Handout #8

Problem Set 2 Due 3:05p, Tuesday, July 8, 1997

Readings

1. Handouts 6, 7.
2. Chapters 9, 10, 12.
3. Sections 8.3, 8.4.2.

Exercises

Exercises are for you to work out, with the assistance of classmates and the course staff. Please don't hand in your solutions.

1. Exercise 8.3-1.
2. Exercise 8.3-4.
3. Let X be a finite set whose elements are easy to enumerate and let Y be a nonempty subset of X of unknown cardinality. Assume we can decide, given $x \in X$, whether or not $x \in Y$. For example, X may be the set of integers between 10 and 10^{10}, and Y can contain the primes in X; in this case, a primality testing predicate $p(x)$ can tell us whether a given $x \in X$ also belongs in Y.

   Our task is to choose a random element of Y according to the uniform distribution, without computing Y's contents more than once; in the above example, we are not allowed to call $p(x)$ more than once for the same x.

   (a) Give an algorithm that solves this problem. Your algorithm may use as much additional memory as necessary. You may assume the existence of a function Uniform$(1,n)$ which returns a uniformly distributed integer in the range $[1, n]$.

   (b) The following algorithm solves this problem using $\Theta(1)$ additional memory:

```
DRAW(X,Y)
    n ← 0
    for each x ∈ X
        do if x ∈ Y
              then n ← n + 1
                   if UNIFORM(1,n)= n
                     then z ← x
    if n > 0
      then return z
      else  return "Error: Y is empty."
```

Prove by induction on the number of elements in Y that this algorithm solves our problem.

4. Exercise 9.1-5.

5. Exercise 9.2-3.

6. Exercise 10.1-2.

7. Exercise 10.3-8.

8. Exercise 12.1-4.

9. Problem 12-3.

Problems

Your problem set grade depends only on the solutions you hand in for the problems below.

1. **[10 points]** Exercise 8.4-1. A formal proof is required; an intuitive argument is not sufficient.

2. **[5 points]** Exercise 35.3-2. You need only read the first 8 lines of section 35.3, in order to solve this problem; you shouldn't have to learn anything about algorithms solving the convex hull problem. *Tip:* if there existed some algorithm which computed the convex hull in $o(n \lg n)$ time, how could you use it to sort n numbers in $o(n \lg n)$ time?

3. **[10 points]** Exercise 9.3-4.

4. **[10 points]** Consider a sorting algorithm analogous to QUICKSORT in which the partitioning is done on the basis of the most significant bit of the elements: if that bit is 1, the element goes into the right hand part of the table; if that bit is 0, the element goes into the left hand part of the table. Subsequent partition is based on the next significant bit of the elements and so on. This method is called RADIX-EXCHANGE sorting.

(a) **[4 points]** Analyze the worst case running time of this algorithm on input of size n. Assume that the elements to be sorted are in the set $S = \{0, 1, 2, \dots, 2^t - 1\}$, i.e. each of the n input numbers can be represented in t or less bits.

(b) **[4 points]** Analyze the average case running time on input of size n. Assume that any input sequence is equally likely. Is the average case behavior different from the worst case one? How important is it for the running time of our algorithm to keep the partition balanced at each step of the recursion?

(c) **[2 points]** Which algorithm would you use to sort elements picked from S: Radix-Exchange or Radix-Sort (section 9.3 of the textbook)?

5. **[5 points]** Exercise 9.4-2.

6. **[10 points]** Exercise 10.1-1.

7. **[15 points]** Exercise 10.3-1. *Tip:* in order to prove the fallacy of statement S, it is not sufficient to attempt a proof for the truthfulness of S and then fail to complete the proof successfully — it's like "proving" that the daytime sky is not blue by trying to show that it's blue and failing to do so.

8. **[20 points]** For n distinct elements $x_1, x_2, \dots, x_n$ with positive weights $w_1, w_2, \dots, w_n$ such that $\sum_{i=1}^{n} w_i = W$, the **weighted 3-median** is the element x_k satisfying

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

and

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

(a) **[5 points]** Show how to compute the weighted 3-median of n elements in $O(n \lg n)$ worst-case time using sorting.

(b) **[15 points]** Show how to compute the weighted 3-median in $\Theta(n)$ worst-case time using a linear-time median algorithm such as Select from the text.

9. **[5 points]** Exercise 12.3-5.

10. **[10 points]** In an application that requires a very large hash table, it may be impractical to store the hash table in primary memory. In particular, one might choose to store the hash table on disk, with one disk page

playing the role of one slot in the hash table. (For a review of disk storage, read pages 382–384 of the textbook.)

We presume that a disk page is large enough to hold many records. For example, a typical disk page holds 2^{12} bytes of data, and a record may contain only 2^5 bytes. We shall store the records that hash to a single page in a linear order on the page. If a page overflows because it contains too many records, the excess records are stored in an overflow area somewhere else on disk.

A SEARCH consists of hashing to the correct disk page and then linearly searching through the records on that page for one with the query key. If the page is full, then the overflow area must be searched in addition. Since the time to access a disk page is typically at least 10 milliseconds, the cost of the linear search on the page is negligible. Thus, we shall concentrate on the number of disk accesses as our cost measure. For the SEARCH operation, the cost is $\Theta(1)$ if we find the record in its "proper" page, but it may be considerably greater if we must in addition search the overflow area. Consequently, the goal of this problem is to ensure that pages seldom overflow, while using as little disk space as possible.

We shall assume that we are hashing n keys to disk pages, where each disk page holds r records. We would like to know the number m of disk pages so that with high confidence, we can search for any of the n keys with a single disk access. Moreover, we would like $m = O(n/r)$ so that at most a constant fraction of the storage is wasted.

We shall make the assumption of simple, uniform hashing (see page 224 of the textbook). *Tip:* inequalities (6.9) and (6.22) will prove useful.

(a) **[5 points]** For some disk page chosen arbitrarily, argue that the probability that this particular page overflows is at most $\binom{n}{r}(1/m)^r$. Please be explicit about why this bound is an overestimate.

(b) **[1 point]** Argue that the probability that any of the m pages overflows is at most $m\binom{n}{r}(1/m)^r$.

(c) **[1 point]** Argue that the probability that any of the m pages overflows is at most $m(ne/mr)^r$.

(d) **[1 point]** Given any constant $\epsilon \geq ne/r2^{r-1}$, show that by choosing $m = 2ne/r$, then the probability that any of the m pages overflows is at most ϵ.

(e) **[1 point]** Calculate a bound on the probability ϵ when hashing $n = 2^{20}$ records to $m = 2ne/r$ pages, where each page contains $r = 2^7$ records. Is the constraint $\epsilon \geq ne/r2^{r-1}$ liable to be critical in practice?

(f) **[1 point]** What is the largest load factor $\alpha = n/m$ that you can obtain with $n = 2^{20}$ and $r = 2^7$ so that the probability of a page overflowing is at most 1 in a billion? *Tip:* use the results of part (b) directly.