Introduction to Algorithms November 7, 2005
Massachusetts Institute of Technology 6.046J/18.410J
Professors Erik D. Demaine and Charles E. Leiserson Handout 22

Problem Set 7

MIT students: This problem set is due in lecture on **Monday, November 14, 2005.** There will be two homework labs for this problem set, one held 6–8 P.M on Wednesday, November 9, 2005 and one held 2–4 P.M. on Sunday, November 13, 2005.

Reading: Chapter 15, 16.1–16.3, 22.1, and 23.

Problem 7-1 is mandatory. Failure to turn in a solution will result in a serious and negative impact on your term grade! Both exercises and problems should be solved, but *only the problems* should be turned in. Exercises are intended to help you master the course material. Even though you should not turn in the exercise solutions, you are responsible for material covered in the exercises.

Mark the top of each sheet with your name, the course number, the problem number, your recitation section, the date and the names of any students with whom you collaborated. **Please staple and turn in your solutions on 3-hole punched paper**.

You will often be called upon to "give an algorithm" to solve a certain problem. Your write-up should take the form of a short essay. A topic paragraph should summarize the problem you are solving and what your results are. The body of the essay should provide the following:

1. A description of the algorithm in English and, if helpful, pseudo-code.
2. At least one worked example or diagram to show more precisely how your algorithm works.
3. A proof (or indication) of the correctness of the algorithm.
4. An analysis of the running time of the algorithm.

Remember, your goal is to communicate. Full credit will be given only to correct solutions *which are described clearly*. Convoluted and obtuse descriptions will receive low marks.

Exercise 7-1. Do Exercise 15.4-3 on page 356 of CLRS.

Exercise 7-2. Do Exercise 16.1-3 on page 379 of CLRS.

Exercise 7-3. Do Exercise 16.3-2 on page 384 of CLRS.

Exercise 7-4. Do Exercise 22.1-5 on page 530 of CLRS.

Exercise 7-5. Do Exercise 23.1-5 on page 566 of CLRS.

Exercise 7-6. Do Exercise 23.2-4 on page 574 of CLRS.

Exercise 7-7. Do Exercise 23.2-5 on page 574 of CLRS.

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`	alyo̲rithm	4	4
replace by `s`	alysr̲ithm	4	8
replace by `i`	alysii̲thm	4	12
replace by `s`	alysist̲hm	4	16
delete	alysish̲m	2	18
delete	alysism̲	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.

(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.

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

(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.

(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.

(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 = `"electrical engineering"`,
y = `"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.

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 operations 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.)

(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.

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

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

(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 $1/2$.

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

(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.