Introduction to Algorithms November 18, 2005
Massachusetts Institute of Technology 6.046J/18.410J
Professors Erik D. Demaine and Charles E. Leiserson Handout 25

Problem Set 7 Solutions

Problem 7-1. Edit distance

In this problem you will write a program to compute edit distance. This problem is mandatory. Failure to turn in a solution will result in a serious and negative impact on your term grade! We advise you to start this programming assignment as soon as possible, because getting all the details right in a program can take longer than you think.

Many word processors and keyword search engines have a spelling correction feature. If you type in a misspelled word x, the word processor or search engine can suggest a correction y. The correction y should be a word that is close to x. One way to measure the similarity in spelling between two text strings is by "edit distance." The notion of edit distance is useful in other fields as well. For example, biologists use edit distance to characterize the similarity of DNA or protein sequences.

The ***edit distance*** $d(x, y)$ of two strings of text, $x[1\,..\,m]$ and $y[1\,..\,n]$, is defined to be the minimum possible cost of a sequence of "transformation operations" (defined below) that transforms string $x[1\,..\,m]$ into string $y[1\,..\,n]$.[1] To define the effect of the transformation operations, we use an auxiliary string $z[1\,..\,s]$ that holds the intermediate results. At the beginning of the transformation sequence, $s = m$ and $z[1\,..\,s] = x[1\,..\,m]$ (i.e., we start with string $x[1\,..\,m]$). At the end of the transformation sequence, we should have $s = n$ and $z[1\,..\,s] = y[1\,..\,n]$ (i.e., our goal is to transform into string $y[.\,.\,n]$). Throughout the tranformation, we maintain the current length s of string z, as well as a *cursor position* i, i.e., an index into string z. The invariant $1 \le i \le s + 1$ holds at all times during the transformation. (Notice that the cursor can move one space beyond the end of the string z in order to allow insertions at the end of the string.)

Each transformation operation may alter the string z, the size s, and the cursor position i. Each transformation operation also has an associated cost. The cost of a sequence of transformation operations is the sum of the costs of the individual operations on the sequence. The goal of the edit-distance problem is to find a sequence of transformation operations of minimum cost that transforms $x[1\,..\,m]$ into $y[1\,..\,n]$.

There are five transformation operations:

[1] Here we view a text string as an array of characters. Individual characters can be manipulated in constant time.

Operation	*Cost*	*Effect*
left	0	If $i = 1$ then do nothing. Otherwise, set $i \leftarrow i - 1$.
right	0	If $i = s + 1$ then do nothing. Otherwise, set $i \leftarrow i + 1$.
replace	4	If $i = s+1$ then do nothing. Otherwise, replace the character under the cursor by another character c by setting $z[i] \leftarrow c$, and then incrementing i.
delete	2	If $i = s+1$ then do nothing. Otherwise, delete the character c under the cursor by setting $z[i \,..\, s] \leftarrow z[i + 1 \,..\, s + 1]$ and decrementing s. The cursor position i does not change.
insert	3	Insert the character c into string z by incrementing s, setting $z[i + 1 \,..\, s] \leftarrow z[i \,..\, s - 1]$, setting $z[i] \leftarrow c$, and then incrementing index i.

As an example, one way to transform the source string `algorithm` to the target string `analysis` is to use the sequence of operations shown in Table 1, where the position of the underlined character represents the cursor position i. Many other sequences of transformation operations also transform `algorithm` to `analysis`—the solution in Table 1 is not unique—and some other solutions cost more while some others cost less.

Operation	z	*Cost*	*Total*
initial string	a̲lgorithm	0	0
right	al̲gorithm	0	0
right	alg̲orithm	0	0
replace by `y`	aly̲orithm	4	4
replace by `s`	alys̲rithm	4	8
replace by `i`	alysi̲thm	4	12
replace by `s`	alysis̲hm	4	16
delete	alysis̲m	2	18
delete	alysis̲	2	20
delete	alysis_	2	22
left	alysi̲s	0	22
left	alys̲is	0	22
left	aly̲sis	0	22
left	al̲ysis	0	22
left	a̲lysis	0	22
insert `n`	an̲lysis	3	25
insert `a`	ana̲lysis	3	28

Table 1: Transforming `algorithm` into `analysis`

(a) It is possible to transform `algorithm` to `analysis` without using the "left" operation. Give a sequence of operations in the style of Table 1 that has the same cost as in Table 1 but does not use the "left" operation.

Solution:

Operation	z	*Cost*	*Total*
initial string	`algorithm`	0	0
right	`algorithm`	0	0
insert `n`	`anlgorithm`	3	3
insert `a`	`analgorithm`	3	6
right	`analgorithm`	0	6
replace by `y`	`analyorithm`	4	10
replace by `s`	`analysrithm`	4	14
replace by `i`	`analysiithm`	4	18
replace by `s`	`analysisthm`	4	22
delete	`analysishm`	2	24
delete	`analysism`	2	26
delete	`analysis_`	2	28

Table 2: Transforming `algorithm` into `analysis` without moving left

(b) Argue that, for any two strings x and y with edit distance $d(x, y)$, there exists a sequence S of transformation operations that transforms x to y with cost $d(x, y)$ where S does not contain any "left" operations.

Solution: We argue that there is a sequence S that transforms x to y with cost $d(x, y)$ without using any "left" operations by contradiction. Suppose that no such sequence S exists. Consider a sequence S' that does transform x to y with cost $d(x, y)$ that uses "left" operations. Consider the characters inserted by the operations in S'. If a character is inserted and then later deleted, then both operations can be removed from S' to produce a sequence S'' that produces the same result at lower cost. If a character a is inserted and then later replaced by a character b, then the insert operation can be changed to insert b and the replace operation can be removed to produce a sequence S'' that produces the same result at lower cost. If a character is replaced by a character a and then replaced again by a character b, then both operations can be replaced by "replace b". This means that all inserted and replaced characters are never changed after performing the insertion or replacement. Notice that after removing these operations that introduce dependencies, any sequence of insert, delete, and replace operations can be reordered so that they occur from left to right without affecting the outcome of the transformation.

(c) Show that the problem of calculating the edit distance $d(x, y)$ exhibits optimal substructure. (*Hint:* Consider all suffixes of x and y.)

Solution:

We show that computing edit distance for strings x and y can be done by finding the edit distance of subproblems. Define a cost function

$$c_{xy}(i,j) = d(x, y[1\,..\,i] \| x[j+1\,..\,m]) \tag{1}$$

That is, $c_{xy}(i,j)$ is the minimum cost of transforming the first j characters of x into the first i characters of y. Then $d(x,y) = c_{xy}(n,m)$. Now consider a sequence of operations $S = \langle o_1, o_2, \ldots, o_k \rangle$ that transforms x to y with cost $C(S) = d(x,y)$. Let S_i be the subsequence of S containing the first i operations of S. Let z_i be the auxilliary string after performing operations S_i, where $z_0 = x$ and $z_k = y$.

Theorem 1 *If $C(S_i) = d(x, z_i)$, then $C(S_{i-1}) = d(x, z_{i-1})$.*

That is, the optimal solution to $d(x, z_i)$ contains optimal solutions to subproblems $d(x, z_{i-1})$. We prove this claim by contradiction using cut and paste. Suppose that $C(S_{i-1}) \neq d(x, z_{i-1})$. There are two cases, $C(S_{i-1}) < d(x, z_{i-1})$ or $C(S_{i-1}) > d(x, zi-1)$. If $C < d(x, z_{i-1})$, then we can transform x to z_{i-1} using operations S_{i-1} with lower cost than $d(x, z_{i-1})$, which is a contradiction. If $C(S_{i-1}) > d(x, z_{i-1})$, then we could replace S_{i-1} with the sequence of operations S' that transforms x to z_{i-1} with cost $d(x, z_{i-1})$. Then the sequence of operations $S' \cup o_i$ transforms x to y with cost $C(S' \cup o_i)$.

$$\begin{aligned} C(S' \cup o_i) &= d(x, z_{i-1}) + C(o_i) \\ &< C(S_{i-1}) + C(o_i) \\ &= C(S_i) \\ &= d(x, z_i) \end{aligned}$$

This means that $d(x, z_i)$ is not the edit distance between x and z_i, which is a contradiction. Therefore Theorem 1 is correct and the edit distance problem exhibits optimal substructure.

(d) Recursively define the value of edit distance $d(x,y)$ in terms of the suffixes of strings x and y. Indicate how edit distance exhibits overlapping subproblems.

Solution:

We can calculate the edit distance $d(x,y)$ using the definition of $c_{xy}(i,j)$ from Equation 1. Recall that $d(x,y) = c_{xy}(m,n)$. Since we showed in part (a) that there exists a sequence of operations that achieves $d(x,y)$ without using the "left" operation, we only need to consider the four operations "right", "replace", "delete", and "insert". We can calculate $c_{xy}(m,n)$ recursively. The base case is when no transformation operations have been performed, so $c_{xy}(0,0) = 0$.

$$c_{xy}(i,j) = \min \begin{cases} 0 & \text{if } i = 0 \text{ and } j = 0 \\ c_{xy}(i-1,j-1) & \text{if } i > 0 \text{ and } j > 0 \text{ and } x[i-1] = y[j-1] \\ c_{xy}(i-1,j-1) + 4 & \text{if } i > 0 \text{ and } j > 0 \\ c_{xy}(i-1,j) + 2 & \text{if } i > 0 \\ c_{xy}(i,j-1) + 3 & \text{if } j > 0 \end{cases} \tag{2}$$

This recursive solution exhibits overlapping subproblems. For example, computing $c_{xy}(i,j)$, $c_{xy}(i-1,j)$, and $c_{xy}(i,j-1)$ all require recursively computing the subproblem $c_{xy}(i-1,j-1)$.

(e) Describe a dynamic-programming algorithm that computes the edit distance from $x[1\,..\,m]$ to $y[1\,..\,n]$. (Do not use a memoized recursive algorithm. Your algorithm should be a classical, bottom-up, tabular algorithm.) Analyze the running time and space requirements of your algorithm.

Solution: Construct a table T where each entry $T[i,j] = c_{xy}(i,j)$. Since each value of $c_{xy}(i,j)$ only depends on $c_{xy}(i',j')$ where $i' \le i$ and $j' \le j$, we can compute the entries of T row by row using Equation 2:

EDIT-DISTANCE$(x[1..m], y[1..n])$
1 $T[0,0] \leftarrow 0$
2 **for** $i \leftarrow 1$ **to** n
3 **for** $j \leftarrow 1$ **to** m
4 **do** $T[i,j] \leftarrow \min \begin{cases} T[i-1,j-1] & \text{if } i > 0 \text{ and } j > 0 \text{ and } x[i-1] = y[j-1] \\ T[i-1,j-1] + 4 & \text{if } i > 0 \text{ and } j > 0 \\ T[i-1,j] + 2 & \text{if } i > 0 \\ T[i,j-1] + 3 & \text{if } j > 0 \end{cases}$
5 **return** $T[n,m]$

The running time of this algorithm is $\Theta(mn)$. This algorithm requires $\Theta(mn)$ space.

(f) Implement your algorithm as a computer program in any language you wish.[2] Your program should calculate the edit distance $d(x,y)$ between two strings x and y using dynamic programming and print out the corresponding sequence of transformation operations in the style of Table 1. Run your program on the strings

$$x = \texttt{"electrical engineering"},$$
$$y = \texttt{"computer science"}.$$

Submit the source code of your program electronically on the class website, and hand in a printout of your source code and your results.

[2] Solutions will be provided in Java and Python.

Solution: Source code for the solutions in Java and Python can be found on the class web site. The output of program on the above inputs is:

```
x: electrical engineering
y: computer science
Edit Distance: 54

Oper    | c |Total| z
initial | 0 |   0 | *electrical engineering
delete  | 2 |   2 | *lectrical engineering
delete  | 2 |   4 | *ectrical engineering
delete  | 2 |   6 | *ctrical engineering
right   | 0 |   6 | c*trical engineering
insert  | 3 |   9 | co*trical engineering
insert  | 3 |  12 | com*trical engineering
insert  | 3 |  15 | comp*trical engineering
insert  | 3 |  18 | compu*trical engineering
right   | 0 |  18 | comput*rical engineering
insert  | 3 |  21 | compute*rical engineering
right   | 0 |  21 | computer*ical engineering
delete  | 2 |  23 | computer*cal engineering
delete  | 2 |  25 | computer*al engineering
delete  | 2 |  27 | computer*l engineering
delete  | 2 |  29 | computer* engineering
right   | 0 |  29 | computer *engineering
delete  | 2 |  31 | computer *ngineering
replace | 4 |  35 | computer s*gineering
replace | 4 |  39 | computer sc*ineering
right   | 0 |  39 | computer sci*neering
delete  | 2 |  41 | computer sci*eering
delete  | 2 |  43 | computer sci*ering
right   | 0 |  43 | computer scie*ring
delete  | 2 |  45 | computer scie*ing
delete  | 2 |  47 | computer scie*ng
right   | 0 |  47 | computer scien*g
insert  | 3 |  50 | computer scienc*g
replace | 4 |  54 | computer science*
```

Note that this solution is not necessarily unique. There may be other transformation sequences that have the same cost.

Sample input and output text is provided on the class website to help you debug your program. These solutions are not necessarily unique: there may be other sequences of transformation oper-

ations that achieve the same cost. As usual, you may collaborate to solve this problem, but you must write the program by yourself.

(g) Run your program on the three input files provided on the class website. Each input file contains the following four lines:

1. The number of characters m in the string x.
2. The string x.
3. The number of characters n in the string y.
4. The string y.

Compute the edit distance $d(x, y)$ for each input. Do not hand in a printout of the transformation operations for this problem part. (Extra bonus kudos if you can identify the source of all the texts, without searching the web.)

Solution:

Input File	$d(x, y)$
Input 1	1816
Input 2	1824
Input 3	1829

(h) If z is implemented using an array, then the "insert" and "delete" operations require $\Theta(n)$ time. Design a suitable data structure that allows each of the five transformation operations to be implemented in $O(1)$ time.

Solution: All the transformation operations can be implemented in $O(1)$ time using two stacks, L and R. Initially, L is empty and R contains all the characters of x in order.

Operation	*Implementation*
left	If not $\textsc{Empty}(L)$, then $\textsc{Push}(R, \textsc{Pop}(L))$
right	If not $\textsc{Empty}(R)$, then $\textsc{Push}(L, \textsc{Pop}(R))$
replace by c	If not $\textsc{Empty}(R)$, then $\textsc{Pop}(R)$, $\textsc{Push}(L, c)$
delete	If not $\textsc{Empty}(R)$, then $\textsc{Pop}(R)$
insert c	$\textsc{Push}(L, c)$

Each stack operation requires $O(1)$ time, and each transformation operation requires only $O(1)$ stack operations. Therefore each operation requires $O(1)$ time.

Problem 7-2. GreedSox

GreedSox, a popular major-league baseball team, is interested in one thing: making money. They have hired you as a consultant to help boost their group ticket sales. They have noticed the following problem. When a group wants to see a ballgame, all members of the group need seats (in the bleacher section), or they go away. Since partial groups can't be seated, the bleachers are often not

full. There is still space available, but not enough space for the entire group. In this case, the group cannot be seated, losing money for the GreedSox.

The GreedSox want your recommendation on a new seating policy. Instead of seating people first-come/first-serve, the GreedSox decide to seat large groups first, followed by smaller groups, and finally singles (i.e., groups of 1).

You are given a set of groups, $G[1\,..\,m] = [g_1, g_2, \ldots, g_m]$, where g_i is a number representing the size of the group. Assume that the bleachers seat n people. Consider the following greedy seating algorithm, where the function ADMIT(i) admits group i, and REJECT(i) sends away group i.

```
SEAT(G[1..m], n)
 1  admitted ← 0
 2  remaining ← n
 3  G ← SORT(G)                    ▷ Sort groups largest to smallest.
 4  for i ← 1 to m
 5      do if G[i] ≤ remaining
 6            then ADMIT(i)
 7                 remaining ← remaining − G[i]
 8                 admitted ← admitted + G[i]
 9            else REJECT(i)
10  return admitted
```

The SEAT algorithm first sorts the groups by size. It then iterates through the groups from largest to smallest, seating any group that fits in the bleachers. It returns the number of people admitted.

(a) The GreedSox owners are right: the greedy seating algorithm works pretty well. Show that if, given G and n, it is possible to admit k people, then the greedy seating algorithm admits at least $k/2$ people.

Solution: We begin by proving a lemma about the number of people admitted by the algorithm SEAT.

Lemma 2 *Algorithm* SEAT *either admits all groups of size* $\leq n$*, or it admits* $\geq n/2$ *people.*

Proof. Assume algorithm SEAT does *not* admit all groups of size $\leq n$. That is, there is some group g_i of size $\leq n$ that SEAT does not admit.

There are two cases to consider. First, assume that $g_i \geq n/2$. Then, since the algorithm is greedy, we know that it must have admitted some group *larger* than g_i; otherwise, group g_i would have been admitted. Therefore, we can conclude that the algorithm seats $\geq n/2$ people, as required.

For the second case, assume that $g_i < n/2$. Since g_i is not admitted, we know that at some point *remaining* $< g_i < n/2$. Since *remaining* is non-increasing, we thus conclude that at least $n/2$ people are seated.

We argue that this lemma immediately implies that the SEAT algorithm is 2-competitive. First, if SEAT admits all groups of size $\leq n$, then it admits exactly the same number of people as the optimal seating algorithm. Second, if SEAT admits at least $n/2$ people, we know that the optimal seating algorithm can seat at most n people. Hence $n/2 > k/2$, as required.

(b) Unfortunately, the SEAT algorithm does not work perfectly. Show that SEAT is not optimal by giving a counterexample in which, asymptotically as n gets large, the ratio between greedy seating and optimal seating approaches $1/2$.

Solution: Consider groups $G = \{(n+2)/2, n/2, n/2\}$. Notice that the greedy seating algorithm admits the group of size $(n+2)/2$, and then cannot admit any of the other groups. The optimal algorithm admits the two groups of size $n/2$, filling all n seats. Notice that asymptotically $((n+2)/2)/n$ approaches $1/2$ as n gets large.

When you present your results to the GreedSox owners, they point out the following problem: unlike numbers in a computer's memory, real people are hard to move around. In particular, people waiting in line do not like to be "sorted." The GreedSox owners ask you to develop a version of the greedy seating algorithm that does not modify the set G. (You can think of G as being stored in read-only memory.) You suggest the following algorithm:

```
RESEAT(G[1..m], n)
 1  admitted ← 0
 2  remaining ← n
 3  for j ← 1 to ⌈lg n⌉
 4       do for i ← 1 to m
 5            do if G[i] ≥ n/2^j and G[i] ≤ remaining
 6                 then ADMIT(i)
 7                      remaining ← remaining − G[i]
 8                      admitted ← admitted + G[i]
 9                 else if G[i] > remaining
10                        then REJECT(i)
11  return admitted
```

The RESEAT algorithm iterates through the list of groups several times. In the first iteration, it admits any group of size at least $n/2$. In the second iteration, it admits any group of size at least $n/4$. It continues in the same manner seating smaller and smaller groups until the bleachers are filled. When RESEAT finishes, it returns the number of people admitted.

(c) Assume that, given G and n, it is possible to admit at least k people. Show that the RESEAT algorithm still seats at least $k/2$ people.

Solution:

The argument in this case is quite similar to the proof in part (a). We begin by reproving the same lemma about the number of people admitted by the algorithm RESEAT.

Lemma 3 *Algorithm* RESEAT *either admits all groups of size* $\leq n$*, or it admits* $\geq n/2$ *people.*

Proof. Assume algorithm RESEAT does *not* admit all groups of size $\leq n$. That is, there is some group g_i of size $\leq n$ that RESEAT does not admit.

There are two cases to consider. First, assume that $g_i \geq n/2$. Then, since the algorithm first considers all groups of size $\geq n/2$ (when $j = 1$), we know that it must have admitted some group of size $\geq n/2$; otherwise, group g_i would have been admitted in the loop when $j = 1$. Therefore, we can conclude that the algorithm seats $\geq n/2$ people, as required.

For the second case, assume that $g_i < n/2$. Notice that when $j = \lceil \lg n \rceil$, $g_i \geq n/2^j$. Therefore, if g_i is not admitted, we can conclude that $g_i >$ *remaining*. That is, *remaining* $< g_i < n/2$. Since *remaining* is non-increasing, we thus conclude that at least $n/2$ people are seated.

As before, this lemma immediately implies that the algorithm RESEAT is 2-competitive, as desired.

(d) The RESEAT algorithm runs in $O(m \lg n)$ time. Devise a new algorithm that runs in $O(m)$ time and still guarantees that if k people can be seated, your algorithm seats at least $k/2$ people.

Solution:

```
FAST-RESEAT(G[1..m], n)
 1  admitted ← 0
 2  remaining ← n
 3  for i ← 1 to m
 4       do  if G[i] ≥ n/2 and G[i] ≤ remaining
 5             then ADMIT(i)
 6                  admitted ← admitted + G[i]
 7                  remaining ← remaining − G[i]
 8  for i ← 1 to m
 9       do  if G[i] ≤ remaining
10             then ADMIT(i)
11                  admitted ← admitted + G[i]
12                  remaining ← remaining − G[i]
13  return admitted
```

The argument showing this algorithm is correct is essentially the same as the proof in part (c). In particular, we again show the same lemma about the number of people seated.

Lemma 4 *Algorithm* FAST-RESEAT *either admits all groups of size* $\leq n$*, or it admits* $\geq n/2$ *people.*

Proof. Assume algorithm FAST-RESEAT does *not* admit all groups of size $\leq n$. That is, there is some group g_i of size $\leq n$ that FAST-RESEAT does not admit.

There are two cases to consider. First, assume that $g_i \geq n/2$. Then, since the algorithm first considers all groups of size $\geq n/2$, we know that if g_i was rejected, it must have admitted some group of size $\geq n/2$; otherwise, group g_i would have been admitted. Therefore, we can conclude that the algorithm seats $\geq n/2$ people, as required.

For the second case, assume that $g_i < n/2$. If g_i is not admitted, we can conclude that $g_i >$ *remaining*. That is, *remaining* $< g_i < n/2$. Since *remaining* is non-increasing, we thus conclude that at least $n/2$ people are seated.

As before, this lemma immediately implies that the algorithm FASTRESEAT is 2-competitive, as desired.