Data Structure and Algorithm II
Homework #2
Due: 13pm, Monday, October 31, 2011

=== Homework submission instructions ===

- Submit the answers for writing problems (including your programming report) through the CEIBA system (electronic copy) or to the TA in R432 (hard copy). Please write down your name and school ID in the header of your documents. You also need to submit your programming assignment (problem 1) to the Judgegirl System(`http://katrina.csie.ntu.edu.tw/judgegirl/`).
- Each student may only choose to submit the homework in one way; either all as hard copies or all through CEIBA except the programming assignment. If you submit your homework partially in one way and partially in the other way, you might only get the score of the part submitted as hard copies or the part submitted through CEIBA (the part that the TA chooses).
- If you choose to submit the answers of the writing problems through CEIBA, please combine the answers of all writing problems into only one file in the doc/docx or pdf format, with the file name in the format of "hw2_[student ID].{pdf,docx,doc}" (e.g. "hw2_b99902010.pdf"); otherwise, you might only get the score of one of the files (the one that the TA chooses).
- For each problem, please list your references (they can be the names of the classmates you discussed the problem with, the URL of the information you found on the Internet, or the names of the books you read). The TA can deduct up to 100% of the score assigned to the problems where you don't list your references.

Problem 1. (30%) It is always very nice to have little brothers or sisters. You can tease them, lock them in the bathroom or put red hot chili in their sandwiches. But there is also a time when all meanness comes back!

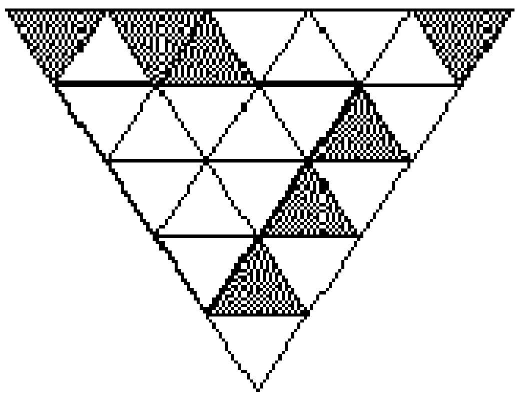


Figure 1: Triangles

As you know, in one month it is Christmas and this year you are honored to make the big star that will be stuck on the top of the Christmas tree. But when you get the triangle-patterned silver paper you realize that there are many holes in it. Your little sister has already cut out smaller triangles for the normal Christmas stars. Your only chance is to find an algorithm that tells you for each piece of silver paper the size of the largest remaining triangle.

Given a triangle structure with white and black fields inside you must find the largest triangle area of white fields, as shown in Figure 2(a).

Input:

The input file contains several triangle descriptions. The first line of each description contains an integer n ($1 \leq n \leq 100$), which gives the height of the triangle. The next n lines contain characters of the set {space, #, -} representing the rows of the triangle, where '#' is a black and '-' a white field. The spaces are used only to keep the triangle shape in the input by padding at the left end of the lines. (Compare with the sample input. The first test case corresponds to the figure.)

For each triangle, the number of the characters '#' and '-' per line is odd and decreases from 2n - 1 down to 1. The input is terminated by a description starting with n = 0.

There are at most 1000 triangles in each test case. Time limit for this problem is 3 seconds.

Output:

For each triangle in the input, first output the number of the triangle, as shown in the sample output. Then print the line “The largest triangle area is a.”, where a is the number of fields inside the largest triangle that consists only of white fields. Note that the largest triangle can have its point at the top, as in the second case of the sample input. Output a blank line after each test case.

Sample Input:

```
5
#-##----#
 -----#-
  ---#-
   -#-
    -
4
#-#-#--
 #---#
  ##-
   -
0
```

Sample Output:

```
Triangle #1
The largest triangle area is 9.

Triangle #2
The largest triangle area is 4.
```

Write a program to solve this problem using dynamic programming. Please also submit a report in which you give a clear description of your algorithm. (20 points for 10 test cases and 10 points for the report.)

Sol: Without lost of generality, we consider the upward triangle in figure 2(b). From top to bottom, we denote each triangle $t_{i,j}$, where i is the row and j is the column of the triangle, and each row i contains $2i-1$ columns in figure 2(b). Furthermore, we denote $T_{i,j}$ is the set of the triangles which top-most triangle is $t_{i,j}$, and denoting $MaxLen(T_{i,j})$ is the length of maximum empty triangle area of the set $T_{i,j}$. Now we can start to think about how to find the maximum empty triangle area in this problem.
In order to decide the value of $MaxLen(T_{i,j})$, we must consider three conditions. Firstly, the condition 1 is to check whether the $t_{i,j}$ is empty. If not, the $MaxLen(T_{i,j})$ is 0. If condition 1 is true, condition 2 will check whether $t_{i+1,j+1}$ is empty. If not, the $MaxLen(T_{i,j})$ is 1. In condition 3, suppose both condition 1 and 2 are true, the value of $MaxLen(T_{i,j})$ is the sum of 1 and the minimum value of $MaxLen(T_{i+1j})$ and $MaxLen(T_{i+1,j+2})$. We now briefly explain the corectness of condition 3. In figure 2(b), assume that the value of $v_{2,1} = MaxLen(T_{2,1})$ and $v_{2,3} = MaxLen(T_{2,3})$ are found, we can form an empty triangle from $1st$ row to $1 + min(v_{2,1}, v_{2,3})'th$ row. Choosing the minimum value $v_{min} = min(v_{2,1}, v_{2,3})$ ensures that there are all empty triangles from $1st$ row to $(1 + v_{min})$'s row. Thus, we can determine the maximum empty triangle area of $T_{1,1}$. The value of $v_{i,j}$ would be determined recursively by the equation 1 if $t_{i,j}$ is an upward triangle.

$$v_{i,j} = \begin{cases} 0, & \text{if } t_{i,j} \text{ is not empty} \\ 1, & \text{if } t_{i,j} \text{ is empty and } t_{i+1,j+1} \text{ is not empty} \\ 1 + min(v_{i+1,j}, v_{i+1,j+2}), & \text{if } t_{i,j} \text{ and } t_{i+1,j+1} \text{ are both empty} \end{cases} \tag{1}$$

Algorithm 1 provides the basic idea of finding the maximum empty area using dynamic programming. In algorithm 1, we need to consider both upward and downward cases. There are subtle differences between finding upward and downward empty triangles, but actually they are the same method. The upward triangle would be search from bottom to top and the downward traingle is searched from top to bottom. This algorithm is linear to the total number of triangles. During programming, be careful in handling the boundary value case. You may find hw2 sample code from the url [1]

[1] `http://katrina.csie.ntu.edu.tw/hw2/hw2.cpp`

Algorithm 1 $maxArea$ = FindMaxTri($t[maxRow][MaxCols]$)

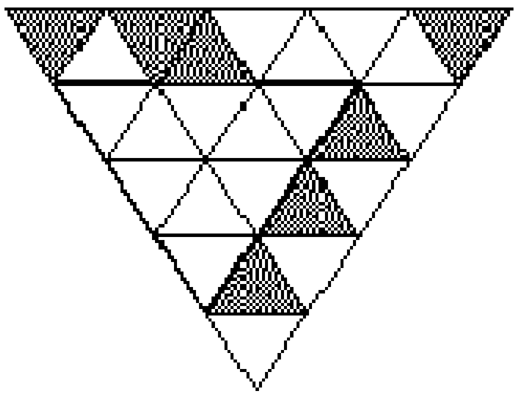
```
1: Initialize $v[maxRow][MaxCols]$
2: for all upward triangle $t_{i,j}$ from bottom to top do
3:     if $t_{i,j}$ is not empty then
4:         $v_{i,j} = 0$
5:     else if $t_{i+1,j+1}$ is not empty then
6:         $v_{i,j} = 1$
7:     else
8:         $v_{i,j} = 1 + min(v_{i+1,j}, v_{i+1,j+2})$
9: for all downward triangle $t_{i,j}$ from top to bottom do
10:    if $t_{i,j}$ is not empty then
11:        $v_{i,j} = 0$
12:    else if $t_{i-1,j+1}$ is not empty then
13:        $v_{i,j} = 1$
14:    else
15:        $v_{i,j} = 1 + min(v_{i-1,j}, v_{i-1,j+2})$
16: $maxLen \leftarrow MaxVal(v)$
17: return $maxLen * maxLen$
```

Problem 2. Solve the following problems on the textbook:

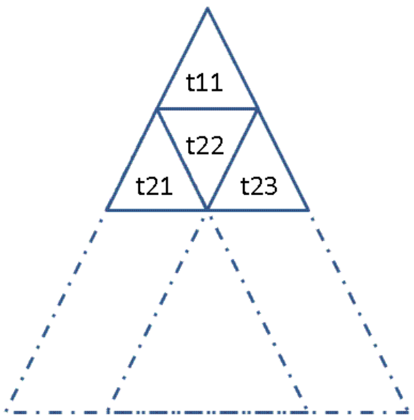
1. (5%) 15.2-1 on p.378

   Sol: Figure 3 shows the result computed by the algorithm MATRIX-CHAIN-ORDER on p.375 of the textbook for the input sequence of dimensions $\langle 5, 10, 3, 12, 5, 50, 6 \rangle$. Therefore by calling PRINT-OPTIMAL-PARENS(s,1,6) on p.377 we get the optimal parenthesization $((A_1A_2)((A_3A_4)(A_5A_6)))$.

2. (5%) 15.2-3 on p.378



(a) Triangles



(b) Upward triangle $t_{i,j}$ is afftected by $T_{i+1,j}$, $t_{i+1,j+1}$, and $T_{i+1,j+2}$

Figure 2: Triangles of problem 1

Sol: Suppose that $P(i) \geq c2^i$, for all $i < n$. Then

$$\begin{aligned} P(n) &= \sum_{k=1}^{n-1} P(k)P(n-k) \\ &\geq \sum_{k=1}^{n-1} (c2^k)(c2^{n-k}) \\ &= c^2 \sum_{k=1}^{n-1} 2^n \\ &= c^2(n-1)2^n \\ &\geq c2^n, \end{aligned}$$

for $n \geq 2$. Hence, $P(n) = \Omega(2^n)$.

3. (10%) 15.4-5 on p.397

   Sol: Let x be the input sequence and y be a sequence of elements in x sorted in ascending order. It's easy to verify that z is a longest monotonically increasing subsequence of x if and only if z is a longest common subsequence of x and y. Therefore, the result of computing the longest common subsequence by the algorithm LCS-LENGTH(x, y) and PRINT-LCS$(b, x, x.length, x.length)$ on p.394-395 gives a monotonically increasing subsequence of x. The running time is $O(n^2)$.

Problem 3. (15%) In the class, we have talked about how to use the divide-and-conquer

j = 1	2	3	4	5	6	i
0	150	330	405	1655	2010	1
	0	360	330	2430	1950	2
		0	180	930	1770	3
			0	3000	1860	4
				0	1500	5
					0	6

(a) The m table

j = 2	3	4	5	6	i
1	2	2	4	2	1
	2	2	2	2	2
		3	4	4	3
			4	4	4
				5	5

(b) The s table

Figure 3: The m and s tables computed by MATRIX-CHAIN-ORDER

technique to solve the maximum subarray problem. In this problem, we ask you to use dynamic programming to solve this problem. Describe your algorithm and show that the running time of your algorithm can be better than $\Theta(n \log n)$.

Here we re-state the maximum subarray problem. Given an array A, find the nonempty, contiguous subarray of A whose values have the largest sum. The output should include the start and end indices and the sum of the maximum subarray of A.

Sol: Here we use dynamic programming strategy to solve this problem. Let $s_{i,j}(A) = \sum_{k=i}^{j} A[k]$, and $maxSum(A[1..n])$ denote the sum of the maximum subarray of $A[1..n]$. Observe that there is an optimal substructure of the maximum subarray problem, which is of the form $maxSum(A[1..n]) = \max\{maxSum(A[1..n-1]), s_{1,n}(A), s_{2,n}(A), ..., s_{n,n}(A)\}$. Based on the optimal substructure, the following procedure returns the sum of the maximum subarray of A starting from position p_1 and ending at position p_2 by dynamic programming. We use the variable r to keep the value $\max\{s_{k,i}(A) \mid k \in \{1, ..., i\}\}$ at the ith iteration, and s to record the current starting point of the maximum subarray. If the largest sum r is gaining at the current position, we update $maxSum$ to r, p_1 to r and p_2 to i, where p_2 is the ending postition of the maximum subarray so far. Note that if $\max\{s_{k,i}(A) \mid k \in \{1, ..., i\}\}$ is negative, which implies that the maximum subarray doesn't include $A[i]$, we need only to assign 0 to r to record the sum of a new subarray starting from position $i + 1$.

```
MAXIMUM-SUBARRAY($A$)
1  $maxSum = 0$
2  $r = 0$
3  $s = 0$
4  **for** $i = 1$ **to** $n$
5      **if** $r + A[i] < 0$
6          $r = 0$
7          $s = i + 1$
8      **else**
9          $r = r + A[i]$
10     **if** $r > maxSum$
11         $maxSum = r$
12         $p_1 = s$
13         $p_2 = i$
14 **return** the sum of the maximum subarray $maxSum$ from position $p_1$ to $p_2$
```

The algorithm MAXIMUM-SUBARRAY(A) runs in $\Theta(n)$ time, which is strictly lower than $\Theta(n \log n)$. Although this algorithm does not work for the case when all the entries in A are negative, it's easy to handle such an exception.

Problem 4. (15%) Solve problem 15-2 on p.405 of the textbook.

Sol: Let $A[1..n]$ denote the input string. Suppose that the length of the longest palindrome subsequence of $A[i..j]$ is $2k_{i,j} - 1$, for some $k_{i,j} \geq 1$, and let $m[i, j] = k_{i,j}$. It is easy to prove by contradiction that the optimal substructure holds in this problem, that is, we can extend the longest palindrome subsequence of any substring of A to construct an optimal solution of A.

Since the property of optimal substructure holds, we can find the recurrence relation of the length of the longest palindrome subsequence from the subproblems. Let $S_{i,j}[1..k_{i,j}]$ denote the first $k_{i,j}$ characters of a longest palindrome subsequence of $A[i..j]$. If $A[i] = A[j]$, then

$S_{i,j}[1] = A[i]$, otherwise we can construct a palindrome subsequence longer than $S_{i,j}$. In this case, $S_{i,j}[2..k_{i,j}]$ is the first $k_{i,j} - 1$ characters of a longest palindrome subsequence of $A[i+1..j-1]$. Suppose on the contrary that $A[i] \neq A[j]$, then $S_{i,j}$ must be the subsequence of either $A[i..j-1]$ or $A[i+1..j]$. Therefore the recurrence relation of $m[i,j]$ is given by

$$m[i,j] = \begin{cases} m[i+1, j-1] + 1, \text{ if } A[i] = A[j], \\ max\{m[i, j-1], m[i+1, j]\}, \text{ if } A[i] \neq A[j]. \end{cases}$$

Based on the recurrence relation above, Algorithm 2 shows how to find a longest palindrome subsequence of the input string A, and the running time of the algorithm is $O(n^2)$.

Algorithm 2 LPS-LENGTH(A)

```
1: $n = A.length$
2: let $m[1..n, 1..n]$ and $s[1..n, 1..n]$ be new tables
3: **for** $i = 1$ to $n$ **do**
4:     $m[i, i] = 1$
5: **for** $l = 2$ to $n$ **do**
6:     **for** $i = 1$ to $n - l + 1$ **do**
7:         $j = i + l - 1$
8:         **if** $A[i] == A[j]$ **then**
9:             $m[i, j] = m[i+1, j-1] + 1$
10:            $s[i, j] =$ "$\swarrow$"
11:        **else**
12:            **if** $m[i+1, j] \geq m[i, j-1]$ **then**
13:                $m[i, j] = m[i+1, j]$
14:                $s[i, j] =$ "$\downarrow$"
15:            **else**
16:                $m[i, j] = m[i, j-1]$
17:                $s[i, j] =$ "$\leftarrow$"
18: **return** $m$ and $s$
```

The procedure PRINT-LPS(s,A,i,j) as shown below enables us to construct a longest palindrome subsequence of A using the table s by initially calling PRINT-LPS(s,A,1,$A.length$).

```
PRINT-LPS($s$,$A$,$i$,$j$)
1  if $i == j$
2        print $A[i]$
3        return
4  if $s[i,j] ==$"$\swarrow$"
5        print $A[i]$
6        PRINT-LPS($s$,$A$,$i+1$,$j-1$)
7        print $A[i]$
8  elseif $s[i,j] ==$"$\downarrow$"
9        PRINT-LPS($s$,$A$,$i+1$,$j$)
10 else PRINT-LPS($s$,$A$,$i$,$j-1$)
```

Since the procedure PRINT-LPS(s,A,i,j) takes time $O(n)$, the overall time to find a longest palindrome subsequence of A takes time $O(n^2)$.

Problem 5. (20%) Solve problem 15-5 a. on p.406-407 of the textbook.

Sol: Let $A[i,j]$ denote the cost of an optimal solution to the problem of transforming $x[1..i]$ to $y[1..j]$. Observe that the optimal substructure of the Minimum Edit Distance problem is

$$A[i,j] = \min \begin{cases} A[i-1,j-1]\text{+cost(copy), if } x[i] = y[j], \\ A[i-1,j-1]\text{+cost(replace), if } x[i] \neq y[j], \\ A[i-1,j]\text{+cost(delete)} \\ A[i,j-1]\text{+cost(insert)} \\ A[i-2,j-2]\text{+cost(twiddle), if } i,j \geq 2 \text{ and } x[i-1] = y[j] \text{ and } x[i] = y[j-1], \\ \min\limits_{0\leq k<m} A[k,n]\text{+cost(kill), if } i = m \text{ and } j = n. \end{cases}$$

Given two strings x and y of size m and n respectively, Algorithm 3 returns an optimal solution of transforming x to y. In the algorithm, we build up a cost matrix $A[0..m, 0..n+1]$ and set the initial value $A[0,0] = 0$ to dynamically programming the cost evaluation. The first column $A[i,0]$, for $i = 1,...,m$, and the first row $A[0,j]$, $j = 1,...,n$, are used to set up the initial condition while comparing to an empty string. The variables c, r, d, in, t and k are used to record the optimal cost for $A[i, j]$ while the last operation being Copy, Replace, Delete, Insert, Twiddle or Kill, respectively. What we do is to find the minimum one between them except for k, which cannot be evaluated when $j \neq n$. The last column $A[i, n + 1]$, for $i = 1, ..., m$, is used to determine whether it is economic to perform a Kill operation. In addition, we use $label[i, j]$ to record the best operation performed according to $A[i, j]$, so that an optimal operation sequence can be printed out in the end. Both the time and the space complexity of Algorithm 3 are $\Theta(mn)$.

Problem 6. (+5% bonus) We are about a month into the semester now. How do you feel about this course? How much time did you spend on the homework 1 and 2? Please also give some constructive suggestions to the course about the homework, the lectures, or another other related things. Feel free to say ANYTHING about the course. :) (You will get all 5-point bonus if your suggestions or comments are somehow constructive.)
Sol: Skipped.

Algorithm 3 Minimum-Edit-Distance(x,m,y,n)

```
1: for $i = 1$ to $m$ do
2:     $A[i, 0] = i*$cost(delete)
3:     $label[i, 0]$ =DELETE
4: for $j = 1$ to $n + 1$ do
5:     $A[0, j] = j*$cost(insert)
6:     $label[0, j]$ =INSERT($y[j]$)
7: for $i = 1$ to $m$ do
8:     for $j = 1$ to $n + 1$ do
9:         if $j < n + 1$ then
10:            $d = A[i - 1, j]$+cost(delete)
11:            $in = A[i, j - 1]$+cost(insert)
12:            if $x[i] = y[j]$ then
13:                $c = A[i - 1, j - 1]$+cost(copy)
14:                $r = \infty$
15:            else
16:                $c = \infty$
17:                $r = A[i - 1, j - 1]$+cost(replace)
18:            if $i$,$j \geq 2$ and $x[i - 1] = y[j]$ and $x[i] = y[j - 1]$ then
19:                $t = A[i - 2, j - 2]$+cost(twiddle)
20:            else
21:                $t = \infty$
22:            $A[i, j] = \min\{c, r, d, in, t\}$
23:        else
24:            if $i < m$ then
25:                $k = A[i, n]$+cost(kill)
26:            else
27:                $k = A[i, n]$
28:            $A[i, j] = \min\{k, A[i - 1, j]\}$
29:        set $label[i, j]$ properly
30: return $A$ and $label$
```