

1 Introduction

There is no programming in this homework, all the code is given to you. On the MyClasses page for this assignment there are two files, `Homework08.cpp` and `Homework08_2.cpp`. Create a new project and copy the contents of the `Homework08.cpp` file into the main. At this point you are ready to go.

This exercise is a timing analysis of the three sorting algorithms we discussed in class, the bubble sort, insertion sort and selection sort. In addition, I included another sorting algorithm, the quick sort, which belongs to a different class of sorting algorithms. The quick sort is one of several divide-and-conquer algorithms. You do not need to know how it works but if you go on to Computer Science II you will learn how it works and analyze it along with other algorithms of the same type. You may find it interesting to look at the code, the `quickSort` function calls itself. Functions that call themselves are called recursive functions, although it may seem strange to do something like that, the ability to do this is a very powerful technique. We will look at recursive functions later in the class.

When you analyze and compare algorithms what you usually do is count the number of operations that the algorithm does. For example, how many arithmetic operations are done (additions, subtractions, multiplications, or divisions), how many array entries are moved or changed, or how many comparisons are done (conditional statements that need to be evaluated). In other words, you look at the portions of the code that take most time and ask how often that portion of code is run to complete the algorithm. We will not do a strict mathematical analysis of these algorithms here, but if you go further into computer science you will learn the techniques for this analysis.

What we are doing in this homework is an empirical analysis and comparison of the sorting algorithms using the run-time of the algorithm's implementation. So we will create an array of random numbers and time how long it takes for each sorting algorithm to sort the array. Timing analysis is not as precise as operation counting but it still gives a good indication of the work done to complete an algorithm. There are a few things you need to be careful with in a timing analysis. First, a timing analysis is dependent on the hardware you are using. If you time a process on a slow computer and then do the same timing on a fast computer the results will clearly be different. **So when you do this assignment you must do all of the runs on the same computer.** Another downfall with timing analysis is that with modern operating systems where are many processes running in the background so your program does not have 100% access to your CPU. For the most part this is out of your control.

When analyzing an algorithm you want to answer three questions,

1. How well does it run in the best case scenario?
2. How well does it run in the worst case scenario?
3. How well does it run on average?

For sorting an array,

1. The best case scenario is usually (but not always) how long does it take the algorithm to sort an array that is already sorted? If the array is already sorted then there should be no movement of array entries and in general the algorithm should do the least amount of work.
2. The worst case scenario is usually (but not always) how long does it take the algorithm to sort an array that is in reverse order? If the array is in reverse order then there should be a lot of movement of array entries and in general the algorithm should do the most amount of work.

3. The average case scenario is usually (but not always) how long does it take the algorithm to sort an array that is random? In this case, it is more informative to run several random arrays through the algorithms and take an average of the times.

Before we get started on timings and analyze the algorithms let's look at a couple segments of the program. We have functions for the three sorting algorithms discussed in class and some support functions, like copying the array and printing the array (which we do not use). In the main, there is a timer that is around each call to the sorting algorithm. The code below starts the stopwatch, runs the function we wish to time, stops the stopwatch and then calculates the elapsed time in seconds.

```
start = high_resolution_clock::now();
selectionSort(A, size);
stop = high_resolution_clock::now();
duration = duration_cast<microseconds>(stop - start);
cout << "Selection sort = " << duration.count() / 1000000.0 << " seconds" << endl;
```

Another block of code we need to discuss is the following,

```
for (int i = 0; i < size; i++) {
    A[i] = rand();    // Average
    //A[i] = i;      // Best
    //A[i] = size - i; // Worst
}
```

This is populating the array and it is set up to create our best, worst, and average cases. The first line creates a random, the second line one that is already sorted, the third line creates an array that is in reverse order. So when we are examining the best case scenario we will uncomment one line and comment the other two.

2 Analysis of the Sorting Algorithms

Show all of your work and answers to the following questions in a Word, LibreOffice Writer (odt) file or PDF that contains all of the charts, graphs, and your explanations. You will be submitting this report to the MyClasses site. As with any other paper you hand in make the presentation of it as neat and orderly as possible.

1. Do the following for each scenario, best, worst, and average cases.

- (a) Run the program on array sizes, 10,000, 20,000, 30,000, 40,000, 50,000, 60,000, and 70,000. Note that the array sizes go up by the same amount each time, we did this for a reason. You should do several runs at the same size. If you get a time that is out of the ordinary you should probably ignore it since there could have been another process running that was using your computer resources.
- (b) Using a spreadsheet, make a chart from the data that has the array size as the first column, the bubble sort times as the second column, the insertion sort times as the third column, the selection sort times as the fourth column, and the algorithm library sort times in the fifth column.
- (c) Create an XY-Scatter plot of sort times verses array size, have the array size on the horizontal axis and the sort times on the vertical axis. So the graph should have all four lines on the same chart, one for the bubble sort, one for the insertion sort, and one for the selection sort and one for the algorithm library sort. Use the line and point options for the graph.

- (d) Give a detailed answer to each of the following questions that are to be based on your chart and graph from the previous exercises. Answer these for each scenario, best, worst, and average cases.
- From the time analysis, which algorithm is consistently better than the others? Which algorithm is consistently worse than the others? Do any of the lines cross? This would indicate that one algorithm is better for smaller arrays but another is better for larger arrays.
From the timing data, does there seem to be any consistencies between the algorithms? As an example, is one consistently 10 times slower than another one, are two fairly close to each other on every run, ...?
 - From the time analysis, does the line graph appear to be straight or curved? If the line is straight then this would imply a linear relationship and hence as the array size grows the time to sort it grows at the same rate. If the graph is not a straight line then there is a different relationship. Does the graph suggest a linear relationship or an different relationship?
2. Let's expand on the last question. **Your analysis for this question is to be done on the average case scenario only.**

Algorithms run in what is called polynomial time if you can represent their run-time increase with the increase in the complexity of the problem as a polynomial equation. So for us, if as the array size increases the time to sort the array increases along a polynomial curve then the algorithm runs in polynomial time. If the algorithm does not run in polynomial time then it may run in exponential time, that is, the graph of problem size verses run-time follows an exponential curve. Some algorithms can be even worse than this and follow curves that increase faster than an exponential.

In general polynomial time algorithms are good, at least they are better than exponential time algorithms. Usually, polynomial time algorithms are those that are worth implementing and can produce an answer in a reasonable amount of time. Exponential time algorithms usually become too slow too quickly to be very useful.

If an algorithm is polynomial time we can use an easy mathematical trick to determine if it actually is and what degree the polynomial is. We do this by taking differences in the y values, for us these will be the sorting times we got from our tests above. We then take the differences of the differences, then the differences of these, and so on until we end up with the differences all being the same number (that is, a constant). The difference that resulted in a constant is the degree of the polynomial.

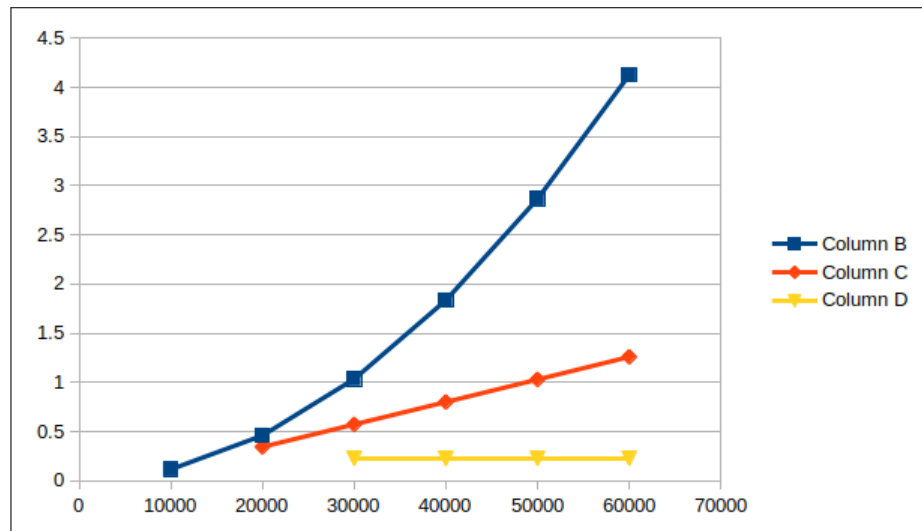
If we have timing data that does not eventually become constant then we are probably dealing with an algorithm that is exponential, or worse, in nature. We do need to take a little caution here since we are dealing with experimental data that will not be exact. So graphing these differences will come in handy when we make our analysis. We would be looking for an approximate straight line that is not horizontal followed by the next difference being an approximate horizontal line.

This analysis relies on the x values (our array sizes) all being equally spaced. Since ours are 10,000, 20,000, ..., 70,000 we have already made sure of that. Let's do a couple examples of this process.

In the example below, the x and y values in the following chart are from the equation $y = x^2 + 3x + 7$. We then take the first and second differences in the following two columns. Notice that in the second difference all of the values are 2. So we deduce that the equation is polynomial and of degree two.

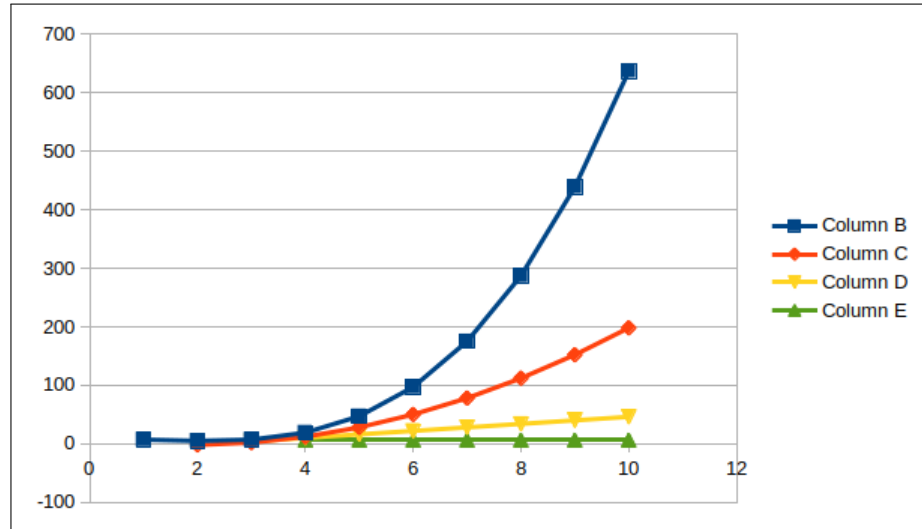
x	y	diff 1	diff 2
1	11		
2	17	6	
3	25	8	2
4	35	10	2
5	47	12	2
6	61	14	2
7	77	16	2
8	95	18	2
9	115	20	2
10	137	22	2

If we graph the data we see that the final line is horizontal and the one before it is straight but not constant.



As another example take the equation $y = x^3 - 4x^2 + 3x + 7$. The chart and graph are below.

x	y	diff 1	diff 2	diff 3
1	7			
2	5	-2		
3	7	2	4	
4	19	12	10	6
5	47	28	16	6
6	97	50	22	6
7	175	78	28	6
8	287	112	34	6
9	439	152	40	6
10	637	198	46	6



Again this would tell us that the relationship was polynomial and the degree here is three.

For the math folks this should feel very reminiscent of derivatives. If you have a cubic polynomial then its derivative is a quadratic, its second derivative is linear and its third derivative is constant. We are doing the same thing here except that instead of using derivatives we are using differences.

Using your timing data and this technique of differences determine if the average case for each of the bubble, selection, and insertion sorts are polynomial or not. If they are polynomial find the degree of the polynomial. Remember that you are doing this on experimental data not on exact equations so the differences will not come out to be perfectly constant. If you get a set of differences that are close to each other then these are probably showing a constant stage.

- Copy the code from the `Homework08_2.cpp` over the code you were using. This program does the same timing analysis except that it just uses the algorithm library sort and the quick sort. It also uses a different way declare arrays. We discussed in class the difference between program memory and heap memory, this method places the array in the heap and hence we can use larger arrays.

Run the program on array sizes 1,000,000, 2,000,000, 3,000,000, 4,000,000, and 5,000,000. Record the results in a table. Graph the XY-Scatter plot of the data.

- From the graph, does the run-time of these appear to be linear or not?
- If not, use the difference technique to determine if the run-time is polynomial or not, and if it is find the degree of the polynomial.

3 Code Files

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <algorithm>
#include <chrono>

using namespace std;
using namespace std::chrono;

void bubbleSort(int[], int);
void selectionSort(int[], int);
void insertionSort(int[], int);

void CopyArray(const int[], int[], int);
void printArray(int *array, int size, bool newline = false);

int main() {
    srand(time(0));
    int size = 0;

    cout << "Input array size: ";
    cin >> size;

    int A[size];
    int B[size];
    for (int i = 0; i < size; i++) {
        A[i] = rand(); // Average
        //A[i] = i; // Best
        //A[i] = size - i; // Worst
    }

    CopyArray(A, B, size);

    auto start = high_resolution_clock::now();
    bubbleSort(A, size);
    auto stop = high_resolution_clock::now();
    auto duration = duration_cast<microseconds>(stop - start);

    cout << "Bubble sort = " << duration.count() / 1000000.0 << " seconds"
        << endl;

    CopyArray(B, A, size);

    start = high_resolution_clock::now();
    selectionSort(A, size);
    stop = high_resolution_clock::now();
    duration = duration_cast<microseconds>(stop - start);

    cout << "Selection sort = " << duration.count() / 1000000.0 << " seconds"
        << endl;

    CopyArray(B, A, size);

    start = high_resolution_clock::now();
    insertionSort(A, size);
    stop = high_resolution_clock::now();
    duration = duration_cast<microseconds>(stop - start);

    cout << "Insertion sort = " << duration.count() / 1000000.0 << " seconds"
        << endl;

    CopyArray(B, A, size);

    start = high_resolution_clock::now();
    sort(A, A + size);
    stop = high_resolution_clock::now();
    duration = duration_cast<microseconds>(stop - start);

    cout << "Algorithm sort = " << duration.count() / 1000000.0 << " seconds"
        << endl;

    return 0;
}
```

```

void CopyArray(const int A[], int B[], int size) {
    for (int i = 0; i < size; i++)
        B[i] = A[i];
}

void bubbleSort(int array[], int size) {
    int maxElement;
    int index;

    for (maxElement = size - 1; maxElement > 0; maxElement--)
        for (index = 0; index < maxElement; index++)
            if (array[index] > array[index + 1])
                swap(array[index], array[index + 1]);
}

void selectionSort(int array[], int size) {
    int minIndex, minValue;

    for (int start = 0; start < (size - 1); start++) {
        minIndex = start;
        minValue = array[start];
        for (int index = start + 1; index < size; index++) {
            if (array[index] < minValue) {
                minValue = array[index];
                minIndex = index;
            }
        }
        swap(array[minIndex], array[start]);
    }
}

void insertionSort(int array[], int size) {
    for (int itemsSorted = 1; itemsSorted < size; itemsSorted++) {
        int temp = array[itemsSorted];
        int loc = itemsSorted - 1;
        while (loc >= 0 && array[loc] > temp) {
            array[loc + 1] = array[loc];
            loc = loc - 1;
        }
        array[loc + 1] = temp;
    }
}

void printArray(int array[], int size, bool newline) {
    for (int i = 0; i < size; i++) {
        cout << array[i];
        if (newline)
            cout << endl;
        else
            cout << " ";
    }
    cout << endl;
}

void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}

```

```

#include <iostream>
#include <cstdlib>
#include <ctime>
#include <algorithm>
#include <chrono>

using namespace std;
using namespace std::chrono;

void quickSort(int*, int, int);
int partition(int*, int, int);
void swap(int&, int&);

```

```

void CopyArray(const int*, int*, int);
void printArray(int *array, int size, bool newline = false);

int main() {
    srand(time(0));
    int size = 0;

    cout << "Input array size: ";
    cin >> size;

    // Don't worry about the change in the way the array is created
    // at this point.
    // This puts the array in the heap so we can use larger arrays.

    int *A = new int[size];
    int *B = new int[size];
    for (int i = 0; i < size; i++) {
        A[i] = rand(); // Average
        //A[i] = i; // Best
        //A[i] = size - i; // Worst
    }

    CopyArray(A, B, size);

    auto start = high_resolution_clock::now();
    quickSort(A, 0, size - 1);
    auto stop = high_resolution_clock::now();
    auto duration = duration_cast<microseconds>(stop - start);

    cout << "Quick sort = " << duration.count() / 1000000.0 << " seconds"
        << endl;

    CopyArray(B, A, size);

    start = high_resolution_clock::now();
    sort(A, A + size);
    stop = high_resolution_clock::now();
    duration = duration_cast<microseconds>(stop - start);

    cout << "Algorithm sort = " << duration.count() / 1000000.0 << " seconds"
        << endl;

    delete[] A;
    delete[] B;

    return 0;
}

void CopyArray(const int *A, int *B, int size) {
    for (int i = 0; i < size; i++)
        B[i] = A[i];
}

void printArray(int *array, int size, bool newline) {
    for (int i = 0; i < size; i++) {
        cout << array[i];
        if (newline)
            cout << endl;
        else
            cout << " ";
    }
    cout << endl;
}

void quickSort(int *set, int start, int end) {
    int pivotPoint;
    if (start < end) {
        // Get the pivot point.
        pivotPoint = partition(set, start, end);
        // Sort the first sublist.
        quickSort(set, start, pivotPoint - 1);
        // Sort the second sublist.
        quickSort(set, pivotPoint + 1, end);
    }
}

```



```
int partition(int *set, int start, int end) {
    int pivotValue, pivotIndex, mid;
    mid = (start + end) / 2;
    swap(set[start], set[mid]);
    pivotIndex = start;
    pivotValue = set[start];
    for (int scan = start + 1; scan <= end; scan++) {
        if (set[scan] < pivotValue) {
            pivotIndex++;
            swap(set[pivotIndex], set[scan]);
        }
    }
    swap(set[start], set[pivotIndex]);
    return pivotIndex;
}

void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}
```