Homework 3 - Robert Duque

Monday, October 31, 2022 4:27 PM

Robert Duque - 1582182
I understand and agree to abide by the provisions in the University of Houston Undergraduate Academic Honesty Policy, This is my own work.

P0 (20pt): This exercise asks you to develop efficient algorithms to find optimal *subsequence* of various kinds. A subsequence is anything obtained from a sequence by extracting a subset of elements, but keeping them in the same order; the elements of the subsequence need not be contiguous in the original sequence. For example, the strings `C`, `DAMN`, `YAIOAI`, and `DYNAMICPROGRAMMING` are all subsequences of the string `DYNAMICPROGRAMMING`.

a) Let $A[1..m]$ and $B[1..n]$ be two arbitrary arrays. A *common subsequence* of A and B is another sequence that is a subsequence of both A and B. Describe an efficient algorithm to compute the length of the **longest common subsequence** of A and B.

To approach this problem dynamically, we need to make a function that will take parameters:
- The first string (let's call A)
- The second string to compare (let's call B)
- The length of the first string (let's call m)
- The length of the second string (let's call n)

In the function, we will initialize a table of size (m+1)*(n+1) that we will call lengthTable and initialize the first row and first column of the lengthTable to 0. We will then iterate through the table row wise and start with the first row which corresponds to the first character of the string A. If the current row character is equal to the current column character, then we add one to the number that resides in the previous row/column cell (diagonal). If the character of the row does not match the character of the column, then we will just take the max of the previous adjacent cells of the current iteration cell. When we run this, the length of the longest common subsequence of both A and B will be in the last row/column cell of the lengthTable table.

Pseudocode:

```
lcsLength(A, B, m, n):
    lengthTable[m][n]
    For I in range(0,m):
        lengthTable[i][0] = 0
    For I in range(0,n):
        lengthTable[0][i] = 0
    For I in range(0,m):
        For j in range(0,n):
            If(char lengthTable[i] == char lengthTable[j]):
                lengthTable[i][j] = lengthTable[i-1][j-1] + 1
            Else:
                lengthTable[i][j] = max(lengthTable[i-1][j], lengthTable[i][j-1])
    Return lengthTable[m][n]
```

The run time complexity of the algorithm will be O(m*n) since there is a nested for loop in the function that is used to create a dynamic programming array for the following algorithm.

b) Call a sequence $X[1\ldots n]$ of numbers *oscillating* if $X[i] < X[i+1]$ for all even i, and $X[i] > X[i+1]$ for all odd i. Describe an efficient algorithm to compute the length of the **longest oscillating subsequence** of an arbitrary array X of integers.

To approach this problem dynamically, we will make a function that will take the parameters:
1. Array of integers (X)
2. Length of the array X

In the function, we will start by initializing a integer called longestSeq to 1 that will hold the largest oscillating sequence of array. We will also initialize a dynamic array named dpArr of size [len][2]. The dynamic array will hold all inputs of the array as well as a corresponding number that will represent the longest oscillating sequence starting at that specific number in the array. In the function, we will make a for loop that will iterate through the length of the input array (index will be i) and then we will implement a nested for loop that will start from the index of the first loop minus 1 and decrement by 1 (index will be j). In the nested for loop, we will have 4 if statements that will check 2 things: If the first for loop index is either an even number or an odd number and check if arr[j] < arr[i] or arr[i] < arr[j]. If arr[j] < arr[i], we will set dpArr[i][0] to the max of dpArr[j][1] + 1 and dpArr[i][0]. If arr[i] < arr[j], we will set dpArr[i][1] to the max of dpArr[j][0] + 1 and dpArr[i][1]. These calculation will make sure that each index of the dynamic array will hold the maximum amount of oscillating entries of that current index i. After the nested for loop, in the main for loop we will set the variable longestSeq to the max of the longestSeq and the max of dpArr[i][1] and dpArr[i][0]. The function will return longestSeq + 1 since the algorithm will only account for the sequence not including the current index of I in the main for loop.

Pseudocode:

```
findOscSeq(arr, len):
    longestSeq = 1
    dpArr[len][2] = {1}
    For I in range(0,len):
        For j in range(i-1,0,-1):
```

Programming Questions:

Programming Challenge P9:
PSID: 1582182
Submission ID: 38ece01a-b1c4-45bc-b6dc-9de821311a1b

Programming Challenge P5:
PSID: 1582182
Submission ID: 15061bfd-21fb-4b6c-b565-f6a5c0a1df05

```
                If i%2 == 0 && arr[j] < arr[i]:
                        dpArr[i][0] = max(dpArr[j][1]+1,dpArr[i][0])
                If i%2 == 0 && arr[i] < arr[j]:
                        dpArr[i][1] = max(dpArr[j][0]+1,dpArr[i][1])
                If i%2 != 0 && arr[j] < arr[i]:
                        dpArr[i][0] = max(dpArr[j][1]+1,dpArr[i][0])
                If i%2 != 0 && arr[i] < arr[j]:
                        dpArr[i][1] = max(dpArr[j][0]+1,dpArr[i][1])
            longestSeq = max(longestSeq, max(dpArr[i][1],dpArr[i][0]))
    Return longestSeq + 1
```

The run time complexity of the above algorithm will be O(n^2). In the function, every elements of the array that we iterate to, we made a nested for loop that will run i-1 times which will explain the n^2 run time complexity.

P1 (20pt) A string w of parentheses (and) and brackets [and] is balanced if it satisfies one of the following conditions:

- w is the empty string
- $w = (x)$ for some balanced string x
- $w = [x]$ for some balanced string x
- $w = xy$ (concatenation) for some balanced strings x and y

1. Describe and analyze an algorithm to determine whether a given string of parentheses and brackets is balanced.

For this problem, we will simply iterate through the string and check if the string is balanced by implementing a stack that will hold all of the brackets in the string. When we meet an open bracket in the string ((, { , [ ), we will push that symbol onto the stack of characters. When we meet a closing bracket ( ), },]), we will simply pop the top of the stack and if the character that is returned by the pop is of the same bracket type, then the string is balanced. The string is not balanced if the top that was popped from the stack does not equal the current closing bracket. The function will be called isBalanced and will only take the input string as the parameter. The return value will be a boolean that will indicate whether the string is balanced or not.

Pseudocode:

```
isBalanced(inString)  => (inString is the input string)
    Len = size of inString
    Stack symStack
    For I in range(0,len):
        If(inString[i] is '(' or '{' or '[' ):
            symStack.push(inString[i]
            Continue
        If(inString[i] == ')' ):
            If(symStack.top == '(' ):
                symStack.pop()
                Continue
            Else:
                Return false
        If(inString[i] == '}' ):
            If(symStack.top == '{' ):
                symStack.pop()
                Continue
            Else:
                Return false
        If(inString[i] == ']' ):
            If(symStack.top == '[' ):
                symStack.pop()
                Continue
            Else:
                Return false
    Return true
```

The run time complexity of this algorithm will be linear (O(n)) because we are only using one more loop and implementing a stack that will keep track of what insertions are balanced or not. We are only executing this for loop once which gives the algorithm a linear run time complexity.

2. Describe and analyze an algorithm to compute the length of a longest balanced subsequence of a given string of parentheses and brackets.

To implement this version of the previous algorithm, we would simply add a vector of integers variable called longestBal and an integer variable named currBal that will keep track of the length of the current balanced bracket set. We will use the same exact if statements but we will add another layer the nested if statements that check if the top of the stack matches the current closing bracket. First, in the beginning if statement that checks if the current char is an opening bracket, we will set currBal to 0 since this will indicate a restart on a balanced string. In each of the nested if statements where we pop the stack, we will add 1 to currBal as well as add to the else statement which will push_back the currBal number to the vector longestBal and set currBal to 0 instead of returning false. This happens if there is an imbalance of the brackets. We will then initialize a integer called largest to 0 and call a for loop that will loop through the vector of integers and find the largest number in that vector and assign the max number to largest. We will finally return largest*2 since we only add to currBal when the current char is a closing bracket. The previous function was a boolean function that returns true or false but since we are looking for the length of the longest balanced subsequence, we will return an

integer.

Pseudocode:

```
isBalanced(inString)  => (inString is the input string)
        Len = size of inString
        Stack symStack
        currBal = 0
        Vector<int> longestBal
        For I in range(0,len):
                If(inString[i] is '(' or '{' or '[' ):
                        symStack.push(inString[i]
                        currBal = 0
                        Continue
                If(inString[i] == ')' ):
                        If(symStack.top == '(' ):
                                symStack.pop()
                                currBal += 1
                        Else:
                                longestBal.push_back(currBal)
                                currBal = 0
                        continue
                If(inString[i] == '}' ):
                        If(symStack.top == '{' ):
                                symStack.pop()
                                currBal += 1
                        Else:
                                longestBal.push_back(currBal)
                                currBal = 0
                        continue
                If(inString[i] == ']' ):
                        If(symStack.top == '[' ):
                                symStack.pop()
                                currBal += 1
                        Else:
                                longestBal.push_back(currBal)
                                currBal = 0
                        continue
        Largest = 0
        For I in range(0,size of longestBal vector):
                If(longestBal[i] > largest):
                        Largest = longestBal[i]
        Return largest*2
```

The run time complexity of the algorithm should be no different than the previous algorithm since we didn't include vast additions to the algorithm, just introducing new variables that tackle a different problem of the same problem set. Both for loops are executed linearly which gives a run time complexity of O(n+n) = O(n).