Joyce Chen Dis 1F 2:00PM Parshan Teimouri

CS180 Homework #5

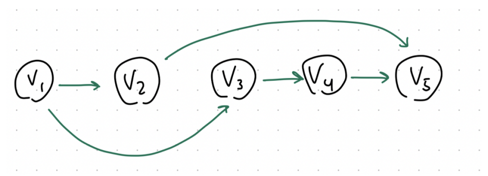
1. **Given an unsorted integer array, find all pairs with a given difference k in it without using any extra space (sorting is allowed).**
   arr = [1, 5, 2, 2, 2, 5, 5, 4] k = 3
   Output: (2, 5) and (1, 4)

Algorithm
- Sort the array from smallest to largest
- Remove all duplicate integers from the array
- Let pointer i and j both start from 0
- While arr[j] - arr[i] >= k or j is not pointing to the last element
    - If arr[j] - arr[i] == k
        - Output (arr[i], arr[j]) as a pair
        - Increment both i and j
    - If arr[j] - arr[i] < k
        - Increment j, such that we get a larger difference
    - If arr[j] - arr[i] > k
        - Increment i, such that we get a smaller difference

Proof of Correctness
- Suppose that there exists a pair (x, y) where y-x = k, but our algorithm doesn't output this pair.
- This means that our algorithm never considered the pair (x, y). There was never a point in the algorithm where arr[i] == x and arr[j] == y simultaneously.
- This can only happen if both i and j are incremented at once, such that there was no consideration for the pairs (arr[i-old], arr[j]) and (arr[i], arr[j-old]).
- Our algorithm only increments i and j when arr[j]-arr[i] == k.
    - Case x !== arr[i] and y !== arr[j]: we would've considered the pair. ><
    - Case x == arr[i] and y == arr[j]: we would've already output the pair in the previous round.
    - Case x == arr[i] and y !== arr[j]: y-x couldn't have equaled k. ><
    - Case x !== arr[i] and y == arr[j]: y-x couldn't have equaled k. ><
- By proof of contradiction, our algorithm works.

Time Complexity
- Sorting the array takes O(NlogN) time.
- We are only going through the array once, which is O(N) time.
- Total time = O(NlogN).
- Not using extra space to store intermediate results, so space complexity is O(1)

2. **Exercise 3 on page 314**
   a) **Example ordered graph:**



The algorithm wouldn't produce a correct result for this graph because it greedily chooses the edge (w, v-j) where j is the smallest, without consideration that an edge (w, v-k) where k > j could be part of the longest path, where v-k isn't necessarily connected to v-j. It would choose (v1, v2) as the first edge, then (v2, v5), outputting 2 as the longest path.
The correct answer in this case is 3: we choose the edges (v1, v3), (v3, v4), and (v4, v5). As we can see, the greedy approach doesn't even consider (v1, v3) as a potential solution.

b) **Give an efficient algorithm that takes an ordered graph G and returns the length of the longest path that begins at v1 and ends at vn.**

Algorithm

- Create an array arr of length n to store the longest path length from v1 to each vertex.
- For each vertex v-i in G, where i = {1…n}:
    - If there are no incoming edges to v-i:
        - Let arr[i] = 0
    - If there are incoming edge(s) (v-a, v-b, …v-h, v-i) to v-i:
        - Let arr[i] = max(arr[a], arr[b], … arr[h]) + 1
- Return arr[n-1], which is the longest path from v1 to vn.

Proof of Correctness

- Base case n = 1:
    - Because there would be no incoming edges to v-1, we return 0.
    - Indeed, the longest path = 0.
- Inductive hypothesis n > 1:
    - Assume that we know the longest path from v1 to v(n-1).
    - To find the longest path to vn, consider all edges leading into vn.
    - Because all edges leading into vn must be from a node i < n, we know the longest path to all these preceding nodes.
    - Therefore, we can find the longest path to vn by taking the max of these paths and adding 1 for the newest edge.
- Our algorithm finds the longest path from v1 to vn.

Time Complexity

- Assume there are n nodes and m edges in the graph.
- Creating an array of length n takes O(n) time
- We are checking all the incoming edges of each node, so we can charge the time to the edges of the graph, which is O(m)
- Total time = O(n + m)

        - Space complexity =O(n), since we are using an auxiliary array of size n.

3. **Exercise 5 on page 316**

   Algorithm
   - Create an array arr of length n to store the max quality scores of each substring starting at index 0.
   - Set arr[0] to be the score of the first letter.
   - For each index i (starting at 1) in the given string y:
     - Let j start from 1 and increment to i.
     - Consider every possible segmentation of the string:
       - Get the max score of substring y(0, j-1) by referencing arr[j-1].
       - Calculate the score of the word y(j, i).
       - Let current final score = arr[j-1] + score[y(j, i)]
     - Set arr[i] to the max of all these final scores.
   - Final answer will be arr[n-1].

   Proof of Correctness
   - This algorithm works on the basis that for any string segmented to yield a max quality score, all its substrings beginning at index 0 will also be optimally segmented. For instance, if *abcde* were optimally segmented (ie. ab cd e), then its substring *abcd* will also be optimally segmented (ab cd), and *abc* (ab c), etc.
   - Base case n = 1
     - There is only 1 letter in the string, so that itself gives the max score.
   - Inductive hypothesis n > 1
     - Assume that we have the max scores for all substrings (0…i) where i < n for a string of length n.
     - Consider the last word in the optimal segmentation, which starts at the index j <= i.
     - The preceding segmentation from 0 to j-1 yields the max score for that substring.
     - Therefore, the max score of the string n will be the score of this segmentation + the score of the last word.
     - Our algorithm considers each index j that the last word could start at (essentially checking every possible “last word combo”), so we will always find the last word that yields the max score in each step.
   - Our algorithm finds the optimal segmentation of a string that yields the max quality score.

   Time Complexity
   - Traversing the string from beginning to end takes O(N) time.
   - For each index i of the string, we consider the max score of every possible substring between 0 and i, which takes O(N^2) time.
   - Total = O(N^2) time, with a space complexity of O(N) because we are using an auxiliary array.

4. **Exercise 10 on Page 321**
   a) **Example input:**

	Minute 1	**Minute 2**	**Minute 3**
A	10	0	0
B	0	20	20

This is an example input that will not work with the given algorithm. The algorithm greedily chooses the machine with the greater of a_1 and b_1, so it would pick A to start off.
However, the optimal plan would be to stick with B for all 3 minutes, which will give 40 steps. If we start off with A, then we would have to switch to B for the 2nd minute which gives us 0 steps. The algorithm given returns 30 steps, when the actual optimal solution is 40 steps.

   b) **Give an efficient algorithm that takes values for a1, a2,..., an and b1, b2,..., bn and returns the value of an optimal plan.**

Algorithm

- Break into subproblems of smaller minute groups.
- Create an array A, where each cell A[i] holds the optimal plan from minute 1 to minute i, ending on machine A.
- Create an array B, where each cell B[i] holds the optimal plan from minute 1 to minute i, ending on machine B.
- Create an array arr, where each cell arr[i] holds the optimal plan from minute 1 to minute i.
- Initialize A[0] = 0, A[1] = a_i, B[0] = 0, B[1] = b_i
- For each minute i = 2 to n:
    - A[i] = max{A[i-1] + a_i, A[i-2] + b_i}
    - B[i] = max{B[i-1] + b_i, B[i-2] + a_i}
    - arr[i] = max{A[i], B[i]}
- Return arr[n]

Proof of Correctness

- We are essentially finding *max{solution(a-1…a-i) + solution(b-i+1…b-n), solution(b-1…b-i) + solution(a-i+1, a-n), solution(a-1…a-n), solution(b-1…b-n)}* for each minute i.
- Base case i = 1
    - Because there is only 1 step in either machine, we just choose the greater of the 2 steps.
- Inductive hypothesis i > 1
    - Assume that we know the maximum steps (optimal plan) from minute 1 all the way up to minute i-1.

        - To figure out the maximum # of steps up to minute i, we consider where we left off at the previous minute i-1.
        - There are 4 choices: we would have either been on machine A, machine B, moving from A to B, or moving from B to A.
        - If we end on A in minute i, the max steps for i would be a_i + the maximum of the previous i-1 steps ending at A and the previous i-2 steps ending at B.
        - If we end on B in minute i, max steps for i would be b_i + max of the previous i-1 steps ending at B and previous i-2 steps ending at A.
        - Therefore, we are able to calculate the max steps up to i by taking the maximum of the 2 ending options.
    - Our algorithm works.

Time Complexity

- Since we are memoizing A[i] for each iteration, referencing it for minutes > i takes constant O(1) time.
- We are calculating A[i] and B[i] (similarly memoized) for n times, so the algorithm takes O(N) total time.
- Space complexity is O(N) since we have 3 auxiliary arrays all of size N.

5. **Given a rod of length n inches and an array of prices that contains prices of all pieces of size smaller than n. Determine the maximum value obtainable by cutting up the rod and selling the pieces. For example, if length of the rod is 8 and the values of different pieces are given as following, then the maximum obtainable value is 22 (by cutting in two pieces of lengths 2 and 6)**

length	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
price	**1**	**5**	**8**	**9**	**10**	**17**	**17**	**20**

Algorithm

- Create a 2D array called arr, where arr[i] stores the max price for a rod of length l_i, and i = {0…n}.
- Initialize arr[0] = 0, because a rod of length 0 will not have a price.
- For i = 1 to n:
    - For each length l_i from 1 to i-1:
        - arr[i] = max{arr[L-l_i] + price(l_i)}
- Return arr[n], the max price for a rod of length n.

Proof of Correctness

- Base case n = 0
    - Since the rod is length 0, the maximum price possible is 0.
    - Our algorithm initializes arr[0] = 0, so this works.
- Inductive hypothesis n > 0
    - Assume that we know the max possible prices for rods of length 0…n-1.

    - Suppose the optimal segmentation of the rod is dividing it into a length of n-x and x, where x < n.
        - If we know the max price for length n-x, the max price for length n = [max price of the rod n-x + price of the remaining length].
    - To figure out whether this segmentation is truly optimal, we consider all possible "optimal" segmentations of the rod, then take the one yielding the max possible price.
- Therefore, our algorithm is correct.

Time Complexity

- For a rod of length n, we break it down into subproblems where each rod has length n-1, n-2, …0, which takes O(n).
- Because we are memoizing the subproblems in our array, it takes O(1) time to access the solution to a subproblem.
- For every rod length we parse, we have an inner for loop that checks for the maximum value out of all rods with a smaller length, which takes O(n).
- Therefore, the total runtime is O(n^2), and space complexity is O(n) because we're using an auxiliary array of length n.

6. **Consider a row of n coins of values v1 . . . vn, where n is even. We play a game against an opponent by alternating turns (you can both see all coins at all times) . In each turn, a player selects either the first or last coin from the row, removes it from the row permanently, and receives the value of the coin. Determine the maximum possible amount of money we can win if we move first, assuming your opponent is using an optimal strategy.**

    Example 1: [5, 3, 7, 10] : The user collects the maximum value as 15 (10 + 5) - Sometimes the greedy strategy works.
    Example 2. [8, 15, 3, 7] : The user collects the maximum value as 22 (7 + 15) - In general the greedy strategy does not work.

Algorithm

- Create a 2D array arr[n][n], where each cell arr[i][j] stores the maximum value our player can get for the sub-array of coins starting at index i and ending at index j.
- For i = n-1; i >= 0; i--
    - For j = n-1; j >= i; j--
        - If i == j: //base case where we only have 1 coin left
            - arr[i][j] = coins[i]
        - else if j-i == 1: //base case, choose max out of 2 coins left
            - arr[i][j] = max(coins[i], coins[j])
        - else: //either pick first or last coin
            - pickFirstCoin = coin[i] + min(arr[i+2][j], arr[i+1][j-1])
            - pickLastCoin = coin[j] + min(arr[i][j-2], arr[i+1][j-1])
            - arr[i][j] = max(pickFirstCoin, pickLastCoin)

Proof of Correctness

- Idea behind the algorithm:
    - Because the opponent is also using an optimal strategy, we will always be left with the subarray that minimizes our score during our turn. This is due to the opponent always choosing the coin that maximizes their score during their turn.
    - Out of this minimum subarray, we must pick either the first or last coin. Obviously, we must choose the coin that maximizes our overall score.
    - We are doing a top-down approach to this recurrence relation.
- Base case n <= 2:
    - If n = 1, there is only 1 coin and we choose that coin
    - If n = 2, there are only 2 coins and we choose the maximum of the 2
- Inductive hypothesis n > 2:
    - Assume that we know the max score we can achieve for a coin array of length n-1.
    - To know the max score for a coin array of length n, we must choose either the first or last coin, whichever yields the max score when combined with the score we got for a subarray of length n-1.
    - Therefore, we will get the maximum possible score for a coins array with length n.
- Our algorithm works.

Time Complexity

- We are iterating through all possible starting indices of the coins array, which takes O(n) time.
- For each starting index, we are also checking all possible last indices so as to partition the coins array in all possible ways, which takes O(n) time.
- Therefore, total runtime is O(n^2).
- Space complexity = O(n^2) because we are using a 2D array.