**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.

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")

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.

**Analysis 1~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

**Quick Sort without setOfMedianThree function**

* **Random Array** 
  + The quick sort **iterative** *without* the median of three helper function ripped right through the array of 1 million randomly sorted integers in less than a second.

Quick Sort - Random Array ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

0 1 1 1 2 2 2 3 5 7

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

999988 999988 999990 999993 999993 999995 999995 999996 999997

# of comparisons: 53658732

# of exchanges: 3961918

**Elapsed time: 0.161195 sec**

* **Sorted Array**
  + The quick sort *without* the median of three helper function ripped right through the array of 1 million sorted integers in less than a second.

Quick Sort - Sorted Array ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

0 1 1 1 2 2 2 3 5 7

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

999988 999988 999990 999993 999993 999995 999995 999996 999997

# of comparisons: 37265727

# of exchanges: 0

**Elapsed time: 0.0549016 sec**

* **Reversed Array**
  + The quick sort *without* the median of three helper function ripped right through the array of 1 million reverse sorted integers in less than a second.

Quick Sort - Reversed Array ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

0 1 1 1 2 2 2 3 5 7

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

999988 999988 999990 999993 999993 999995 999995 999996 999997

# of comparisons: 37766416

# of exchanges: 500000

**Elapsed time: 0.0527851 sec**

* **Identical Array**
  + The quick sort *without* the median of three helper function didn’t do so well on the array of identical integers. I even changed the type of the comparison and exchanges to *long long int* as well as changed the stack size with the pragma statement. **#pragma comment(linker, "/STACK: 80000000");**
    - **Array of 10,000 identical integers –** Lot of comparisons

Quick sort - Identical Array~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

42 42 42 42 42 42 42 42 42 42

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

42 42 42 42 42 42 42 42 42 42

# of comparisons: 100039996

# of exchanges: 0

**Elapsed time: 0.0130423 sec**

* + - **Array of 100,000 identical integers –** A lot more of comparisons

Quick sort - Identical Array~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

42 42 42 42 42 42 42 42 42 42

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

42 42 42 42 42 42 42 42 42 42

# of comparisons: 10000399996

# of exchanges: 0

**Elapsed time: 11.86 sec**

* + - **Array of 300,000 identical integers –** Segmentation fault

# of comparisons: NA

# of exchanges: NA

**Elapsed time: Segmentation fault: 11**

**Quick Sort with setOfMedianThree function**

* **Random Array** 
  + The quick sort *with* the median of three helper function ripped right through the array of 1 million randomly sorted integers in less than a second in about the same time as without the median of three function.
    - **It had ~ 27 million less comparisons than without the median of three**
    - **But it had ~ 8.8million more exchanges**

Quick Sort - Random Array ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

0 1 1 1 2 2 2 3 5 7

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

999988 999988 999990 999993 999993 999995 999995 999996 999997

# of comparisons: 25748785

# of exchanges: 12775219

**Elapsed time: 0.180163 sec**

* **Sorted Array**
  + The quick sort *with* the median of three helper function couldn’t get through the array of 1 million sorted integers, so I changed it to 100,000 sorted integers in and it took ~ 28 sec to sort the 100,000 sorted integers **(Terrible performance)**
  + **Array of 100,000 reverse sorted integers**

Quick Sort - Sorted Array ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

0 1 1 1 2 2 2 3 5 7

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

99959 99960 99960 99963 99964 99964 99965 99965 99966 99966

# of comparisons: 5000050000

# of exchanges: 5000150000

**Elapsed time: 27.9137 sec**

**Array of 1 million sorted integers –** Error: segmentation fault

* **Reversed Array**
  + The quick sort *with* the median of three helper function struggled with the reverse sorted array.
  + **Array of 100,000 reverse sorted integers**

Quick Sort - Reversed Array ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

0 1 1 1 2 2 2 3 5 7

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

999988 999988 999990 999993 999993 999995 999995 999996 999997

# of comparisons: 3405879160

# of exchanges: 1833822075

**Elapsed time: 13.5278 sec**

* + **Array of 1 million reverse sorted Integers –** Error: segmentation fault
* **Identical Array**
  + The quick sort *with* the median of three helper function didn’t do so well on the array of identical integers either. I even changed the type of the comparison and exchanges to *long long int* as well as changed the stack size with the pragma statement. **#pragma comment(linker, "/STACK: 80000000");**
    - **Array of 10,000 identical integers –** Lot of comparisons

Quick sort - Identical Array~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

42 42 42 42 42 42 42 42 42 42

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

42 42 42 42 42 42 42 42 42 42

# of comparisons: 50005000

# of exchanges: 50015000

**Elapsed time: 0.290561 sec**

* + - **Array of 100,000 identical integers –** A lot more of comparisons

Quick sort - Identical Array~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

42 42 42 42 42 42 42 42 42 42

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

42 42 42 42 42 42 42 42 42 42

# of comparisons: 5000050000

# of exchanges: 5000050000

**Elapsed time: 28.2213 sec**

* + - **Array of 300,000 identical integers –** Segmentation fault

# of comparisons: NA

# of exchanges: NA

**Elapsed time: Segmentation fault: 11**

**Analysis 2~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**