Introduction to Algorithms November 27, 2002
Massachusetts Institute of Technology 6.046J/18.410J
Professors Erik Demaine and Shafi Goldwasser Handout 25

Problem Set ∞

This is a *make-up* problem set, only for students who are missing several problems. It is due to your TA on **Monday, December 2.**

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.

Each problem should be done on a separate sheet (or sheets) of three-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 your essay should provide the following:

1. A description of the algorithm in English and, if helpful, pseudocode.
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. Graders will be instructed to take off points for convoluted and obtuse descriptions.

Problem ∞-1. Suppose you are given a set S of n tasks, where task i requires p_i units of processing time to complete, once it has started. You have one computer on which to run these tasks, and the computer can run only one task at a time.

A ***schedule*** assigns which tasks to run during what times on the computer. For any schedule, let c_i denote the ***completion time*** of task i, that is, the time at which task i completes processing. Your goal is to find a schedule that minimizes the average completion time, that is, one that minimizes

$$\frac{1}{n}\sum_{i=1}^{n} c_i \; .$$

(a) Suppose there are two tasks with $p_1 = 3$ and $p_2 = 5$. Consider (1) the schedule in which task 1 runs first, followed by task 2 and (2) the schedule in which task 2 runs first, followed by task 1. In each case, state the values of c_1 and c_2 and compute the average completion time.

(b) Give an algorithm that schedules the tasks so as to minimize the average completion time. Each task must run nonpreemptively, that is, once task i is started, it must run continuously for p_i units of time. Prove that your algorithm minimizes the average completion time, and state the running time of your algorithm.

Suppose now that the tasks are not all ready at once. Specifically, each task has a ***release time*** r_i before which it is not available to be processed. Suppose also that we allow ***preemption***, so that a task can be suspended and restarted at a later time. For example, a task i with processing time $p_i = 6$ may start running at time 1 and be preempted at time 4. It can then resume at time 10 but be preempted at time 11 and finally resume at time 13 and complete at time 15. Task i has run for a total of 6 time units, but its running time has been divided into three pieces.

(c) Give an algorithm that schedules the tasks so as to minimize the average completion time in this new scenario. Prove that your algorithm minimizes the average completion time, and state the running time of your algorithm.

Problem ∞-2. You and your friends thought that playing Paintball over a network of platforms situated in the jungle would be a good way to forget about 6.046 for a few hours. No such luck! The platforms are connected by rickety bridges, each of which has some probability of failure (the bridges fail independently of each other). As every Paintballer knows, the goal of each player is to get from platform s to platform t in one piece in order to gain a better defensive or offensive position. You've created a graph with each platform as a node and each bridge as an edge in order to better analyze your situation.

Give a polynomial-time algorithm to find paths from a specified node s to all other nodes, so that every path has minimum failure probability (a path fails if any of the bridges on that path fail).

Problem ∞-3. We are given an input array A of length n and an integer m such that $m < n/2$. We have to compute the array *mins* defined as follows:

$$\textit{mins}[i] = \min\{A[i], A[i+1], \ldots, A[i+m-1]\}, \text{ for all } 1 \le i \le n-m+1$$

A naive approach to this would be to calculate the minimum for each set of m elements independently. This takes $\Omega(m)$ time for the computation of each *mins*$[i]$ and hence the total time to produce the entire array *mins* would be at least $(n-m+1)\Omega(m) = \Omega((n-m)m)$, which is $\Omega(nm)$ for $m < n/2$. In this problem, we would like to build an algorithm that does better.

A useful way to view the problem is to imagine sliding a window of size m across the input array A. (See Figure 1.) We can have $n-m+1$ different positions for the window; the first position consists of integers $A[1], A[2], \ldots, A[m]$, the second position consisting of elements $A[2], A[3], \ldots, A[m+1]$, and so on. For each position of the window, we would like to find the minimum of all the elements in the window.

(a) To begin with, describe an algorithm that computes the array *mins* in time $O(n \lg m)$.

We now describe an algorithm that takes $O(1)$ amortized time to compute a single element of the array *mins*.

We color the elements in a window either red or green in a fashion mentioned below. We also maintain two minima: *min-red*, which is the minimum value among all the red-colored elements in

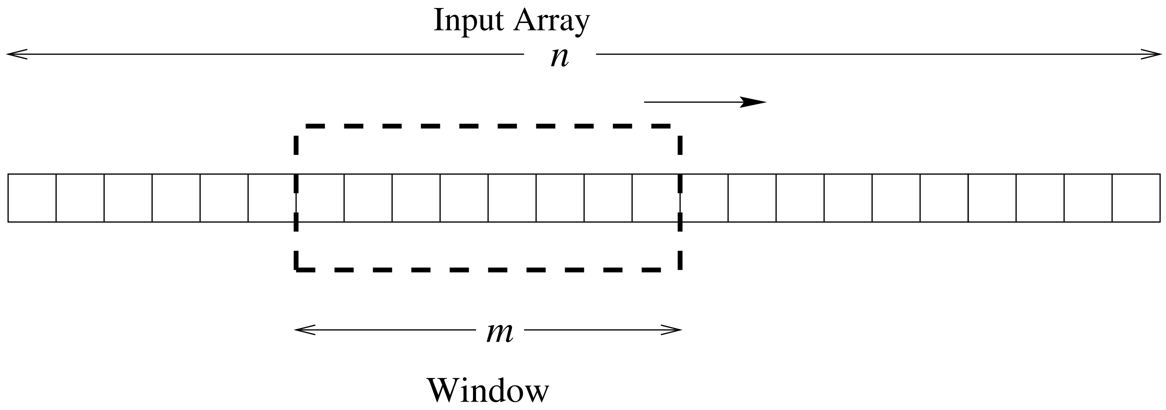

Figure 1: Problem 6-3: Window on an array of numbers

the window, and *min-green*, which is the minimum value of all the green-colored elements in the window. Thus, the overall minimum in the window is the minimum of *min-red* and *min-green*. (See COMPUTE-MINS:line 8 below.) Each red-colored element carries with it a field *min-ahead*, which is the minimum value among all the red-colored elements ahead of it in the window. However, the green-colored elements do not carry with them any such field.

Whenever the window slides past an element (performed by subroutine SLIDE), the incoming element is colored green (SLIDE:line 1) and *min-green* is updated (SLIDE:line 2). If the outgoing element is red, we can easily update *min-red* based on the value of *min-ahead* of the outgoing red-element. However, if the outgoing element is green, we then color all the elements in the window red (see subroutine COLOR-RED) and update *min-ahead* of all the elements in the window. Initially, all the elements in the first window (ie., elements $A[1], A[2], \ldots, A[m]$) are colored green and both *mins*$[1]$ and *min-green* are set to the value of the minimum element in the first window (COMPUTE-MINS:lines 2-5). The pseudo-code for the algorithm is given below.

```
COMPUTE-MINS(A, n, m)
1  min-green ← ∞
2  for i ← 1 to m
3      do color[i] ← green
4         min-green ← min(min-green, A[i])
5  mins[1] ← min-green
6  for i ← m + 1 to n
7      do SLIDE(i)
8         mins[i − m + 1] ← min(min-red, min-green)
```

```
SLIDE(i)
color[i] ← green
min-green ← min(min-green, A[i])
if color[i − m] = green
   then COLOR-RED(i)
min-red ← min-ahead[i − m]
```

```
COLOR-RED(i)
min-green ← ∞
min-red ← ∞
for j ← i downto i − m
      do color[j] ← red
         min-ahead[j] ← min-red
         min-red ← min(min-red, A[j])
```

(b) What is the worst case time taken to compute the value of a single element of the array *mins* (i.e., what is the worst case time taken by SLIDE)?

(c) Show that the time taken by the entire algorithm is $O(n)$ and thus the amortized cost to find the minimum for each set of m consecutive elements is $O(1)$.

Problem ∞-4. Let Σ be the alphabet $\{a, b, c\}$, and suppose the elements of Σ have the following multiplication table:

Right Hand Symbol

Left Hand Symbol

	a	b	c
a	b	b	a
b	c	b	a
c	a	c	c

Note that this multiplication is neither associative nor commutative.

Give an efficient dynamic programming algorithm that examines a string $x_1 x_2 \ldots x_n$ of charecters of Σ and decides whether or not it is possible to parenthesize x in such a way that value of the resulting expression is a. For instance, if $x = bbbba$, your algorithm should return "yes" because $(b(bb))(ba) = a$. (This expression is not unique. For example, $(b(b(b(ba)))) = a$ as well.) Analyze the running time in terms of n. Explain how to modify your algorithm to return a parenthesization that yields a, if one exists.