Homework #5 Solutions

(Due 11:59 PM Nov. 29)

For many of the DP problems, there exists a recursive solution that uses memoization that has the same time complexity as the solutions listed here. Those solutions are acceptable for full credit with accompanying proof and time complexity analysis.

1. **Exercise 3, Page 107 (20pts)**

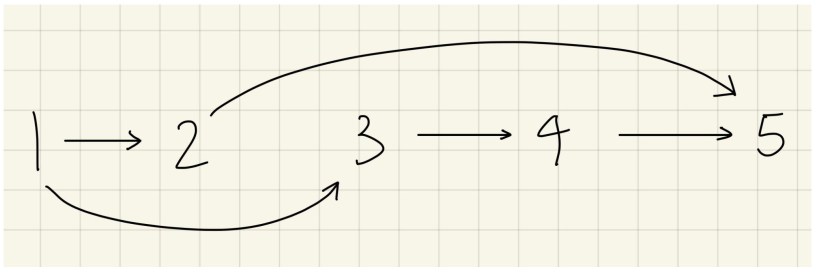
The size of the input/output does not count against the memory complexity of the algorithm, since all algorithms would have to somehow receive the input and return the output anyways.

- Algorithm (12pts):
    - Sort the array.
    - For each element n:
        - Use binary search in the section of the sorted array greater than n to look for if the element n+k exists in the array:
            - If the difference between n and the middle element is less than k, disregard the left half of the array and use binary search on the right half, and vice versa if the difference between n and the middle element is more than k.
        - Store such pairs if they exist.
    - Return all pairs found.
- Proof of correctness (4pts):
    - Assume that the algorithm missed a pair of numbers whose difference is k.
    - Since the algorithm performs binary search on each element, binary search must have failed to find an element's complement.
    - However, the first step of the algorithm sorts the array, thus binary search will not fail, a contradiction.
- Time complexity (4pts):
    - Sorting costs O(nlogn) time.
    - For each element, we perform binary search. Since binary search allows us to get rid of half the search space per iteration, each element incurs logn runtime
    - The whole algorithm thus has O(nlogn) runtime.

Note: There exists a two pointer approach for finding the complementary number that has a difference of k from itself that is acceptable for full credit. That algorithm's runtime is the same since you still need to pay the cost of sorting.

2. **Exercise 3, Page 314 (20pts)**

Part A (4pts):



- The greedy solution would start with the edge (1,2) instead of (1,3) because 2 < 3. From node 2, the only outgoing edge is (2, 5), which leads to the incorrect longest path of 2.
- The correct longest path is (1, 3), (3, 4), (4, 5) which has a length of 3.

Note: Many correct counter examples for this part exist.

Part B:

- Algorithm (12pts):
    - Define OPT(i) to be the longest path length that ends at node i. Initialize OPT(i) to be 0 for all nodes.
    - For all i from 2 through n:
        - For each incoming edge pointing to i coming from node j, update OPT(i) with the maximum of the following:
            - the original value store in OPT(i)
            - OPT(j) + 1
    - Return OPT(n)
- Proof of correctness (2pts):
    - For the base case of this problem, the longest path from node 1 to node 1 is trivially 0.
    - Assuming that the algorithm in its current state has correctly calculated the optimal solution for all nodes before node i:
        - The max path length ending at node i must come from one one of the precursor nodes. The algorithm exhaustively compares all of those edges, and the max of those elements must be OPT(i).
    - Therefore once the algorithm gets to OPT(n), OPT(n) will be optimal.
- Time Complexity (2pts):
    - Initializing an array to store the value of OPT for every node takes O(n) time.
    - The loop processes every node at most once, and performs constant time lookups and additions for each edge, which themselves are processed at most once.
    - Thus this algorithm has runtime O(V+E)

3. **Exercise 5, Page 316 (20pts)**

- Algorithm (12pts)
    - Define OPT(x) to be the optimal segmentation of the first x letters.
    - Start by using the black box to find the quality provided by the first letter in the string, quality(y_1). Set OPT(1) equal to that result.
    - For the remaining string lengths i from 2 to the length of the string:
        - Calculate OPT(i) by taking the maximum of the following:
            - OPT(i-1) + quality(y_i)
            - OPT(i-2) + quality($y_{i-1}y_i$)
            - OPT(i-3) + quality($y_{i-2}y_{i-1}y_i$)
            - …
            - OPT(1) + quality($y_2y_3\dots y_{i-2}y_{i-1}y_i$)
            - quality($y_0y_1\dots y_i$)
        - In addition to updating the max quality of OPT(i), keep track of which of the above segmentations caused OPT(i).
    - Return the segmentation represented by OPT(the complete string).
- Proof of correctness (4pts):
    - The algorithm correctly finds the OPT(1) trivially.
    - Given that the algorithm has correctly calculated the optimal solution for the substring up to character number i-1 correctly:
        - There are only i 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 of highest quality.
- Time complexity(4pts):
    - For an input string of length n, OPT is calculated for each substring starting at index 0, starting at length 1 all the way to length n.
        - Each iteration x does a constant number of lookups and iterations for all x from 1, 2, …, x. => O(n) operations are needed to calculate each OPT(i)
    - Thus the algorithm has a total time complexity of $O(n^2)$.

4. **Exercise 10, Page 321 (20pts)**

Part A (4pts):

Computer	Minute 1	Minute 2	Minute 3
A	4	2	2
B	3	3	5

- The proposed solution does not work because it would start by choosing computer A for minute 1, then switch during minute 2, and use computer B for minute 3 since $b_3 > a_2+a_3$. This would yield a total of 9 cycles.
- The true optimal solution would be to just use computer B for all 3 minutes, to yield a total of 11 cycles.

Note: Many correct counter examples for this part exist.

- Algorithm (12pts):
    - Define OPT(i, n) as the maximum number of cycles achievable with a schedule ending at minute i that is forced to end at computer n. Define VAL(i, n) as the number of cycles available on computer n at minute i.
    - Compute OPT(1, a) and OPT(1, b) as simply the number of cycles available at minute 1 on computers a and b respectively.
    - Compute OPT(2, a) and OPT(2, b) as simply the number of cycles available over minute 1 and 2 on computer a and b respectively.
    - For all remaining minutes i from 3 through n:
        - Calculate OPT(i, a) by taking the maximum of the following:
            - OPT(i-1, a) + VAL(i, a)
            - OPT(i-2, b) + VAL(i, a)
        - Calculate OPT(i, b) by taking the maximum of the following:
            - OPT(i-1, b) + VAL(i, b)
            - OPT(i-2, a) + VAL(i, b)
    - Return the max of OPT(n, a) and OPT(n, b).
- Proof of correctness (2pts):
    - The base cases of OPT(1) and OPT(2) are found trivially.
    - Given that the algorithm has correctly calculated the optimal solution for the substring up to character number i-1 correctly:
        - At each time step, the only options we have are to continue execution on the same computer, or to switch computers, both of which we consider and take the maximum of.
- Time complexity (2pts):

- Assume there are n timesteps: at each timestep we compute a constant number of max, addition, and array lookup operations. Thus the algorithm is O(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. (20pts)**

- Algorithm (12pts):
    - Construct a 2 dimensional array, where the length of the first dimension is the size of the stick, and the size of the second dimension are all the valid lengths we can break the stick up into.
    - Define OPT(i, j) as the optimal solution of breaking a stick of length i into pieces that are at most j inches long. Define VAL(j) as the value of a stick that is j inches long, as determined by the input table.
    - For all i from 1 to n, setOPT(i, 0) as 0. For all j from 0 to n set OPT(0, j) as 0.
    - For all remaining broken up stick lengths i from 2 to n:
        - For all stick lengths j from 1 to n:
            - Compute OPT(i, j) as the maximum of the following, considering each only if i-j is positive:
                - OPT(i-j, j) + VAL(j)
                - OPT(i, j-1)
    - Return the value of OPT(n, n).
- Proof of correctness (4pts):
    - The base cases where we calculate the max value when the stick has length 0 or where we can break up the stick only into pieces of length 0 is trivially 0.
    - Given that the algorithm has correctly calculated the optimal solution for all subproblems smaller than OPT(i, j):
        - The optimal solution of OPT(i, j) can either include another stick of length j, or it may not. The algorithm considers both cases and returns the max to accurately find OPT(i, j)
- Time complexity (4pts):
    - The 2D array for OPT in the worst case has n^2 cells.
    - To compute the OPT of each cell, we do a constant number of array lookups and arithmetic operations, thus the algorithm has runtime complexity $O(n^2)$.

Note: There exists a one dimensional dp solution to this question that is acceptable for full credit. That solution essentially runs for every length i from 1 to n, taking the max of adding a new rod of length 1 all the way through length n. The time complexity of the one dimensional dp solution should still be $O(n^2)$.

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. (20pts)**

- Algorithm (12pts):
    - Create a 2 dimensional array, where we have one dimension denote the start of a contiguous sequence of coins and the other dimension represents the end of a contiguous sequence of coins.
    - Define OPT(i, j) as the best reward the player can get from the contiguous sequence of coins including the i-th coin and ending just before the j-th coin. Define VAL(x) as the value of the x-th coin in the sequence.
    - For all contiguous subsequences that consist of just one coin, set OPT(i, i+1) to be the value of the only coin in the sequence.
    - For all contiguous subsequences that consist of just two coins, set OPT(i, i+2) to be the value of the largest coin in the subsequence.
    - For all lengths of contiguous subsequence j from 3 to n:
        - For all valid starting positions i of that subsequence from 1 to n-j:
            - Calculate OPT(i, i+j) as the maximum of the following:
                - VAL(i) + min(OPT(i+1, i+j-1), OPT(i+2, i+j))
                - VAL(i+j-1) + min(OPT(i, i+j-2), OPT(i+1, i+j-1))
    - Return OPT(1, n+1)
- Proof of correctness (4pts):
    - The base cases are trivially correct.
    - Given that the algorithm has correctly calculated the optimal solution for all smaller subsequences than x:
        - At this step we may only choose the coin on the left, or the coin on the right.
            - If we chose the coin on the left, since the opponent is acting optimally, they will use the same optimal strategy to get as many coins as possible for themselves, which is equivalent to minimizing our own reward, so we consider the minimum of the two options the opponent has at their turn. Thus after the opponent acts, we can determine the next subproblem to solve by taking the minimum OPT of the situation where the opponent took the left coin or the right coin.
            - The same logic applies if we chose the coin on the right.
- Time complexity (4pts):

- There are $O(n^2)$ contiguous subsequences.
- To calculate the OPT of each subsequence, we perform a constant number of operations, thus the runtime of the algorithm is $O(n^2)$.