Massachusetts Institute of Technology
6.046J/18.410J: Introduction to Algorithms
Professors Piotr Indyk and Bruce Tidor

Handout 10
February 25, 2003

Problem Set 3

This problem set is due **at the beginning of** class on *Thursday, March 20, 2003*. Note that you have three weeks to do it in.

Each problem is to be done on a separate sheet (or sheets) of paper. Mark the top of each sheet with your name, 6.046J/18.410J, the problem number, your recitation section, the date, and the names of any students with whom you collaborated.

Problem 3-1. Priorities

Brian Dean is a VERY busy TA. He often has so many things to do he can't even figure out which one he should do first. Nitin Thaper has offered to help him by assigning 'priority' numbers to each of Brian's tasks. Nitin guarantees no two tasks will have the same priority. Now, content with at least having priorities, Brian writes all his tasks down on paper, and simply works on the one that is most important until he finishes it.

As new tasks appear, Nitin is more than happy to assign a 'priority' number to each of them. But Brian would like to be somewhat more efficient in how he processes all that data. Assume that there are n tasks Brian is working on. Each has an integer associated with them that represents the priority. Your task is to analyze the running time of each of these ideas (do not provide an algorithm unless we ask how):

(a) If Brian were to store all his tasks in an array, along with the priority values assigned to them:

- How long does it take to search for a task of specific priority?
- How long does it take to add one more task?
- How long does it take to delete a task? (You can assume you already know where the task is, so searching is unnecessary). Explain how you would do so.
- How long does it take to find the most important task?

(b) If Brian were to store all his tasks in a sorted array, along with the priority values assigned to them (the sorting is done based on priority):

- How long does it take to search for a task of specific priority?
- How long does it take to add one more task?
- How long does it take to delete a task?
- How long does it take to find the most important task?

(c) Mihai (who loves complicated ideas) suggests the following: Store the data in multiple arrays. Let $k = \lceil \lg(n+1) \rceil$. Let the binary representation of $n = \langle n_{k-1}, n_{k-2}, \ldots, n_0 \rangle$. We will keep k sorted arrays $A_0, A_1, \ldots, A_{k-1}$, where for $i = 0, 1, \ldots, k-1$, the length of array A_i is 2^i. Each array is either full or empty, depending of whether $n_i = 1$

or $n_i = 0$. The total number of elements in all k arrays is therefore $\sum_{i=0}^{k-1} n_i 2^i = n$. Each of these arrays is kept sorted, but there is no relationship between elements in different arrays.

- How long does it take to search for a task of specific priority?
- How long does it take to add one more task? Analyze the *worst-case* for an addition, and analyze the *amortized* time for a continuous sequence of additions (no deletions in the middle), starting from an empty data structure. Describe how one performs an addition.
- How do you perform a deletion in time $\Theta(n)$?
- How long does it take to find the most important task? Describe how one does this.

(d) Yoav (whom we last saw sorting nuts and bolts) suggests that Brian store his tasks in a balanced BST, with the keys being the priorities. Additionally, one more pointer should be used to keep track of the position in the tree of the item with the highest priority:

- How long does it take to search for a task of specific priority?
- How long does it take to add one more task? Describe how one does this.
- How long does it take to delete a completed task? Describe how.
- How long does it take to find the most important task?

Problem 3-2. Dynamic random number generator

You want to build a dynamic random number generator. The generator should maintain an array $W[1 \ldots n]$ of non-negative (but not necessarily integer) *weights*. The generator should support the following operations:

- *Modify*(i, w): assigns $W[i] = w$
- *Generate*: returns a random number from $\{1 \ldots n\}$, such that the probability of returning i is equal to $\frac{W[i]}{\sum_j W[j]}$

Show how to implement the generator such that both operations take $O(\log n)$ time. You can use procedure *Rand*(a, b), which (in unit time) returns a number r chosen uniformly at random from the (continuous) interval $[a, b]$. The initial state of the data structure should be defined by weights $W[1] = 1$, $W[i] = 0$, for $i > 1$, but you can ignore the time needed to initialize the data structure.

Problem 3-3. Product Finder

The following problem should be highly reminscent of problem 1-3.

Design an algorithm, which, given an input array $A[1], \ldots, A[n]$ of different integers from the range $\{1 \ldots n^4\}$, and a target integer value x, prints *all* pairs (i, j) such that $A[i] + A[j] = x$.

Your algorithm should have an expected running time of $O(n)$. Your algorithm should allocate no more space than $O(n)$.

You do not need to prove correctness, but you should justify the running time. Note that the running time is different from before. Also, again note that we want the indices, not the values in the array.

Food for thought: Assume now that you would like to check if there is any *triple* (i, j, k) such that $A[i] + A[j] + A[k] = x$. Can you design an algorithm that solves this task in time $O(n^{2-\epsilon})$ for some constant $\epsilon > 0$? Note: none of us can. If you discover one, apply for a professor position at MIT immediately.

Problem 3-4. Professor Indyk's Sock Drawer

Professor Indyk does his laundry fortnightly. Because he hates matching socks after the laundry session, he instead grabs all the socks and throws them in his special sock drawer. The reason his sock drawer is special is because until Professor Indyk pulls out a sock, he has no clue which one he will end up with, and the sock is always a random sock from those remaining.

Say Professor Indyk has n pairs of socks, each unique from the other pairs. Professor Indyk's idea is to pull out a sock, then pull another, and on and on, until he finds at least one pair that matches, at which point he returns all the other socks to the drawer, and puts on the matching pair.

Clearly, this algorithm has a worst-case number of sock pulls of $\Theta(n)$ (On an unlucky day, Prof Indyk could indeed pull out one of every sock). However, what is the *expected* number of pulls before getting a matching pair? Note: we are only interested in the asymptotic (i.e., $\Theta(\cdot)$) answer.

Optional: Assume now that the socks do not have to come in pairs. Instead, we only know that there are $2n$ socks, of at most n different colors. What is the answer in this case ?

Optional Extra Credit: To improve the running time, volunteer to sort Professor Indyk's socks for him.