**Advanced Sorts**

The sort algorithm prototypes are

void quickSort(int \* a, int s);

void quickSort(int \* a, int i, int j);

// sort from index i to index j inclusive

void setMedianOfThree(int a [], int i, int j);

void mergeSort(int a [], int s, int n = false);

// parameter n is false for binary mergeSort, true for natural

void radixSort(int a [], int s);

void countingSort(int a [], int s);

The helper function for quickSort is

void setMedianOfThree(int a [], int i, int j);

which has constant-time performance.

The complete code for the first of the quickSort prototypes is

void quickSort(int a [], int s){

quickSort(a, 0, s-1);

}

It can be shown that any sorting algorithm that compares keys has, at best, ( *n* lg *n* ) performance. Radix sort and counting sort get around this barrier by (in the first case) comparing only *portions* of keys and (in the second) not doing comparisons at all. Both have nearly linear performance, but are restricted in the type of data that they can sort.

As usual, you’ll provide source code and executable on USB drive and a printout of source code, and a report similar to that in the previous assignment. In addition, you will conduct a detailed analysis of quickSort.

Analysis one: benefit of the setMedianOfThree routine.

 (last - first )/2 + first

Run quickSort both with and without the setMedianOfThree function for random, sorted, reversed, and all-identical-element lists. Describe any differences in performance you observe. For what type of lists does the setMedianOfThree have a large impact on performance, and for what types does the impact seem to be minimal?

Note that for a large, sorted (or reversed) list quickSort might generate so many recursive calls that you get a stack overflow. You can up the stack size in Visual C++ by adding a preprocessor directive

#pragma comment(linker, "/STACK: 8000000") // don’t use fix

where 8000000 is some value that is larger than the default stack size.

Analysis two: use of a helper sorting function.

Like many advanced sorts, quickSort chews through most of the job pretty quickly, then spends a disproportionately large time finishing the job. Thus, quickSort often makes use of a helper sort function to tackle small sublists. For a given random list, send various sizes of unsorted to sublists and see if there is a “sweet spot” that maximizes performance. Do this for at least three lists to see if your results are consistent. Include a detailed discussion of your findings in your report.