Problem Set 2

MIT students: This problem set is due in lecture on *Day 8*.

Reading: Chapters 6, 7, §5.1-5.3.

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 instructor and time, the date, and the names of any students with whom you collaborated.

MIT students: 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 2-1. Do Exercise 5.3-1 on page 104 of CLRS.

Exercise 2-2. Do Exercise 6.1-2 on page 129 of CLRS.

Exercise 2-3. Do Exercise 6.4-3 on page 136 of CLRS.

Exercise 2-4. Do Exercise 7.2-2 on page 153 of CLRS.

Exercise 2-5. Do Problem 7-3 on page 161 of CLRS.

Problem 2-1. Average-case performance of quicksort

We have shown that the expected time of randomized quicksort is $O(n \lg n)$, but we have not yet analyzed the average-case performance of ordinary quicksort. We shall prove that, under the assumption that all input permutations are equally likely, not only is the running time of ordinary quicksort $O(n \lg n)$, but it performs essentially the same comparisons and exchanges between input elements as randomized quicksort.

Consider the implementation of PARTITION given in lecture on a subarray $A[p \mathinner{..} r]$:

```
PARTITION$(A, p, r)$
1  $x \leftarrow A[p]$
2  $i \leftarrow p$
3  for $j \leftarrow p + 1$ to $r$
4       do if $A[j] \leq x$
5             then $i \leftarrow i + 1$
6                  exchange $A[i] \leftrightarrow A[j]$
7  exchange $A[p] \leftrightarrow A[i]$
8  return $i$
```

Let S be a set of distinct elements which are provided in random order (all orders equally likely) as the input array $A[p \mathinner{..} r]$ to PARTITION, where $n = r - p + 1$ is the size of the array. Let x denote the initial value of $A[p]$.

(a) Argue that $A[p + 1 \mathinner{..} r]$ is a random permutation of $S - \{x\}$, that is, that all permutations of the input subarray $A[p + 1 \mathinner{..} r]$ are equally likely.

Define $\delta : S \to \{-1, 0, +1\}$ as follows:

$$\delta(s) = \begin{cases} -1 & \text{if } s < x \text{ ,} \\ 0 & \text{if } s = x \text{ ,} \\ +1 & \text{if } s > x \text{ .} \end{cases}$$

(b) Consider two input arrays $A_1[p \mathinner{..} r]$ and $A_2[p \mathinner{..} r]$ consisting of the elements of S such that $\delta(A_1[i]) = \delta(A_2[i])$ for all $i = p, p + 1, \ldots, r$. Suppose that we run PARTITION on $A_1[p \mathinner{..} r]$ and $A_2[p \mathinner{..} r]$ and trace the two executions to record the branches taken, indices calculated, and exchanges performed — but not the actual array values manipulated. Argue briefly that the two execution traces are identical. Argue further that PARTITION performs the same permutation on both inputs.

Define a sequence $F = \langle f_1, f_2, \ldots, f_n \rangle$ to be an (n, k) ***input pattern*** if $f_1 = 0$, $f_i \in \{-1, +1\}$ for $i = 2, 3, \ldots, n$, and $|\{i : f_i = -1\}| = k - 1$.

Define a sequence $F = \langle f_1, f_2, \ldots, f_n \rangle$ to be an (n, k) ***output pattern*** if

$$f_i = \begin{cases} -1 & \text{if } i < k \;, \\ 0 & \text{if } i = k \;, \\ +1 & \text{if } i > k \;. \end{cases}$$

We say that a permutation $\langle s_1, s_2, \ldots, s_n \rangle$ of S ***satisfies*** a pattern $F = \langle f_1, f_2, \ldots, f_n \rangle$ if $\delta(s_i) = f_i$ for all $i = 1, 2, \ldots, n$.

(c) How many (n, k) input patterns are there? How many (n, k) output patterns are there?

(d) How many permutations of S satisfy a particular (n, k) input pattern? How many permutations of S satisfy a particular (n, k) output pattern?

Let $F = \langle f_1, f_2, \ldots, f_n \rangle$ be an (n, k) input pattern, and let $F' = \langle f'_1, f'_2, \ldots, f'_n \rangle$ be an (n, k) output pattern. Define $S|_F$ to be the set of permutations of S that satisfy F, and likewise define $S|_{F'}$ to be the set of permutations of S that satisfy F'.

(e) Argue that PARTITION implements a bijection from $S|_F$ to $S|_{F'}$. (*Hint:* Use the fact from group theory that composing a fixed permutation with each of the $n!$ possible permutations yields the set of all $n!$ permutations.)

(f) Suppose that before the call to PARTITION, the input subarray $A[p+1 \,.\,.\, r]$ is a random permutation of $S - \{x\}$, where $x = A[p]$. Argue that after PARTITION, the two resulting subarrays are random permutations of their respective elements.

(g) Use induction to show that, under the assumption that all input permutations are equally likely, at each recursive call of QUICKSORT(A, p, r), every element of S belonging to $A[p \,.\,.\, r]$ is equally likely to be the pivot $x = A[p]$.

(h) Use the analysis of RANDOMIZED-QUICKSORT to conclude that the average-case running time of QUICKSORT on n elements is $O(n \lg n)$.

Problem 2-2. Analysis of d-ary heaps

A ***d-ary heap*** is like a binary heap, but (with one possible exception) nonleaf nodes have d children instead of 2 children.

(a) How would you represent a d-ary heap in an array?

(b) What is the height of a d-ary heap of n elements in terms of n and d?

(c) Give an efficient implementation of EXTRACT-MAX in a d-ary max-heap. Analyze its running time in terms of d and n.

(d) Give an efficient implementation of INSERT in a d-ary max-heap. Analyze its running time in terms of d and n.

(e) Give an efficient implementation of INCREASE-KEY(A, i, k), which first sets $A[i] \leftarrow \max(A[i], k)$ and then updates the d-ary max-heap structure appropriately. Analyze its running time in terms of d and n.

(f) When might it be better to use a d-ary heap instead of a binary heap?