CS161 Handout 05
Summer 2001 July 12, 2001

Problem Set 3: Searching and Sorting

Due: Wednesday, July 18th at 5:00 p.m.

Problem 1: Understanding QuickSort (10 points)

- Using Figure 8.1 on CLR, p. 155 as a model, illustrate the operation of `Partition` on the array `A = [6,7,3,9,2,4,6,1,8]`. Rather than use the version of **`Partition`** I used in lecture, simply use the one presented on page 154.
- Show that the running time of **`Quicksort`** is $\Theta(n \lg n)$ when all elements of array A are equal. Use the **`Partition`** from the book.
- Show that the running time of **`Quicksort`** is $\Theta(n^2)$ when array A is sorted in nondecreasing order. . Again, use the **`Partition`** from the book.

Problem 2: Stack Depth for Quicksort (20 points, courtesy Problem 8-4 on CLR, p. 169)

The **`QuickSort`** algorithm of Section 8.1 contains two recursive calls to itself. After the call to **`Partition`**, the left subarray is recursively sorted and then the right half is recursively sorted. The second recursive call in **`QuickSort`** is not really necessary; it can be avoided by using an iterative control structure. This technique, called tail recursion, is provided automatically by good compilers. Consider the following version of **`QuickSort`**, which simulates tail recursion.

```
Quicksort(A,p,r)
   while p < r
      do q ← Partition(A,p,r)
         Quicksort(A,p,q)
         p ← q + 1
```

- Argue that **`QuickSort`**`(A,1,length[A])` correctly sorts the array A.

Compilers usually execute recursive procedures by using a **stack** that contains pertinent information, include the parameter values, for each recursive call. The information for the most recent recursive call is at the top of the stack, and the information for the initial call is at the bottom. When a procedure is invoked, its information is **pushed** onto the stack; when it terminates, its information is **popped** from the stack. Since we assume that array parameters are actually represented by pointers, the information for each procedure call on the stack requires $\Theta(1)$ stack space. The stack depth is the maximum amount of stack space used at any time during a computation.

- Describe a scenario in which the stack depth of our new **`QuickSort`** is $\Theta(n)$ on an n-element array.
- Modify the code for our new **`QuickSort`** so that the worst-case stack depth is $\Theta(\lg n)$.

Problem 3: Finding the majority element (10 points)

Consider an integer array of length n. If a single number occurs more than $\lceil n/2 \rceil$ times, we call that number the **majority** element. If no such number occurs, then the array doesn't have a majority element. Note that an array can have at most one majority element.

- Present a $\Theta(n \lg n)$-algorithm that returns the majority element if such an element exists. If no such element exists, then the algorithm can return anything at all.
- Present another algorithm that does exactly the same thing, but improves on the running time and runs in $\Theta(n)$ worst-case time.

Problem 4: Quick and Dirty Algorithms (10 points)

Consider a set S of $n \ge 2$ distinct numbers given in unsorted order. Each of the following five problem parts asks you to give an algorithm to determine two distinct numbers x and y in the set S that satisfy a stated condition. In as few words as possible, describe your algorithm and justify the running time. To keep your answers brief, use algorithms from lectures and the book as subroutines.

- In $O(n)$ time, determine $x, y \in S$ such that $|x-y| \ge |w-z|$ for all $w, z \in S$.
- In $O(n \lg n)$ time, determine $x, y \in S, x \ne y$ such that $|x-y| \le |w-z|$ for all $w, z \in S, w \ne z$.
- In $O(n)$ expected time, determine $x, y \in S$ such that $x + y = Z$ where Z is given, or determine that no two such numbers exist. *Hint: use a hashtable, where insertion and lookup take expected constant time.*
- (Extra credit: 5 points) In $O(n)$ time, determine $x, y \in S$ such that $|x-y| \le \frac{1}{n-1}\left(\max_{z \in S} z - \min_{z \in S} z\right)$—that is, determine any two numbers that are at least as close together as the average distance between consecutive numbers in sorted order. *Hint: use divide-and-conquer*.