CSCI 665 Foundation of Algorithms

Homework 4

Guo, Zizhun (zg2808)

Problem 1

Algorithm Description

For this problem, I implemented a dynamic programming algorithm to find the longest convex subsequence of an array. The heart of the solution is described down below:

S[i][j] = the maximum length of longest convex subsequence started with a^i and ended with a^k.

$$\mathrm{S}[i][\mathrm{j}] = \begin{cases} 2 & for\ each\ i, j\ before\ k\ starts\ to\ loop \\ \max(S[i][j], S[j][k] + 1) & for\ all\ k\ \in [j+1, n], \quad if\ a_i + a_k \geq 2a_j \end{cases}$$

$a_i$and $a_j$are picked from the tail of the array, to move forwarded to the very front. In Pseudocode part, the index for each loop is from back to front.

Running Time Estimate

There exists a two-dimensional matrix constructed, and an extra linear traverse to refresh the value for each element. Each time a new S element updated, it would be comparing with the max value, so in total it's $\mathrm{O}(n^3)$.

Pseudocode

```
dpConvex(array):
    n = array.length
    max = 1
    for each i = n – 2 to 0:
        for each j = n – 1 to  i +  1:
            S[i][j] = 2;
            For each k = j + 1 to n – 1:
                If a[i] + a[k] >= 2*a[j]:
                    S[i][j] = max(S[i][j], S[j][k] + 1)
            max = max (S[i][j], max)
    return max
```

Problem 2

Algorithm Description

For this problem, I implemented the dynamic programming algorithm to find the longest common sequence among all three and print out the longest common sequence that appeared as the subsequence in all three given sequences.

For each *(i, j, k)* pair, it is represented as Node Class which includes three public variables: ***val*** (longest common sequence length so far), ***ele*** (default: -1, element's value when all three elements are same), front (the front linked Node that has the common).

The heart of the solution is described as follows:

$\mathrm{N}[i][j][\mathrm{k}].val$ = the maximum length of common sequence among three arrays a, b, c, where each length is i, j and k.

$$\mathrm{N}[i][j][\mathrm{k}].val = \begin{cases} 0 & if\ i = 0\ or\ j = 0\ or\ k = 0 \\ N[i-1][j-1][k-1].val + 1 & if\ a_i = b_j = c_k \\ \max(N[i-1][j][k], N[i][j-1][k], N[i][j][k-1]).val & if\ a_i != b_j\ or\ a_i != c_k\ or\ b_j != c_k \end{cases}$$

$\mathrm{N}[i][j][\mathrm{k}].ele =$
$the\ equavalent\ element\ value\ if\ ith\ element\ a\ equals\ to\ jth\ element\ in\ b\ and\ kth\ element\ in\ c.$

$$\mathrm{N}[i][j][\mathrm{k}].ele = \begin{cases} -1 & if\ i = 0\ or\ j = 0\ or\ k = 0 \\ a_i & if\ a_i = b_j = c_k \\ -1 & if\ a_i != b_j\ or\ a_i != c_k\ or\ b_j != c_k \end{cases}$$

$$Current.front = \begin{cases} null & if\ i = 0\ or\ j = 0\ or\ k = 0 \\ N[i-1][j-1][k-1] & if\ a_i = b_j = c_k \\ \max(N[i-1][j][k], N[i][j-1][k], N[i][j][k-1]) & if\ a_i != b_j\ or\ a_i != c_k\ or\ b_j != c_k \end{cases}$$

Running Time Estimate

The DP algorithm constructed in a three-dimensional matrix and requires a linked list to fetch all front node of which element not equals to -1, that makes the $\mathrm{O}(\mathrm{pqr} + \mathrm{p} + \mathrm{q} + \mathrm{r})$. For longest common sequence's size is $\mathrm{O}(\mathrm{pqr})$., to print out the sequence is $\mathrm{O}(\mathrm{p} + \mathrm{q} + \mathrm{r})$.

Pseudocode

dpLCS(*array1, array2, array3*):

let *p = array1.length*

let *q = array2.length*

let *r = array3.length*

Initialize all Nodes

For each *i = 0 or j = 0 or k = 0*, there *N[i][j][k] = 0*

For each *i = 1 to p*:

For each *j = 1 to q*:

For each *k = 1 to r*:

If array1[i] = array2[j] = array3[k]:

N[i][j][k].val = N[i - 1][j - 1][k - 1].val + 1

N[i][j][k].ele = array1[i]

N[i][j][k].front = N[i - 1][j - 1][k - 1]

Else

N[i][j][k].front = max(N[i - 1][j][k], N[i][j - 1][k], N[i][j][k - 1])

N[i][j][k].val = N[i][j][k].front.val + 1

Printout *N[p][q][r].val*

Pass all nodes' ele to an new array if *ele != -1*

Printout all elements of new array

Problem 3

Algorithm Description

For this problem, I implemented a dynamic programming algorithm to find the smallest possible weight that has the total cost at least the given C. Here follows the heart of the solution:

$\mathrm{S}[i][\mathrm{j}] = the\ minimum\ weight\ that\ has\ at\ least\ total\ cost\ of\ C.$

i represents the cost range from 1 to C, whereas j represents the $j^{th}\ item$

$$\mathrm{S}[i][j] = \begin{cases} 0 & if\ i = 0\ \ and\ j = 0 \\ 0 & if\ i = 0 \\ \infty & if\ j = 0 \\ \min(S[0][j-1] + w, S[i][j-1] & if\ cost > j \\ \min(S[i-c][j-1] + w, S[i][j-1] & if\ cost \leq j \end{cases}$$

The solution is yielded to by passing the smallest possible weight along the diagonal or vertical in the solution matrix.

For $\mathrm{i} = 0$, which means at current cost of 0, there has no item that cost 0, so all elements here are assigned with 0 for further iteration.

For $\mathrm{j} = 0$, which means at current item none, there has no any level of cost that is produced by no item, so for getting the min value, all elements here are assigned as ∞ whereas in program, it is replaced by 10000 for convenience.

For $cost > j$, which means the cost for current item is greater than current bounded *Cost j*, so when traced back to $j - cost < 0$, that implies the semantics of "*total cost at least C*". It is necessary to manually assign it to $0^{th}\ item\ situation.$

For $cost \leq j$, which means the cost for current item is less or equal to current bounded *Cost j*, either choose the $S[i-c][j-1] + w\ or\ S[i][j-1]$ that has smaller value. The heart of solution sits well to the Knapsack problem taught in the class.

Running Time Estimate

Since the dynamic programming matrix is a two-dimensional array with size of $\mathrm{C} \times \mathrm{n}$, so the Time running time complexity is $\mathrm{O}(\mathrm{nC})$

Pseudocode

dpKnapsack(*items, n, C*):

$S[0][j] = 0$

$S[i][0] = 10000$

$S[0][0] = 0$

For each $i = 1\ to\ C$:

For each $j = 1\ to\ n$:

$if\ item.c > i$:

$S[i][j] = min(S[0][j-1] + item.w, S[i][j-1])$

$else$:

$S[i][j] = min(S[i - item.c][j-1] + item.w, S[i][j-1])$

Return $S[C][n]$

Problem 4

Algorithm Description

For this problem, I implemented a dynamic programming algorithm to compute the minimal cost of for matrix chain multiplication, and the corresponding parenthesizing format. There exists a separate array set to store the parenthesizing index while populating the optimal minimal cost array, so that by tracing back the index array, a recursion function can be employed to print out the parenthesized matrixes' formation. Here is the heart of the solution below:

$S[L][R] = the\ minimal\ number\ of\ steps\ required\ to\ multiply\ matrixes\ from\ L^{th}\ to\ the\ R^{th}.$

$P[L][R] = the\ parenthesized\ index\ under\ current\ values\ of\ L\ and\ R.$

$a\ is\ the\ input\ array\ to\ be\ parenthesized, a_m\ is\ the\ m^{th}\ element\ of\ array\ a.$

$k\ is\ the\ index\ of\ current\ parenthesizing\ index.$

$$S[L][R] = \begin{cases} 0 & L = R \\ \min(S[L][R], S[L][k] + S[k+1][R]) + a_{L-1} \times a_k \times a_R & L < R \\ null & L > R \end{cases}$$

For all *L* and *R* pairs, there exists (R – L) number of possible *differences(d)*. Since the purpose is to find the minimal cost from all possible pairs, so the algorithm should start with the minimal *difference*, which in this case is *d = 1*, so that to incrementally compute the last pair, which is *L = 1, R = R*.

Running time estimate

For *difference* has *(R – L)*'s values, so it is upper bounded by *n*. For *L*, it has *(n - d)* running time, so it is upper bounded by n. For *parenthesizing index*, *k* has *(R - L)* running time, so it is as well upper bounded by *n*. The time complexity of minimal cost of Matrix Chain Multiplication is $O(n^3)$.

The parenthesized matrix's formation is running recursively. The running time determined by the length of array *a*, so it is bounded by *n*. The time complexity for formation is $O(n)$.

Pseudocode

matrixChainMultiplication(a):

for each *i = 1* to *n: S[i][i] = 0*

for each *d = 1* to *n – 1*:

for each *L = 1* to *n – d:*

R = L + d

S[L][R] = ∞

for each *k = L to R:*

temp = *S[L][k] + S[k + 1][R] +* $a_{L-1} * a_k * a_R$

if (temp < S[L][R]) S[L][R] = temp and P[L][R] = k

print S[1][n]

print printMultiplication(P, L, R)

printMultiplication(P, i, j):

if i = j:

"A" + i

if i != j:

"(" + *printMultiplication(P, i, k) + "x" + printMultiplication(P, k + 1, j) +* ")"

Problem 5

Algorithm Description

For this problem, I implemented a dynamic programming algorithm to find the minimum possible length of triangulation of the given polygon. The heart of the solution is as follows:

$S[i][j]$
$= \ the\ minimal\ possible\ length\ of\ triangulation\ of\ the\ polygon\ that\ has\ vertices\ started\ from\ i\ to\ j.$

$$S[i][j] = \begin{cases} 0 & if\ i = j \\ 0 & if\ i+1 = j \\ \min\big(S[i][j], S[i][k] + S[k][j] + distance(V[i], V[j])\big) & if\ i+1 < j \\ S[i][k] + S[k][j] & if\ d = n-1 \end{cases}$$

If *i = j*, it is a dot not a polygon, so $S[i][j] = 0$. If *i + 1 = j*, it is an edge not a polygon either, $S[i][j] = 0$. If *i + 2 <= j*, it is at least a triangle so $S[i][j]$ would be the sum of sub-polygon's optimal minimum cost with its current length of edge D_{ij}. The difference of *i and j* is represented by variable of *d*, where its value ranges from *2 to i – 1*.

The algorithm is similar to that of in matrix china multiplication. The algorithm employs an extra class to store the x and y coordinates of each vertices.

Running time estimate

For *difference* has *(i - 1 – j)*'s values, so it is upper bounded by *n*. For *i*, it has *(n - d)* running time, so it is upper bounded by n. For *segmentation index*, *k* has *(i - j)* running time, so it is as well upper bounded by *n*. The time complexity of minimal cost of Triangulation is $\mathrm{O}(n^3)$.

Pseudocode

triangulation(a):

for each *i = 1* to *n: S[i][i] = 0*

for each *j = i + 1* to *n and i = 1 to n: S[i][j] = 0*

for each *d = 2* to *n – 1*:

for each *L = 1* to *n – d:*

j = i + d

S[i][j] = ∞

for each *k = i + 1 to j - 1:*

temp = *S[i][k] + S[k][j] +* $distance(i, j)$

if (temp < S[i][j]) S[i][j] = temp

print S[1][n] rounded in 4 digits decimal