Com sci 180 Homework 5. Discussion 1D
Krish Patel

Dynamic programming:

Problem 1)

We have an array of length n, where repetitions are allowed. Given the array, we first sort using a nlog n sorting algo, such as heap sort or quicksort. This would allow us to optimize our algorithm efficiently. If the algorithm only has one element, then we output no pair and change the array. We subtract the first value from the last value of the array and if the difference is less than k, we terminate the program. If not, we start by creating two pointers, each initially pointing to the first and second element respectively. We then calculate the difference and compare it to the value k. If the value is the same, then we output this pair and increment the second pointer to the next element. However, if the difference is less than k, we increment the second pointer to move to the next element if the array. If the difference is larger than the value of k, we increment the value of the first pointer. However, if the first and second pointer point to the same element, we increment the second pointer to point to the next element. Repeat until you reach till the end of the array, and then end the program.

This should have a maximum time complexity of O(nlog n), which comes from the sorting part of the algorithm. The rest of the array traversal has a time complexity of O(n) as each of the pointers only visit an element once, thus giving the time complexity as O(2n) = O(n).

Proof that this works:
The algorithm correctly identifies pairs of elements in the array with a difference of k. The sorting step ensures that the array is in ascending order, which simplifies the pair searching process and would shorten the search for the difference between two values. By using two pointers to traverse the sorted array and comparing differences with k, the algorithm successfully identifies pairs that meet the difference critera. Its time complexity analysis supports its efficiency, making it a reliable method for finding pairs with a difference of k in an array.

Problem 2:
Let G = (V,E) be a directed graph with nodes v1,v2,v3…vn

a).

Consider the ordered graph G with vertices (v_1, v_2, v_3, v_4, v_5) and edges: (v_1, v_2), (v_2, v_3), (v_3, v_4), (v_4, v_5), and (v_5, v_1).
Using the algorithm:

1. We first start from v_1.
2. Follow edges to v_2, v_3, v_4, and v_5 in sequence.
3. Finally, from v_5, we choose the edge (v_5, v_1), and set w back to v_1. However, since we have already visited all vertices once, the algorithm terminates. This would return 4

However, The correct path length is 5, which is a cycle (v_1, v_2), (v_2, v_3), (v_3, v_4), (v_4, v_5), (v_5, v_1). Thus this is not ideal.

b)

1. Create an array L of length n, where L[i] represents the longest path from v_1 to v_i.
2. Initialize all elements of L to 0.
3. Iterate through vertices v_i from v_2 to v_n.
    - For each v_i, iterate over vertices v_j with j < i.
        - If there exists a directed edge from v_j to v_i, update L[i] as max(L[i], L[j] + 1).
4. Return L[n] as the length of the longest path from v_1 to v_n.

The time complexity of the algorithm is O(n^2) because it uses two nested loops, each iterating through n terms. The outer loop has O(n) iterations, and for each iteration, the inner loop checks for a directed edge, which is an O(1) operation. The combination of these nested loops results in a quadratic time complexity of O(n^2).

Proof: By induction, assume L[i] is the length of the longest path from v_1 to v_i.
Base case: For our base case wherei = 2, L[2] is the longest path from v_1 to v_2, which is the optimal solution.

Inductive step: Assume L[j] is the longest path from v_1 to v_j where j < i. The optimal path length to v_i is L[i], which is the max of L[j] + 1 if there is a directed edge from v_j to v_i. This holds for all j < i, so L[j] represents the optimal solution for all subproblems.
Thus, using principle of induction, we find that this algorithm would produce the optimal solution for all i>=2. Hence proved>>>

Problem 3:
Algo:
We start with an array of all the letters in an ordered sequential manner. We then follow the following steps:

1) First we initialize a dynamic programming table (a 2d array) with dp[0] initialized as 0 (dp[i] holds the value of the best segmentation at the given i).
2) We iterate through positions of the input string y from left right, and for each I we consider all possible segments ending at position i.
3) For each segment ending at the position I, we calculate the total “quality” of that segment, for the given segment plus the maximum total quality of the previous segment that ends before the start of the current segment. For instance dp[i] = max(dp[i],dp[j] +quality(y[j+1,i+1]). Repeat for all string positions for y
4) The value in dp[n] for n = len(array) would represent the max total quality of the segmentation for the entire string.
5) To backtrack, we create an additional array segment_end where segment_end[i] stores the end position of the last word in the optimal segmentation that ends at position i. we can backtrack through this additional array to reconstruct the segmentation of the entire string

- Time complexity : This algorithm's time complexity is O(n^2), where n is the string length. It checks every position in the string once, and the backtracking step also takes O(n) time. Overall, it's efficient for finding the best way to segment a string based on quality.

Proof of correctness:
By induction:

The base case is established with dp[0] initialized to 0. This correctly represents the maximum total quality of an empty string (or before any characters are processed).

Inductive Step:Assume that dp[j] for all j < i is the maximum total quality for the string up to position j. When processing position i, the algorithm considers all segments that end at i and adds the quality of these segments to the maximum total quality found for the string just before the segment starts (dp[j] for all j < i). Thus, dp[i] represents the maximum total quality for the string till position i.
By induction, we can conclude that if dp[j] is correct for all j < i, then dp[i] is also correct. Since dp[n] (where n = len(string)) is built upon the correctness of all previous dp values, it must represent the maximum total quality for the entire string.

Problem 4:

We have two supercomputer that we need to run jobs on: A and B

a) Scenario where this doesn't work:
Consider there is a scenario where there are n= 3 minutes.
Minute 1: A = 10 , B = 0 A will be chosen because a1 >b1
Minute 2: A = 0 , B = 50 (Algo would stay on a as b3 unknown, but b2 < a2+a3)
Minute 3: A = 0 , B = 0 (It will not move because both options give 0 steps)
Minute 4: A = 0 , B = 100 (Algo would stay on A because it only compares one step ahead and b4 not greater than a4 +a5.
In this scenario, the total number of steps would be 10 +0 +0 +0 = 10
However, the optimal solution would capitalize on the 100 steps on B, and would give 150 steps.

The correct efficient Algo:

If n is 0, we don't need to perform any iterations as there are no minutes. In this case, the maximum number of steps is 0.If n is 1, we handle this by setting dp[1][0] and dp[1][1] considering the first possible states.

1. Start by creating a dynamic programming table dp[i][j], where i represents the minute and j represents the machine. This will hold the maximum number of steps that can be achieved up to minute i if we are on machine j at minute i. This table should have i+1 rows (to include the starting state) and 2 columns.
2. Initialize dp[0][0] and dp[0][1] to be zero, where 0 represents machine A, and 1 represents machine B.
3. For each minute from 1 to n, we fill in the dp table. For Machine A at minute i (i.e., dp[i][0]), we have two choices: Stay on machine A: In this case, we add the steps at minute i for machine A to the total steps until minute i-1 on machine A, which is dp[i-1][0] + a[i]. The other option is to Switch to machine B. In this case, we take the steps until minute i-1 on machine B, which is dp[i-1][1], and add the steps gained from switching to machine A which is b[i]. We update dp[i][0] as the maximum of these two choices.

4. We do the same for machine B as well, except we switch up the machines and scenarios given above. We update dp[i][1] as the maximum of these two choices.
5. Finally, after filling the entire dp table, the maximum number of steps at minute n will be max(dp[n][0], dp[n][1]).

The time complexity for dynamic programming is O(n), as that is the time complexity for dynamic programming approaches. We have a loop from 1 to n, and at each iteration we perform a constant amount of work to update the DP for both A and B.

Proof of correctness: Through induction
Base case: n = 0, thss dp[0][0] and dp[0][1] are initialized to zero.
Inductive step: for n > 0 dp[i][X]: where i is greater than zero and X is the machine, we assume that they are all computed correctly for all i for i < n. We show that dp[n][0] and dp[n][1] is also correct.
Considering a machine A at minute n, we compare staying on machine A vs switching to machine B: dp[n][0] = dp[n-1][0] + a[n] for A and for switching to B: dp[n][0] = dp[n-1][1] + b[n].
By choosing the max value, we ensire that dp[n][0] holds maximum number of steps on machine A till time n. The same logic applies to machine B at minute n. We compare staying on machine B (dp[n][1] = dp[n-1][1] + b[n]) and switching to machine A (dp[n][1] = dp[n-1][0] + a[n]). By choosing the maximum of these two options, we ensure that dp[n][1] holds the maximum number of steps up to minute n on machine B.

Above we showed that the base case is correct, and that the algorithm correctly computes dp[n][0] and dp[n][1] based on the values of dp[n-1][0] and dp[n-1][1], the algorithm is correct for all values of n.

Problem 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

This problem is similar to Knapsack and we should use recursion to solve this problem.
The algorithm for this:

- We first create two arrays, one which represents the prices of the different pieces and the other which represents the length of the different pieces.
- We then create a memorization array, which is an n by n array that stores the results of the previous subproblems. This stores the maximum price that can be obtained for a rod of length k with pieces from the first i. This reuses previously computed values.
- We then use a variable to store the maximum value of the length which is desired. The algorithm returns a zero if the maximum length is 0.

- We go over the length array and if we find that if the current state is memoized, we return the value of the memoized array. If the length of the element in the length array at index i is less than or equal to max length, we take the maximum result of including the current length in the rod vs. excluding it. This decision is based on the prices and lengths.

- If the length of the element in the length array at index i is greater than max_length, it means that the current piece cannot be included in max_length, so it excludes it and repeats the process for all elements in the length array.
- Return the maximum value obtained.

Proof of correctness:

By induction: Base Case: If the rod length is 0, then max_len is 0, and our algorithm correctly returns 0.

- Inductive Step: Let's assume that the algorithm correctly returns the maximum value for all subproblems computed up to a length of k. We want to show that it holds for the next subproblem with a length of k+1.
    - The algorithm considers all possible ways of cutting the rod and returns the most optimal choice at every step. This is done by comparing the results of including the current piece in the rod vs. excluding it. The final value is stored in the memoization array (mem) and is returned without any additional computations.
    - Therefore, by induction, we can conclude that the algorithm correctly computes the maximum obtainable value for any rod length.

Time Complexity:

- The algorithm uses a 2D memoization array with dimensions n x n. As the algorithm traverses through each element of this 2D array only once, the worst-case time complexity is O(n^2), where n is the length of length array.

Problem 6:

To solve this, we can implement the following algorithm:

1. Create an n x n 2D array: This helps to prevent any redundancy in the problem. We then fill the diagonals with zero
2. Iterate through all the elements of the array, and using this we initialize three variables:
    - If [i+2] <= [j], a = array[i+2][j], else this value a is updated to zero
    - If [i+1] <= [j-1], b = array[i+1][j-1], else this value b is updated to zero
    - If [i] <= [j-2], c = array[i][j-2], else this value c is updated to zero

3. The value at the array[i][j] would the the maximum of the sum of the coins array at [i] and min(a,b), and the sum of the coins array at [j] and min(b,c). We would then return the value at array[0][n-1]

Proof of correctness:

For a single coin, our algorithm assigns its value to a cell in the 2D array. This is because the diagonal of the array conta ins 0's, and our algorithm simply selects the coin's value for that cell.

Inductive Step:

Assume that our algorithm correctly computes the maximum possible value for n coins. We want to show that it also works for n+1 coins.

Our algorithm iterates through each cell [i][j] of the array and calculates the maximum possible value for that cell.
Proof via induction”
For n+1 coins at positions i and j:

- Value a considers excluding the first coin and finding the optimal solution for the remaining n coins between i+2 and j.
- Value b considers excluding the last coin and finding the optimal solution for the remaining n coins between i+1 and j-1.
- Value c considers excluding both the first and last coins and finding the optimal solution for the remaining n coins between i and j-2. The algorithm selects the maximum value between:
- Choosing the first coin at coins[i] and finding the optimal solution for the remaining n coins between i+1 and j (b).
- Choosing the last coin at coins[j] and finding the optimal solution for the remaining n coins between i and j-1 (a). The maximum of these two options represents the optimal solution for n+1 coins between positions i and j.

By induction, we have shown that if our algorithm correctly computes the maximum value for n coins, it also correctly computes the maximum value for n+1 coins. Therefore, through the principle of mathematical induction, this algorithm is correct for any number of coins. The final result is the maximum possible value that can be obtained for the entire set of n coins, which is stored in array[0][n-1].

Time Complexity:
The time it takes for our algorithm is O(n^2). We get this because for every possible pair of indices [i] and [j], we do n computations for [i] and n computations for [j], adding up to n^2 in total.