CS-GY 6033 : Design and Analysis of Algorithms , Section C : Assignment-4

Khwaab Thareja — N15911999 — kt3180

November 2024

Problem 4-1: Billboard Placement

Suppose you are managing the construction of billboards on an east-west highway that extends in a straight line. The possible sites for billboards are given by numbers $x_1, x_2, \ldots, x_n$ with $0 \le x_1 < x_2 < \cdots < x_n$, specifying their distance in miles from the west end of the highway. If you place a billboard at location x_i, you receive payment $p_i > 0$. Regulations imposed by the county's Highway Department require that any pair of billboards be more than 5 miles apart. You'd like to place billboards at a subset of the sites so as to maximize your total revenue, subject to that placement restriction.

Example. Suppose $n = 4$, with

$$\langle x_1, x_2, x_3, x_4 \rangle = \langle 6, 7, 12, 14 \rangle,$$

and

$$\langle p_1, p_2, p_3, p_4 \rangle = \langle 5, 6, 5, 1 \rangle.$$

The optimal solution is to place billboards at x_1 and x_3, for a total revenue of $p_1 + p_3 = 10$. (a) Professor Manfine proposes the following "greedy" algorithm,

which operates by placing billboards as early as possible to satisfy the 5-mile restriction:

Algorithm 1: Billboard Placement Algorithm

```
1. Place a billboard at $x_1$.
2. lastboard $\leftarrow x_1$
3. $P \leftarrow p_1$                                  total revenue
4. for $i \leftarrow 2$ to $n$ do
    (a) if $x_i >$ lastboard $+ 5$ then
```

```
            i. Place a billboard at $x_i$.
           ii. lastboard $\leftarrow x_i$.
          iii. $P \leftarrow P + p_i$.
      (b) end if
   5. end for
   6. return $P$.
```

Give an example (i.e., choose input $\{x_i\}, \{p_i\}$) for which Professor Manfine's algorithm does not return the maximum revenue.

The remainder of the problem asks you to provide a dynamic-programming algorithm that takes as input an instance (i.e., locations $\{x_i\}$ given in sorted order and their prices $\{p_i\}$) and returns the maximum revenue obtainable from a valid subset of sites. Your running time should only depend on n, not on the value of x_n or any of the p_i.

(b) For this problem part, suppose you have access to a function `prev`(j) that returns the latest billboard i that precedes billboard $j > i$ by more than 5 miles, i.e., `prev`$(j) = \max\{i < j \mid x_i < x_j - 5\}$. Define a recursive formula for the value of the solution. You will want to use `prev`(j).

(c) Write pseudocode for a dynamic program that returns the maximum possible revenue, again using `prev` as a blackbox.

(d) Give an algorithm to implement `prev`(j). Analyze the running time of your dynamic program given this algorithm for `prev`.

It's actually possible to achieve a $\Theta(n)$-time algorithm, but it's likely that your first crack at part d led to something much slower. That's okay — a slower (polynomial time) solution is still worth full credit. However, it's worth thinking about how to improve your solution. Can you figure out how to reduce the running time to $O(n \log n)$ or, even better, $\Theta(n)$?

This problem highlights a general algorithm design principle — your high-level algorithm should specify what you want, not how to do it. I've already pushed you in that direction by defining the `prev` function. Only after you've given the main ideas of the algorithm should you resolve implementation details. Doing it this way yields a more flexible algorithm with better modularity: if you can improve the solution for any particular step, you can swap it in without breaking your main algorithm.

Answer a

Consider the following billboard locations and revenues:

- **Locations**: $x = \langle 6, 7, 13, 14 \rangle$
- **Revenues**: $p = \langle 5, 6, 5, 1 \rangle$

Greedy Algorithm Process

1. Place a billboard at $x_1 = 6$, earning 5.
2. Skip $x_2 = 7$ as it is within 5 miles of x_1.
3. Place a billboard at $x_3 = 13$, adding another 5.
4. Skip $x_4 = 14$ as it is less than 5 miles from x_3.

Total revenue from the greedy approach: 10 (billboards at x_1 and x_3).

Optimal Solution

Place billboards at $x_2 = 7$ and $x_3 = 13$ for a total revenue of $6 + 5 = 11$.

Conclusion

The greedy algorithm, while simple, can miss the optimal solution. In this case, placing billboards at x_2 and x_3 results in a higher revenue of 11. A dynamic programming method would be needed to determine the optimal placement.

Python Code for Maximum Revenue Calculation

```python
def max_revenue(n, x, p):

    # M[i] stores the maximum revenue up to the i-th site
        (1-indexed)
    M = [0] * (n + 1)

    for i in range(1, n + 1):
        # Option 1: Do not place a billboard at the i-th
            site
        M[i] = M[i - 1]

        # Option 2: Place a billboard at the i-th site
        # Find the last valid position j such that x[j] <= x
            [i] - 5
        j = i - 1
        while j > 0 and x[i - 1] - x[j - 1] < 5:
            j -= 1

        # If j is valid, add the revenue up to j; otherwise,
            just take the current p[i-1]
            if j > 0:
                M[i] = max(M[i], p[i - 1] + M[j])
            else:
                M[i] = max(M[i], p[i - 1])

    return M[n]
```

The algorithm iterates through all n sites in the main loop (`for i in range(1, n + 1):`), resulting in $O(n)$ operations.

For each site i, it performs a nested search to find the last valid j such that $x[j] \leq x[i] - 5$. This search is performed using a `while` loop, which in the worst case can run up to $O(n)$ iterations if no valid j is found early.

Time Complexity:

- The outer loop runs $O(n)$ times.
- The inner `while` loop can also run $O(n)$ in the worst case.

Thus, the overall time complexity is $O(n^2)$ in the worst case.

Answer b

Define dp$[j]$ as the maximum revenue possible up to the j-th site. To follow the 5-mile rule, we use a helper function prev(j) that finds the last valid site i before j such that $x[i] < x[j] - 5$. The formula is:

$$\text{dp}[j] = \max(\text{dp}[j-1], p_j + \text{dp}[\text{prev}(j)])$$

- dp$[j-1]$: The revenue without placing a billboard at j, using the best result up to $j - 1$.
- $p_j + \text{dp}[\text{prev}(j)]$: The revenue from placing a billboard at j, plus the best revenue from a valid previous site.

Base Case:

$$\text{dp}[0] = p[0] \quad \text{(if placing a billboard at the first site)}$$

Helper Function:

$$\text{prev}(j) = \text{last } i \text{ such that } i < j \text{ and } x[i] < x[j] - 5$$

This function finds the last valid spot before j that maintains the 5-mile distance.

```
def MAX_REVENUE(j, x, p):
1.  if j == 0:
2.      return 0  # Base case: no sites, no revenue
3.
4.  # Option 1: Do not place a billboard at site j
5.  revenue_without_j = MAX_REVENUE(j - 1, x, p)
6.
7.  # Option 2: Place a billboard at site j
8.  i = PREV(j, x)  # Find the latest valid i such that x[i]
     < x[j] - 5
9.  if i is not None:  # Check if a valid i exists
10.     revenue_with_j = p[j - 1] + MAX_REVENUE(i, x, p)
11. else:
12.     revenue_with_j = p[j - 1]
13.
14. # Return the maximum revenue at site j
15. return max(revenue_without_j, revenue_with_j)
```

Answer c

```
def max_revenue(n, x, p):
    # Initialize an array M to store the maximum revenue up
       to each site
    M = [0] * (n + 1)

    # Iterate over each billboard site
    for j in range(1, n + 1):
        # Option 1: Do not place a billboard at site j
        M[j] = M[j - 1]

        # Option 2: Place a billboard at site j
        i = prev(j, x)  # Find the latest valid i such that
            x[i] < x[j] - 5

        # If a valid i is found, calculate the revenue with
            placing at site j
        if i >= 0:
            M[j] = max(M[j], p[j - 1] + M[i + 1])
        else:
            M[j] = max(M[j], p[j - 1])
```

```python

    return M[n]
```

Answer d

```python
def prev(j, x):
    # Iterate backwards from j-1 to find the last valid site
    i=j-1
    while i>=0:
        if x[j]-x[i]>5:
            return i
        i=i-1
    return -1  # No valid site found
```

Time Complexity Explanation

1. `prevIndex` Function

- The `prevIndex` function checks backward from the current index j to find the nearest site that is more than 5 miles away. - It only needs to check up to 5 previous sites at most. If no valid site is found within these 5 checks, there will not be one. - Therefore, each call to `prevIndex` takes constant time, or $O(1)$.

2. Main Loop in `billboardCost`

- The main loop runs $O(n)$ times, where n is the number of sites. - Each iteration calls `prevIndex`, which, as explained, takes $O(1)$ time.

3. Total Time Complexity

- The main loop runs $O(n)$ times, and each `prevIndex` call takes $O(1)$ time. - Thus, the total time complexity is:

$$O(n) \times O(1) = O(n)$$

Summary

- The `prevIndex` function runs in constant time, and the main loop iterates n times, resulting in an overall time complexity of $O(n)$.

Problem 4-2: Chain Splitting

Suppose Scrooge McDuck wants to split a gold chain into $n+1$ smaller chains to distribute to his numerous nephews and nieces as their inheritance. Scrooge has carefully marked the n locations where he wants the chain to be cut.

McDuck's jeweler, Fenton Crackshell, agrees to cut any chain of length ℓ into two smaller pieces, at any specified location, for a fee of ℓ. To cut a chain into more pieces, Scrooge must pay Crackshell separately for each individual cut. As a result, the cost of breaking the chain into multiple pieces depends on the order that Crackshell makes his cuts. Obviously, Scrooge wants to pay Crackshell as little as possible.

Example. Suppose the chain is 12 inches long, and the cut locations are at 3 inches, 7 inches, and 9 inches from one end of the chain. All of these positions must be cut—the only question is what order should the cuts be made in. If Crackshell cuts first at the 3-inch mark, then at the 7-inch, then at the 9-inch mark, Scrooge must pay $12+9+5=26$. If Crackshell cuts the chain first at the 9-inch mark, then at the 7-inch mark, and then at the 3-inch mark, Scrooge must pay $12+9+7=28$. Finally, if Crackshell makes his first cut at the 7-inch mark, Scrooge only owes him $12+7+5=24$. This last solution is the best answer for this input.

Input format. The input to the algorithm is a set of integers $p_0, p_1, p_2, \ldots, p_{n+1}$, with $0 = p_0 < p_1 < p_2 < \cdots < p_{n+1}$. The number $p_0 = 0$ is the beginning of the chain, and the number p_{n+1} is the end of the chain (or equivalently its total length). The numbers $p_1, \ldots, p_n$ specify the positions at which cuts need to be made. In the above example, the input would be $0, 3, 7, 9, 12$.

(a) Scrooge's great-nephew Huey suggests the following greedy strategy: cut the median position first, then recurse. This algorithm is given by the following pseudocode, starting with a call to `MAKECUTS(0, n + 1)`.

Algorithm 2: `MAKECUTS(i, j)` $\triangleright$ Cut the subchain $p_i, p_{i+1}, \ldots, p_j$ at $p_{i+1}, \ldots, p_{j-1}$

```
1. if $j \leq i+1$ then
2.     No cut to be made. return
3. end if
4. mid $\leftarrow \frac{i+j}{2}$
5. Cut at position $p_{\texttt{mid}}$
6. MAKECUTS(i, mid)
7. MAKECUTS(mid, j)
```

Provide an input for which Huey's algorithm does not produce the cheapest ordering of cuts. You don't need an input larger than $n = 3$, but the example above doesn't break Huey's algorithm.

The remainder of the problem is about designing a good dynamic-programming solution.

(b) Define a recursive formula for the optimal value of cutting the chain.

(c) Write pseudocode for a dynamic program that determines the value of the optimal solution. (You do not have to output the order of cuts.) To simplify the code, you can feel free to use mathematical expressions like taking the min of multiple choices.

(d) Analyze the running time of your algorithm.

Answer a

Analysis of Example $p = [0, 3, 9, 19, 31]$

Input Details:

- The total chain length is 31 inches.
- Cut positions are at 3, 9, 19, and 31 inches.

Applying Huey's Median-Cut Algorithm:

Initial Cut:

- The median cut is made at position 9.
- The cost for this cut is \$31 (the length of the full chain).

Recursive Subchain Cuts:

- For the subchain [0, 3, 9], cut at the median position 3.
- The cost for this cut is \$9 (length of the subchain).
- For the subchain [9, 19, 31], cut at the median position 19.
- The cost for this cut is \$22 (length of the subchain).

Total Cost Using Huey's Method:

$$\text{Total cost} = \$31 \text{ (cut at 9)} + \$9 \text{ (cut at 3)} + \$22 \text{ (cut at 19)} = \$62.$$

Finding the Optimal Solution:

Best Sequence of Cuts:

- Start with the cut at position 19 (cost = \$31).
- Next, cut at position 9 (cost = \$19).
- Finally, cut at position 3 (cost = \$9).

Total Cost for the Optimal Solution:

$$\text{Total cost} = \$31 \text{ (cut at 19)} + \$19 \text{ (cut at 9)} + \$9 \text{ (cut at 3)} = \$59.$$

Comparison:

- Huey's algorithm cost: \$62.
- Optimal solution cost: \$59.

Conclusion:

This example $p = [0, 3, 9, 19, 31]$ illustrates that Huey's median-cut strategy does not always result in the lowest cost. To find the minimum cutting cost, a dynamic programming approach is needed.

Answer b

To find the minimum cost of cutting a chain, we need to establish a recursive formula that computes the lowest cost for making cuts between specified positions. Let $C(i, j)$ represent the minimum cost of cutting the subchain from p_i to p_j (where $p_0 = 0$ is the start of the chain, and p_{n+1} is the chain's total length). This formula evaluates all potential intermediate cut positions to find the most cost-effective split.

Recursive Formula:

$$C(i,j) = \begin{cases} 0, & \text{if } j \le i+1 \quad \text{(no cuts needed)} \\ \min_{k=i+1}^{j-1} \left(C(i,k) + C(k,j) + (p_j - p_i) \right), & \text{otherwise} \end{cases}$$

Explanation:

- **Base Case**: When $j \le i + 1$, no cuts are needed, so the cost is 0.
- **Recursive Case**:

- Loop through each valid cut position k between i and j (i.e., $i < k < j$).
- Calculate the cost of making a cut at k, including the cost of recursively solving the subchains $C(i, k)$ and $C(k, j)$.
- Include the cost of cutting the entire subchain, $p_j - p_i$.
- Find the minimum cost from all evaluated k.

This recursive method ensures that all potential cutting sequences are considered, and the one with the lowest total cost is chosen.

```python
def min_cuts_cost_recursive(p, i, j):
    # Base case: If the subchain has no cuts between i and j, the cost is 0
    if j <= i + 1:
        return 0

    # Initialize minimum cost to a large value
    min_cost = float('inf')

    # Try every possible cut position between i and j and calculate the cost
    for k in range(i + 1, j):
        # Cost of making the current cut and recursively cutting the left and right subchains
        cost = (p[j] - p[i]) + min_cuts_cost_recursive(p, i, k) + min_cuts_cost_recursive(p, k, j)
        # Update the minimum cost
        min_cost = min(min_cost, cost)

    return min_cost
```

```python
def min_cuts_cost(p):
    n = len(p) - 1  # Total number of positions including start and end
    return min_cuts_cost_recursive(p, 0, n)
```

Answer c

```python
def min_cuts_cost(p):
    n = len(p) - 2  # n is the number of cuts, excluding the endpoints 0 and p[n+1]
    # Create a 2D array to store the minimum cost for
      cutting subchains
    C = [[0] * (n + 2) for _ in range(n + 2)]

    # Fill the DP table
    for length in range(2, n + 2):  # length of subchain (
      starts from 2 to include endpoints)
        for i in range(n + 2 - length):
            j = i + length
            C[i][j] = float('inf')
            for k in range(i + 1, j):  # Try every cut point
                 between i and j
                C[i][j] = min(C[i][j], (p[j] - p[i]) + C[i][
                   k] + C[k][j])

    return C[0][n + 1]
```

Answer d

Time Complexity Analysis

1. Outer Loop:

The outer loop iterates through subchains of increasing length, starting from 2 up to $n + 1$. This loop runs $O(n)$ times, where n is the number of cuts.

2. Middle Loop:

The loop for i runs from 0 to $n + 2 - \text{length}$, also with a time complexity of $O(n)$.

3. Innermost Loop:

The innermost loop goes through each potential cut position k between i and j, with a worst-case time complexity of $O(n)$.

Overall Time Complexity:

The total time complexity is $O(n)$ for the outer loop, $O(n)$ for the middle loop, and $O(n)$ for the innermost loop. This results in:

$$O(n) \times O(n) \times O(n) = O(n^3)$$

Final Time Complexity:

$$O(n^3)$$

Problem 4-3: Biking

You've decided to go on a long-distance bicycle trip across the country. You've already designed your route and identified reasonable campsites along the way. The question is where you should set up camp. Your entire trip can be modeled as a straight line, starting at mile marker 0. The problem input specifies the campsites along the way, at mile markers $0 < x_1 < x_2 < \cdots < x_n$, where x_i is the distance from the starting point of the i-th possible campsite. You must stop at the final campsite (at distance x_n), which is your destination.

The most pleasant ride for you is 50 miles a day, but this may not be possible (depending on the spacing of campsites). If you travel x miles during a day, the penalty for that day is $(50 - x)^2$. You'd like to plan your campsites to minimize the total penalty—that is, the sum, over all days, of the daily penalties.

(a) Clearly define a recursive formula for the minimum total penalty possible.

(b) Write pseudocode for a dynamic program that computes the minimum total penalty for your trip.

(c) Analyze the running time of your algorithm.

(d) Augment your code to also output which campsites to stop at.

Answer a

Let the penalty function be defined as follows:

$$\text{Penalty}(x) = (50 - x)^2$$

where x is the distance traveled in a single day.

Let MinPenalty(i) represent the minimum penalty to reach the i-th campsite at x_i. To calculate the minimum total penalty, we need to consider that each day's ride could end at a different campsite before reaching the final one, so we can set up the recursive formula as follows:

$$\text{MinPenalty}(i) = \min_{j<i} \left(\text{MinPenalty}(j) + (50 - (x_i - x_j))^2 \right)$$

Where:

- $x_i - x_j$ represents the distance traveled from campsite j to campsite i,
- MinPenalty(j) is the minimum penalty up to campsite j,
- The base case is MinPenalty(0) = 0, assuming there's no penalty to start from mile marker 0.

The recursive formula captures the minimum penalty by considering the minimum penalty from any previous campsite j to the current campsite i, plus the penalty incurred by the distance traveled on that day.

Answer b

We can solve this problem using dynamic programming with the following pseudocode:

Algorithm 1 Biking Minimum Penalty

```
Input: A list of campsites campsites
Output: Minimum penalty and the optimal stops
Let $n$ be the length of campsites
Initialize dp array of size $n$, where dp[$i$] stores the minimum penalty to reach campsite $i$
Set dp[0] = 0
Initialize previous_stop array to track optimal stops
for each campsite $i$ from 1 to $n-1$ do
    for each previous campsite $j$ from 0 to $i-1$ do
        distance = campsites[$i$] − campsites[$j$]
        penalty = $(50 - \text{distance})^2$
        if dp[$j$] + penalty < dp[$i$] then
            dp[$i$] = dp[$j$] + penalty
            previous_stop[$i$] = $j$
        end if
    end for
end for
return dp[$n-1$], previous_stop =0
```

Answer c

The time complexity of this dynamic programming solution is as follows:

- The outer loop iterates over each campsite i from 1 to $n-1$, which takes $O(n)$.
- For each campsite i, the inner loop iterates over each previous campsite j, which also takes $O(n)$.

Therefore, the time complexity is:

$$O(n^2)$$

Answer d

To also retrieve the sequence of campsites to stop at, we can backtrack from the last campsite using the previous_stop array that we filled in the DP step. Here's the modified code:

Explanation of the Backtracking Process

After filling in the dp array with minimum penalties, we backtrack from the final campsite (index $n-1$) using the previous_stop array. Each entry in

Algorithm 2 Biking Minimum Penalty with Stops

```
**Input:** A list of campsites campsites
**Output:** Minimum penalty and the optimal stops
Let $n$ be the length of campsites
Initialize dp array of size $n$, where dp[$i$] stores the minimum penalty to reach campsite $i$
Set dp[0] = 0
Initialize previous_stop array to track optimal stops
for each campsite $i$ from 1 to $n-1$ do
  for each previous campsite $j$ from 0 to $i-1$ do
    $\text{distance} = \text{campsites}[i] - \text{campsites}[j]$
    $\text{penalty} = (50 - \text{distance})^2$
    if $\text{dp}[j] + \text{penalty} < \text{dp}[i]$ then
      $\text{dp}[i] = \text{dp}[j] + \text{penalty}$
      $\text{previous\_stop}[i] = j$
    end if
  end for
end for
$\text{min\_penalty} = \text{dp}[n-1]$
Initialize stops = []
current = $n-1$
while current $\neq -1$ do
  Add campsites[current] to stops
  current = previous_stop[current]
end while
Reverse the list of stops to get the sequence from start to finish
return $\text{min\_penalty}, \text{stops}$ =0
```

previous_stop[i] tells us the previous campsite where we stopped before reaching campsite i. By repeatedly following these indices, we can construct the list of campsites where we stopped. Finally, we reverse the list to get the sequence from start to finish.

Problem 4-4: Minimizing Chapter Unbalancedness

Mr. Hugo has written a long article consisting of n paragraphs. The i-th paragraph contains a_i words. As an editor, your task is to help Mr. Hugo assemble his article into several chapters by dividing the paragraphs under the following conditions:

1. Paragraphs must remain in their original order.
2. Each paragraph must belong entirely to one chapter. It cannot be split between chapters.

Your goal is to split the article into at least two chapters such that the unbalancedness of the book is minimized. The unbalancedness is defined as the largest difference between the number of words in the longest and the shortest chapters.

Example

Given the following input:

- Paragraph word counts: $A = [4, 3, 8, 5, 2, 6]$
- Explanation: We will divide all paragraphs into 4 chapters. The first chapter contains paragraphs $A[1]$, $A[2]$, the second chapter contains paragraph $A[3]$, the third chapter contains paragraphs $A[4]$, $A[5]$, and the fourth chapter contains paragraph $A[6]$. The unbalancedness is $8 - 6 = 2$, where 8 is the word count of the longest chapter (Chapter 2), and 6 is the word count of the shortest chapter (Chapter 4).

The algorithm should find the optimal way to split these 6 paragraphs into 4 chapters, minimizing the unbalancedness.

(a) Define a recursive formula for the optimal value of splitting the article into chapters to minimize unbalancedness.

(b) Write pseudocode describing your algorithm.

(c) Analyze the running time of your algorithm.

Answer a

Let dp$[i][k]$ be the minimum difference in word count between the longest and shortest chapters when dividing the first i paragraphs into k chapters.

- **Base Case**: If we put all i paragraphs into one chapter ($k = 1$), then:

$$\text{dp}[i][1] = \text{prefix}[i] = \sum_{j=1}^{i} A[j]$$

  This is just the total word count from the first paragraph to the ith paragraph.

- **Recursive Case**: For more than one chapter ($k > 1$), we find the minimum difference by checking all possible places to split the paragraphs:

$$\text{dp}[i][k] = \min_{j<i} \left(\max(\text{dp}[j][k-1].\text{max}, \text{currChapterSum}) - \min(\text{dp}[j][k-1].\text{min}, \text{currChapterSum})\right)$$

  where:

$$\text{currChapterSum} = \text{prefix}[i] - \text{prefix}[j]$$

This formula looks at all places j where we can split the paragraphs up to i, calculates the word count for the new chapter from $j+1$ to i, and finds the split that results in the smallest difference between the largest and smallest chapter word counts.

Answer b

Algorithm 3 Minimize Unbalancedness

```
function minimizeUnbalancedness(totalParagraphs, words)
  Initialize prefixSum array of size totalParagraphs + 1 with prefixSum[0] = 0
  for paraIndex = 1 to totalParagraphs do
    prefixSum[paraIndex] = prefixSum[paraIndex  1] + words[paraIndex  1 {Calculate cumulative word count}
  end for
  Initialize 2D array dp[paraIndex][chapters] of pairs {minWords, maxWords}, initialized with {INT_MAX, INT_MIN}
  for paraIndex = 1 to totalParagraphs do
    dp[paraIndex][1] = {prefixSum[paraIndex], prefixSum[paraIndex]} {Base case: one chapter}
  end for
  for chapterCount = 2 to totalParagraphs do {Iterate over number of chapters}
    for paraIndex = chapterCount to totalParagraphs do {Ensure enough paragraphs for chapters}
      Initialize minUnbalancedness = INT_MAX
      Initialize currentMax = INT_MIN, currentMin = INT_MAX
      for splitPoint = chapterCount  1 to paraIndex  1 do
        currentChapterSum = prefixSum[paraIndex]  prefixSum[splitPoint]
        previousMax = dp[splitPoint][chapterCount  1].maxWords
        previousMin = dp[splitPoint][chapterCount  1].minWords
        updatedMax = max(previousMax, currentChapterSum)
        updatedMin = min(previousMin, currentChapterSum)
        unbalancedness = updatedMax  updatedMin
        if unbalancedness < minUnbalancedness then
          minUnbalancedness = unbalancedness
          currentMax = updatedMax
          currentMin = updatedMin
        end if
      end for
      dp[paraIndex][chapterCount] = {currentMax, currentMin}
    end for
  end for
  Initialize finalResult = INT_MAX
  for chapterCount = 2 to totalParagraphs do
    finalResult = min(finalResult, dp[totalParagraphs][chapterCount].maxWords
      - dp[totalParagraphs][chapterCount].minWords)
  end for
  return finalResult {Return minimum achievable unbalancedness}
end function=0
```

Answer c

1. **Calculating Prefix Sums**:
    - Creating the prefix sum array takes $O(n)$ time, where n is the number of paragraphs. This is done with a single pass through the array.
2. **Filling the DP Table**:
    - The dynamic programming table $\text{dp}[i][k]$ is filled using three loops:
        - The outer loop (for chapters k) runs $O(n)$ times.
        - The middle loop (for paragraphs i) also runs $O(n)$ times.
        - The innermost loop (to find the split point j) runs up to $O(n)$ times.
    - Overall, filling the table takes $O(n) \times O(n) \times O(n) = O(n^3)$ time.
3. **Getting the Final Answer**:
    - Finding the minimum unbalancedness from the last row of the table takes $O(n)$ time.

Overall Time Complexity:

$$O(n) + O(n^3) + O(n) = O(n^3)$$

The final time complexity is $O(n^3)$, which is the most important term.

Problem 4-6: Inversions Revisited 2

Given an array A of n numbers, we say that a 5-tuple $(i_1, i_2, \ldots, i_5)$ of integers is inverted if:

$$1 \leq i_1 < i_2 < i_3 < \cdots < i_5 \leq n \quad \text{and} \quad A[i_1] > A[i_2] > A[i_3] > \cdots > A[i_5].$$

(a) Give an $O(n^2)$-time algorithm to count the number of such inverted tuples with respect to A.

(b) Give an $O(n \log n)$-time algorithm to count the number of such inverted tuples with respect to A.

Answer a

Algorithm

Algorithm 4 Count Decreasing Subsequences of Length 5

```
Initialize dp[i][k] to 0 for all $i$ and $k$
Set dp[i][1] $= 1$ for all $i$
for $i = 1$ to $n - 1$ do
    for $j = 0$ to $i - 1$ do
        if $A[j] > A[i]$ then
            for $k = 2$ to $5$ do
                dp[i][k] += dp[j][k - 1]
            end for
        end if
    end for
end for
count $= \sum_{i=0}^{n-1}$ dp[i][5]
return count $=0$
```

Explanation of the Algorithm

- **Initialization**:
    - Let `dp[i][k]` be a 2D array where each entry represents the number of decreasing subsequences of length k that end at the i-th element in the array A.
    - Initialize `dp[i][k] = 0` for all i and k.

  - Set `dp[i][1] = 1` for all i, because each individual element is a subsequence of length 1.

- **Nested Loop Structure**:
  - Iterate through the array A using a loop from $i = 1$ to $n - 1$:
    - For each i, iterate through all j such that $0 \leq j < i$.
    - If $A[j] > A[i]$, indicating a decreasing relationship:
      - Update `dp[i][k]` for $k = 2$ to 5 as follows:

        ```
        dp[i][k] += dp[j][k-1]
        ```

        This step accumulates the number of decreasing subsequences ending at index i with length k.

- **Final Count**:
  - The total number of decreasing subsequences of length 5 in the array can be found by summing over all `dp[i][5]`:

    $$\texttt{count} = \sum_{i=0}^{n-1} \texttt{dp[i][5]}$$

  - Return `count`.

Algorithm Implementation

Answer b

Pseudocode for Counting Inverted 5-Tuples

Algorithm 5 Count Inverted 5-Tuples

```
1: Input: Array $A$ of length $n$
2: Output: Total number of inverted 5-tuples
3: Initialize sorted lists sorted_list_1, sorted_list_2, sorted_list_3,
   sorted_list_4, sorted_list_5
4: Initialize DP arrays dp1, dp2, dp3, dp4, dp5 to zero of size $n$
5: for $i = 0$ to $n - 1$ do
6:     dp1[i] = 1
7:     Add $A[i]$ to sorted_list_1
8:     dp2[i] = count of elements greater than $A[i]$ in sorted_list_1
9:     Add $(A[i], dp2[i])$ to sorted_list_2
10:    dp3[i] = sum of counts for elements less than $A[i]$ in sorted_list_2
11:    Add $(A[i], dp3[i])$ to sorted_list_3
12:    dp4[i] = sum of counts for elements less than $A[i]$ in sorted_list_3
13:    Add $(A[i], dp4[i])$ to sorted_list_4
14:    dp5[i] = sum of counts for elements less than $A[i]$ in sorted_list_4
15:    Add $(A[i], dp5[i])$ to sorted_list_5
16: end for
17: return sum of dp5 =0
```