ECS 122A – Algorithm & Analysis
Homework 05

Due: Sunday, November 07, 2021, 11:59pm PT

Note:

- Identify the corresponding pages for each question according to the outline on Gradescope.
- If you handwrite your solutions, make sure they are clear and readable.

Question 1: Suboptimality Property for LCS (15 points)

Prove the suboptimality property for LCS's dynamic programming algorithm we defined in class.
(Hint: Prove the different cases individually.)

Answer:

Let $X_m = \langle x_1, x_2, ..., x_m \rangle$ and $Y_n = \langle y_1, y_2, ..., y_m \rangle$ be the two input sequences, and $OPT = \langle o_1, o_2, ..., o_k \rangle$ be a LCS of X_m and Y_n.

The proof has three cases depending on x_m, y_n, and o_k.

- Case 1: To prove that if $x_m = y_n$, then $B = \langle o_1, o_2, ..., o_{k-1} \rangle$ is a LCS of X_{m-1} and Y_{n-1}.

  Proof by contradiction:

  Assume that B is not a LCS of X_{m-1} and Y_{n-1}.
  Then there exists a sequence B' which is a common sequence of X_{m-1} and Y_{n-1}.
  $B' + \langle o_k \rangle$ is a LCS of X_m and Y_n and $|B' + \langle o_k \rangle| = |B'| + 1 > |B| + 1 = |OPT|$.
  This is a contradiction because no common sequence of X_m and Y_n can be longer than OPT.

- Case 2: To prove that if $x_m \neq y_n$ and $o_k \neq x_m$, then OPT is a LCS of X_{m-1} and Y_n.

  Proof by contradiction:

  Assume that OPT is not a LCS of X_{m-1} and Y_n.
  Then there exists a sequence B which is a common sequence of X_{m-1} and Y_n.
  B is a common sequence of X_m and Y_n and $|B| > |OPT|$.
  This is a contradiction because no common sequence of X_m and Y_n can be longer than OPT.

- Case 3: To prove that if $x_m \neq y_n$ and $o_k \neq y_n$, then OPT is a LCS of X_m and Y_{n-1}.

  Proof by contradiction:

Assume that OPT is not a LCS of X_m and Y_{n-1}.
Then there exists a sequence B which is a common sequence of X_m and Y_{n-1}.
B is a common sequence of X_m and Y_n and $|B| > |OPT|$.
This is a contradiction because no common sequence of X_m and Y_n can be longer than OPT.

Question 2: Maximum Subarray (50 points total)

We have defined the maximum-subarray problem and solved it using divide and conquer. In fact, the problem can be solved using greedy and dynamic programming. Define a greedy algorithm and a dynamic-programming algorithm for this problem.

Specifically, given an input which is an array $A[0..n-1]$ of n numbers, output 1) indices i and j such that the subarray $A[i..j]$ has the greatest sum of any nonempty subarray of A; and 2) the sum of the values in $A[i..j]$.

1. (25 points) Greedy algorithm:

   (a) Define a greedy choice for the problem

   Answer:

   Keep a local maximum sum $local$ and a global maximum sum $global$.
   For each number in the array, we choose between two options:

   - if $local <= 0$, we reset $local$ to the current number;
   - otherwise, we add the number to $local$.

   Then we update $global$ if $local$ is larger than $global$.

   Note:

   - *You do not have to specify how to calculate/record the indices in the greedy choice.*
   - *Another greedy choice is we either add the number to local or rest local to the number, depending on which one is larger. For homework 06, please prove the one above.*

   (b) Write the pseudo-code for a greedy algorithm based on your greedy choice

   Answer:

```
maxSubarray(A[0..n-1]):
    local = A[0]
    local_left = 0
    local_right = 0

    global = A[0]
    left = 0
    right = 0

    for i = 1 to n-1:
        if local <= 0:
            local = A[i]
            local_left = i
            local_right = i
        else:
            local = local + A[i]
            local_right = i

        if local > global:
            global = local
            left = local_left
```

```
            right = local_right
    return (left, right, global)
```

Note: We assume the input array is of size at least 1. You may add a base case such that if A is empty, return an invalid output.

(c) Analyze the time complexity of your greedy algorithm

Answer:

$T(n)$ is $\Theta(n)$ for the for loop and we go through each number in the input array once.

2. (25 points) Dynamic programming:

(a) Define the recursive formula for the problem

Answer:

$$maxSum[i] = \begin{cases} A[i] & \text{if } maxSum[i-1] <= 0 \\ maxSum[i-1] + A[i] & \text{otherwise} \end{cases}$$

(b) Write the pseudo-code for a dynamic-programming algorithm based on your recursive formula

Answer:

```
maxSubarray(A[0..n-1]):
    maxSum, left, right: arrays with size n
    maxSum[0] = A[0]
    left[0] = 0
    right[0] = 0

    for i = 1 to n-1:
        if maxSum[i-1] <= 0:
            maxSum[i] = A[i]
            left[i] = i
            right[i] = i
        else:
            maxSum[i] = maxSum[i-1] + A[i]
            left[i] = left[i-1]
            right[i] = i

    idx = 0
    for i = 1 to n-1:
        if maxSum[idx] < maxSum[i]:
            idx = i

    return (left[idx], right[idx], maxSum[idx])
```

(c) Analyze the time complexity of your dynamic-programming algorithm

Answer:

$T(n)$ is $\Theta(n)$ for the for loop.

(Note: Subarrays are contiguous.)

Question 3: Print Neatly (35 points)

Consider the problem of neatly printing a paragraph with a monospaced font (all characters having the same width) on a printer. The input text is a sequence of n words of lengths l_1, l_2, ..., l_n, measured in characters. We want to print this paragraph **neatly** on a number of lines that hold a maximum of M characters each.

Our criterion of "neatness" is as follows. If a given line contains words i through j, where $i \leq j$, and we leave exactly one space between words, the number of extra space characters at the end of the line is $M - j + i - \sum_{k=i}^{j} l_k$, which must be nonnegative so that the words fit on the line. We wish to minimize the sum, over all lines except the last, of the cubes of the numbers of extra space characters at the ends of lines. We call the sum the *cost* of printing neatly.

Inputs:

- an array $l = [l_1, l_2, ..., l_n]$ such that l_i is the number of characters in the ith word
- a non-negative integer M for the maximum number of characters can be fit in each line

Output:

- an integer C for the minimum cost of printing the words neatly
- an array $P = [p_1, p_2, ...,]$ where p_j is the index of the last word printed on line j

1. (15 points) Define the recursive formula for the problem

   Answer:

   Let $s[i, j]$ be the number of spaces at the end of a line when the line contains words $i+1, i+2, ..., j$.

$$s[i, j] = M - j + i - 1 - \sum_{k=i+1}^{j} l_k$$

   But not all of these words can be put on the same line. We define the cost of printing a line with words $i+1, ..., j$ as

$$c[i, j] = \begin{cases} \infty & \text{if } s[i, j] < 0 \\ 0 & \text{if } j = n \text{ and } s[i, j] \geq 0 \\ s[i, j]^3 & \text{otherwise} \end{cases}$$

   The recursive formula for printing words $1, .., j$ is

$$C[j] = \begin{cases} 0 & \text{if } j = 0 \\ \min_{0 \leq i < j}(C[i] + c[i, j]) & \text{otherwise} \end{cases}$$

   For a more detailed explanation: `https://www.cs.helsinki.fi/webfm_send/1449`

2. (15 points) Write the pseudo-code for a dynamic-programming algorithm based on your recursive formula

   Answer:

```
printNeatly(L[1..n], M):
    C, last_word: two arrays with size (n+1)
    C[0] = 0
    last_word[0] = 0

    for j = 1 to n:
        // find the minimum of all i's
        m = infinity
        word_idx = 0
        for i = 0 to j-1:
            // compute s[i,j]
            s = M - j + i - 1 - sum(l[i+1], ..., l[j])
            // compute c[i,j]
            c = infinity
            if s >= 0:
                if j == n:
                    c = 0
                else:
                    c = s

            if m > C[i] + c:
                m = C[i] + c
                word_idx = i
        C[j] = m
        last_word[j] = word_idx    // the last word printed on previous line is word_idx

    W = traceback(last_word)
    return (C[n], W)


traceback(last_word[0..n]):
    W = [n]
    i = n
    while i > 0:
        W.prepend(last_word[i])
        i = last_word[i]
    return W
```

3. (5 points) Analyze the time complexity of your dynamic-programming algorithm

   Answer:

   $T(n)$ is $\Theta(n^2)$ for the nested for loop.