CSC 505, Homework 3 Solution
Due date: Wednesday, October 24, 9:00 PM

Problem 1d [Radi] *of homework 2)* (6 points) Given a set of arbitrary denominations C =(c_1, ..., c_d), describe an algorithm that uses **dynamic programming** to compute the minimum number of coins required for making the change. You may assume that C contains 1 cent, that all denominations are different, and that the denominations occur in an increasing order.

Ans:

Pseudo code:

```
Input: V, {C[1], C[2], …, C[d]}
Output: minCoin     // minimum number of coins to make change

Table[0] = 0        // initialization
for(i = 1; i <= V; i++){
     Table[i] = infinity      // initialization
     for(j = 1; j <= d; j++){
          if(C[j] > i) break
          if(Table[i - C[j]] < infinity and  // recurrence
               1 + Table[i - C[j]] < Table[i])
                    Table[i] = 1 + Table[i - C[j]]
     }
     minCoin = Table[V]  // minCoin(V, C)
     print (minCoin)
}
```

Textual Description and Argument for Correctness:

The above algorithm results from the following recurrence relation:

minCoin(V, C) = 0, when V = 0
minCoin(V, C) = min(1 + minCoin(V - Ci, C)), when V > 0 and Ci <= V over 1 <= i <= d

The base case is trivial as a value of 0 needs 0 coins to make the change.
The recurrence relation follows from the fact that after using a coin Ci as part of the change, we still need to make change for the value of (V-Ci) using the same set of denominations. We have to make change for (V - Ci) using the minimum number of coins. So, in total, we need that number + 1 coin, which is Ci. Now, we can use any coin Ci from the set of coins C to make the first change so long as Ci <= V. Thus, overall Ci, which are <= V, we will use one for which 1 + minCoin(V - Ci, C) is the minimum to ensure that minCoin(V, C) is assigned the smallest possible value. Since 1 is part of the coin set, there will always be at least one Ci <= V available.

The problem has the optimal substructure property since we have constructed the solution for V using the optimal solution for smaller values such as V - Ci. We have initialized a DP table, Table, with the value 0 at the 0th index, and an infinity at the indices from 1 to V. We have computed the table bottom up starting at index 1, i.e., V=1, and ending at index V. For each value, we have iterated over the set of coins to find out which one, being used as the first change, gives the optimal answer. Since the optimal solution for the values smaller than the current value is already computed, we have simply looked up the table.

Example:

V = 13, C = {1, 6, 10}
Table = {0, 1, 2, 3, 4, 5, 1, 2, 3, 4, 1, 2, 2, 3}
minCoin = Table[13] = 3.

Time and space complexity:

The initialization of the table takes θ (V) time. Then, for each value, we have to iterate from C1 until Ci > V. So, in the worst case, for a value we need O(d) time. Hence, for V values we need O(Vd) time. Therefore the time complexity of the algorithm is O(Vd). The space complexity is θ (V+d) for the table with V items and the list of coins with d items.

Problem 1e [Radi] *of homework 2)* (5 points) Implement the algorithm described in d). We will use the **AutoGradr site (https://app.autogradr.com/)** for grading your programs. **Instructions for signing up at the site and submitting the programming task are given at the bottom of this assignment.**

Problem 2. [Radi] *Purpose: reinforce your understanding of dynamic programming and the matrix-chain multiplication problem.* (10 points) Please follow the description in our textbook (Chapter 15) and implement a dynamic programming version of the algorithm for solving the matrix-chain multiplication problem. Your program should take an array of integers representing the dimensions of the matrices in the matrix-chain as input, and it should produce the optimal number of scalar multiplications needed to compute the matrix-chain product as output. In addition, your algorithm should also output an optimal parenthesization of the matrix chain.

Again, like in problem 1e of homework 2, We will use the **AutoGradr site (https://app.autogradr.com/)** for grading your programs. Please submit your implementation of problem 2 in the **"Homework 3: MCM"** project, which will be posted in **AutoGradr**. You will find the detailed description of the problem in the site, including input and output specifications, limits, and formatting. Some test cases will be made visible to you as samples. The other test cases will be hidden. Your program will run against each test case automatically, and you will receive an instant feedback on how many test cases you have passed. Usually, the percentage of passed test cases will be proportional to your grade

in this question. We may cut off some marks for poorly written codes. The time limit will be set sufficiently high, so you do not need to worry so long as you have correctly implemented a dynamic-programming-based solution with the expected time complexity. Our test cases will be designed in a way that the optimal cost, as well as, all intermediate values should fit in 32-bit signed integers. Also, none of our test cases will have more than one optimal parenthesization order.

Problem 3.[Neeraj&Effat] *Purpose: practice algorithm design using dynamic programming.* A subsequence is palindromic if it is the same whether read left to right or right to left. For instance, the sequence A,C,G,T,G,T,C,A,A,A,A,T,C,G has many palindromic subsequences, including A,C,G,C,A and A,A,A,A (on the other hand, the subsequence A,C,T is not palindromic). Assume you are given a sequence x[1...n] of characters. Denote L(i,j) the length of the longest palindrome in the substring x[i,...,j]. The goal of the Maximum Palindromic Subsequence Problem (MPSP) is to take a sequence x[1,...,n] and return the length of the longest palindromic subsequence L(1,n).

a) (4 points) Describe the optimal substructure of the MPSP and give a recurrence equation for L(i,j).

A problem is said to have optimal substructure if an optimal solution can be constructed from optimal solutions of its subproblems.

Optimal substructure of MPSP can be viewed as:

Let L(1,k) be length of longest palindromic subsequence. Let S(1,k) be the string under consideration of length k. If the 1st character and kth character of the string S(1,k) is the same, then L(1,k) will be 2 + L(2,k-1) else if there are not the same, then L(1,k) will be maximum of L(1,k-1) and L(2,k).

Recurrence equation:

L(i,j)	= 1	if i == j
	= 2	if i+1 == j and S[i] == S[j]
	= L(i+1,j-1) + 2	if S[i] == S[j]
	= max(L(i,j-1),L(i+1,j))	otherwise

If i=j the substring is just one character and the length of the longest palindromic subsequence is one. For i+1=j, if S[i] == S[j], the palindromic subsequence consists of the two character, ergo L(i,j)=2.

For the remaining cases, the recurrence holds because if the first and the last character are the same, we can conclude 2 + L(i+1,j-1)>= max(L(i,j-1),L(i+1,j)). Ergo, these character can be included in the optimal solution and we get L(i,j)=2 + L(i+1,j-1). If the characters are

unequal at most one of them will be part of the optimal solution and we obtain the length of the optimal solution via max(L(i,j-1),L(i+1,j)).

b) (6 points) Describe an algorithm that uses dynamic programming to solve the MPSP. The running time of your algorithm should be $O(n^2)$.

Pseudocode:

```
//Indexing starts from 1 till n

MPSP(s){
        // s is the string

        n = len(s)
        if n == 0:
                return 0

        L = int[n][n]          // L is n*n 2d matrix

        for i <- 1 to n:
                L[i][i] = 1

        for i <- n-1 to 1:
                for j <- i+1 to n:
                        if  i - j == 1 and s[i] == s[j]:
                                L[i][j]  =  2
                        else:
                                if s[i] == s[j]:
                                        L[i][j] = 2 + L[i+1][j-1]
                                else:
                                        L[i][j] = max(L[i+1][j], L[i][j-1])


        return L[1][n]
}
```

Textual Description: The algorithm tries to find the longest palindromic subsequence in the given string s. It calculates the length of the longest palindrome for every substring of s. However, it calculates the solution for each substring only once. It remembers the result by storing it in a n*n 2d matrix. Every cell[i,j] in the matrix denotes the maximum palindromic length of substring of s which starts at ith character and ends at jth character. Hence, the given matrix becomes an upper diagonal matrix. The calculation of the given cell is performed using the optimal substructure as described above.

Example:

Let string be “acbdcb”

So, we will calculate the matrix L for every value of i,j
For i =j, L[i][j] = 1

For i = n-1 and j = n,
s[i] = ‘c’, s[j] = ‘d’
s[i] != s[j], Hence, L[n-1][n] = max(L[n][n],L[n-1][n-1]) = 1

We will calculate for each values and get a table like this:

i ↓ j→	a	c	b	d	c	b
a	1	1	1	1	3	3
c		1	1	1	3	3
b			1	1	1	3
d				1	1	1
c					1	1
b						1

Hence, the answer is 3.

Proof of correctness:

The base cases are strings of length 0, 1 and 2. If length is 0, algorithm returns 0. For length 1, it is 1. For length 2, if both the characters are same, the length is 2, else it will be max of (1,1) and (2,2) which both are of length 1, so it would be 1. Hence, it is correct.

The problem follows optimal substructure property. We calculated the answer of string of length n from the values of substrings which are of length n-1(unequal) or n-2(if the starting and ending characters of string of length n are equal). This is done by looking up the answer in the matrix L and getting the values of required substrings.

Time and space complexity:

Initially there is one for loop for initializing the diagonal elements to 1, which runs in O(n). Additionally, we can see in the algorithm, there are two nested for loops. These loops run in

the order of n each. Hence, runtime for these loops is O(n^2). Hence the **time complexity of the algorithm is O(n^2)**.

The algorithm requires additional space for matrix L which is n*n matrix. Hence, the **space complexity is O(n^2)**.
(Worth thinking: Use can optimize the space complexity and reduce it to O(n) space)

Problem 4. [Kornelia] *Purpose: practice designing greedy algorithms.* (10 points) Suppose you have a long straight country road with houses scattered at various points far away from each other. The residents all want cell phone service to reach their homes and we want to accomplish this by building as few cell phone towers as possible.
More formally, think of points x1, …, xn, representing the houses, on the real line, and let d be the maximum distance from a cell phone tower that will still allow reasonable reception. The goal is to find a minimum number of points y1,…,yk so that, for each i, there is at least one j with | yj - xi | ≤ d.
Describe a greedy algorithm for this problem. If the points are assumed to be sorted in increasing order your algorithm should run in time O(n). Be sure to describe the greedy choice and how it reduces your problem to a smaller instance. Prove that your algorithm is correct.

SOLUTION:

Assume that the points are sorted so that $x_1 < x_2 < \ldots < x_n$

Make the greedy choice: The choice to put a tower that serves x_1, i.e. a tower at a point y in the interval $[x_1 - d, x_1 + d]$ will reduce the problem to a subproblem that find towers to cover points $x_i, \ldots, x_n$, where $x_1, \ldots, x_{i-1} \in [y - d, y + d]$.

Algorithm and analysis: The greedy choice is to put the first tower at position $x_1 - d$. Suppose $x_1, \ldots, x_{i-1}$ are covered by the first tower, then finding towers to cover the remaining points $x_i, \ldots, x_n$ becomes a subproblem. Iterate the same procedure for the remaining points $x_i, \ldots, x_n$ until all houses are covered.

Example:
For example, there are 5 houses x_1, x_2, x_3, x_4, x_5, based on the algorithm, the first tower is $y_1 = x_1 + d$, and y_1 can cover all the houses in the range of $[x_1, x_1 + 2d]$. Let us assume that x_1, x_2, x_3 fall in the range. For the remaining houses, we pick the second tower $y_2 = x_4 + d$, and y_2 will cover the range, $[x_4, x_4 + 2d]$ let's say that $x_4 \le x_5 \le x_4 + 2d$, so x_4 and x_5 can be covered by the second tower. So y_1 and y_2 are our solution.

Proof the correctness of the algorithm:
The greedy choice will result in an optimal solution based on the following argument. Assume greedy does not always result in an optimal solution. Choose a counter example with minimum number of towns. Let y^* be the position of the first tower in an optimal

solution. If $y^* = x_1 + d$, we are done, since for the reduced subproblem, due to our assumption of a minimum counter example, greedy will result in an optimal solution; if not, it cannot be the case that $y^* > x_1 + d$, otherwise the house at x_1 would not be covered by any tower. So if $y^* < x_1 + d$, the reduced subproblem for the optimal choice would be one with houses at position $x_j, \ldots, x_n$, where $x_i < y^* + d$ for $1 \le i \le j - 1$. This subproblem has at least as many houses as the one resulting from the greedy choice and therefore requires at least many towers, again using the assumption that greedy will produce an optimal solution for the reduced subproblem due to our assumption that we start from a counter example of minimum size. So the optimum choice leads to a solution with at least as many towers as the greedy one.

The time complexity of the algorithm:
During the execution of the algorithm we scan the sorted list of houses from left to right to identify the first house x_j that is not covered by the tower that was selected last. In particular, we compare the position of the current house x_j with the position t_{last} of the tower that was selected last. If $x_j - t_{last} > d$ the house is the anchor point for the next tower at position $x_j + d$, and the scan proceeds until no further uncovered houses exist. Since there are n houses, and during the scan, each house is compared only once to a tower location, and might trigger at most one new tower, the algorithm runs in O(n) worst case time.

Problem 5. [Hyunwoo] *Purpose: reinforce your understanding of data structures for disjoint sets. For background on binomial trees and binomial heaps please read 19-2 on page 527.* (6 points) Please solve Problem 21.3-3 on page 572 of our textbook.

21.3-3 Give a sequence of m MAKE-SET, UNION, and FIND-SET operations, n of which are MAKE-SET operations, that takes $\Omega(mlgn)$ time when we use union by rank only.

Ans:

1. Perform ***n* MAKE-SET** operations. Then, we'll have n sets of $\{x_1\}, \{x_2\}, \ldots, \{x_n\}$.

2. Create a binomial tree using ***n-1* UNION** operations. First, make sets of size 2 and then increase the size continuously ($2^1 \to 2^2 \to 2^3 \to \ldots$). We know that a binomial tree of degree $k+1$ is formed by taking the UNION of any two binomial trees of degree k. Thus, we need $2^{\lfloor log_2 n \rfloor} - 1$ UNION operations to build the tree, and the resulting binomial tree has one node k at depth $\lfloor log_2 n \rfloor$.

3. We now perform ***m-n-(n-1)*** operations **FIND-SET(k)** operations. Since each FIND-SET(k) operation takes time proportional to its depth $\lfloor log_2 n \rfloor$, this takes $(m - 2n + 1) * \lfloor log_2 n \rfloor$.

➢ Since each MAKE-SET and UNION operation takes at least O(1), the total time to perform the entire sequence can be bounded from below by the time to perform all

the FIND-SET operations, resulting in $\Omega((m-2n)*log_2 n)$. If m is sufficiently large compared to $2n$, it follows $\Omega((m-2n)*log_2 n) = \Omega(mlog_2 n)$.