Burt Rosenberg

Problem Set 6

OUT: 31 OCTOBER, 1995
DUE: 14 NOVEMBER, 1995

Quicksort

Write a quicksort subroutine. Your main routine should call quicksort with the array of integers to sort and the integer size of the array. Quicksort returns with the array sorted in ascending order. The main generates some test data, prints the array before the sort, calls the sort, then prints the array after the sort.

The heart of quicksort is the partitioning of a section of the array into numbers smaller than or equal to the pivot and numbers larger than the pivot. To make life easier, let us divide the work of partitioning into a separate pivot selection routine and a split routine. The split routine assumes that the pivot has been placed in the zeroth location of the region it is to partition. At first, the pivot selection routine does nothing. Once the more difficult split routine is working, go back and improve the pivot selection routine to use the median-of-three heuristic.

The split routine takes an array and two integer indices, assumes that the pivot is positioned at the bottom of the region to split, and returns an integer index indicating where the pivot element ended up after the split:

```
int split( int A[], int bottom, int top )
{
   /* Split A[bottom..top] at the pivot = A[bottom]. That is,
   originally the pivot is placed in the bottom location.
   Return the pivot position after the split. */
   ...
   return( where_the_pivot_is_now ) ;
}
```

The split routine is tricky, and you should use the following loop invariant to guide you:

> Integers i and j are such that bottom $\leq i \leq j \leq$ top, and for any integer k between bottom and i, $A[k] \leq$ pivot, and for any integer k between $k+1$ and top, $A[k] >$ pivot.

You should ask yourself the following questions:

1. To what values will i and j set set initially to make the loop invariant true.

2. What relationship between i and j imply loop termination.

3. How do I move towards termination? Precisely, how is the loop invariant reestablished each time before the bottom of the loop. (Hint: If i and j

both defy the invariant, then swapping the values in `A[i]` and `A[j]` will fix this.)

Notes on Big-Oh

To talk easily and precisely about the efficiency of algorithms, when we are less interested in the constant factors than we are in how the algorithm performs for large data sets, we use the Big-Oh notation. We say that insertion and selection sorts are $O(n^2)$, "Big-Oh of n-squared," and merge and quick sort are $O(n \log n)$, "Big-Oh of n-log-n."

A function f on the positive integers is "Big-Oh" of another function g, $f = O(g)$, if there exists an integer N and a constant C such that,

$$|f(n)| < C|g(n)|, \text{ for all } N < n.$$

The choice of the constant C lets us group together functions which differ only by constant factors, and the choice of N lets us neglect perhaps bad behavior of the algorithm on small problem sizes.

Insertion sort is $O(n^2)$, so that given any machine, and any programmer, there is a C and an N such that whenever you sort n numbers, $n > N$, using this programmer's code on the specified machine, it will take less than Cn^2 seconds. Merge sort is $O(n \log n)$, so that given any machine, and any programmer, there is a C' and an N' such that whenever you sort n numbers, $n > N'$, using that programmer's code on the specified machine, it will take less than $C'n \log n$ seconds. But, insertion sort is *not* $O(n \log n)$, that is, no matter how big you choose C, for large enough n the sort will take *more* than $Cn \log n$ seconds.

There are some rules for using Big-Oh notation:

1. If $f = O(g)$ and $g = O(h)$ then $f = O(h)$. This is called *transitivity.*
2. If $f_1 = O(g)$ and $f_2 = O(g)$, and A and B are constants, then $Af_1 + Bf_2 = O(g)$. Hence the collection of all $O(g)$ functions forms a *vector space.*
3. It is possible that $f = O(g)$ while $g \neq O(f)$ — in fact, this is often the case. An example has already been given.
4. If $f = O(g)$ then $O(f + g) = O(g)$ — adding a "small" function to a "big" function won't change the big function's size.