Homework 6 Solutions
Fundamental Algorithms, Spring 2008, Professor Yap

Due: Mon May 5, during the last class.
HOMEWORK with SOLUTION, prepared by Instructor and T.A.s

INSTRUCTIONS:

- Please read questions carefully. When in doubt, please ask.
- Please write succinctly, to the point. Every sentence must be an complete English sentence.
- You may post general questions to the homework discussion forum in class website. Also, bring your questions to recitation on Monday.

---

1. (2 Points)
   Exercise 6.5, Lect.V.

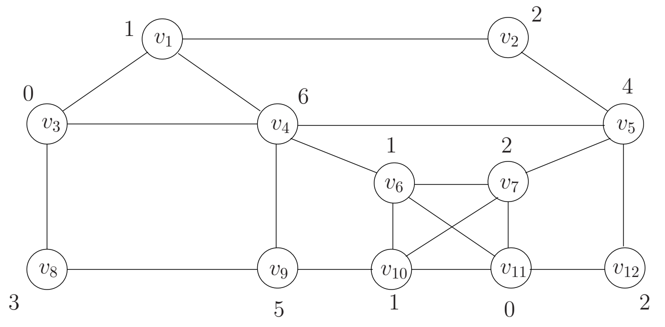

Figure 1: The house graph: The cost of edge $v_i{-}v_j$ is defined as $C(v_i) + C(v_j)$, where $C(v)$ is the value indicated next to v. E.g. $C(v_1{-}v_4) = 1 + 6 = 7$.

Do hand simulation of Kruskal's Algorithm on the graph of Figure 1. We consider each edge in turn, maintaining a partition of $V = \{1, \ldots, 12\}$ into disjoint sets. Let $L(i)$ denote the set containing vertex i. Initially, each node is in its own set, i.e., $L(i) = \{i\}$. Whenever an edge $i{-}j$ is added to the MST, we merge the corresponding sets $L(i) \cup L(j)$. E.g., in the first step, we add edge $1{-}3$. Thus the lists $L(1) = \{1\}$ and $L(3) = \{1\}$ are merged, and we get $L(1) = L(3) = \{1, 3\}$. To show the computation of Kruskal's algorithm, for each edge, if the edge is "rejected", we mark it with an "X". Otherwise, we indicate the merged list resulting from the union of $L(i)$ and $L(j)$: Please fill in the last two columns of the table (we have filled in the first 4 rows for you).

Sorting Order	Edge	Weight	Merged List	Cumulative Weight
1	1–3:	1	$\{1,3\}$	1
2	6–11:	1	$\{6,11\}$	2
3	10–11:	1	$\{6,10,11\}$	3
4	6–10:	2	X	3
5	7–11:	2		
6	11–12:	2		
7	1–2:	3		
8	3–8:	3		
9	6–7:	3		
10	7–10:	3		
11	2–5:	6		
12	3–4:	6		
13	5–7:	6		
14	5–12:	6		
15	9–10:	6		
16	1–4:	7		
17	4–6:	7		
18	8–9:	8		
19	4–5:	10		
20	4–9:	11		

SOLUTION:

Sorting Order	Edge	Weight	Merged List	Cumulative Weight
1	1–3:	1	$\{1,3\}$	1
2	6–11:	1	$\{6,11\}$	2
3	10–11:	1	$\{6,10,11\}$	3
4	6–10:	2	X	3
5	7–11:	2	$\{6,7,10,11\}$	5
6	11–12:	2	$\{6,7,10,11,12\}$	7
7	1–2:	3	$\{1,2,3\}$	10
8	3–8:	3	$\{1,2,3,8\}$	13
9	6–7:	3	X	13
10	7–10:	3	X	13
11	2–5:	6	$\{1,2,3,5,8\}$	19
12	3–4:	6	$\{1,2,3,4,5,8\}$	25
13	5–7:	6	$\{1,2,3,4,5,6,7,8,10,11,12\}$	31
14	5–12:	6	X	31
15	9–10:	6	$\{1,2,3,4,5,6,7,8,9,10,11,12\}$	37
16	1–4:	7	X	37
17	4–6:	7	X	37
18	8–9:	8	X	37
19	4–5:	10	X	37
20	4–9:	11	X	37

REMARK: if we kept track of the number of edges added to the MST, we could stop after this number is equal to $n-1=11$. In particular, we could have stopped after stage 15, after we added the 11th edge.

2. (15 Points)
Exercise 6.6, Lect.V.

This question considers two concrete ways to implement Kruskal's algorithm. Let $V=\{1,2,\ldots,n\}$ and $D[1..n]$ be an array of size n that represents a **forest** $G(D)$ with vertex set V and edge set $E=\{(i,D[i]) : i \in V\}$. More precisely, $G(D)$ is an directed graph that has no cycles except for self-loops (i.e., edges of the form (i,i)). A vertex i such that $D[i] = i$ is called a **root**. The set V is thereby partitioned into disjoint subsets $V = V_1 \cup V_2 \cup \cdots \cup V_k$ (for some $k \geq 1$) such that each V_i has a unique root r_i, and from every $j \in V_i$ there is a path from j to r_i. For example, with $n = 7$, $D[1] = D[2] = D[3] = 3$, $D[4] = 4$, $D[5] = D[6] = 5$ and $D[7] = 6$ (see Figure 2). We call V_i a **component** of the graph $G(D)$ (this terminology is justified because V_i is a component in the usual sense if we view $G(D)$ as an *undirected* graph).

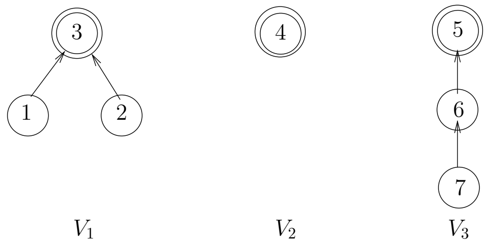

Figure 2: Directed graph $G(D)$ with three components (V_1, V_2, V_3)

(i) Consider two restrictions on our data structure: Say D is **list type** if each component is a linear list. Say D is **star type** if each component is a star (i.e., each vertex in the component points to the root). E.g., in Figure 2, V_2 and V_3 are linear lists, while V_1 and V_2 are stars. Let ROOT(i) denote the root r of the component containing i. Give a pseudo-code for computing ROOT(i), and give its complexity in the 2 cases: (1) D is list type, (2) D is star type.
(ii) Let COMP(i) $\subseteq V$ denote the component that contains i. Define the operation $MERGE(i,j)$ that transforms D so that COMP(i) and COMP(j) are combined into a new component (but all the other components are unchanged). E.g., the components in Figure 2 are $\{1,2,3\}, \{4\}$ and $\{5,6,7\}$. After $MERGE(1,4)$, we have two componets, $\{1,2,3,4\}$ and $\{5,6,7\}$. Give a pseudo-code that implements $MERGE(i,j)$ under the assumption that i,j are roots and D is list type which you must preserve. Your algorithm *must* have complexity $O(1)$. To achieve this complexity, you need to maintain some additional information (perhaps by a simple modification of D).
(iii) Similarly to part (ii), implement $MERGE(i,j)$ when D is star type. Give the complexity of your algorithm.
(iv) Describe how to use ROOT(i) and $MERGE(i,j)$ to implement Kruskal's algorithm for computing the minimum spanning tree (MST) of a weighted connected undirected graph H.
(v) What is the complexity of Kruskal's in part (iv) if (1) D is list type, and if (2) D is star type. Assume H has n vertices and m edges.

SOLUTION: (i) The obvious algorithm for ROOT(i) this:

```
ROOT($i$):
    $j \leftarrow i$;
    While ($D[j] \neq j$)
        $j \leftarrow D[j]$;
    Return($j$);
```

For list type structure, ROOT(i) can be found in $O(n)$ time. For star type structure, ROOT(i) takes $O(1)$ time.
(ii) Let D be list type. To achieve $O(1)$, we need to know both ends of a linked list. We modify D so that if i is a leader, and j is the last element in the linked list corresponding to COMP(i) then $D[i] = -j$. Then $MERGE(i, j)$ amounts to concatenating the lists corresponding to i and j:

```
$MERGE(i, j)$:
    $k \leftarrow D[j]$;
    $D[j] \leftarrow -D[i]$;
    $D[i] \leftarrow k$;
```

This algorithm is $O(1)$, as desired.
(iii) Let D be star type. In this case, we do not need to modify D.

```
$MERGE(i, j)$:
    For $k = 1$ to $n$
        If $D[k] = j$ then $D[k] = i$;
```

This algorithm is $O(n)$.
(iv) In Kruskal's algorithm we consider each edge $e = (i, j)$ in order of non-decreasing weight. Let T be the current set of edges chosen to be in the MST. This induces a partition of V into components, represented by the data structure D. Initially, $D[k] = k$ for all k. To decide whether e can be added to T, we first compute $i' =$ ROOT(i) and $j' =$ ROOT(j). If $i' = j'$, then we cannot add e. Otherwise we compute $MERGE(i', j')$.
(v) Kruskal's algorithm as described in (iv) makes $2m$ calls to ROOT() and $n - 1$ calls to $MERGE()$. Note that there is a preliminary cost of $O(m \log n)$ to sort the edges. It remains to assess the cost of operations on D.
For list type D, ROOT(i) takes $O(n)$ time and $MERGE(i, j)$ takes $O(1)$ time. Hence the overall cost is $O(mn + n) = O(mn)$ (assuming $m \geq 1$).
For star type D, ROOT(i) takes $O(1)$ time and $MERGE(i, j)$ takes $O(n)$ time. Hence the overall cost is $O(m + n^2) = O(n^2)$.

3. (0 Points)
   Exercise 1.1, Lect.VI.

   Our model and analysis of counters can yield the exact cost to increment from any initial counter value to any final counter value. Show that the exact number of work units to increment a counter from 68 to 125 is 190.

SOLUTION: It costs $2n - \Phi(D_n)$ work units to count from 0 to n. Since $125 = (1, 111, 101)_2$ and $68 = (1, 000, 100)_2$, and so it costs exactly $2(125 - 68) - \Phi(D_{125}) + \Phi(D_{68}) = 194 - 6 + 2 = 190$.

4. (4 Points)
   Exercise 1.2, Lect.VI.

A simple example of amortized analysis is the cost of operating a special kind of pushdown stack. Our stack S supports the following two operations: $S.\texttt{push}(K)$ simply add the key K to the top of the current stack. But $S.\texttt{pop}(K)$ will keep popping the stack as long at the current top of stack has a key smaller than K (the bottom of the stack is assume to have the key value ∞). The cost for push operation is 1 and the cost for popping $m \geq 0$ items is $m + 1$.
(a) Use our potential framework to give an amortized analysis for a sequence of such push/pop operations, starting from an initially empty stack.
(b) How tight is your analysis? E.g., can it give the *exact cost*, as in our Counter Example?

SOLUTION: We will use the accounting method. For each Push, we will charge 2 units of work. One of these units will be used for the actual push, the other will be saved for when that element is later popped off the stack. For the Pop operation, we will charge nothing. Observe that the extra credit stored equals the number of elements in the stack. Now observe that for any sequence of n operations, we charge at most $T(n) = 2 * n$ work, obtaining an amortized cost of $T(n)/n = O(1)$ work per operation. Another solution for this problem involves setting a potential function to be equal to the number of elements in the stack. Think through how such a proof would work using this framework. In both cases this analysis is tight. Imagine inserting n elements and then the final Pop, pops the entire stack.

5. (10 Points)
Exercise 1.3, Lect.VI.

Let us generalize the example of incrementing binary counters. Suppose we have a collection of binary counters, all initialized to 0. We want to perform a sequence of operations, each of the type

$$\texttt{inc}(C), \quad \texttt{double}(C), \quad \texttt{add}(C, C')$$

where C, C' are names of counters. The operation $\texttt{inc}(C)$ increments the counter C by 1; $\texttt{double}(C)$ doubles the counter C; finally, $\texttt{add}(C, C')$ adds the contents of C' to C while simultaneously set the counter C' to zero. Show that this problem has amortized constant cost per operation.

We must define the cost model. The length of a counter is the number of bits used to store its value. The cost to double a counter C is just 1 (you only need to prepend a single bit to C). The cost of $\texttt{add}(C, C')$ is the number of bits that the standard algorithm needs to look at (and possibly modify) when adding C and C'. E.g., if $C = 11, 1001, 1101$ and $C' = 110$, then $C + C' = 11, 1010, 0011$ and the cost is 9. This is because the algorithm only has to look at 6 bits of C and 3 bits of C'. Note that the 4 high-order bits of C are not looked at: think of them as simply being "linked" to the output. Here is where the linked list representation of counters is exploited. After this operation, C has the value $11, 1010, 0011$ and C' has the value 0.
HINT: The potential of a counter C should take into account the number of 1's as well as the bit-length of the counter.

You couldn'
do this witl
arrays!

SOLUTION: Let m be the number of counters and let us represent each binary counter C_i, $1 \leq i \leq m$, with a linked list that initially is empty. We also denote the value stored in C_i by 'C_i'. Let L_i be the length of C_i, O_i be the number of 1's in C_i, and E_i be the length of the maximum suffix of 1's in C_i in the current state.
Now define the potential function of the set of counters as follows:

$$\Phi = \sum_{k=1}^{m} (L_i + O_i).$$

For each operation on the set of counters, we will show that it has a constant amortized cost. Consider the most interesting case of `add`(C_1, C_2). In this case, $\Delta\Phi \leq -\min(L_1, L_2) - K$ where K is the number of carry bits beyond $\min(L_1, L_2)$ in the addition process. This release enough potential, except for some small constant A, to pay for the cost of addition. This small constant A can be our amortized cost.

(a) Inc(C_i): The actual cost is $E_i + 1$ since it resets E_i bits and sets one bit to a 1. The number of 1's in C_i after this operation is $O_i - E_i + 1$ and the length of C_i is at most $L_i + 1$. Thus the potential difference is

$$\begin{aligned} \Delta\Phi &\leq [(O_i - E_i + 1) + (L_i + 1)] - [O_i + L_i] \\ &= 2 - E_i. \end{aligned}$$

The amortized cost is therefore

$$(E_i + 1) + \Delta\Phi \leq (E_i + 1) + (2 - E_i) = 3.$$

(b) Double(C_i): The actual cost is 1 since it is the cost of appending a 0-bit to the end of the linked list for C_i. After this operation, the number of 1's is not changed but the length is increased by 1. Thus the potential difference is 1. Therefore the amortized cost is 2.

(c) Add(C_i, C_j): Let us assume that $C_i \geq C_j$. Let t_{01} be the number of times a bit is flipped from 0 to 1 in C_i and t_{10} be the number of times a bit is flipped from 1 to 0 when C_j is added to C_i. After the addition, the number of 1's in C_i is $O_i + t_{01} - t_{10}$ and the length of C_i is at most $L_i + 1$. The actual cost of the addition is at most $L_j + t_{10} + 1$ because all bits of C_j is scanned to be added to C_i and after that a carry will flip bits of C_i from 1 to 0. Thus the potential difference is

$$\begin{aligned} \Delta\Phi &\leq [(L_i + 1) + (O_i + t_{01} - t_{10})] - [(L_i + L_j) + (O_i + O_j)] \\ &= 1 + t_{01} - t_{01} - L_j - O_j. \end{aligned}$$

Therefore the amortized cost is

$$\begin{aligned} (L_j + t_{10} + 1) + (1 + t_{01} - t_{10} - L_j - O_j) &= 2 + (t_{01} - O_j) \\ &\leq 2, \end{aligned}$$

since $O_j \geq t_{01}$.

6. (0 Points)
Exercise 2.1 and 2.2, Lect.VI.

7. (5 Points)
Fix a key K and a splay tree T_0. Let $T_{i+1} \leftarrow$ `splay`(K, T_i) for $i = 0, 1, \ldots$.
(a) Under what conditions will the T_i's stabilize (become a constant)?
(b) What must be the case if the T_i's do not stabilize.

SOLUTION: (a) The T_i's stabilizes provided either key K is in T_0, or $K = +\infty$, or $K = -\infty$.
(b) The T_i's do not stabilize iff K does not lie in tree T_0, and moreover, K lies strictly between two keys K' and K'' which are K's successor and predecessor (resp.). Then the T_i's eventually alternates between having K' as root or K'' as root.

8. (8 Points)
Exercise 2.7, Lect.VI.

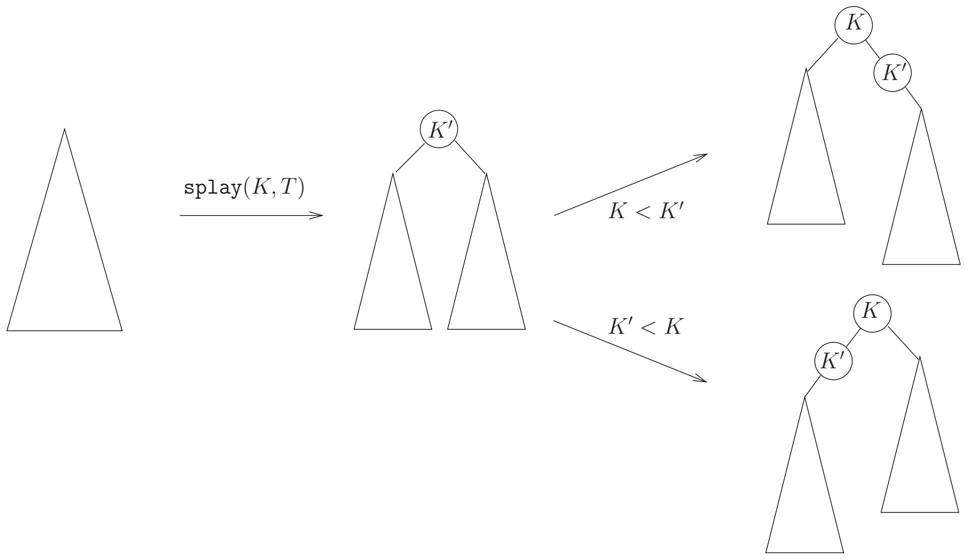

Figure 3: Alternative method to insert a key K.

In our operations on splay trees, we usually begin by performing a splay. This is not the case in our insertion algorithm. But consider the following variant insertion that follows this paradigm. To insert an item X into T:
1. Perform $\texttt{splay}(X.\texttt{key}, T)$ to give us an equivalent tree T'.
2. Now examine the key K' at root of T': if $K' = X.\texttt{key}$, we declare an error (recall that keys must be distinct).
3. If $K' > X.\texttt{key}$, we install a new root containing X, and K' becomes the right child of X; the case $K' < X.\texttt{key}$ is symmetrical. In either case, the new root has key equal to $X.\texttt{key}$. See Figure 3.
(a) Prove that the amortize complexity this insertion algorithm remains $O(\log n)$. NOTE: To do this, you must understand Theorem 1 (VI.§1) and might be helpful to look at the proof of Theorem 5 as well.
(b) (Ignore this part)

SOLUTION: When we insert a new element, we not only add a node to the tree, but that node has its own potential which was not accounted for before the insertion. Letting r denote the root of the tree, the potential at r becomes $\log(n+1)$. The potential generated by adding node X is also at most $\log(n)$, so we see that after insertion the potential increases by at most, $\log(n+1) + \log(n) \leq 2\log(n)$.

9. (15 Points)
Recall the convex hull problem in Lect.VI, §4. Consider an array-based representation of upper hulls. An upper hull $U = (v_0, \ldots, v_k)$ is represented by an array $U[0..k]$, where $U[i] = v_i$. We want to build U incrementally using METHOD ONE in the text. Assume the set of input points is non-degenerate (no 3 points are collinear).

(a) Describe in detail how to carry out the operation of finding LeftTangent of a given point p in U.
(b) Describe how to implement the insertion of a new point p.
(c) Carry out a complexity analysis of your algorithm.
(d) Show how to modify your algorithms if the input can be degenerate. You must discuss how to implement the changes using the LeftTurn(p,q,r) predicate. NOTE: For this problem, ASSUME that an input point p is in U if it does NOT lie in the interior of the convex hull.

SOLUTION: Before we attempt to find LeftTangent, first verify that the point p does not lie within the interior of the convex hull. By assumption, the array of points is already sorted by x values. Let p lie between points v_j and v_{j+1} for some $j = 0, \ldots, k$ (if $j = 0$ then v_j is undefined, and if $j = k$ then v_{j+1} is undefined). In case $j = 0$ is undefined, we can conclude that the left tangent is $v_S = (0, -\infty)$ (south pole). Knowing j, we can check if p lies inside the upper hull. If so, we return $\uparrow$. Otherwise, we need to only search points $v_0, v_2, \ldots, v_{k-1}$ for the necessary vertex. We will now perform a binary search looking for the proper vertex. Let $v_i = v_{k/2}$ be a point and look at the line connecting p and v_i. We now verify that both of points v_i neighbors lies beneath this line. If so, they we have found a new point. Otherwise, if v_{i+1} lies above the line, we perform a binary search on the set $(v_{i+1}, \ldots, v_k)$ or symmetrically if v_{i-1} lies above the line, we search on the other half.

10. (2 Points)
Exercises 1.3, Lect.VII.

11. (10 Points)
Exercises 1.10, Lect.VII. Let X, Y be strings.
(a) Prove that $L(XX, Y) \leq 2L(X, Y)$.
(b) Give examples where the inequality is strict, and where it is not strict.
(c) Prove that $L(XX, YY) \leq 3L(X, Y)$. How tight is this upper bound?

SOLUTION:
(a) Two proofs from students: [Michael Haag, 2003] Let $L(XX, Y) = m + n$, where m of the first X, and n of the second X, are used in an optimal solution in $LCS(XX, Y)$. Then $L(X, Y) \geq \max\{m, n\}$. Hence $2L(X, Y) \geq m + n = L(XX, Y)$.
[Yanjun Wang, 2003] Let $Z \in LCS(X, Y)$. Then $L(XX, Y) \leq L(XX, Z) + L(Z, Y)$ by triangular inequality. But $|Z| = L(XX, Z) = L(Z, Y) = L(X, Y)$. Hence $L(XX, Y) \leq 2L(X, Y)$.
(b) Let $X = a^n$ and $Y = a^{n+1}$. Then $L(XX, Y) = n + 1 < 2n = 2L(X, Y)$. Let $X' = a^n$ and $Y' = a^{2n}$. Then $L(X'X', Y') = 2n = 2L(X', Y')$.
(c) Let $Z \in LCS(XX, YY)$ of length ℓ. Define the increasing function f so that $Z[i] = XX[f(i)]$ for $i = 1, \ldots, \ell$. Similarly, define g so that $Z[i] = YY[g(i)]$. Suppose $f(i_0) \leq |X|$ and $f(i_0+1) > |X|$. There are two cases: either $g(i_0) > |Y|$ or $g(i_0) \leq |Y|$. In the former case, $i_0 \leq L(X, YY) \leq 2L(X, Y)$ and $|Z| - i_0 \leq L(X, Y)$. The latter case is symmetrically argued.
This inequality is tight: let $X = a^n b^n c^n x^n$ and $Y = y^n c^n b^n a^n$ be strings of length $4n$. Then $L(X, Y) = n$ and $L(XX, YY) = 3n$.

12. (12 Points)
Exercise 1.18, Lect.VII. Researchers are using LCS computation to fight computer viruses. A virus that is attacking a machine has a predictable pattern of messages it sends to the machine. We view the concatenation of all these messages that a potential virus sends as a single string. Call the first 1000 bytes than from any source (i.e., potential virus) the **signature** of that source. Let X be the signature of an unknown source and Y is the signature of a known virus. To test the source is the Y-virus, we compute $L(X, Y)$. Empirically, suppose it is shown that if $L(X, Y) > 500$, then that our source is likely to be Y-virus.

(a) Design a practical and efficient algorithm for the decision problem $L(X, Y, k)$ which outputs "PROBABLY VIRUS" if $L(X, Y) > k$ and "PROBABLY NOT VIRUS" otherwise. Give the pseudo-code for an efficient practical algorithm. NOTE: The obvious algorithm is to use the standard algorithm to compute $L(X, Y)$ and then compare n to k. But we want you to do better than this. HINT: There are two ideas we want you to exploit – most students only think of one idea.
(b) Quantify the complexity of your algorithm, and compare its performance to the obvious algorithm (which first computes $L(X, Y)$). First do your analysis using the general complexity parameters of $m = |X|, n = |Y|$ and k, and also $\ell = L(X, Y)$. Also discuss this for the special case of $m = n = 1000$ and $k = 500$.

SOLUTION:

(a) We exploit two ideas: EARLY ACCEPTANCE and EARLY REJECTION. Early acceptance means that once we determined that $L(X, Y) > k$, we can stop, without computing the true value of $L(X, Y)$. Early rejection means that once we determined hat $L(X, Y) \leq k$, we can stop. Here is the algorithm:

```
$L(X, Y, k)$:
Output: If $L(X, Y) > k$, then print "PROBABLY VIRUS"
    Else print "PROBABLY NOT VIRUS"
▷ Initialization
    For $i = 0$ to $|X|$ do:
        $L[i, 0] := 0$.
    For $j = 0$ to $|Y|$ do:
        $L[0, j] := 0$.
▷ Main Loop
    For $i = 1$ to $|X|$ do:
        For $j = 1$ to $|Y|$ do:
            If $X[i] = Y[j]$ then $L[i, j] := 1 + L[i-1, j-1]$
            Else $L[i, j] := \max\{L[i-1, j], L[i, j-1]\}$
            If $L[i, j] > k$, return("PROBABLY VIRUS")
            If $L[i, j] \leq k - \max(|X| - i, |Y| - j)$, return("PROBABLY NOT VIRUS")
```

You can improve this computation by filling in the matrix L using the "square filling method". This rule says that you must not fill in $L[i, j]$ unless both $L[i-1, j]$ and $L[i, j-1]$ have already been filled. So the filled entries are essentially a square (except for the last row and last column).
Most students got the early acceptance idea, but most students do not see early rejection idea unless we give the hint! In fact, since most sources are not viruses, it is the early rejection idea that is more useful.

(b) We measure the complexity as a function of $m = |X|, n = |Y|$ and k, and also $\ell = L(X, Y)$. The first remark should be that our heuristics in (a) do not give any asymptotic improvement in the worst case: it is still $\Theta(mn)$ like the straightforward algorithm.
In the best case, we might achieve a complexity of $O(k^2)$. For our numerical example where $m = n = 1000$ and $k = 500$, we can at most save a factor of 4.
Next, suppose $L(X, Y) = \ell$. If $\ell > 500$, then we could detect probable virus after filling in $1500 - \ell$ rows of the matrix L. I.e., we do not fill in at least $\ell - 500$ rows. Thus we save a factor of $(\ell/1000) - 0.5$. If we use the "square filling method" then we say a factor of $1 - (1.5 - (\ell/1000))^2$.

13. (2 Points)

Exercise 2.4, Lect.VII. Compute $A(X, Y)$ where X, Y are the strings `AATTCCCGA` and `GCATATT`. Assume δ has gap penalty 2, $\delta(x, x) = -2$ and $\delta(x, y) = 1$ if $x \neq y$. You must organize this computation systematically as in the LCS problem.

SOLUTION: Solution by Iuliana Ionita (2003). We fill in the usual table row by row:

	_	G	C	A	T	A	T	T
_	0	2	4	6	8	10	12	16
A	2	1	3	2	4	6	8	10
A	4	3	2	1	3	2	4	6
T	6	5	4	3	-1	1	0	2
T	8	7	6	5	1	0	-1	-2
C	10	9	5	7	3	2	1	0
C	12	11	7	6	5	4	3	2
C	14	13	9	8	7	6	5	4
G	16	12	11	10	9	8	7	6
A	18	14	13	9	11	7	9	8

The value we are interested in is at the bottom right corner of the table, so $A(X, Y) = 8$.