11/6/2021

=== P0 ===

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.

Form a Matrix AB of size m x n .
For each row r:
    For each col c in row r:
        If A[r] == B [c]
            If r == 0|| c==0
                AB[r][c] = 1;
            Else:
                AB[r][c] = A[r-1][c-1] + 1
        Else:
            If r==0:
                If c == 0:
                    AB[r][c] = 0
                Else:
                    AB[r][c] = AB[r][c-1]
            Else
                If c== 0:
                    AB[r][c] = AB[r-1][c]
                Else:
                    AB[r][c] = max(AB[r-1][c], AB[r][c-1]
Return AB[m-1][n-1]

Time complexity is O(m x n)

b) Call a sequence $X[1..n]$ of numbers *bitonic* if there is an index i with $1<i<n$, such that the prefix $X[1..i]$ is increasing and the suffix $X[i..n]$ is decreasing. Describe an efficient algorithm to compute the length of the **longest bitonic subsequence** of an arbitrary array X of integers.

Given that X > 2

Make two arrays LIS and LDS (longest increasing subsequence and longest decreasing subsequence)

LIS[i] = length of longest increasing subsequence from 1 -> i
LDS[i] = length of longest decreasing subsequence from i -> n

To compute, first compute LIS

```
LIS[0 -> n - 1] = 1

For i  = 1; i < n; i++
        For j = 0; j < i; j++
                If X[i] > X[j] && LIS[i] < LIS[j] + 1
                        LIS[i] = LIS[j] + 1
```

Then LDS

```
LDS[0 -> n - 1] = 1

For i  = n-2; i >= 0; i--
        For j = n - 1; j > i; j--
                If X[i] > X[j] && LIS[i] < LIS[j] + 1
                        LIS[i] = LIS[j] + 1;
```

Return max(LIS[i] + LDS[i] - 1)

Time complexity is O(n ^ 2)

=== P1 ===

My strategy would work by finding the largest rectangle duplicate a point.

r = 0 (row)
c = 0 (col)

The following rules apply to make it easy to explain rectangle duplicate finding.

Origin is the top left point of my original rectangle
Examined is the top left point of the rectangle that is being checked to see if it is a duplicate of the Origin rectangle

M[r][c] is the origin point
M[i][j] is the Examined point

O is the origin rectangle, the rectangle starting at M[r][c] and going to M[r + l][c + w]
E is the examined rectangle, the rectangle starting at M[i][j] and going to M[i + l][j + w]

comparison = (rc, rt) => { return M[r + rt][c + ct] == M[i + rt][j + ct]; } //js :)

Mcr = min(n - r, n - i); //max comparison row
Mcc = min(n - c, n - j); // max comparison col

Mor = n - r //max origin row
Moc = n - c //max origin col

Mer = n - i //max examined row
Mec = n - j //max emanied col

MRect = (current largest rectangle)
OLF = Mor x Moc (the largest formable rectangle given origin)
ELF = Mer x Mec (the largest Formable for the examined)
CLF = Mcc x Mcr (the largest formable rectangle for the current comparison)

```
LargestRectCpy(r, c) {
        Max = 1;
        For i = r, j = c + 1; i < n; i++;
                For j < n; j++;
                        If ELF < MR
                                Break;
                        Temp = largestRectfromPts(r, c, i, j);
                        If temp > max:
                                Max = temp;
                J = 0;
                If ELF < MR
                        break;
        Return max;
}

LargestRectFromPts(r, c, i, j) {
        Mcr = min(n - r, n - i); //max comparison row
        Mcc = min(n - c, n - j); //max comparison col
        Lc = 0 (largest column)
        For rt  = 0; rt < Mcr; rt++;
                For rc = 0; rc < Mcc; rc++;
                        If !comparison(rc, rt)
                                 If rc x Mcr < Mrect:
                                        Return Lc x rt;
                                Lc = rc;
                                break;
                        If Lc != 0 && (rc == Lc):
                                Break;
        Return Mcr X Mcc;
}

LargestRect() {
        For r = 0; r < n; r++;
                For c = 0;  c < n; c++;
                        If LF < Mrect
                                Return Mrect;  // cannot form a larger rectangle so return MR
                        Temp = LargestRectCpy(r, c);
                        If temp > MRect:
                                Mrect = temp;
        Return  Mrect;
}
```

Time complexity is O(n^5)

P2 (20pt): Suppose you are given a sequence of integers separated by + and − signs; for example:

$$1+3-2-5+1-6+7$$

You can change the value of this expression by adding parentheses in difference places. For example:

$$\begin{aligned} 1+3-2-5+1-6+7 &= -1 \\ (1+3-(2-5))+(1-6)+7 &= 9 \\ (1+(3-2))-(5+1)-(6+7) &= -17 \end{aligned}$$

Describe and analyze an algorithm to compute, given a list of integers separated by + and − signs, the maximum possible value the expression can take by adding parentheses. Parentheses must be used only to group additions and subtractions; in particular, do not use them to create implicit multiplication as in 1 + 3(-2)(-5) + 1- 6 + 7 = 33.

```
Lowest(exp){
        Min = eval(exp);
        If exp only contains one operator:
                Return Min;
        For i in exp.len:
                If exp[i] = '+':
                        Temp = lowest (exp[0 : i  - 1]) + lowest (exp[i + 1]);
                        if(temp < min):
                                Min = temp;
                Else if Exp[i] = '-':
                        Temp = lowest (exp[0 : i  - 1]) - highest (exp[i + 1]);
                        if(temp < min):
                                Min = temp;
        Return Min;
}
Highest(exp, l, r){
        //if this subproblem has already been calculated, no need to recalculate
        If DP [l][r] != empty
                Return DP[l][r];

        Max = eval(exp);
        If exp only contains one operator:
                Return Max;
        For i in n:
                If exp[i] = '+':
                        Temp = highest (exp, l, i - 1) + highest (exp, i + 1, r);
                        if(temp > max):
                                Max = temp;
                Else if Exp[i] = '-':
                        Temp = highest (exp, l, i  - 1]) - lowest (exp, i + 1, r);
```

```
                        if(temp > max):
                                Max = temp;
        Return Max;
}
MaxVal(exp){
        // with n being exp.len
        global DP [n][n]; //empty DP matrix;
        return Highest(exp);
}
```

Simply put the expression into in order to obtain the highest value parenthesis combination.

This algorithm runs in O(n ^ 3) time complexity, using DP to remain efficient by not recalculating subproblems.

P3 (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

For example, the string
$w = ([()][]())[()()]()$ is balanced, because $w = xy$, where $x = ([()][]())$ and $y = [()()]()$

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

1.
I use the stack method as it is the best way to determine balanced brackets in linear time and space

```
balancedBrackets(exp){
        Stack s;
        For i = 0; i < exp.len; i++;
                If exp[i] = '[' or '('  //an opening bracket
                        s.push(exp[i])
                Else //a closing bracket
                        If s.empty:
                                Return false; //closing bracket with no matching opening bracket
                        If exp[i] == ']' && s.pop() != '['
                                Return false; //same as next condition
                        If exp[i] == ')' && s.pop() != '('
                                Return false; //opening and closing brackets do not match

        Return S.empty
         //if stack is not empty, then we have an opening bracket without a closing bracket
}
```

O(n) time complexity, it's a linear scan

2.
I would use the same algorithm that I used for the first question but with some modifications…

```
balancedBrackets(exp){
        Stack s;
        Length = 0
        For i = 0; i < exp.len; i++;
                If exp[i] = '[' or '('  //an opening bracket
                        s.push(exp[i])
                Else //a closing bracket
                        If exp[i] == ']' && s.pop() == '['
                                Length++ //allowed expansion
                        If exp[i] == ')' && s.pop() == '('
                                Length++ //as this is a valid addition to the subsequence

        Return length
}
```

Again this runs in O(n) time complexity, it's a linear scan.

Greedy algorithms: Please provide complete proof of correctness! The proof usually involves the exchange argument. You don't have to prove you are correct if you are not using greedy algorithm.

P4 (20pt) Let X be a set of n intervals on the real line. We say that a subset of intervals $Y \subseteq X$ covers X if the union of all intervals in Y is equal to the union of all intervals in X . The size of a cover is just the number of intervals.

Describe and analyze an efficient algorithm to compute the smallest cover of X. Assume that your input consists of two arrays $L[1..n]$ and $R[1..n]$, representing the left and right endpoints of the intervals in X.

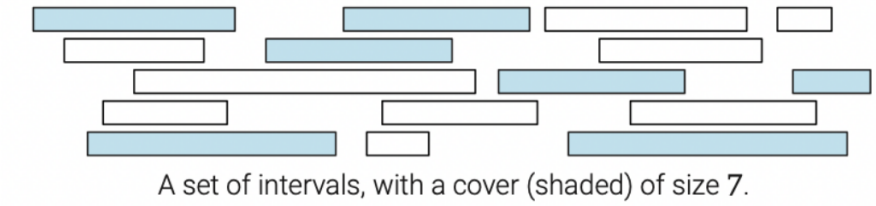

Figure 1. Example of a cover. This is not the smallest cover.

My solution is a greedy algo.

If n == 0 return 0 //no intervals, no point!

```
Let size = 1
Let i = interval where L[i] = 0 and max(R[i]) //start at beginning and cover as much as possible
While R[i] != max(R) //while we have not covered all of X yet
        Max = 0
        For j = 1; j < n; j++;
                If j == i || L[j] < L[i]
                        Continue; //no point in these ones, they are end before or are interval i
                If j <= R[i] + 1 && R[j] > max //finding next interval
                        Max = j;
        i = Max
        Size++
Return Size
```

This algo runs in O(n ^ 2) time.

The basis is that the interval i starts at 0 but goes further than others that start at 0.
Then we repeat but instead of 0, the interval j covers the end of i, being (R[i] + 1) and then R[j] being greater than any other interval that covers R[i] + 1.

We of course do not have to check any interval along the way such that L[interval] <= L[i] as this interval would have been selected earlier instead of i to begin with.