

**CIS 291 – Data Structures in C++ - Spring 2005**  
**Homework #5**  
**HW #5 due: TUESDAY, March 1st, BEGINNING of lab**

(the above is NOT a typo. Since this is being turned in on paper, and I will be out of town on when this "should" be due --- Thursday, Feb. 24<sup>th</sup> --- it just seemed simpler to allow it to be turned in at the next available class meeting.)

**Purpose:**

To encourage you to think more deeply about 5 classic sorting algorithms, and to practice with experimentation.

**How this will be turned in:**

You will turn this in on-paper by the time and date mentioned above.

**NOTE: you may work TOGETHER on experimentation and analysis for THIS HW (HW #5), but the final WRITE-UP to be TURNED IN must be INDIVIDUAL WORK.** (ASK ME if this is not clear!!)

AND NOW, FOR SOMETHING COMPLETELY DIFFERENT...  
MODIFIED from the assignment by:

**David B. Levine, Computer Science Department, St. Bonaventure University**

This assignment has the purpose of encouraging you to think more deeply about many of the different sorts that we discussed in lecture (note that radix sort is not included).

Since there are only write-ups involved in this lab, you should turn in the three required components described below ON-PAPER by the beginning of class on the time and date specified above.

On the course web page, via "Homeworks and Handouts", you will find a link named **291hw05.jar**. This file is also available from `~st10/291hw05.jar` on cs-server.

You need to execute a program in this Java jar (java archive) file named **SortDetective**. Here are two of a number of ways to do so:

- \* IF you are on the Windows side on a machine in NHW 244, open up a browser, and copy the 291hw05.jar file from the public course web page --- BUT, copy it to the actual PC's Desktop.
  - \* (be careful --- I don't mean the Desktop under Users. If, after you've saved the file, you see the little icon labeled 291hw05.jar on the desktop, then it's in the right place. If you don't, you probably saved it to the other one --- and for some reason, it doesn't run when you run the command below (!?.))
  - \* then, go to the Start button, select the Run... command, and in the dialog that appears, in the textfield labeled Open:, type this command:

```
java -classpath 291hw05.jar SortDetective
```

...and click OK. SortDetective should start right up (although its window may hide behind other open windows...)

- \* IF you are on the Linux side on a machine in NHW 244, open up a terminal program, and type the following:

```
cp ~st10/291hw05.jar .          # NOTE the period at the end!!!  
java -classpath 291hw05.jar SortDetective
```

...and SortDetective should start right up (although its window may hide behind other open windows...)

Note, too, that this **java -classpath 291hw05.jar SortDetective** command OUGHT to work on any machine that has Java installed, if called from the directory where 291hw05.jar is stored... I've done it successfully on Mac OS X, too.

**SortDetective** lets you practice running implementations of **several different** sort implementations, with lists of sizes you choose and various features. It then shows you the **number of comparison** and **number of movements** that that run of that sort took.

The primary objective of this assignment is for you to apply your theoretical knowledge of sorting algorithms to solve a problem of poor user interface design. More specifically, SortDetective is designed to measure comparisons, data movements, and execution time for the several sorting algorithms discussed in class. Unfortunately, the designer of the program did not label the buttons properly. You must apply your understanding of the general properties of the algorithms to determine the proper labeling of the buttons.

The secondary objective of this lab is for you to gain experience writing a concise but complete analysis of a system.

The sorts have dummy names; these are actually implementations of the discussed sorts **selection sort, insertion sort, bubble sort, merge sort, and quick sort.**

**CONSIDER:** you know the **basic run-time** complexity of these; you know a bit about how they work. What kinds of tests could you run to make an **educated guess** about which sort is which?

As you know from class, if you double the size of the data set that you give to a quadratic algorithm, it will do four times the work; by contrast, an  $O(N\log N)$  algorithm will do a bit more than twice as much; and, a linear algorithm will do only twice as much work. As you also know, the characteristics of the input data set can affect the expected performance of many of our sorting algorithms. Before you begin the planning, you can see how it would be helpful for you to review the **expected** performance of the algorithms on various data sets.

### **FIRST ITEM TO TURN IN:**

**First**, make a **plan** --- what kinds of tests could you run to **GUESS** which sort is which? **Write** this plan/strategy out, and **turn it in** as part of this assignment. (You may discuss potential plans/strategies with one another, but each should write out their plan **individually**.) Note the this plan should be **word-processed** rather than hand-written.

What should this plan/strategy look like? Think of the procedure for a science experiment --- if you gave this to two different people, both could go through these steps and be able to compare their results. That is, it should be specific! What sizes of lists will you run the different sorts for? What characteristics will these lists have? What data will you then record for each test? And, you should also include your hypotheses for what you hope to discover for each set of tests.

For example, if you were making a plan to try to distinguish sequential search from binary search, you might have a plan such as:

- \* Run each search on a list of size 32 for an element not in the list. Record the number of comparisons. Repeat for 5 different non-list elements:
  - (one smaller than any in the list,
  - one larger than any in the list,
  - one near the center of the "range" of the list,
  - one near the beginning of the "range" of the list,
  - one near the end of the "range" of the list)

Hypotheses:

- \* the binary search should take about the same number of steps for each non-list element.
- \* the sequential search may vary [depending on what "shortcuts" are built in --- you'd probably know what these were at the time you are making the plan. You will know for the sorts, as you will see below.]
- \* Run each search on a list of size 64 for an element not in the list. Record the number of comparisons. Repeat for 5 different non-list elements:
  - (one smaller than any in the list,
  - one larger than any in the list,
  - one near the center of the "range" of the list,
  - one near the beginning of the "range" of the list,
  - one near the end of the "range" of the list)

Hypotheses:

- \* the binary searches here will also take about the same number of comparisons for each search, and should only be a few more steps than the size-32 searches took.
  
- \* the sequential searches here may vary, but at least some should take roughly twice as many steps as the corresponding size-32 searches.

**SECOND ITEM TO TURN IN:**

Then, **execute** your plan, running the tests and **recording** the results. (You may hand-record these results, rather than word-process or type them, if you wish.) (Yes, we are practicing a little scientific experimentation here, too.) You may discuss and brainstorm together during this testing, also.

Note that you will likely have a number of tables/charts as the result of executing your plan and recording the results (the results will be recorded in tables or charts, which, again, may be either hand-written or typed).

What if you find you need additional tests? Run them, but record for each what you did, and why you did them, along with their results.

**THIRD ITEM TO TURN IN:**

Finally, you will take your notes from those experiments, and **write up** your **guesses** as to which sort is which. You will **first summarize** your guesses, and **then justify** them; the format should be like that given on the example on the next page. **(Although you may experiment, speculate, and brainstorm with other class members, THIS write-up is INDIVIDUAL work)** Like the original plan, this final report should be word-processed rather than hand-written.

Searching Lab Report  
A. Beauregard Clump  
2-14-05

### **General Results**

Button Name	Searching Algorithm
Sigma	Binary Search
Tau	Sequential Search

### **Rationale**

I chose to search data sets of size 32 and 64, looking for 5 elements not in the list each time: one smaller than any in the list, one larger than any in the list, one that is about in the middle of the "range" of elements in the list, one that is near the beginning of the "range" of elements in the list, and one that is near the end of the "range" of elements in the list. Button Tau took very different numbers of steps for the 5 different searches in the 2 lists, but the largest numbers of its comparisons were doubled for the size-64 list as compared to the size-32 list. Button Sigma was very consistent in its number of comparisons for each item in the size-32 list, and this number increased by only 2 for the size-64 searches.

Since a binary search is logarithmic, and the number of comparisons for Tau only increased by a small amount when the list was doubled, I conclude that Button SIGMA is BINARY SEARCH.

Since a sequential search is linear in the worst and average cases, and the highest-comparison searches did double in number of comparisons when the list size was doubled, I conclude that Button TAU is SEQUENTIAL SEARCH.

---

### **GRADING NOTES:**

- \* There is no coding in this assignment. Thus, you should expect that a significant portion of the grade for this assignment will be determined by the quality of the writing of the report. This includes the completeness of the report, the clarity of the writing, and general presentation. You could get a poor grade for poor writing, even if you "get" all of the right matches.
- \* Some of the sorts may be difficult to distinguish. A carefully outlined experiment may compensate for an error in these cases if the writing makes it clear that your conclusions/guesses are substantiated by the data.
- \* Remember that your report (the 3<sup>rd</sup> item you are turning in) needn't detail every experiment you ran. Rather, it should give sufficient information to justify your conclusions. It is possible to write a very short report that is completely correct if your experiments are well-chosen. After you determine the matching, you might consider whether there was a shorter way to arrive at your conclusion!
- \* Carefully note the requirements given; not turning in some required pieces, or structuring them differently, etc., could have a negative impact on your grade.

### **REFERENCES:**

Below, you will find the Java implementations of the sorts used within the SortDetective program. Again, these should be readable by someone with a knowledge of C++. Why might you want to look at the implementations? To answer such questions as, does this bubble sort stop "early" if it can? How does this quick sort choose its pivot? Does this merge sort use a temporary "helper" array for the merge step? Does this insertion sort always recopy the current element into its insertion location in the sorted part of the array, even if it is already there? etc.

---

```
/* Here are the implementations of the sorts as used in SortDetective. */
/* (note that they have had code added to count comparisons and data */
/* movements) */

/* Code and Comments by David B. Levine, Computer Science Department, */
/* St. Bonaventure University */

/* BubbleSort */

// Code basically taken from Sedgewick, "Algorithms"

protected void bubbleSort(int list[])
{
    int temp;
    do {
        movements++;
        temp = list[0];
        for (int j=1; j<list.length; j++)
        {
            comparisons++;
            if (list[j-1]>list[j])
            {
                movements += 3;
                temp = list[j];
                list[j] = list[j-1];
                list[j-1] = temp;
            }
        }
        comparisons++;
    } while (temp != list[0]);
}
```

**/\* Selection Sort \*/**

// Code from memory

```
protected void selectionSort(int list[])
{
    for (int j=list.length-1; j>0; j--)
    {
        int maxpos = 0;
        for (int k=1; k<=j; k++)
        {
            comparisons++;
            if (list[k]>list[maxpos])
            {
                maxpos = k;
            }
        }
        if (j != maxpos) // Only move if we must
        {
            movements += 3;
            int temp = list[j];
            list[j] = list[maxpos];
            list[maxpos] = temp;
        }
    }
}
```

**/\* Insertion Sort \*/**

// Code basically taken from Sedgewick, "Algorithms"

```
protected void insertionSort(int list[])
{
    for (int j=1; j<list.length; j++)
    {
        movements++;
        int temp = list[j];
        int k = j;
        while( k > 0 && list[k-1]>temp )
        {
            comparisons++;
            movements++;
            list[k] = list[k-1];
            k--;
        }
        if (k > 0)
        {
            comparisons++;
        }
        movements++;
        list[k] = temp;
    }
}
```

```
/* MergeSort */

// Code basically taken from Horowitz and Sahni, "Fundamentals of Computer
// Algorithms"

protected void mergeSort(int list[])
{
    msort(list, 0, list.length-1);
}

private void msort(int list[], int low, int high)
{
    if (low<high)
    {
        int mid = (low+high)/2;
        msort(list, low, mid);
        msort(list, mid+1, high);
        merge(list, low, mid, high);
    }
}

private void merge(int list[], int low, int mid, int high)
{
    int h = low;
    int i = 0;
    int j = mid+1;
    int otherList[] = new int[high-low+1];

    while ((h <= mid) && (j <= high))
    {
        comparisons++;
        if (list[h] <= list[j])
        {
            movements++;
            otherList[i] = list[h];
            h++;
        }
        else
        {
            movements++;
            otherList[i] = list[j];
            j++;
        }
        i++;
    }
    if (h>mid)
    {
        for (int k=j; k<=high; k++)
        {
            movements++;
            otherList[i] = list[k];
            i++;
        }
    }
    else
    {
        for (int k=h; k<=mid; k++)
        {
            movements++;
            otherList[i] = list[k];
            i++;
        }
    }

    for (int m=0; m<otherList.length; m++)
    {
        movements++;
        list[low+m] = otherList[m];
    }
}
}
```

```
/* QuickSort */

// Code basically taken from Cormen, Leiserson, and Rivest, "Introduction to
// Algorithms"

protected void quickSort(int list[])
{
    qsort(list, 0, list.length-1);
}

private void qsort(int list[], int low, int high)
{
    if (low<high)
    {
        int pivot = partition(list, low, high);
        qsort(list, low, pivot);
        qsort(list, pivot+1, high);
    }
}

private int partition(int list[], int low, int high)
{
    movements++;
    int temp = list[low];
    int i = low-1;
    int j = high+1;
    while(true)
    {
        do {
            j--;
            comparisons++;
        } while (list[j] > temp);

        do {
            i++;
            comparisons++;
        } while (list[i] < temp);

        if (i < j)
        {
            movements += 3;
            int swapTemp = list[i];
            list[i] = list[j];
            list[j] = swapTemp;
        }
        else
        {
            return j;
        }
    }
}
}
```