CS 141 Assignment 4

Due on Sunday 08/29/2021 at 03:00 PM Pacific Time

Anand Mahadevan SID - 862132182

1. *(6 points)* Given a number x and an exponent n, we would like to compute x^n.

   (a) *(2 points)* Develop an iterative algorithm for solving this problem. Your algorithm should run in $O(n)$.

   EXP_ITER(x, n):
       int ans = x^0
       for i in range(n):
           ans = ans * x
       return ans

   (b) *(2 points)* Develop a divide and conquer algorithm for solving this problem. Your algorithm should run in $O(log(n))$.

   //Strategy: $2^4 = (2^2)^2$; $2^3 = 2 * (2^2)^1$
   EXP_DC(x, n):
       if n $\leq$ 1:
           return x^n
       if n is even:
           return EXP_DC(x^2, n/2)
       else if n is odd:
           return x * EXP_DC(x^2, n/2)

   (c) *(2 points)* Prove that the time complexity of your algorithm is $O(log(n))$ by coming up with the running time recurrence relation and using the Master Theorem.

   T(n) = T(n/2) + O(1)
   Case: $0 = log_2 1 = 0$
   T(n) = $O(n^0 * log(n)) = O(log(n))$

2. *(6 points)* Given a set of n strings, we would like to find the longest common prefix among all the given strings. For example, given the set {“sandra”, “sand”, “sandiego”, ”sam”}, the lonest common prefix among all the given strings is “sa”. Assume that the longest string in the set has a constant length of 10.

   (a) *(3 points)* Develop a divide and conquer algorithm for solving this problem.Your algorithm should have a linear time complexity (i.e.,O(n)).

   ```
   //Strategy: Divide array by halves until we get 2 strings, then //return the largest prefix between the two.
   //Let A be the array of n strings
   //Let low and high be the index bounds for A
   //Initial call will be low = 1 and high = n
   LONG_PREFIX(A, low, high):
       if low equals high:
           return A[low]
       //low can never be > high
       mid = (low + high) / 2
       left = LONG_PREFIX(A, low, mid)
       right = LONG_PREFIX(A, mid+1, high)
       return the longest prefix between left and right // O(1)
   ```

   (b) *(3 points)* Show that the time complexity of your algorithm is O(n).

   T(n) = 2*T(n/2) + O(1)
   Case: $0 < log_2 2 = 1$
   T(n) = $O(n^{log_2 2}) = O(n)$

3. *(6 points)* Given n arrays of size m each where each array has positive integers. We would like to find the maximum sum obtained by selecting a number from each array such that the element selected from the i^{th} array is larger than the element selected from the $(i-1)^{th}$ array. Assume that the maximum sum with the given constraints always exists. Consider the following example: $A_1 = \{1, 7, 3, 4\}$, $A_2 = \{4, 2, 5, 1\}$, $A_3 = \{9, 5, 1, 8\}$; the maximum sum under the given constraints is 18, which is the result of adding $A_1[4] + A_2[3] + A_3[1]$.

   (a) *(2 points)* Design a greedy algorithm to find the maximum sum with the given constraints.

   1) Get Max element of each array. - O(m * n)
   2) Order the n arrays by smallest max to largest max. Assume all max elements are distinct from one another. - O(n * lg(n))
   3) Add the max element of each array in the sorted list together. - O(n)
   4) Return this sum.

   (b) *(2 points)* Prove the correctness of your algorithm.

   Let $A_1, A_2, ..., A_{n-1}, A_n$ represent n arrays where A_1 has the smallest max element out of all arrays and A_n has the largest.
   The greedy ordering α from my algorithm is simply $A_1, A_2, ..., A_{n-1}, A_n$.
   Let α^* be an optimal ordering better than α, where there are two consecutive arrays i and j where i > j, thus an array with a larger max than the one after it.
   Focusing on these two arrays A_i and A_j in α^*, the largest number we can choose for Array **j** is its max element, and the max element we can choose for Array **i** is certainly not its max as its max is larger than the max of Array **j**.

   If we were to make a new schedule from $\sigma*$, we switch the order of these Arrays i, j to have Array j before Array i. The largest number we can choose for Array **i** is its max element, and the max element we can choose for Array **j can** be its max element as it is indeed < the max element of Array **i**.

   Max before swap: $max(A[j]) + not\_the\_max(A[i])$.
   Max after swap: $max(A[j]) + max(A[i])$.
   This is a **contradiction** since we get a larger sum from the swap compared to without it, meaning there is a benefit from the swap of the order of Arrays i and j from $\sigma*$.

This means that our assumption that $\sigma*$ was the optimal schedule was incorrect, and in fact our algorithm produces the optimal schedule.

(c) *(2 points)* What is the time complexity of your algorithm?

$O(m * n + n * lg(n))$

4. *(6 points)* Consider the problem of making change for n cents using the fewest number of coins.

   (a) *(2 points)* Describe a greedy algorithm to make change consisting of the fewest number of coins given an amount of cents n. The possible denominations are: quarter (25 cents), dime (10 cents), nickel (5 cents), and penny (1 cent).

   1) Choose as many "q" quarters until the amount remaining from n is $<$ 25 cents.
   2) Choose as many "d" dimes until the amount remaining from n is $<$ 10 cents.
   3) Choose as many "c" nickels until the amount remaining from n is $<$ 5 cents.
   4) Choose as many "p" pennies until the amount remaining from n is 0 cents.
   5) Return the total number of coins used (q + d + c + p)

   (b) *(2 points)* Prove that your algorithm is correct (i.e., it produces the fewest number of coins).

   Let q, d, c, *and* p represent the number of quarters, dimes, nickels, and pennies needed to make n cents from my algorithm be called β.
   Let β^* be an optimal coin distribution better than β, where there is a total lesser amount of coins used to make n cents.
   Focusing on this β^*, let us examine what type of coin could have its total amount decreased compared to β. At least 1 category of coin **must** have decreased from β to β^* for the total coin amount to decrease.

   If we lowered β's quarter distribution, then we would need to make 25 cents out of dimes, nickels, and pennies, all possible combinations results in more than 1 coin being used. So we cannot decrease the number of quarters.
   If we lowered β's dime distribution, then we would need to make 10 cents out of nickels and pennies, all possible combinations results in more than 1 coin being used. So we cannot decrease the number of dimes.
   If we lowered β's nickel distribution, then we would need to make 5 cents out of pennies, which is simply 5 pennies per 1 nickel decreased. So we cannot decrease the number of nickels.
   And we simply cannot decrease β's penny distribution, as pennies would be the only possible replacement. So, finally, pennies cannot decrease.

Thus, this is a contradiction, as I have shown that no category of coin from β can be decreased, thus our assumption that β^* was an optimal coin distribution compared to β was incorrect.
In fact, this result implies that my algorithm produces the optimal, fewest amount of coins needed to make n cents.

(c) *(2 points)* What is the time complexity of your algorithm?
Time Complexity - O(n)

5. *(6 points)* Given a binary square matrix A with dimensions $n \times n$, where each entry $a_{i,j} \in \{0,1\}, 1 \leq i,j \leq n$, a square submatrix of ones is described using a triple (y,x,k) such that $1 \leq y,x,k \leq n$, and

$$\forall i,j \text{ such that } i \in [y, y+k-1] \text{ and } j \in [x, x+k-1] : a_{i,j} = 1.$$

We would like to find the largest square of ones (i.e., a square submatrix with the largest value of k). In the matrix below, the largest square submatrix is described by the triple $(5,3,3)$. That is, the square starts at position $(5,3)$ and it has a size of 3×3 (i.e., $k = 3$).

$$\begin{matrix}
0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\
1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\
1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\
0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\
1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\
0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
1 & 0 & 0 & 1 & 1 & 0 & 1 & 1
\end{matrix}$$

(a) *(2 points)* Develop a recursive formula for the size of the largest square block of ones. Hint: Try to create a formula that calculates the size of the largest square of ones with upper-right corner (i,j).

(i,j) represents the row,column of the top right corner of the square of 1's. Formula will return k, which is the largest k x k matrix of 1s possible with top-right corner (i,j)

Base Case:
i = n or j = 1 //out of bounds for recursive calls
Recursive Formula:
if $a_{i,j}$ equals 1:
  big_ones(i,j) = min{big_ones(i,j-1),big_ones(i+1,j-1),big_ones(i+1,j)}
    + 1
else if $a_{i,j}$ equals 0:
  big_ones(i,j) = 0

(b) *(2 points)* Use your formula to develop a bottom-up dynamic programming algorithm. Write a pseudo code for your algorithm.

```
ONES_DP(A):
    most_ones = [A.rows][A.col]
    for i in range(A.rows , 1)
        for j in range(1 , A.col)
            if i equals A.rows or j equals 1:
                most_ones[i][j] = A[i][j]
            else:
                if A[i][j] equals 1:
                    most_ones[i][j] = min{ most_ones[i][j-1], most_ones[i+1][j-1],
                                           most_ones[i+1][j] } + 1
                else if A[i][j] equals 0:
                    most_ones[i][j] = 0
    return max(most_ones)
```

(c) *(2 points)* Apply your algorithm to the example above by drawing the dynamic programming table and filling it. Circle the optimal answer in the table.

most_ones table - Optimal answer is circled below in red.

0	1	0	0	1	1	1	1
1	0	0	1	1	0	0	1
1	1	1	0	0	0	0	1
1	0	0	1	2	1	1	0
0	1	2	3	3	0	0	1
1	1	2	2	2	0	1	1
0	1	1	1	2	1	1	0
1	0	0	1	1	0	1	1

6. *(6 points)* A *palindrome* is a non-empty string that reads the same backward as forward. Examples of palindromes include: *civic*, *racecar*, *aibohphobia*; all strings of length 1 are palindromes. We would like to find the length of the longest palindrome substring in a given string using dynamic programming. For example, given the input string *character*, the longest palindrome substring *carac* has a length of 5.

   (a) *(2 points)* Develop a recursive formula for the length of the longest palindrome substring. Discuss the correctness of your formula.

   // Initial call will be str, 1, len(str), 0 for the string, low, high,
   // and max respectively
   Base Cases:
   if low equals high: return max+1 //length 1 palindrome
   if low > high: return max //out of bounds
   Recursive Formula:
   // VERY important to reset max if outer edges don't match (for cases like "cabc" for example which should return 1)
   if str[low] equals str[high]:
   return max{len_palin(str, low+1, high-1, max+2),len_palin(str, low, high-1, 0), len_palin(str, low+1, high, 0)}
   if str[low] does not equal str[high]:
   return max{len_palin(str, low, high-1, 0),len_palin(str, low+1, high, 0)}

   // Recursive calls discussion: When ends are equal, need 3 calls:
   // 1) +2 to max and check inside the ends, 2) Check left side
   // (minus 1 element on the right), 3) Check right side
   // (minus 1 element on the left)
   // When ends are not equal, need only 2 calls:
   // 1) Check left side (minus 1 element on the right), 2) Check right
   // side (minus 1 element on the left)
   // This algorithm will indeed return the max length palindrome
   // substring in the string str

(b) *(2 points)* Use your formula to design a bottom-up dynamic programming algorithm that finds the length of the longest palindrome substring.

```
LONG_DP(str):
    len = len(str)
    is_pal = [len][len] // boolean array to simulate resetting max
    //must do first two diagonals first for bottom up to work
    for i in range(len):
        is_pal[i][i] = True // all length 1 are palindromes
    max = 1 // min max palindrome is of course 1
    for i in range(len-1):
        if str[i] equals str[i+1]: // palindrome length 2
                is_pal[i][i+1] = True
                max = 2 // max palindrome now 2
    for level in range(3, len): // all remaining diagonals
        for low in range(len - level + 1): // row traversal
            high = low + level - 1 // column traversal
            // First check if the innards is a palindrome and if the outside is equal
            // This means that the whole string is a palindrome
            if is_pal[low+1][high-1] == True and str[low] == str[high]:
                is_pal[low][high] = True
                if level > max: // level equals length of palindrome substring found
                    max = level
    return max
```

(c) *(2 points)* What is the time complexity of your algorithm?

Let n be the length of the string passed in
O(n^2)