Homework 7

Tushar Jain, N12753339
CSCI-GA 1170-003 - Fundamental Algorithms

October 20, 2017

Problem 1.a. Show, by means of a counterexample, that the following greedy strategy does not always determine an optimal way to cut rods. Define the density of a rod of length i to be p_i/i, that is, its value per inch. The greedy strategy for a rod of length n cuts off a first piece of length i, where $1 \leq i \leq n$, having maximum density. It then continues by applying the greedy strategy to the remaining piece of length $n-i$.

Solution.

Let prices be as following:

$$p_1 = 1,$$
$$p_2 = 6,$$
$$p_3 = 10,$$
$$p_4 = 0$$

Then the densities would be as follows:

$$p_1/1 = 1,$$
$$p_2/2 = 3,$$
$$p_3/3 = 3.33$$

For $n = 4$, the greedy solution would be 3 and 1 with total value of $1 + 10 = 11$.
However, the dynamic programming solution would be 2 and 2 with a total value of $6 + 6 = 12$. □

Problem 1.b. As stated, in dynamic programming we first solve the subproblems and then choose which of them to use in an optimal solution to the problem. Professor Betsy claims that we do not always need to solve all the subproblems in order to find an optimal solution. She suggests that we can find an optimal solution to the matrix-chain multiplication problem by always choosing the matrix A_k at which to split the subproduct $A_i A_{i+1} A_j$ (by selecting k to minimize the quantity $p_{i-1}p_k p_j$) before solving the subproblems. Find an instance of the matrix-chain multiplication problem for which this greedy approach yields a suboptimal solution.

Solution.

Assumption: Let us assume to be given the following matrices:

$$A_1, A_2, A_3, and A_4$$

with the following dimensions such that:

$$p_0 = 1000, p_1 = 100, p_2 = 20, p_3 = 10, p_4 = 1000$$

Then $p_0 p_k p_4$ is minimized when k = 3, so we need to solve the subproblem of multiplying $A_1 A_2 A_3$, and also A_4 which is solved automatically. By her algorithm, this is solved by splitting at k = 2.

Thus, the full parenthesization is $(((A_1A_2)A_3)A_4)$. This requires $1000{\cdot}100{\cdot}20{\cdot}1000{\cdot}20{\cdot}10{+}1000{\cdot}10{\cdot}1000 = 12,200,000$ scalar multiplications.

On the other hand, suppose we had fully parenthesized the matrices to multiply as $((A_1(A_2A_3))A_4)$. Then we would only require $100 \cdot 20 \cdot 10 + 1000 \cdot 100 \cdot 10 \cdot 1000 \cdot 10 \cdot 1000 = 11,020,000$ scalar multiplications, which is fewer than Professor Betsys method. Therefore her greedy approach yields a suboptimal solution. $\square$

Problem 2a. Given a tree (not necessarily binary), show how to calculate the height of the subtree rooted at every node. Your algorithm should take time $O(n)$ where n is the number of nodes in the tree.

Solution:
Assumption: We assume the height of leaf node to be 1 and the node structure to have an attribute called height.

Pseudo-Code:

```
def height(node):
    if node is None:
        return 0
    else:
        max_height = max([height(child) for child in node.children])
        node.height = max_height + 1
        return node.height
```

Running Time: As we are traversing each node in the tree, the running time is $O(n)$.

$\square$

Problem 2b. Suppose that we are given a directed acyclic graph $G = (V, E)$ with real-valued edge weights and two distinguished vertices s and t (see CLRS Sect. B.4 for definitions). You can assume that each node u stores an array *edges*, that contains pairs (v, k) that encodes an edge from u to v with weight k. Describe a dynamic-programming approach for finding a longest weighted simple path from s to t. What does the subproblem graph look like? What is the efficiency of your algorithm?

Solution:
This can be solved using topological sort or using memoization technique. Here I shall describe the later. The function *Longest* get the longest weighted path from s to t using dynammic programming recursion

$$LONGEST(G, s, t) = weight_{s,s'} + max_{s'}(LONGEST(G_{V-s}, s', t)$$

Assumption: The given graph is DAG.
Pseudo-Code:

```
memo = {}
def get_longest(from_node, to_node):
    if (from_node,to_node) in memo:
        return memo[(from_node,to_node)]
    if from_node == to_node:
        memo[(from_node,to_node)] = 0
        return memo[(from_node,to_node)]
    best = 0
    for from_node2, weight in G[from_node].edges:
        best = max(best, get_longest(from_node2, from_node) + weight)
    memo[to_node] = best
    return best
```

The subproblem graph is a subset of the original graph without the all vertices from the source to the immediate previous vertice (i.e., previous subproblem source). Thus, the search space keeps on reducing.

Running Time: Assuming n to be the number of vertices. The running time is $O(n^2)$ as at most $(n-1)+(n-2)+...+1=(n-1)n/2$ times the max would be calculated using the for loop in line 9. □

Problem 3. A palindrome is a nonempty string over some alphabet that reads the same forward and backward. Examples of palindromes are all strings of length 1, `civic`, `racecar`, and `aibohphobia` (fear of palindromes).
Give an efficient algorithm to find the longest palindrome that is a subsequence of a given input string. For example, given the input `character`, your algorithm should return `carac`. What is the running time of your algorithm?

Solution:
By taking the LCS of the original and the reversed string, we will be able to get the longest palindrome subsequence.

Pseudo-Code:

```
def lcs(X, Y):
    m = len(X)
    n = len(Y)
    L = [[0 for x in range(n+1)] for x in range(m+1)]

    for i in range(m+1):
        for j in range(n+1):
            if i == 0 or j == 0:
                L[i][j] = 0
            elif X[i-1] == Y[j-1]:
                L[i][j] = L[i-1][j-1] + 1
            else:
                L[i][j] = max(L[i-1][j], L[i][j-1])

    index = L[m][n]

    lcs = [""] * (index)

    i = m
    j = n
    while i > 0 and j > 0:
        if X[i-1] == Y[j-1]:
            lcs[index-1] = X[i-1]
            i-=1
            j-=1
            index-=1

        elif L[i-1][j] > L[i][j-1]:
            i-=1
        else:
            j-=1

    return "".join(lcs)

def lps(s):
    return lcs(s, s[::-1])
```

Running Time: Since lcs takes $O(n^2)$ time and reverse takes linear time, the total running time of LPS is $O(n^2)$. □

Problem 4. Suppose you are given an array $A[1, ..., n]$ of numbers, which may be positive, negative, or zero.

Problem 4a. Let $S_{i,j}$ denote $A[i]+A[i+1]++A[j]$. Use dynamic programming to give an $O(n^2)$ algorithm to compute $S_{i,j}$ for all $1 \leq i \leq j \leq n$, and hence compute $max_{i,j}S_{i,j}$.

Solution:

Pseudo-Code:

```
 1  def Sij(A):
 2      S = [[0 for x in range(n)] for x in range(n)]
 3      best = A[1]
 4      for i in range(n):
 5          S[i][i] = A[i]
 6          for j in range(i+1, n):
 7              S[i][j] = A[i][j-1] + A[j]
 8              if S[i][j] > best:
 9                  best = S[i][j]
10      return S, best
```

Here *best* is the $max_{i,j}S_{i,j}$.
Running Time:
Looking at for loops starting at lines 2, 3 and 5, we can clearly see the total running time would be $O(n^2)$ as all other lines have $O(1)$ running time individually. □

Problem 4b. Let $L[j]$ denotes $max_i S_{i,j}$. Give a recurrence relation for $L[j]$ in terms of $L[1, ..., j1]$. Use your recurrence relation to give an $O(n)$ time dynamic programming algorithm to compute $L[1...n]$, and hence compute $max_{i,j}Si, j$.

Solution:

Recurrence relation:

For $j > 1$,

$$L[j] = max(L[j-1] + A[j], A[j])$$

And for $j = 1$, $L[j] = A[j]$.
Pseudo-Code:

```
1  def max_Sij(A):
2      n = len(A)
3      L = [0] *(n)
4
5      for i in range(2, n):
6          L[i] = max(A[i], A[i] + L[i-1])
7
8      return max(L)
```

Running Time: Looking at for loops starting at line 5 and *max* at line 8, we can clearly see the total running time would be $O(n)$ as all other lines have $O(1)$ running time individually. □

Problem 4c. Consider the naive recursive algorithms (without memorization, like Cut-Rod from class) based on the recurrence relations to compute the answers to part(a) and part(b). Will both running times stay at $O(n^2)$ and $O(n)$, respectively, only one of them (which one?), or none?

Solution:

None of the previous part answers will remain at the same running time.

For part (a), we would require 2 for loops for calculating $S_{i,j}$ without memoization and need $O(n)$ time for sum and thus, would require $O(n^3)$ running time in total.

For part (b), we would require 2 for loops for checking all possible A's sub-arrays and thus, would require $O(n^2)$ running time in total □

Problem 4d. Suggest appropriate modifications to your algorithm in part (b) to give an $O(n)$ algorithm to compute $max_{i,j} P_{i,j}$, where $P_{i,j} = A[i] \cdot A[i+1] \cdots A[j]$. Assume that multiplication of any two numbers takes O(1) time.

Solution:

Pseudo-Code:

```
def max_Sij(A):
    n = len(A)
    bestmax, total_max, bestmin = 1
    for i in range(n):
        if A[i]> 0:
            bestmax = bestmax * A[i]
            bestmin = bestmin * A[i]
        elif A[i] == 0:
            bestmax = bestmin = 1
        else:
            bestmax, bestmin = max(bestmin*A[i], 1), bestmax * A[i]

        total_max = max(total_max,bestmax)

    return total_max
```

Running Time: Assuming 1 positive number in A, the running time will be $O(n)$ and if all the numbers are negative, the max product will be product of all. □

Problem 5. Let $X[1...m]$ and $Y[1...n]$ be two given arrays. A common supersequence of X and Y is an array $Z[1...k]$ such that X and Y are both subsequences of $Z[1...k]$. Your goal is to find the shortest common super-sequence (SCS) Z of X and Y , solving the following sub-problems.

Problem 5a. First, concentrate on finding only the length k of Z. Proceeding similarly to the longest common subsequence problem, define the appropriate array $M[0...m, 0...n]$ (in English), and write the key recurrence equation to recursively compute the values $M[i, j]$ depending on some relation between $X[i] and Y[j]$. Do not forget to explicitly write the base cases $M[0, j] and M[i, 0]$, where $1 \leq i \leq m, 1 \leq j \leq n$.

Solution:

Following is the require recurrence equation:

$$
\begin{aligned}
M[i,j] &= j && \text{(if i=0)}\\
M[i,j] &= i && \text{(if j=0)}\\
M[i,j] &= 1 + Z[i-1, j-1], && \text{(if X[i] = Y[j])}\\
M[i,j] &= 1 + min(M[i-1,j], M[i,j-1]), && \text{(else)}
\end{aligned}
$$

□

Problem 5b. Translate this recurrence equation into an explicit bottom-up O(mn) time algorithm that computes the length of the shortest common supersequence of X and Y .

Solution: **Pseudo-Code:**

```
1   def SCS(X,Y):
2       m, n = len(X), len(Y)
3       M = [[0 for x in range(n+1)] for x in range(m+1)]
4
5       for i in range(m+1):
6           for j in range(n+1):
7               if i == 0:
8                   M[i][j] = j
9               elif j ==0:
10                  M[i][j] = i
11              elif X[i-1] == Y[j-1]:
12                  M[i][j] = 1 + M[i-1][j-1]
13              else:
14                  M[i][j] = 1 + min(M[i][j-1], M[i-1][j])
15
16      return M[m][n]
```

Running Time: Looking at for loops starting at lines 11 and 12, we can clearly see the total running time would be $O(mn)$ as all other lines have $O(1)$ running time individually.

□

Problem 5c. Find the SCS of X = BARRACUDA and Y = ABRACADABRA. (Notice, you need to find the actual SCS, not only its length.)

Solution:
Here there are many possible SCS solutions. I'm presenting one of them here.

In the case of $M[i-1][j] = M[i][j-1]$ (and $X[i-1] \neq Y[j-1]$), choosing to go up in the table:

	0	A	B	R	A	C	A	D	A	B	R	A
0	0	**1**	2	3	4	5	6	7	8	9	10	11
B	1	2	**2**	3	4	5	6	7	8	9	10	11
A	2	2	**3**	4	4	5	6	7	8	9	10	11
R	3	3	**4**	4	5	6	7	8	9	10	10	11
R	4	4	5	**5**	6	7	8	9	10	11	11	12
A	5	5	6	6	**6**	7	8	9	10	11	12	12
C	6	6	7	7	7	**7**	**8**	9	10	11	12	13
U	7	7	8	8	8	8	**9**	10	11	12	13	14
D	8	8	9	9	9	9	10	**10**	**11**	**12**	**13**	14
A	9	9	10	10	10	10	10	11	11	12	13	**14**

Here the SCS is: ”ABARRACAUDABRA”.

□

Problem 5d. Show that the length k of the array Z computed in part (a) satisfies the equation $k = m+n-l$, where l is the length of the longest common subsequence of X and Y.

Hint: Use the recurrence equation in part (a), then combine it with a similar recurrence equation for the LCS, and then use induction. There the following identity is very handy: $min(a, b) + max(a, b) = a + b$.

Solution:

To Prove: $|SCS(X, Y)| = |X| + |Y| - |LCS(X, Y)|$

Proof By Contradiction:
Let us assume that the RHS does not provide us to the length of the SCS as a shorter subsequence is possible,(or consider because the SCS algorithm is not correct).
$\implies$ The length of SCS is less than that given on the RHS. Let us assume the length of correct SCS' to be:

$$\begin{aligned}|SCS'(X, Y)| &= |X| + |Y| - |LCS(X, Y)| - c\\ &= |X| + |Y| - (|LCS(X, Y)| + c)\\ &\text{where } c \in \mathbb{N}\end{aligned}$$

But this shortest super-sequence has both X and Y as subsequences. Judging by the sizes of the string that would mean that X and Y intersect in (—LCS(X,Y)— + c) places. That is there's a LCS of size (—LCS(X,Y)— + c) which is a contradiction to the LCS algorithm and definition.

Thus, our assumption is incorrect and the length of SCS equals length of LCS subtracted from sum of individual sequences.

Proof By Induction:
Assuming: m=—X—, n = —Y—

Base Case: m+n = 0, i.e. m=n=0
$\implies$ Both X and Y are empty. Then —SCS(X,Y)—=—LCS(X,Y)—= 0. Thus, the claim holds.

Assuming inductive hypothesis that claim is true for all $m + n < p$ and considering the following cases when $m + n = p > 0$: Suppose X = X + X[m] if m ¿ 0 and Y = Y + Y[n] if n ¿ 0

- Case #1: $m \geq 0, n = 0$ As —Y— = 0, —LCS— = 0 and the shortest common super-sequence (SCS) would simply be of length, —SCS— = —X— = n.
  $\implies$ LHS = n and RHS = n + 0 - 0 = n.
  $\implies$ LHS = RHS.
- Case #2: $m = 0, n > 0$. Similar to previous case, the claim holds.
- Case #3: $m > 0, n > 0$ and $X[m] = Y[n]$. Now:

$$\begin{aligned}|LCS(X, Y)| &= |LCS(X', Y')| + 1\\ |SCS(X, Y)| &= |SCS(X', Y')| + 1\end{aligned}$$

By inductive hypothesis, we have $|SCS(X', Y')| = |X'| + |Y'| - |LCS(X', Y')|$.

Using the above 3 equations we have,

$$\begin{aligned}
|SCS(X', Y')| &= |X'| + |Y'| - |LCS(X', Y')| \\
|SCS(X, Y)| - 1 &= (|X| - 1) + (|Y| - 1) - (|LCS(X, Y)| - 1) \\
|SCS(X, Y)| - 1 &= |X| + |Y| - |LCS(X, Y)| - 1 \\
|SCS(X, Y)| &= |X| + |Y| - |LCS(X, Y)|
\end{aligned}$$

Thus, the claim holds.

- Case #4: $m > 0, n > 0$ and $X[m] \neq Y[n]$ and $M[m-1][n] > M[m][n-1]$. Now:

  $$\begin{aligned}
  |LCS(X, Y)| &= |LCS(X, Y')| \\
  |SCS(X, Y)| &= |SCS(X, Y')| + 1
  \end{aligned}$$

  By inductive hypothesis, we have $|SCS(X, Y')| = |X| + |Y'| - |LCS(X, Y')|$.

  Using the above 3 equations, in a similar manner to case 3, we get,

  $$|SCS(X, Y)| = |X| + |Y| - |LCS(X, Y)|$$

  Thus, the claim holds.

- Case #5: $m > 0, n > 0$ and $X[m] \neq Y[n]$ and $M[m-1][n] \leq M[m][n-1]$. Now:

  $$\begin{aligned}
  |LCS(X, Y)| &= |LCS(X', Y)| \\
  |SCS(X, Y)| &= |SCS(X', Y)| + 1
  \end{aligned}$$

  By inductive hypothesis, we have $|SCS(X', Y)| = |X'| + |Y| - |LCS(X', Y)|$.

  Using the above 3 equations, in a similar manner to case 3, we get,

  $$|SCS(X, Y)| = |X| + |Y| - |LCS(X, Y)|$$

  Thus, the claim holds.

As the claim holds true for all the above cases, the claim is true for all m+n.

□