Solutions to CS / MCS 401 Exercise Set #4 (Summer 2007)

Exercise 8.1-1 The minimum depth of a leaf node is $n-1$, as every comparison sorting algorithm requires at least $n-1$ comparisons, even in the best case. Suppose the sorted order for an array a of size n is $a[i_1], a[i_2], a[i_3], ..., a[i_{n-1}], a[i_n]$.

The sorting algorithm must compare $a[i_1]$ with $a[i_2]$; otherwise it has no way to distinguish the order

$$a[i_1], a[i_2], a[i_3], ..., a[i_{n-1}], a[i_n]$$

from

$$a[i_2], a[i_1], a[i_3], ..., a[i_{n-1}], a[i_n],$$

since every comparison other than that of $a[i_1]$ with $a[i_2]$ turns out the same in both cases.

Likewise, it must compare $a[i_2]$ with $a[i_3]$, ..., $a[i_{n-1}]$ with $a[i_n]$.

Exercise 8.1-3 Suppose a comparison sorting algorithm runs in linear time from some fraction $\delta(n)$ of its inputs. This means that there exists a constant C (not depending on n) such that, for all n sufficiently large, the algorithm performs at most Cn comparisons for $\delta(n)\,n!$ of its $n!$ inputs. In the decision tree, there must be at least $\delta(n)\,n!$ leaves at depth Cn or less. But we know that the number of leaves at depth Cn or less is bounded by 2^{Cn} . So $\delta(n)n! \le 2^{Cn}$, or $\delta(n) \le 2^{Cn}/n!$. Approximating $n!$ by Stirling's formula gives

$$\delta(n) \le 2^{Cn}/n! \le 2^{Cn}/\left((n/e)^n sqrt(2\pi n)\right) = \left(2^C e/n\right)^n/sqrt(2\pi n).$$

Exercise 8.1-3 asks specifically about the case $\delta(n) = 1/2$, $\delta(n) = 1/n$, and $\delta(n) = 1/2^n$. In none of these cases is $\delta(n) \le \left(2^C e/n\right)^n/sqrt(2\pi n)$ for some constant C and all n sufficiently large. If $\delta(n) = 1/2^n$, then $\delta(n)/\left(\left(2^C e/n\right)^n/sqrt(2\pi n)\right) = \left(n/2^{1+C} e\right)^n sqrt(2\pi n)$ approaches ∞ as n approaches ∞, since $n/2^{1+C} e > 1$ for all n sufficiently large. So a comparison sorting algorithm cannot run in linear time even for $1/2^n$ of its inputs.

Exercise H

i	j,p												r
	36	83	75	48	14	71	64	22	91	69	58	88	72

i,p	j											r
36	83	75	48	14	71	64	22	91	69	58	88	72

i,p		j										r
36	83	75	48	14	71	64	22	91	69	58	88	72

i,p			j								r	r
36	75	83	48	14	71	64	22	91	69	58	88	72

p	i			j								r
36	48	83	75	14	71	64	22	91	69	58	88	72

p		i			j							r
36	48	14	75	83	71	64	22	91	69	58	88	72

p			i			j						r
36	48	14	71	83	75	64	22	91	69	58	88	72

p				i			j					r
36	48	14	71	64	75	83	22	91	69	58	88	72

p					i			j				r
36	48	14	71	64	22	83	75	91	69	58	88	72

p					i				j			r
36	48	14	71	64	22	83	75	91	69	58	88	72

p						i				j		r
36	48	14	71	64	22	69	75	91	83	58	88	72

p							i				j	r
36	48	14	71	64	22	69	58	91	83	75	88	72

p							i					j,r
36	48	14	71	64	22	69	58	91	83	75	88	72

p							i					j,r
36	48	14	71	64	22	69	58	72	83	75	88	91

Exercise I Elements that are shaded will be exchanged in the next step.

left												right
36	83	75	48	14	71	64	22	91	69	58	88	72

left	lo									hi	hi	right hi
64	83	75	48	14	71	36	22	91	69	58	88	72

left		lo					hi	hi	hi			right
64	58	75	48	14	71	36	22	91	69	83	88	72

left			lo	lo	lo	hi						right
64	58	22	48	14	71	36	75	91	69	83	88	72

left					hi	lo						right
64	58	22	48	14	36	71	75	91	69	83	88	72

left												right
36	58	22	48	14	64	71	75	91	69	83	88	72

partition() returns *left*+5

Exercise J

Consider first the general case in which the pivot is chosen as the median of k elements, where k is odd and $k > 1$. We also assume $n >> k^2$, so we may approximate choice of k distinct elements by choice of k elements with repetition allowed..

Let $A_S = \{a[i] \mid a[i] < n/4^{th}$ smallest element of $a\}$, and let $A_L = \{a[i] \mid a[i] > n/4^{th}$ largest element of $a\}$. In general, the probability that among k randomly chosen elements of a exactly i lie in an $n/4$ element subset of a is very close to $C(k,i)(1/4)^i(3/4)^{k-i}$, since n is large. In order to obtain a bad split, at least $(k+1)/2$ elements from our k–element subset must lie in A_S, or at least $(k+1)/2$ must lie in A_L. Each of these alternatives occurs with probability

$$\sum_{i=(k+1)/2}^{k} C(k,i)(1/4)^i(3/4)^{k-i},$$

so the probability of a bad split is twice this, or

$$\sum_{i=(k+1)/2}^{k} 2C(k,i)(1/4)^i(3/4)^{k-i},$$

For $k = 1$, 3, 5, and 7, this evaluates as followed:

$k = 1$: 1/2 = **0.500**

$k = 3$: $2\cdot3(1/16)(3/4) + 2\cdot1(1/64) = 20/64 =$ **0.312**

$k = 5$: $2\cdot10(1/64)(9/16) + 2\cdot5(1/256)(3/4) + 2\cdot1\cdot(1/1024) = 212/1024 =$ **0.207**

$k = 7$: $2\cdot35(1/256)(27/64) + 2\cdot21(1/1024)(9/16) + 2\cdot7(1/4096)(3/4) + 2\cdot1(1/16384) =$ **0.139**.

Exer 9.3-1

Let us consider the general case where the input elements are divided into groups of q elements each, where q is odd. For simplicity, assume n is a multiple of q, so the number of groups is n/q. Let $T_q(n)$ be the time to find k^{th} smallest element. In class and in the text, $q = 5$ was used. For a fixed q, the *Select*() algorithm uses constant time to sort (or at least to find the median of) each group of q elements, and hence time linear in n to sort all q-element groups. (However, the constant multiplying n does increase as q increases.) Then *Select*() invokes itself recursively to find the median of the n/q medians, requiring time $T(n/q)$. Next *Select*() invokes *partition*(), modified to used the median of the n/q medians as the pivot. This takes linear time, and produces a split in which at least $(q+1)/2 \cdot n/2q - 1$of the n elements are less than the pivot, and at least this number are greater. Ignoring the -1, the fraction of elements on either side of the pivot is at least $(q+1)/(4q)$, meaning that the fraction on either side of the pivot can be at most $1 - (q+1)/(4q) = (3q-1)/(4q)$. The time for the recursive call to *Select*() on the left or right subarray is at most $T\big((3q-1)/(4q)\cdot n\big)$. So our recurrence for the running time is

$$T(n) = T(n/q) + T\big((3q-1)/(4q)\cdot n\big) + \Theta(n).$$

The sum of the subproblem sizes is $n/q + (3q-1)/(4q)\cdot n + \Theta(n) = 3(q+1)/4q \cdot n$. We showed in class that the solution was $\Theta(n)$ when $q = 5$; the proof relied only on the fact that the sum of the subproblem sizes was at most cn for some constant c less than 1. On the other hand, if the subproblems have size αn and βn with $\alpha+\beta = 1$, the solution is $\Theta(n\lg(n))$. (We proved this in class for $\alpha = 2/3$, $\beta = 1/3$, but the proof works for any α and β with $\alpha < 1$, $\beta <$

1, and $\alpha+\beta = 1$. So for *Select*() to run in linear time in the worst case, we need $3(q+1)/4q < 1$, or $3(q+1) < 4q$, or $q > 3$.

Thus, with groups of 3, the worst-case running time of *Select*() is not linear, but with groups of 5, 7, or any other odd integer it is linear.

Exer 9.3-8

First note: Let S be any set. If, for some q with $q < |S|/2$, we remove from S both q elements less than or equal to the (old) median of S and q elements greater than or equal to the median, then the median of S remains unchanged.

To keep things simple, let us assume $n = 2^k-1$ for some k.

If $k = 1$, then we are finding the median of two elements; simply choose the smaller (for the lower median).

If $k > 1$, then $n \geq 3$. Let $m = (n+1)/2 = 2^{k-1}$, the middle position of X and Y. Since X and Y are sorted, the medians of X and Y are X[m] and Y[m], respectively. If X[m] = Y[m], we are done; X[m] is the median of the two arrays combined. If X[m] < Y[m], then

$$X[m] \leq \text{(combined median)} \leq Y[m].$$

We may discard X[1..m] (m elements ≤ combined median) and Y[m..n] (m elements ≥ the combined median). This leaves two sorted subarrays, X[m+1..n] and Y[1..m−1] with $2^{k-1}-1$ each, which have the same combined median as the two original arrays, so we can invoke our algorithm recursively to find the median of these two subarrays. Similarly, if

```
// Invoke initially as median( X, 1, n, Y, 1, n).  T is the element type of X and Y.  Note
//  this code assumes for simplicity that n is one less than a power of 2.  Note at all times
//  xRight−xLeft = yRight−yLeft.

T median( T[] X, int xLeft, int xRight, T[] Y, int yLeft, int yRight)
        if ( xLeft == xRight )
                return min( X[xLeft], Y[yLeft] );
        xMid = (xLeft + xRight) / 2;
        yMid = (yLeft + yRight) / 2;
        if ( X[xMid] < Y[yMid] )
                return median( X, xMid+1, xRight, Y, yLeft, yMid−1);
        else if ( X[xMid] > Y[yMid] )
                return median( X, xLeft, xMid−1, Y, yMid+1, yRight);
        else
                return X[xMid];
```