Preyasi Gaur Disc 1A

Time: 8:00AM - 9:50 AM

TA: Vincent Li

Computer Science 180

Homework 5

**Question 1**

Algorithm:

* Begin by initializing two pointers:
  + left: at the start of the array (0)
  + right: at the end of the array (len(array) - 1)
* Make a while loop with the condition that left < right:
  + Calculate the current sum of the elements at the left and right pointers.
    - If the current sum is equal to k:
      * Output the pair (array[left], array[right]).
      * Increment the left pointer to left + 1.
      * Decrement the right pointer to right - 1.
    - If the current sum is less than k:
      * Increment the left pointer to left + 1 to find a larger sum.
    - If the current sum is greater than k
      * decrement the `right` pointer to right - 1 to find a smaller sum.
* End the loop when left is not less than right.
* The pairs that add up to k have been found and output during the loop.

Proof:

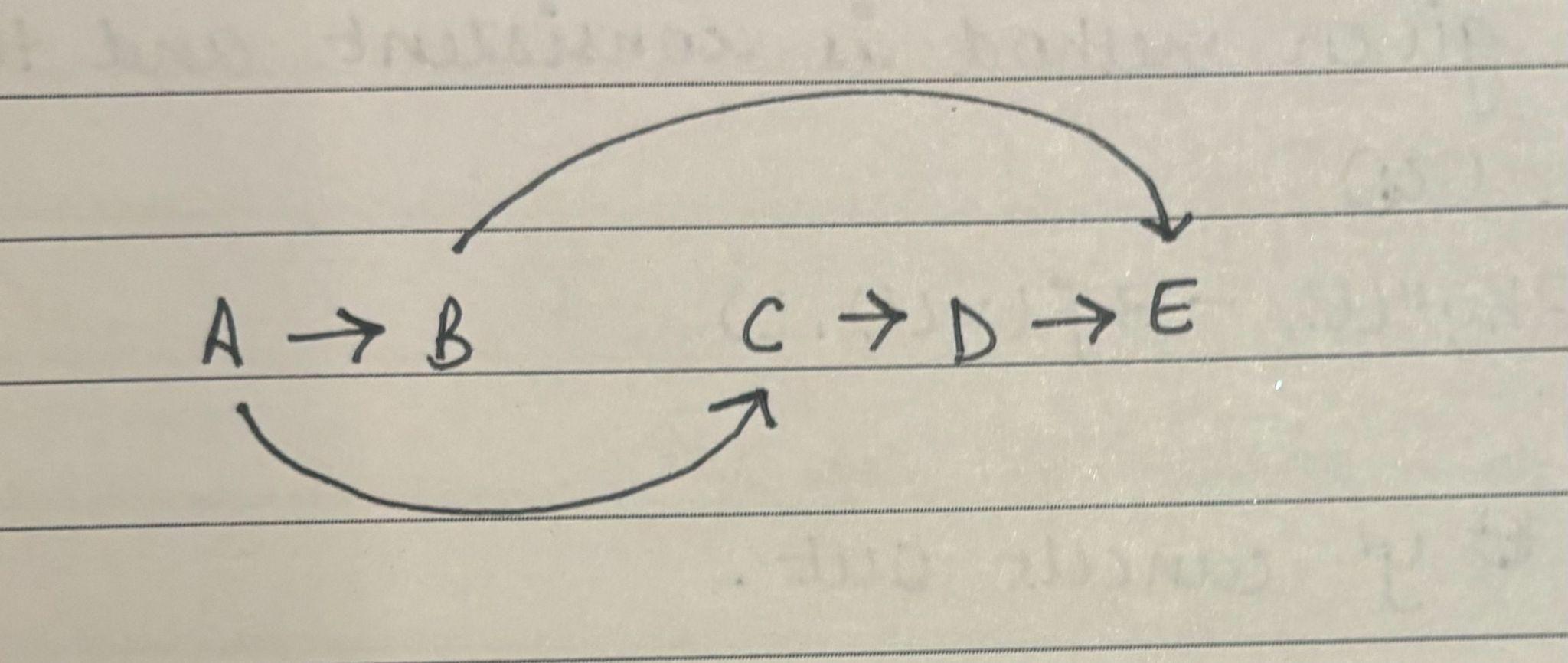
* Assume the algorithm is incorrect, meaning there is at least one pair (a, b) in the array such that a + b = k, which is not identified by the algorithm.
* The algorithm starts with two indices, left at the beginning (0) and right at the end of the array (length - 1).
* If the sum of the elements at left and right is less than k, the algorithm moves the left index to the right (increments left), since the array is sorted and this is the only way to increase the sum.
* If the sum is greater than k, the algorithm moves the right index to the left (decrements right), as this is the only way to decrease the sum in a sorted array.
* If the sum equals k, the algorithm records the pair.
* Now, consider the pair (a, b) such that a + b = k. There must be a moment during the execution of the algorithm when left is at the index of a and right is at the index of b.
* If the algorithm doesn't record (a, b), it must be because it either moved left past a or right past b without recording the pair.
* If left moved past a, it implies that the sum at left and right was less than k at that point, which is impossible because b is at most right and a + b = k.
* Similarly, if right moved past b, it implies that the sum was greater than k, which is also impossible because a is at least left and a + b = k.
* Since neither left nor right can move past a or b without recording the pair if a + b = k, we have a contradiction with our initial assumption that there exists a pair that the algorithm fails to find.
* Therefore, by contradiction, the algorithm must correctly identify all pairs (a, b) in the array such that a + b = k.

Time Complexity:

* The two pointers traverse the array at most once each, making the complexity linear with respect to the array size: O(n)
* Total runtime: O(n)

**Question 2**

**Part A**

Suppose we have the linked list:  


* In this case, the greedy solution will start (A, B) as it will pick the first edge, and as B < C.
* Now, from node B, the only outgoing edge is (B, E), which leads to the incorrect longest path of 2.
* In this case, the correct longest path is (A, C), (C, D), (D, E) which has a length of 3

**Part B**

Algorithm:

* Define a variable path\_len(i) to be the longest path length that ends at some arbitrary node i. Initialize path\_len(i) to be 0 for all nodes.
* For all i from 2 to n:
  + For each incoming edge pointing to i coming from node j, update path\_len with max(path\_len(i), and path\_len(j) + 1)
* Return path\_len(n)

Proof:

* We will prove this by induction.
* Base Step: For the base case, the longest path from a node to itself is 0.
* Inductive step: Assuming that the algorithm has correctly calculated the solution for all nodes before the ith node:
  + The max path length ending at node i must come from one of the nodes that we have previously covered.
  + Our algorithm at every step compares all the edges and only stores that max.
  + Thus, once the algorithm reaches path\_len(n), it will be the optimal solution.

Time complexity:

* Creating the array to store the value of path\_len for all the nodes: O(n)
* Our algorithm goes through every node once, and uses contant time to lookup and add each edge: O(1)
* Thus, total runtime: O(V+E)

**Question 3**

Algorithm:

* Let's begin by defining OPT(x) as the optimal segment spanning the first x letters.
* Start by utilizing the black box to determine the quality provided by the initial letter in the string, denoted as quality().
* Set OPT(1) to be equal to this value. For the subsequent string lengths i ranging from 2 to the total length of the string:
* Compute OPT(i) by selecting the maximum among the following options:
  + OPT(i-1) + quality()
  + OPT(i-2) + quality()
  + OPT(i-3) + quality()
  + ...
  + OPT(1) + quality()
  + quality()
* Additionally, while updating the maximum quality of OPT(i), maintain a record of which segmentation among the aforementioned options led to OPT(i). Finally, return the segmentation indicated by OPT(the entire string).

Proof:

* We will prove its correctness by using mathematical induction:
* Base Case: it correctly finds the OPT(1) trivially.
* Inductive Case: Assume that the algorithm correctly calculated the optimal solution for the given substring upto i-1 characters. Then,
  + There are only i number of different ways to append the new character to the end of the string.
  + These represent considering adding the i-th character to a word that consists of just itself, just the last two characters, just the last three characters, ..., and a word that consists of all the first i characters.
* The algorithm exhaustively takes the max of all those possibilities, and thus OPT(i) is correctly calculated. Thus by the end of the string, the algorithm will correctly find the correct segmentation.

Time complexity:

* Given an input string of len n, OPT is determined for every substring starting at each index and all the way to n: O(n)
* Every iteration does a constant time lookup: O(n)
* Total runtime:

**Question 4**

**Part A**

| Computer | Minute 1 | Minute 2 | Minute 3 |
| --- | --- | --- | --- |
| A | 20 | 10 | 10 |
| B | 15 | 15 | 25 |

* The given algorithm does not work as it is greedy.
* It would start by picking computer A for Minute 1
* Then switch for Minute 2
* Finally it will pick computer B for Minute 3 as
* Thus, it yields an answer of 45, when in this case the correct answer would be 55 (by picking Computer B throughout).

**Part B**

Algorithm:

* Let OPT(i, n) represent the maximum number of cycles attainable with a schedule concluding at minute i and termination at computer n. VAL(i, n) is defined as the quantity of cycles available on computer n at minute i.
* Determine OPT(1, a) and OPT(1, b) by simply evaluating the cycle count available at minute 1 on computers a and b, respectively.
* Determine OPT(2, a) and OPT(2, b) by simply evaluating the cumulative cycle count available over minutes 1 and 2 on computers a and b, respectively.
* For all subsequent minutes i from 3 through n:
  + Compute OPT(i, a) by selecting the maximum from the following:
    - OPT(i-1, a) + VAL(i, a)
    - OPT(i-2, b) + VAL(i, a)
  + Compute OPT(i, b) by selecting the maximum from the following:
    - OPT(i-1, b) + VAL(i, b)
    - OPT(i-2, a) + VAL(i, b)
* Return the maximum of OPT(n, a) and OPT(n, b).

Proof:

* We will prove this by mathematical induction.
* Base Case: OPT(1) and OPT(2) are found trivially.
* Inductive Case: Assume that it has correctly calculated the solution for substring up to character i-1 correctly:
  + At each subsequent time step, our options
    - Continue executing on the same computer
    - Switch computers
  + We consider both of these cases we take the maximum of.

Time Complexity:

* Assume that we have n timesteps.
* Then we iterate through every time step and do constant time operations: O(n)
* Total runtime: O(n)

**Question 5**

Algorithm:

* Initialize an array max\_prices of size n+1 to store the maximum price that can be obtained for each length up to n. Set max\_prices[0] = 0 because a rod of length 0 has no value.
* For each length i from 1 to n, determine the maximum price max\_prices[i] that can be obtained by cutting the rod and selling the pieces.
* This is done by iterating through all possible cuts j from 0 to i-1 and using the formula: max\_prices[i] = max(max\_prices[i], prices[j] + max\_prices[i - (j + 1)]) where prices[j] is the price of a piece of length j + 1, and max\_prices[i - (j + 1)] is the maximum price of the remaining length after the cut.
* The final answer is max\_prices[n], which represents the maximum price obtainable for a rod of length n.

Proof:

* For the rod cutting problem, this principle can be applied as follows:
* To get the maximum price for a rod of length i, you need to make the best cut at some point j that maximizes the sum of the price of a piece of length j and the maximum price of the remaining rod of length i - j. This means that if we know the optimal solutions (maximum prices) for rods of lengths 1 through i-1, we can use these solutions to compute the optimal solution for a rod of length i.
* The algorithm makes use of previously computed solutions for smaller lengths, ensuring that the maximum price for each smaller length is already optimal. Therefore, by using these optimal solutions, the final solution for the rod of length n is also optimal.

Time Complexity:

* The outer loop runs n times, for each length from 1 to n.
* The inner loop runs up to i times for each iteration of the outer loop.
* Thus, in the worst case, the inner loop runs (1 + 2 + ... + n) times, which sums up to n(n+1)/2, giving a time complexity of O().

**Question 6**

Algorithm:

* Create a 2D array dp where dp[i][j] represents the maximum value the player can collect from coins in the range i to j.
* Fill this 2D array as follows:
  + If i == j, dp[i][j] is simply the value of the single coin available.
  + If the number of coins is odd ((j - i) % 2 == 1), the player will pick the coin that leaves the opponent with a less valuable choice. Hence, dp[i][j] = max(coins[i] + min(dp[i+2][j], dp[i+1][j-1]), coins[j] + min(dp[i+1][j-1], dp[i][j-2])).
  + If the number of coins is even, it is the opponent's turn, and they will leave the less valuable choice for the player. Thus, dp[i][j] = min(dp[i+1][j], dp[i][j-1]).
* The final answer is dp[0][n-1], which gives the maximum value the player can collect from all the coins.

Proof:

* The algorithm is correct because it uses a bottom-up approach to solve smaller subproblems first, which are then used to solve larger subproblems. Each dp[i][j] value is calculated using the optimal solutions of its subproblems, following the principle of optimality.
* This proof can be formalized by induction:
* Base case: When i == j, the solution is trivially correct.
* Inductive step: Assume that for all k < (j - i), dp[i'][j'] gives the optimal solution for the subproblem from i' to j'. When solving for dp[i][j], the algorithm correctly considers the optimal solutions for the subproblems dp[i+1][j], dp[i][j-1], dp[i+2][j], and dp[i][j-2] to find the optimal solution for dp[i][j].
* Because we're assuming that both players are playing optimally, we have to consider that each player will make the move that maximizes their gain while minimizing the opponent's gain.

Time complexity:

* Filling a 2D array of size n\*n and each cell calculation takes constant time, assuming that the number of coins n is even as given in the problem.
* Total runtime: O()