Base Cases

- $dp[0][0] = 1$ (one way to start)
- $dp[i][0] = 1$ for all i (only one way to reach points on the x-axis)
- $dp[0][j] = 0$ for all $j > 0$ (cannot reach y-axis points except origin)
- $dp[i][j] = 0$ if $i < j$ (positions below the diagonal are unreachable)

Recurrence Relation

For $i \geq j$ and $i, j > 0$:

$$dp[i][j] = dp[i-1][j] + dp[i][j-1]$$

This is because we can reach (i, j) either from $(i-1, j)$ by moving up, or from $(i, j-1)$ by moving right.

Algorithm Steps

1. Initialize a 2D array dp of size $(n+1) \times (n+1)$ with all elements set to 0.
2. Set base cases:
   - $dp[0][0] = 1$
   - For i from 1 to n: $dp[i][0] = 1$
3. Fill the dp table:
   - For i from 1 to n:
     - For j from 1 to i: (Ensure we stay on or above diagonal)
       - $dp[i][j] = dp[i-1][j] + dp[i][j-1]$
4. Return $dp[n][n]$

Pseudocode

function CountWays(n):

Initialize dp array of size $(n+1) \times (n+1)$ with all zeros

$dp[0][0] \leftarrow 1$

for $i \leftarrow 1$ **to** n **do**

$dp[i][0] \leftarrow 1$

for $i \leftarrow 1$ **to** n **do**

  for $j \leftarrow 1$ **to** i **do**

    $dp[i][j] \leftarrow dp[i-1][j] + dp[i][j-1]$

return $dp[n][n]$

Time and Space Complexity

- **Time Complexity:** $O(n^2)$ as we fill approximately half of an $n \times n$ table.
- **Space Complexity:** $O(n^2)$ for the dp table.

Correctness

The algorithm correctly computes all possible paths because:

- It respects the constraints (only moving right or up, staying on/above the diagonal).
- It considers all possible ways to reach each point.
- It builds up the solution from smaller subproblems to the final destination.
- The base cases and recurrence relation cover all scenarios.

Example for $n = 4$

The dp table (final state) for $n = 4$:

$$\begin{array}{ccccc} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 2 & 2 & 0 & 0 \\ 1 & 3 & 5 & 5 & 0 \\ 1 & 4 & 9 & 14 & 14 \end{array}$$

Note: The dp table is $(n+1) \times (n+1)$ in size, which is why it's a 5x5 table for $n = 4$.

The answer, $dp[4][4] = 14$, represents the number of valid paths from $(0, 0)$ to $(4, 4)$.

Verification

We can verify this result by listing all possible paths:

- RRRRUUUU

- RRRURUUU
- RRRUURUU
- RRRUUURU
- RRRUUUUR
- RRURRUU
- RRURURUU
- RRURUURU
- RRURUUUR
- RRUURURR
- RURRRUU
- RURURUU
- RURURUR
- RUURRRU

where R represents a right move and U represents an up move.

This confirms that there are indeed 14 valid paths, matching our dynamic programming result.

Ex.4 (Cool Numbers) You are given **distinct** integers $a_1, \ldots, a_n$. A pair of numbers (a_i, a_j) is called **cool** if $a_i = a_j + 20$.

- For the case where the n integers are unsorted, design an algorithm that runs in time $\Theta(n \log n)$ and returns the total number of cool pairs as the output. Justify your answer. (Hint: start with a small example and count the "cool" pairs.)
- A pair of numbers (a_i, a_j) is called **super-cool** if $a_i = a_j + 42$. For the case where the n given integers are initially sorted, design an algorithm that runs in linear time $\Theta(n)$ and returns the total number of "super-cool" pairs. Justify your answer.

Ex.4 (Cool Numbers) You are given **distinct** integers $a_1, \ldots, a_n$. A pair of numbers (a_i, a_j) is called **cool** if $a_i = a_j + 20$.

- For the case where the n integers are unsorted, design an algorithm that runs in time $\Theta(n \log n)$ and returns the total number of cool pairs as the output. Justify your answer. (Hint: start with a small example and count the "cool" pairs.)

- A pair of numbers (a_i, a_j) is called **super-cool** if $a_i = a_j + 42$. For the case where the n given integers are initially sorted, design an algorithm that runs in linear time $\Theta(n)$ and returns the total number of "super-cool" pairs. Justify your answer.

Solution.

Exercise 4: Cool Numbers

Given **distinct** integers $a_1, \ldots, a_n$, we define:

- A pair of numbers (a_i, a_j) is called **cool** if $a_i = a_j + 20$.
- A pair of numbers (a_i, a_j) is called **super-cool** if $a_i = a_j + 42$.

Part 1: Finding Cool Pairs in Unsorted Array

Design an algorithm that runs in time $\Theta(n \log n)$ and returns the total number of cool pairs.

Algorithm

1. Sort the array $a_1, \ldots, a_n$ in ascending order.
2. Initialize a counter for cool pairs to 0.
3. Use two pointers, $left$ and $right$, starting at the beginning of the sorted array.
4. While $right < n$:
   - If $a_{right} - a_{left} = 20$, increment the counter and move both pointers.
   - If $a_{right} - a_{left} < 20$, move $right$ forward.
   - If $a_{right} - a_{left} > 20$, move $left$ forward.
5. Return the counter.

Pseudocode

function FindCoolPairs($a[0 \ldots n-1]$):

sort(a)

$left, right \leftarrow 0, 0$

$coolCount \leftarrow 0$

```
    while left < n and right < n do
        if a[right] − a[left] = 20 then
            coolCount ← coolCount + 1
            left ← left + 1
            right ← right + 1
        else if a[right] − a[left] < 20 then
            right ← right + 1
        else
            left ← left + 1
    return coolCount
```

Time Complexity

- Sorting: $O(n \log n)$
- Two-pointer traversal: $O(n)$
- Overall: $\Theta(n \log n)$

Correctness

The algorithm correctly counts all cool pairs because:

- Sorting allows us to efficiently identify pairs with a difference of 20.
- The two-pointer approach ensures we don't miss any cool pairs and don't count any pair twice.
- We only move forward, ensuring $O(n)$ time for the traversal after sorting.

Part 2: Finding Super-Cool Pairs in Sorted Array

Design an algorithm that runs in linear time $\Theta(n)$ and returns the total number of super-cool pairs, given that the input array is initially sorted.

Algorithm

1. Initialize two pointers, $left = 0$ and $right = 0$, and a counter for super-cool pairs to 0.
2. While $right < n$:
   - If $a_{right} - a_{left} = 42$, increment the counter and move both pointers.
   - If $a_{right} - a_{left} < 42$, move $right$ forward.
   - If $a_{right} - a_{left} > 42$, move $left$ forward.
3. Return the counter.

Pseudocode

function FindSuperCoolPairs($a[0 \dots n-1]$):

$left, right \leftarrow 0, 0$

$superCoolCount \leftarrow 0$

while $left < n$ **and** $right < n$ **do**

if $a[right] - a[left] = 42$ **then**

$superCoolCount \leftarrow superCoolCount + 1$

$left \leftarrow left + 1$

$right \leftarrow right + 1$

else if $a[right] - a[left] < 42$ **then**

$right \leftarrow right + 1$

else

$left \leftarrow left + 1$

return $superCoolCount$

Time Complexity

- The algorithm performs a single pass through the array with two pointers.
- Each pointer moves at most n times.
- Therefore, the time complexity is $\Theta(n)$.

Correctness

The algorithm correctly counts all super-cool pairs because:

- The array is already sorted, so we can directly apply the two-pointer technique.
- This approach ensures we find all pairs with a difference of 42 without any duplicates.
- The linear time complexity is achieved by never moving backwards and processing each element at most twice.

Justification

Both algorithms implement the sorted nature of the array either after sorting in Part 1 or given in Part 2 to efficiently find pairs with a specific difference. The two-pointer technique allows to solve the problem without nested loops thus having the required time complexities of $\Theta(n \log n)$ for the unsorted case and $\Theta(n)$ for the pre-sorted case.

Ex.5 (Collecting Stamps) We have collected n stamps $s_1, s_2, \ldots, s_n$ each associated with a different positive integer value $v_1, v_2, \ldots, v_n$. We want to determine if there is a subset of the stamps whose total sum of values is equal to a given integer V amount of dollars. (Hint: we saw similar problems in class, like the Knapsack and Coin Changing problems.)

For example, if we have $n = 5$ stamps with values $v_1 = 5, v_2 = 6, v_3 = 18, v_4 = 45, v_5 = 79$ and we want to create the total sum $V = 90$, the answer is YES since $v_1 + v_2 + v_5 = 5 + 6 + 79 = 90$, so taking the first, second, and fifth stamp will do the job.

- Design a dynamic programming algorithm that runs in time $O(nV)$ and answers the question.
- Moreover, design a reconstruction algorithm that outputs the subset of the stamps whose total sum of values equals V.

Solution.

Exercise 5: Collecting Stamps

We have collected n stamps $s_1, s_2, \ldots, s_n$ each associated with a different positive integer value $v_1, v_2, \ldots, v_n$. We want to determine if there is a subset of the stamps whose total sum of values is equal to a given integer V amount of dollars.

Part 1: Dynamic Programming Solution

Design a dynamic programming algorithm that runs in time $O(nV)$ and answers the question.

Problem Formulation

- Input: n stamps with values $v_1, v_2, \ldots, v_n$, and target value V
- Output: Boolean indicating whether a subset of stamps sums to V

Dynamic Programming Approach

Let $dp[i][j]$ be a boolean value indicating whether it's possible to achieve a sum of j using a subset of the first i stamps.

Base Cases

- $dp[0][0] = true$ (empty subset sums to 0)
- $dp[0][j] = false$ for $j > 0$ (can't make positive sum with no stamps)

Recurrence Relation

For i from 1 to n and j from 0 to V:

$$dp[i][j] = dp[i-1][j] \vee (j \geq v_i \wedge dp[i-1][j-v_i])$$

This relation considers two cases:

- Not including the i-th stamp: $dp[i-1][j]$
- Including the i-th stamp if possible: $j \geq v_i \wedge dp[i-1][j-v_i]$

Algorithm

Initialize $dp[n+1][V+1]$ with $false$

Set $dp[0][0] \leftarrow true$

for $i \leftarrow 1$ **to** n **do**

  for $j \leftarrow 0$ **to** V **do**

    $dp[i][j] \leftarrow dp[i-1][j]$

    if $j \geq v_i$ **then**

      $dp[i][j] \leftarrow dp[i][j] \vee dp[i-1][j-v_i]$

return $dp[n][V]$

Complexity Analysis

- Time Complexity: $O(nV)$ - we fill a table of size $(n+1) \times (V+1)$
- Space Complexity: $O(nV)$ for the DP table

Space Optimization Note

The space complexity can be optimized to $O(V)$ by using a 1D array instead of a 2D array, as we only need the previous row's information at each step.

Part 2: Subset Reconstruction

Design a reconstruction algorithm that outputs the subset of the stamps whose total sum of values equals V.

Algorithm

Subset Reconstruction Algorithm [1] $dp[n][V]$ is $false$ "No valid subset exists" Initialize an empty list 'subset' Set $i = n$, $j = V$ $i > 0$ and $j > 0$ $dp[i-1][j]$ is $false$ Add stamp i to 'subset' $j = j - v_i$ $i = i - 1$ 'subset'

Reconstruction Complexity

The reconstruction algorithm runs in $O(n)$ time as it traverses back through at most n stamps.

Correctness

- The DP approach considers all possible combinations of stamps.
- The recurrence relation correctly captures the decision to include or exclude each stamp.
- The reconstruction algorithm traces the decisions made during the DP process.

Example

Given $n = 5$ stamps with values $v_1 = 5, v_2 = 6, v_3 = 18, v_4 = 45, v_5 = 79$ and target sum $V = 90$:

- The DP table will have $dp[5][90] = true$
- The reconstruction will return $[1, 2, 5]$, corresponding to stamps with values 5, 6, and 79.

- Indeed, $5 + 6 + 79 = 90$, verifying the correctness of our solution.

Ex.6 (Celebrity Show) Near Hollywoo (no "d") in LA lies the Bojack Horseman universe, where there is a celebrity show called "Hollywoo stars and Celebrities: what do they know? do they know things? let's find out!" presented by Mr. Peanutbutter. In this show, there are exactly n different categories of trivia questions, arranged in a row in front of the players. Each question if answered correctly gives the player some value, denoted $v_1, \ldots, v_n$ for each of the n categories. The values are known to the players. Fortunately for the players, Mr. Peanutbutter knows all the answers to the multiple choice questions, and unconsciously, he is giving them away by raising his ears every time he reads through the correct choice. So we can assume players will always get the questions right and score the associated value points.

Bojack faces Pinky Penguin and they play by alternating turns. In each turn, a player selects either the **first** or **last** question category from the row, removes it from the row permanently, and receives the value of the category (since the player will answer correctly). Bojack, being a famous celebrity, has the choice of either starting first, or letting his opponent start answering questions first.

Given the row with the n values $v_1, \ldots, v_n$, determine whether or not Bojack should go first. That is, **compute the maximum possible value Bojack can definitely win if he moves first or second**, and pick the best for him. Then, give a **reconstruction algorithm that tells Bojack exactly which categories (and in what row) to pick**. (Hint: your algorithm needs to be efficient, so brute-force algorithms are excluded. Instead, use dynamic programming.)

Note: Pinky Penguin is as clever as Bojack.

Before attempting the exercise, think of the following examples: In the first example, the values are $50, 30, 70, 100$. In this case, it's better if Bojack goes first because he can score 150, by starting with the question of value 100, then Pinky Penguin will select the question of value 70, and then Bojack will select the question of value 50. In the second example, we have values $80, 150, 30, 70$. Bojack here should go first and will get 220: first he picks 70 (**and not 80**), then Pinky Penguin should select 80, then Bojack selects 150 and so he gets 70+150=220. Notice that the greedy strategy of picking the largest available number is not correct.

Solution.

Exercise 6: Celebrity Show

Dynamic Programming Solution

We'll use a 2D array dp where $dp[i][j]$ represents the maximum score difference Bojack can achieve over Pinky for the subarray from index i to j.

Base Cases

- For $i = j$: $dp[i][i] = v[i]$ (only one category left, player takes it)

Recurrence Relation

For $i < j$:

$$dp[i][j] = \max(v[i] - dp[i+1][j], v[j] - dp[i][j-1])$$

This represents Bojack choosing either the first or last category and subtracting the best Pinky can do with the remaining categories.

Algorithm

function CelebrityShow($v[0 \dots n-1]$):

Initialize $dp[n][n]$ with 0

for $i \leftarrow 0$ **to** $n-1$ **do**

$dp[i][i] \leftarrow v[i]$

for $len \leftarrow 2$ **to** n **do**

for $i \leftarrow 0$ **to** $n - len$ **do**

$j \leftarrow i + len - 1$

$dp[i][j] \leftarrow \max(v[i] - dp[i+1][j], v[j] - dp[i][j-1])$

if $dp[0][n-1] > 0$ **then**

return "Bojack should go first"

else

return "Bojack should go second"

Reconstruction Algorithm

To determine Bojack's moves:

function ReconstructMoves($v[0 \dots n-1], dp[0 \dots n-1][0 \dots n-1]$):

$i \leftarrow 0$, $j \leftarrow n-1$

while $i \leq j$ **do**

if $v[i] - dp[i+1][j] > v[j] - dp[i][j-1]$ **then**

Print "Bojack selects first category (value $v[i]$)"

$i \leftarrow i + 1$

else

Print "Bojack selects last category (value $v[j]$)"

$j \leftarrow j - 1$

if $i \leq j$ **then**

if $dp[i+1][j] > dp[i][j-1]$ **then**

$i \leftarrow i + 1$ // *Pinky's move*

else

$j \leftarrow j - 1$ // *Pinky's move*

Time and Space Complexity

- Time Complexity: $O(n^2)$ to fill the dp table
- Space Complexity: $O(n^2)$ for the dp table

Correctness

The dynamic programming approach considers all possible game scenarios and accounts for both players playing optimally. The reconstruction algorithm follows the optimal decisions made during the DP process.

Examples

1. For $v = [50, 30, 70, 100]$:

- $dp[1][4] = 50 > 0$, so Bojack should go first
- Optimal moves: Bojack takes 100, Pinky takes 70, Bojack takes 50
- Bojack's score: 150

2. For $v = [80, 150, 30, 70]$:

- $dp[1][4] = 70 > 0$, so Bojack should go first
- Optimal moves: Bojack takes 70, Pinky takes 80, Bojack takes 150
- Bojack's score: 220