Introduction to Algorithms September 25, 2002
Massachusetts Institute of Technology 6.046J/18.410J
Professors Erik Demaine and Shafi Goldwasser Handout 9

Problem Set 3

This problem set is due **in recitation** on **Friday, October 4.**

Reading: Chapter 9; §8.4; §11.1-11.3, 11.5

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 by 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.

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.

Exercise 3-1. Do exercise 9.3-1 on page 192 of CLRS.

Exercise 3-2. Do exercise 9.3-5 on page 192 of CLRS.

Exercise 3-3. Do exercise 11.2-1 on pages 228–229 of CLRS.

Exercise 3-4. Use induction to prove the correctness of RADIX-SORT. Where does your proof use stability of the sorting subroutine?

Problem 3-1. We have seen that COUNTING-SORT requires $O(n + k)$ time to sort n numbers in the range $1, \ldots, k$, while RADIX-SORT requires $O(nd)$ time to sort n number of d digits each. By a judicious combination of these algorithms, we can get a linear running time, as in COUNTING-SORT, when operating on elements from a wider range, as in RADIX-SORT.

(a) Given a number x, show how to get the ith digit of its base-r representation in $O(1)$ time. (The 1st digit is the least-signicant one.)

(b) What is the running time of RADIX-SORT on an array of n numbers in the range $0, \ldots, n^5 - 1$, when using base-10 representations?

(c) Give an algorithm to sort an array of n numbers in the range $0, \ldots, n^5 - 1$ in only $O(n)$ time. Does your technique extend to wider ranges of elements? Explain.

Problem 3-2. An array $A[1, \ldots, n]$ has a ***majority element*** x if x appears more than $n/2$ times in A. (Clearly, not every array has a majority element.)

(a) Give an $O(n)$-time algorithm that returns the majority element of an n-element array, or $\perp$ if none exists.

(b) Given an array, a $1/k$***-plurality element*** is one which appears in the array more than n/k times. (Note that an array may have zero, one, or several $1/k$-plurality elements.) Give an $O(nk)$-time algorithm that takes an array of length n, and returns all of its $1/k$-plurality elements (or $\perp$ if none exist).

Problem 3-3. There are many useful operations we might want to perform on sets, where all elements are distinct. For this problem, assume that a set is represented as an unordered array.

(a) Suppose you are given a set S of integers and another integer x, and you want to know if there exist two elements in S whose sum is x. (We saw in problem set 1 how to solve this problem using sorting in $(n \log n)$ time.)

Give a randomized algorithm to solve this problem that runs in $O(n)$ *expected*-time.

(b) Suppose you are given two sets S and T, and you want to determine if $S = T$, i.e. if they contain exactly the same elements.

Using sorting, give a deterministic algorithm to solve this problem in $O(n \log n)$ time. Then give a *randomized* algorithm to solve this problem that runs in $O(n)$ *expected*-time. *Hint:* S and T are equal if $S \subseteq T$ and $T \subseteq S$.

Problem 3-4. Let $\mathcal{H} = \{h\}$ be a class of hash functions in which each h maps the universe U of keys to $\{0, 1, \ldots, m-1\}$. We say that $\mathcal{H}$ is 2-***universal*** if, for every fixed pair (x, y) of keys where $x \neq y$, and for h chosen randomly from $\mathcal{H}$, the pair $\langle h(x), h(y) \rangle$ is equally likely to be any of the m^2 pairs of elements from $\{0, 1, \ldots, m-1\}$. (The probability is taken over *only* the random choice of the hash function.)

(a) Show that if $\mathcal{H}$ is 2-universal, then it is universal.

(b) Construct a family $\mathcal{H}$ that is universal, but not 2-universal (justify your answer). Write down the family as a table, with one column per key, and one row per function. Try to make m, $|\mathcal{H}|$, and the number of keys as small as possible.

(c) Suppose there is an adversary who controls which keys we hash, and wants to force a collision. Furthermore, the adversary knows the family $\mathcal{H}$. Suppose $\mathcal{H}$ is universal, and the following scenario takes place: we choose a hash function h randomly from $\mathcal{H}$ (keeping it secret from the adverary), then the adversary chooses a key x and learns the value $h(x)$. Can the adversary now force a collision? In other words, can it find a $y \neq x$ such that $h(x) = h(y)$ with probability greater than $1/m$?

If so, write down a particular universal hash family (in the same format as in part **(b)**), and describe how an adversary can force a collision in this scenario. *If not*, prove that the adversary cannot force a collision with probability greater than $1/m$.

(d) Answer the question from part **(c)** if $\mathcal{H}$ is 2-universal.

Problem 3-5. (Optional, extra credit.) For n distinct elements $x_1, x_2, \ldots, x_n$ (not necessarily in sorted order) with positive weights $w_1, w_2, \ldots, w_n$ such that $W = \sum_{i=1}^{n} w_i$, the ***weighted 3-median*** is the element x_k satisfying

$$\sum_{x_i < x_k} w_i \leq \frac{W}{3}$$

and

$$\sum_{x_i > x_k} w_i \leq \frac{2W}{3} .$$

(a) Give an algorithm to compute the weighted 3-median of n elements in $O(n \lg n)$ worst-case time using sorting.

(b) Give an algorithm to compute the weighted 3-median in $\Theta(n)$ worst-case time using a linear-time order statistic algorithm such as SELECT.