DAA (CS 6363) - Homework 3
Karthik Ragunath Ananda Kumar (KXA200005)
Vijay Anand Varma Indukuri (VXI210000)

Problem 1. Longest Convex Subsequence (30 points) Call a sequence [x1,..., xm] of numbers convex if x[i] < (x[i−1]+x[i+1] / 2) for all 1 < i < m. Use dynamic programming to give an O(n^3) time algorithm to compute the length of the longest convex subsequence of an arbitrary array A[1...n] of n integers.

Recursive Solution
The base is when length of given array is less than 3, then, longest convex subsequence is not possible and we return 0. We are calculating the Longest Convex Subsequence from right to left and hence LCS(prev_prev, prev, start) indicates the longest convex subsequence length possible between the indices start and n considering all elements greater than prev_prev.

Pseudo Code:

```
LCS(prev_prev, prev, start, n, A):
    if start > n then: # assuming
        return 0
    ignore = LCS(prev_prev, prev, start + 1, n, A)
    best = ignore
    if A[prev] < ((A[prev_prev] + A[start]) / 2) then:
        include = 1 + LCS(prev, start, start + 1, n, A)
        if include > ignore then:
            best = include
    return best

solution(A):
    if len(A) < 3:
        return 0
    else:
        return LCS(1,2,3,len(A), A)
```

where A here indicates the array for which we are computing the Longest Convex Subsequence and initial values of prev_prev, prev are 1 and 2 (assuming we are using array indexing starting at 1)

Correctness of Algorithm:
The hypothesis here is that LCS(prev_prev, prev, start) returns the longest convex subsequence possible between start and n indices such that all considered elements are greater than prev_prev. At each recursive step, we consider two cases, either the start element will be part of the solution or ignored. Both cases are computed recursively and we take the best possible solution of the two cases. Hence, at the end of the recursive call we will end up with the best possible solution possible.

Sample Execution

edit fork download copy

```
def LCS(prev_prev, prev, start, n, A):
    if start >= n:
        return 0
    ignore = LCS(prev_prev, prev, start + 1, n, A)
    best = ignore
    if A[prev] < ((A[prev_prev] + A[start]) / 2):
        include = 1 + LCS(prev, start, start + 1, n, A)
        if include > ignore:
            best = include
    return best

def solution(A):
    n = len(A)
    if n < 3:
        return 0
    else:
        lcs = LCS(0, 1, 2, n, A)
        return lcs

A = [1, 3, 9, 8, 16]
print(solution(A))
```

Success #stdin #stdout 0.02s 9132KB comments (0)

stdin copy

```
Standard input is empty
```

stdout copy

```
2
```

If we want to compute a solution where we need to consider the first two elements as start elements too, we can simply modify the ‘A’ array by adding 2 zeros before it.
Running the code for the same input with minor modification we stated above:

```
def LCS(prev_prev, prev, start, n, A):
    if start >= n:
        return 0
    ignore = LCS(prev_prev, prev, start + 1, n, A)
    best = ignore
    if A[prev] < ((A[prev_prev] + A[start]) / 2):
        include = 1 + LCS(prev, start, start + 1, n, A)
        if include > ignore:
            best = include
    return best

def solution(A):
    A = [0,0] + A
    n = len(A)
    if n < 3:
        return 0
    else:
        lcs = LCS(0, 1, 2, n, A)
        return lcs

A = [1, 3, 9, 8, 16]
print(solution(A))
```

Success #stdin #stdout 0.02s 9084KB

stdin

```
Standard input is empty
```

stdout

```
4
```

Converting this into DP solution:

Dynamic Programming Pseudo-Code:

```
LCS(prev_prev, prev, start, n, A):
    lcs_dp = [[[None] * (n + 1)] * n] * (n - 1)
    for i in range(0, n - 1):
        for j in range(0, n):
            lcs_dp[i][j][n] = 0
    for start in range(n - 1, 1, -1):
        for prev in range(start - 1, 0, -1):
            for prev_prev in range(prev - 1, -1, -1):
                ignore = lcs_dp[prev_prev][prev][start + 1]
                best = ignore
                if A[prev] < ((A[prev_prev] + A[start]) / 2):
                    include = 1 + lcs_dp[prev][start][start + 1]
                    if include > ignore:
                        best = include
                    lcs_dp[prev_prev][prev][start] = best
    return lcs_dp[0][1][2]

solution(A):
    n = len(A)
    if n < 3:
        return 0
    else:
        lcs = LCS(0, 1, 2, n, A)
        return lcs
```

Here, the logic is the same as the recursive solution except that we are storing the result of dp[prev_pre][prev][start] combination at each iteration which gives maximum Longest Convex Subsequence between indices from ‘start’ to ‘n’ considering all elements greater than prev_prev. In order to simplify the visualization, we consider the 3-D data store as 2-D where the first dimension is formed by merging for loops iterating over prev_prev and prev elements and hence have O(n^2) elements. We can clearly see that LCS(prevOfP rev, prev, start) depends on LCS(prevOfP rev, prev, start + 1) and LCS(prev, start, start + 1) and hence in 2-D space. Hence the dynamic programming table is filled from greater to smaller elements in x-dimension (right to left) and again from greater value to smaller value in y-dimension (bottom to top).

Time Complexity of DP Solution:

There are totally n^2 * n = n^3 entries in the dynamic programming table and each entry is filled in constant O(1) time. Hence, the total time complexity of the dynamic programming solution is O(n^3).

Problem 2:

You have mined a large slab of marble from a quarry. For simplicity, suppose the marble slab is a rectangle measuring n inches in height and m inches in width. You are also given a price table P[1...n][1...m], where P[x][y] is the price for selling an x-inch by y-inch marble rectangle. You can assume the prices are non-negative, but otherwise you should not make any assumptions; in particular, larger rectangles could have significantly smaller prices. You also have a marble saw that can make either horizontal or vertical cuts at any integer inch position for free. Note any cut must go all the way across from one side of the slab to the other. Given the table P of prices and integers n and m as input, describe an efficient algorithm that computes the maximum profit obtainable by cutting an n × m marble rectangle into smaller rectangles for sale.

Recursive Solution:

The base case of the recursive solution will be when n = 0 or m = 0 which means that there are no more slabs to cut. Let recursive_cuts(n, m, p) denote the maximum profit obtainable by cutting an n∗m marble rectangle into smaller rectangles computed using price matrix P[1...n][1...m] where P[x][y] is the price for selling an x-inch by y-inch marble rectangle.

Pseudo Code:

```
recursive_cuts(n,m,p):
    if (n == 0  or  m == 0) then:
        return 0 # Base Case.
    else:
        max_val = p[n - 1][m - 1] # Full Slab
        for i in range(1, int(n/2) + 1): # Horizontal Cuts.
            max_val = max(max_val, (recursive_cuts(i,m,p) + recursive_cuts(n-i,m,p)))
        for j in range(1, int(m/2) + 1): # Vertical Cuts.
            max_val = max(max_val, (recursive_cuts(n,j,p) + recursive_cuts(n,m-j,p)))
    return max_val

def solution(p):
    n = len(p)
    m = len(p[0])
    max_profit = recursive_cuts(n,m,p)
    return max_profit
```

Correctness of the solution:

At each recursive step, we compute the max_val by considering three possibilities. The price at which we could sell the slab as it is, the max we could get by making optimal horizontal cuts and the profit we could make by using optimal vertical cuts. Since, the three cases considered encloses all possible cuts, we get the max profit for slab dimension n*m at the end of each recursive function call - recursive_cuts(n,m,p) where p indicates price for selling an x-inch by

y-inch marble rectangle.

Sample Execution:

edit fork download copy

```python
def recursive_cuts(n,m,p):
    if (n == 0  or  m == 0):
        return 0 # Base Case.
    else:
        max_val = p[n - 1][m - 1] # Full Slab
        for i in range(1, int(n/2) + 1): # Horizontal Cuts.
            max_val = max(max_val, (recursive_cuts(i,m,p) + recursive_cuts(n-i,m,p)))
        for j in range(1, int(m/2) + 1): # Vertical Cuts.
            max_val = max(max_val, (recursive_cuts(n,j,p) + recursive_cuts(n,m-j,p)))
    return max_val

def solution(p):
    n = len(p)
    m = len(p[0])
    max_profit = recursive_cuts(n,m,p)
    return max_profit

p = [[3,45,8], [12,23,34]]
max_profit = solution(p)
print(max_profit)
```

Success #stdin #stdout 0.02s 9252KB comments (0)

stdin copy

```
Standard input is empty
```

stdout copy

```
102
```

Converting into DP Solution:

We can convert the recursive solution into dp solution by using the same principle we used for the recursive solution except that we memoize the solution for each slab rectangular cut. We fill the dp matrix from 1 to n (left to right) and 1 to m (top to bottom). Hence, the overall solution is found at dp[n][m].
DP Pseudo Code:

```
recursive_cuts(n,m,p):
    cuts_dp = [[0] * (m + 1)] * (n + 1)
    for i in range(1, n + 1):
        for j in range(1, m + 1):
            max_profit = p[i - 1][j - 1]
```

```
            for k in range(1, int((i/2) + 1)):
                max_profit = max(max_profit, (cuts_dp[k][j] + cuts_dp[n - k][j]))
            for k in range(1, int((j/2) + 1)):
                max_profit = max(max_profit, (cuts_dp[i][k] + cuts_dp[i][m - k]))
            cuts_dp[i][j] = max_profit
    return max_profit

solution(p):
    n = len(p)
    m = len(p[0])
    max_profit = recursive_cuts(n,m,p)
    return max_profit
```

Time Complexity:

The dp table has n*m entries and it takes O(1) to update each entry. Hence, the overall time complexity of the dp solution is O(n * m).

Problem 3:

Recursive Solution

Pseudo Code:

```
def Solution (E [], D [], R [] [], left, current, right):
        if (left = 0 && right = N+1)   return 0;
        if (left = 0) {
                return (Solution (E [], D [], R [] [], left, current, right + 1)) + (R (right, N) *
E[right] – E[current]);
        }
        if (right = N+1) {
                return (Solution (E [], D [], R [] [], left-1, current, right)) + (R (1, left) *
E[current] – E[left]);
        }

        costLeft = (E(current) – E(left)) * (R (1, left) + R (right, N))
        costRight = (E(right) – E(current)) * (R (1, left) + R (right, N))

        updatedLeftCost = costLeft + Solution (E [], D [], R [] [], left-1, left, right)
        updatedRightCost = costRight + Solution (E [], D [], R [] [], left, right, right+1)

        return min (updatedLeftCost, updatedRightCost)
```

Above pseudo code runs in $O(2^n)$ time because of the recursion. Now we try to induce memorization into the above recursive solution to get better time complexity.

Correctness of the Algorithm:

Intuitively, we can say that the number of ounces can be minimized by minimizing the total distance travelled by all the students while on the bus. Minimizing (Students on bus * Distance travelled by the bus).

Similar to LIS, we can consider two options for all the exits. One is going left from that exit and another is going right. Initially, we take a given starting point as our current exit, exit before starting point as left pointer and exit after starting point as our right pointer. If we go left, right pointer will remain unchanged and both left and current pointers will be reduced by one exit. Similarly, going right will not change the left pointer but both right and current pointer will increase by one exit.

Cost incurred after taking a left step, cost[left] = (E(current) – E(left)) * (R (1, left) + R (right, N))
(E(current) – E(left)) gives the distance in miles between current exit and left exit.
(R (1, left) + R (right, N)) gives the total number of students travelling from current exit to left exit

Cost after taking a right step, cost[right] = (E(right) – E(current)) * (R (1, left) + R (right, N))
(E(right) – E(current)) gives the distance in miles between current exit and right exit.
(R (1, left) + R (right, N)) gives total number of students travelling from current exit to right exit

Recursively left step incurs a total cost of: cost[left] + Solution (left-1, left, right)
Recursively right step incurs a total cost of: cost[right] + Solution (left, right, right+1)

Total cost at an exit will be Minimum of (cost[left] + Solution (left-1, left, right)) and (cost[right] + Solution (left, right, right+1))

Base case will be when the left exit becomes zero or right exit becomes N+1. Also, when both left becomes zero and right becomes N+1, we return zero.

Pseudo Code with Memoization:

In this we try to use the previous move of the bus (LEFT or RIGHT). Using this move we get the current exit of the bus. Let's call this variable as prev_move and its values will be LEFT or RIGHT. Let **cache** be a 3D array where the first dimension will be left exits, second will be right exits and third will give us information about our previous move. If our previous move is LEFT, then the current exit will be left + 1. If our previous move is RIGHT, then the current exit will be right – 1.

```
def Solution (E [], D [], R [] [], cache [] [] [], left, right, prev_move):
        if (cache [left] [right] [prev_move] ! = null) {
                return cache [left] [right] [prev_move];
        }
        if (left = 0 && right = N+1) {
                return 0;
        }
        if (prev_move = "LEFT") {
                current = left + 1
        } else if (prev_move = "RIGHT"){
                current = right - 1
        }
        if (left = 0) {
                return (Solution (E [], D [], R [] [], left, right + 1, prev_move)) + (R (right, N) *
E[right] – E[current]);
        }
        if (right = N+1) {
                return (Solution (E [], D [], R [] [], left-1, right, prev_move)) + (R (1, left) *
E[current] – E[left]);
        }

        costLeft = (E(current) – E(left)) * (R (1, left) + R (right, N))
        costRight = (E(right) – E(current)) * (R (1, left) + R (right, N))
```

```
updatedLeftCost = costLeft + Solution (E [], D [], R [] [], left-1, right, “LEFT”)
updatedRightCost = costRight + Solution (E [], D [], R [] [], left, right+1, “RIGHT”)

cache [left] [right] [prev_move] = minimum (updatedLeftCost, updatedRightCost)
return minimum (updatedLeftCost, updatedRightCost)
```

Converting into DP Solution - The dp solution is very similar to the memoized solution mentioned above where we cache the result at each instance of dp[left][right][prev_move] as we have explained in the memoized solution.

Time Complexity:

This pseudo code of recursion with memoization and dynamic programming runs in $O(n^2)$ time because we are creating a table size of $n*n*2 = 2n^2$ and each entry will be created in constant time.