Introduction to Algorithms February 24, 2004
Massachusetts Institute of Technology 6.046J/18.410J
Professors Erik Demaine and Shafi Goldwasser Handout 8

Problem Set 2

This problem set is due **in class** on **Tuesday, February 24**.

Reading: Chapters §5.1-5.3, 7, 9

There are **four** problems. Each problem is to be done on a **separate sheet** (or sheets) of paper. 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.

You will often be called upon to "give an algorithm" to solve a certain problem. Giving an algorithm entails:

1. A description of the algorithm in English and, if helpful, pseudocode.
2. A proof (or argument) of the correctness of the algorithm.
3. An analysis of the running time of the algorithm.

It is also suggested that you include at least one worked example or diagram to show more precisely how your algorithm works. Remember, your goal is to communicate. Graders will be instructed to take off points for convoluted and obtuse descriptions. If you cannot solve a problem, give a brief summary of any partial results.

Problem 2-1. Fuzzy Sorting of Intervals **(Extra Credit)**

Consider the sorting problem where all the numbers are not known exactly. Instead, for each number, you know an interval on the real line to which it belongs. That is, you are given n closed intervals of the form $[a_i, b_i]$ where $a_i \leq b_i$. Assume that no interval contains any other interval. That is, if $a_i \leq a_j$ then $b_i \leq b_j$. You are asked to ***fuzzy-sort*** these intervals, i.e., produce a permutation $\langle i_1, i_2, \ldots, i_n \rangle$ of the intervals, such that for all j, there exists a $c_j \in [a_{i_j}, b_{i_j}]$ satisfying $c_1 \leq c_2 \leq \cdots \leq c_n$.

Suppose that each interval is guaranteed to overlap at least $d-1$ other intervals. Give an $O(n \log \frac{n}{d})$ algorithm to fuzzy-sort n intervals with d degrees of overlap. Thus, if $d = \Theta(1)$, the algorithm runs in $O(n \log n)$ time, and if $d = \Theta(n)$, the algorithm runs in $O(n)$ time.

Solution:
Overview:
We adopt a divide-and-conquer strategy similar to Quicksort. For each call of our algorithm FUZZY SORT, we pick an interval as our "pivot" and partition the intervals into two groups: a group of intervals to the left of the pivot and a group to the right. We call FUZZY SORT recursively

on each group until the size of the group is at most d, in which case the group will be sorted with a procedure in $O(d)$ time.

The invariant of the algorithm will be that each set S of intervals can be split into three groups—left, middle, and right—satisfying the following properties:

1. The left group is the "smallest" group. In other words, any interval in the left group can be listed before any interval in the middle group or right group in the sorted output.
2. The right group is the "largest" group. In other words, any interval in the right group can be listed after any interval in the middle group or left group in the sorted output.
3. Every pair of intervals in the left group overlap each other.
4. Every interval in the middle group overlaps at least $d-1$ other intervals (not necessarily from the middle group).
5. Every pair of intervals in the right group overlap each other.

Algorithm Description:
The algorithm is based on Deterministic Quicksort. We consider the left endpoints of all the intervals and pick the median in $O(n)$ time. We pick the corresponding interval as our pivot and partition the intervals according to the left endpoints. This divides the original problem into two subproblems, namely the left subproblem and the right subproblem, each with size $\lfloor \frac{n}{2} \rfloor$.

Since all the intervals in the left subproblem have left endpoints smaller than the pivot, all of them can be listed before the pivot in the output. Similarly, all the intervals in the right group can be listed after the pivot in the output. As a result, FUZZY SORT is recursively called on the left subproblem and the right subproblem independently. Combining the two sorted results along with the pivot will take $O(n)$ time.

We argue that the invariant is preserved by the recursive calls. Recall that the invariant states that the set of intervals can be partitioned into three groups with aforementioned properties. Initially, the invariant holds because the left and right groups are empty and the middle group consists of the entire set of intervals. Now assume that the invariant holds for the set S of intervals on a subsequent call of FUZZY SORT. By induction hypothesis, S can be partitioned into S_L, S_M, S_R which are the desired left, middle and right groups respectively. We argue that after the partitioning of S, the invariant is perserved for both the left subproblem and the right subproblem. Here we consider the left subproblem.

Let S^l be the set of intervals of the left subproblem. Consider the following two cases:

1. The pivot is in S_L. We set $S^l_L := S^l, S^l_R := \emptyset, S^l_M := \emptyset$. We have the left, right and middle groups with the desired properties.
2. The pivot is not in S_L. This implies $S_L \subset S^l$. Let T be the set of intervals in S^l that overlap the pivot. We set $S^l_L := S_L, S^l_R := T \backslash S^l_L$ and $S^l_M := S^l \backslash (S^l_L \cup S^l_R)$. We also have the left, right and middle groups with the desired properties.

Therefore, the invariant holds for the left subproblem. Similarly, the invariant holds for the right subproblem as well.

We next claim that a problem of size at most d can be solved in linear time. Suppose we have a set of at most d intervals. By the invariant, the intervals can be partitioned into three groups. There are two cases. If the middle group is empty, then a valid sorting is to list the left group in arbitrary order followed by the right group in arbitrary order. If the middle group has at least one interval, then that interval overlaps every other interval by Condition 2 (because there are only $d-1$ other intervals). Therefore the sorting is again easy: list all intervals overlapping the left endpoint of the interval, then list the interval, then list all intervals overlapping the right endpoint of the interval.

Therefore we can cut the recursion tree when we get to problems of size at most d, so the depth is $\Theta(\lg(n/d))$, so the running time is $O(n \lg(n/d))$.

Problem 2-2. Randomized Quicksort Variants

Regularly, RANDOMIZED QUICKSORT selects a single pivot element uniformly at random. In this problem you will analyze two possible modifications to this choice of pivot.

(a) For an integer parameter $k \geq 1$, the HYBRID RANDOMIZED QUICKSORT algorithm uses regular RANDOMIZED QUICKSORT whenever $n > k$, but uses INSERTION SORT whenever $n \leq k$. Analyze the expected running time of HYBRID RANDOMIZED QUICKSORT in terms of n and k. For what values of k does the algorithm run in $O(n \lg n)$ time?

Solution:

Let S_i represent a sublist of size $i \in [1, k]$ created by a RANDOMIZED QUICKSORT pivot. The sublist S_i will be a leaf in the RANDOMIZED QUICKSORT recursion. We can find the runtime contribution of INSERTION SORT by tallying the expected work done over all sublist leaves.

Since the pivot was chosen uniformly at random, all resulting sublist lengths in the range $[1, k]$ are equally likely and $E[|S_i|] = k/2$. If the expected size of an arbitrary S_i is $k/2$, then we expect there to be $n/(k/2) = 2n/k$ sublist leaves in the recursion tree. Each leaf of size i takes time ci^2 to process, therefore the contribution of work done by INSERTION SORT is $(2n/k) * c(k/2)^2 = cnk/2 = O(nk)$.

Now we wish to find the expected contribution of work by RANDOMIZED QUICKSORT. If INSERTION sort were not called on the leaves of expected size $k/2$, RANDOMIZED QUICKSORT would have continued for an expected $O(\log k)$ more levels. Thus, the hybrid scheme yields an expected $O(\log n - \log k) = O(\log \frac{n}{k})$ recursion depth. At each level of recursion we perform $O(n)$ work, so total expected contribution of RANDOMIZED QUICKSORT is $O(n \log \frac{n}{k})$.

Thus the total amount of work done by the algorithm is $O(nk + n \log \frac{n}{k})$. If $k = 1$, this is just regular QUICKSORT, so takes time $O(n \log n)$. If $k = n$, this is just INSERTION SORT and takes time $O(n^2)$.

In the CAUTIOUS RANDOMIZED QUICKSORT algorithm, the pivot is chosen by repeatedly selecting random candidate pivots and stopping only once a "good" pivot is found. A pivot is *good* if it partitions an array of n elements into two subarrays each with at least $n/3$ elements. To determine whether a pivot is good, CAUTIOUS RANDOMIZED QUICKSORT runs the PARTITION subroutine, computes the split, and if the split is not good it undoes the actions made by PARTITION in linear time.

(b) What is the probability of selecting a good pivot after a single trial?

Solution: The probability of selecting a good pivot is $1/3$, because there are $n/3$ elements that produce partitions of size at least $n/3$.

(c) What is the maximum recursion depth of CAUTIOUS RANDOMIZED QUICKSORT?

Solution: Because this algorithm always produce a subproblem of size at most $2n/3$, the maximum depth is $O(\log_{\frac{3}{2}} n) = O(\log n)$.

(d) What is the expected running time of CAUTIOUS RANDOMIZED QUICKSORT?

Solution: Testing whether a pivot is good takes $O(n)$ time. On each level of recursion, the expected number of random selections until a good pivot is found is three. So, the expected amount of work done on each recursive level is $O(n)$ and the depth of the recursion is $O(\log n)$. Therefore, this algorithm still has an $O(n \log n)$ expected runtime.

If an adversary controls the random choices, they can force you to always choose a bad pivot. If you didn't keep track of previously chosen pivots, they could force you to run forever. However if you mark prior choices, they can force you to select at most $2n/3$ points on each level of recursion. Testing each pivot takes $O(n)$ time, so your algorithm would perform $O(n^2)$ work on each level and thus run in time $O(n^2)$.

Problem 2-3. In-Place Median

You are given a DVD-ROM storing n values and wish to find the median. Since the disk is read-only, you cannot swap or move elements. Your computer has $O(\log n)$ read/write memory, so you can't simply copy the DVD into memory. Give an $O(n \log n)$ expected-time algorithm to find the median without writing to disk.

Solution:

IN-PLACE-MEDIAN(Array A):
1 $low \leftarrow \min_i\{A[i]\}$
2 $high \leftarrow \max_i\{A[i]\}$
3 **while** $(low < high)$:
4 $size \leftarrow$ number of elements in A with values in $[low, high]$
5 Select a random $r \in [1, size]$.
6 Let p be the rth element in A among those with value in $[low, high]$.
7 $SizeLow \leftarrow$ number of elements in A with values at most p
8 **if** $(SizeLow \geq n/2)$ **then** $high \leftarrow p$ **else** $low \leftarrow p$
9 **return** low

This code first counts the number of elements between the low and $high$ values. Then it selects a random element between those values, denoted p. The algorithm then counts the values between low and p. If p is less than the median, we continue with $low \leftarrow p$. Otherwise, we continue with $high \leftarrow p$.

Each iteration performs $O(n)$ work. Let us call an iteration *successful* if it reduces the number of candidates to at most $3/4$ of its original size. The probability of any given iteration being successful is at least $(1 - 2 \cdot (1/4)) = 1/2$. Therefore, the expected number of trials up to and including a successfull iteration is at most 2, and since $\log_{4/3} n$ successful iterations are required for termination, we have $O(\log n)$ expected iterations.

Problem 2-4. Clothing Store

You decide to open a "Short'n'Tall" clothing store, catering only to very short and very tall people. As part of market research, you measure n people who arrive at times $a_1, \ldots, a_n$. Their heights are the positive numbers $h_1, \ldots, h_n$. (Person i arrives at time a_i and has height h_i.) Neither the arrival times nor the heights are sorted.

(a) You decide that your store will only sell clothing to the top and bottom $(1/k)$th of the population. You wish to separate out the tallest n/kth people and the shortest n/kth people. Give an $O(n)$-time algorithm to find both of these groups.

Solution: Run the linear time SELECT algorithm to find the n/kth highest and lowest order statistics. The two groups can then be separated out in $O(n)$ time.

After a lawsuit from the Association of Average People, you decide to make custom clothes for people of all heights. Suppose that the amount of material to clothe a person is proportional to their height. If $\sum_{i=0}^{n} h_i = C$, then C units of material can clothe all n people. Unfortunately you have only $C/2$ units of material on hand.

You decide to clothe as many people as possible, but give priority to those who arrived first. As a result, everyone under the *weighted median* will get their clothes. A weighted median is an index k such that:

$$\sum_{i:a_i<a_k} h_i \leq \frac{C}{2} \quad and \quad \sum_{j:a_k<a_j} h_j \leq \frac{C}{2}$$

(b) Give an algorithm to compute the weighted median of n elements in $O(n \lg n)$ worst-case time.

Solution: We first sort the n elements in increasing order by a_i. Then we scan the array of sorted a_i's, starting with the smallest element and accumulating h_i values as we scan until the total exceeds $C/2$. The last element, a_k, whose h_i value caused the total to exceed $C/2$, is the weighted median. Notice that the total weight of all elements smaller than a_k is less than $C/2$, because a_k was the first element that caused the total weight to exceed $C/2$. Similarly, the total weight of all elements larger than a_k is also less than $C/2$, because the total weight of all the other elements exceeds $C/2$.

The sorting phase can be done in $O(n \lg n)$ worst-case running time (using Merge Sort or Heapsort), and the scanning phase takes $O(n)$ time. The total worst-case running time is therefore $O(n \lg n)$.

(c) Give an algorithm to compute the weighted median in $\Theta(n)$ worst-case time.

Solution:

We find the weighted median in $O(n)$ worst-case time using the $O(n)$ deterministic median algorithm. The algorithm works by first finding the actual median a_k of the n elements, and partitioning around it. It then computes the total weights of the two halves. If the weights of the two halves are each less than $C/2$, then the weighted median is $p = a_k$. Otherwise, the weighted median should be in the half with total weight exceeding $C/2$. The total weight of the "light" half is lumped to a_k, and the search continues on the half that weighs more than $C/2$. The sketch of the code for the algorithm follows.

```
1  Find the median a_k of a_1, a_2, ..., a_n
2  Partition around a_k
3  Compute H_L = sum_{a_i<a_k} h_i and H_G = sum_{a_i>a_k} h_i
4  if H_L < C/2 and H_G < C/2
5     then return a_k
6  if H_L > C/2
7     then Change h_k to h_k + H_G
8          Recurse on the elements ≤ a_k
9     else Change h_k to h_k + H_L
10         Recurse on the elements ≥ a_k
```

The recurrence for the worst-case running time of the algorithm is $T(n) = T(n/2) + O(n)$, since there is only one recursive call on half the number of elements, and all the extra work can be done in $O(n)$ time. The solution of the recurrence is $T(n) = O(n)$.