Oregon State University

CS 325 - Spring 2021

Prof. Julianne Schutford

Homework 1

Merge Sort and Insertion Sort

by

Abdullah Saydemir

saydemia@oregonstate.edu

April 10, 2021

Q1) Pseudocode for my algorithms is as follows.

```
FUNCTION insertion_sort (Vector V)
    FOR i <- 1 to length(V) DO
        pivot <- V[ i ]                         // take the current element
        j <- i                                  // and the position
        WHILE j > 0 and V[ j ] > pivot DO      // find an element less than pivot
            V [ j ] <- V [ j -1 ]
            j--
        END WHILE
        V[ j ] <- pivot                         // insert pivot
    END FOR
END FUNCTION
```

Merge sort consists of two algorithms. One for sorting vectors and one for merging two sorted vectors.

```
FUNCTION merge(Vector V, Vector L, Vector R)
    i,j,k   <- 0

    WHILE j < length(L) and k < length(R) DO   // compare the first elements in
        IF L[ j ] < R[ k ] THEN                 // vectors and insert smaller one
            V [ i ] <- L [ j ]
            j++
        ELSE THEN
            V [ i ] <- R [ k ]
            k++
        END IF
        i++
    END WHILE
    WHILE j < length(L) DO                      // if anything left in L take them
        V [ i ] <- L [ j ]
        i++, j++
    END WHILE
```

```
    WHILE k < length(R) DO                          // if anything left in R take them
        V [ i ] <- R [ k ]
        i++, k++
    END WHILE
END FUNCTION

FUNCTION merge_sort (Vector V)

    IF length(V) < 2 THEN                           // vectors of size 1 and 0
        return                                      // are already sorted
    END IF

    LET mid <- length(V) / 2
    LET L <- merge_sort( V [ 0 .. mid ] )                   // sort left half
    LET R <- merge_sort( V [ (mid+1) .. length( V ) ] )     // sort right half

    merge(V, L, R)                                          // merge
END FUNCTION
```

Pseudocode of *main()* for both insertsort.cpp and mergesort.cpp is as follows.

```
FUNCTION main()
    FILE input := read(data.txt)
    IF file is NOT correctly opened THEN
        print "Error reading file"
    ELSE THEN
        WHILE input has next integer DO
            LET line_size <- cin
            LET to_sort <- new Vector()
            FOR i <- 0 to line_size DO
                to_sort[ i ] <- cin
            END FOR
            merge_sort(to_sort)          // insertion_sort() can be called here

            FOR i <- 0 to length(to_sort) DO
                print to_sort[ i ] + "  "
            END FOR
        END WHILE
    END IF
END FUNCTION
```

Q2) To collect times from insertion sort and merge sort I had to change the code a little bit.

First, I included another for loop *(line 38 in Figure 1)* to get the average running time of the algorithm for each size. For each *n*, sorting algorithms run 3 times and running time of each pass is added up to a variable called *total_time_elapsed.* After 3 passes, the value stored in this variable is divided by 3 and passed to *cout*.

To measure time, I used *chrono* library *(lines 47 and 49 in Figure 1)* instead of *ctime.* Because I had some static casting issues with the compiler and could not work it out. An advantage of using the *chrono* library was that I did not need to increase the size in merge sort because I was able to measure the time in microseconds.

```
36        auto total_time_elapsed = 0.0;
37
38        for (auto attempts = 0; attempts < 3 ; ++attempts) // to smooth the running time
39        {
40            auto to_sort = vector<int>();
41
42            for (auto j = 0; j < i ; ++j)
43            {
44                to_sort.push_back(uni(rng));
45            }
46
47            auto start = chrono::steady_clock::now();
48            insertion_sort(to_sort);
49            auto end = chrono::steady_clock::now();
50
51            auto time = chrono::duration_cast<chrono::microseconds>(end - start).count(); // in microseconds
52            total_time_elapsed += time; // just add up, average is calculated later
```

(Figure 1)

Another detail is how the random integers are chosen because this may change the running time of insertion sort. I did not use *srand()* and *rand()* functions because it is known that these functions do not give uniformly random integers. Instead, I preferred a commonly used *Mersenne Twister (line 30 in Figure 2)* engine with *uniform_int_distibution (line 31 in Figure 2)* which provides both random and uniformly distributed integers.

```
26  int main()
27  {
28      // This code is to get uniformly distributed random integers.
29      random_device rd;
30      mt19937 rng(rd());
31      uniform_int_distribution<int> uni(0,10000);
```

(Figure 2)

Pseudocode for both *insertTime* and *mergeTime* is as follows:

```
FUNCTION main()
    Initialize the random generator engines
    LET N <- 5000

    FOR i <- N to 10*N DO                        // to collect multiple points
        LET total_time <- 0.0
        FOR attempts <- 0 to 3 DO               // to get average
            LET V <- new Vector ()
            FOR j <- 0 to i DO              // push random integers to vector
                V.push( random int )
            END FOR
            clock.start()
            merge_sort( V )                 // insertion_sort(V) can be called here
            clock.end()
            total_time += clock.end() - clock.start()
        END FOR
    print "Size " + i + " Time " + total_time / 3
    END FOR
END FUNCTION
```

Q3)

a) Running times are collected on *flip* servers. Before the experiment *uptime* command gave *1.28%* user activity on the servers which is very good to get reliable results. The data collected is as follows.

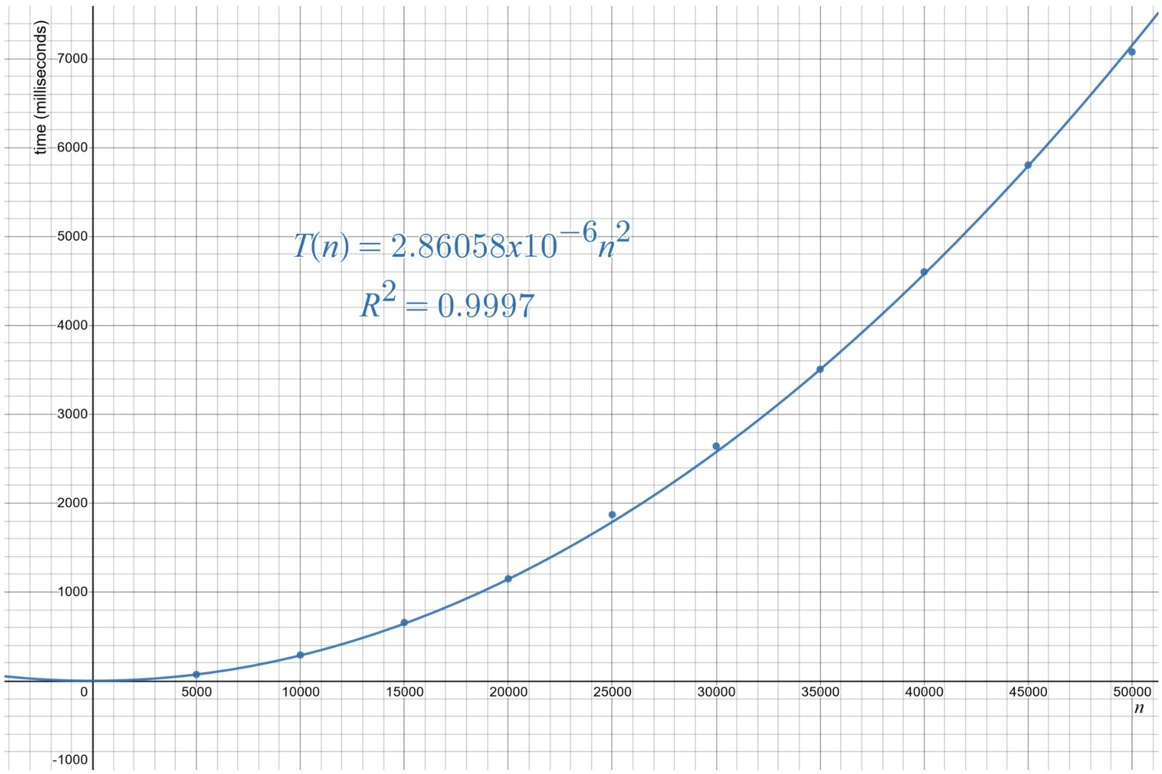
	Running Time (milliseconds)	
Size (n)	**Insertion Sort**	**Merge Sort**
5000	71.69	8.86
10000	291.52	17.93
15000	656.50	27.45
20000	1148.87	37.15
25000	1870.66	46.23
30000	2642.07	54.24
35000	3506.39	64.37
40000	4602.21	74.07
45000	5804.41	85.83
50000	7077.57	93.78

These results show average case running time for insertion sort. Because, integers were neither nearly sorted *(best case)* nor reversely sorted *(worst case)*. For merge sort, It doesn't really matter how sorted the data is. It is always *n * lgn* and we can call it average.

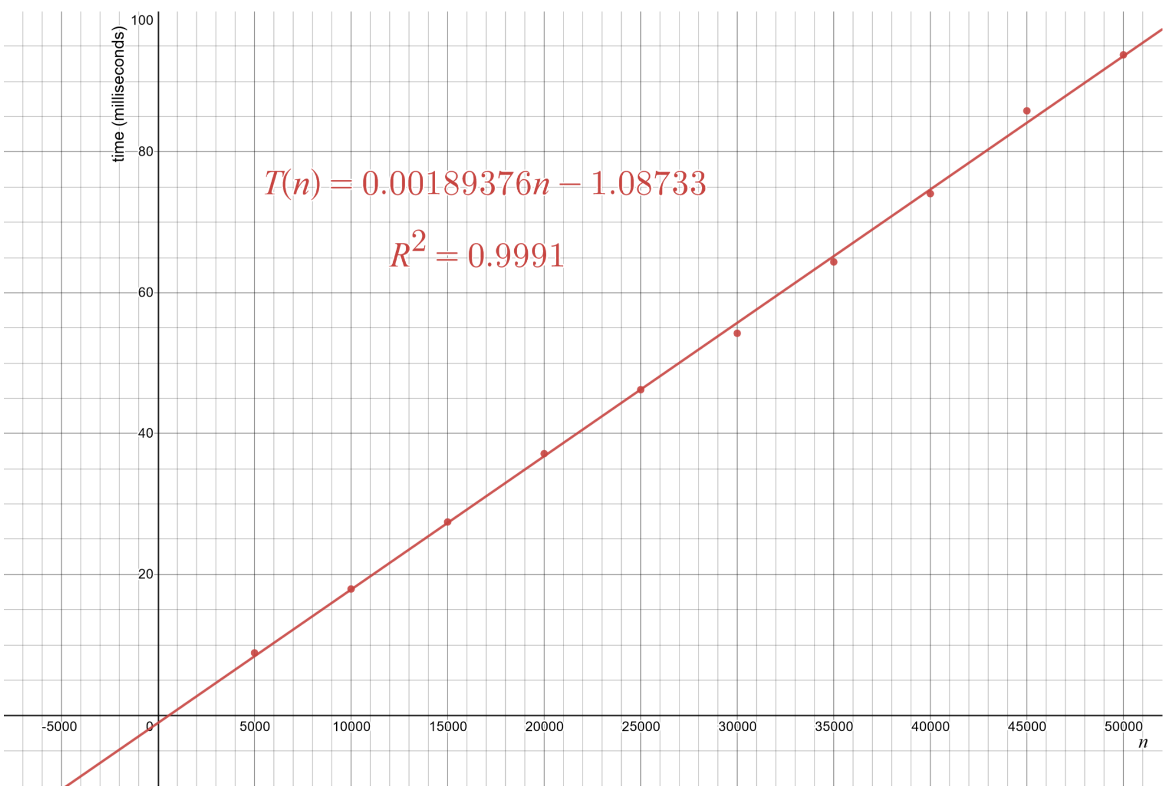
b) I used Desmos[1] to fit curves to collected running times. Following graphs show the best fit curves and R^2 values in individual graphs. Best fit curves were quadratic and linear curves for insertion sort and merge sort, respectively. For merge sort, the R^2 value of the linearithmic curve was less than that of the linear curve, so I used linear one. High R^2 values confirm that experimental running times are similar to theoretical running times.

[1] Link for my calculations : https://www.desmos.com/calculator/y6jeq1u5qj

Running Time for Insertion Sort

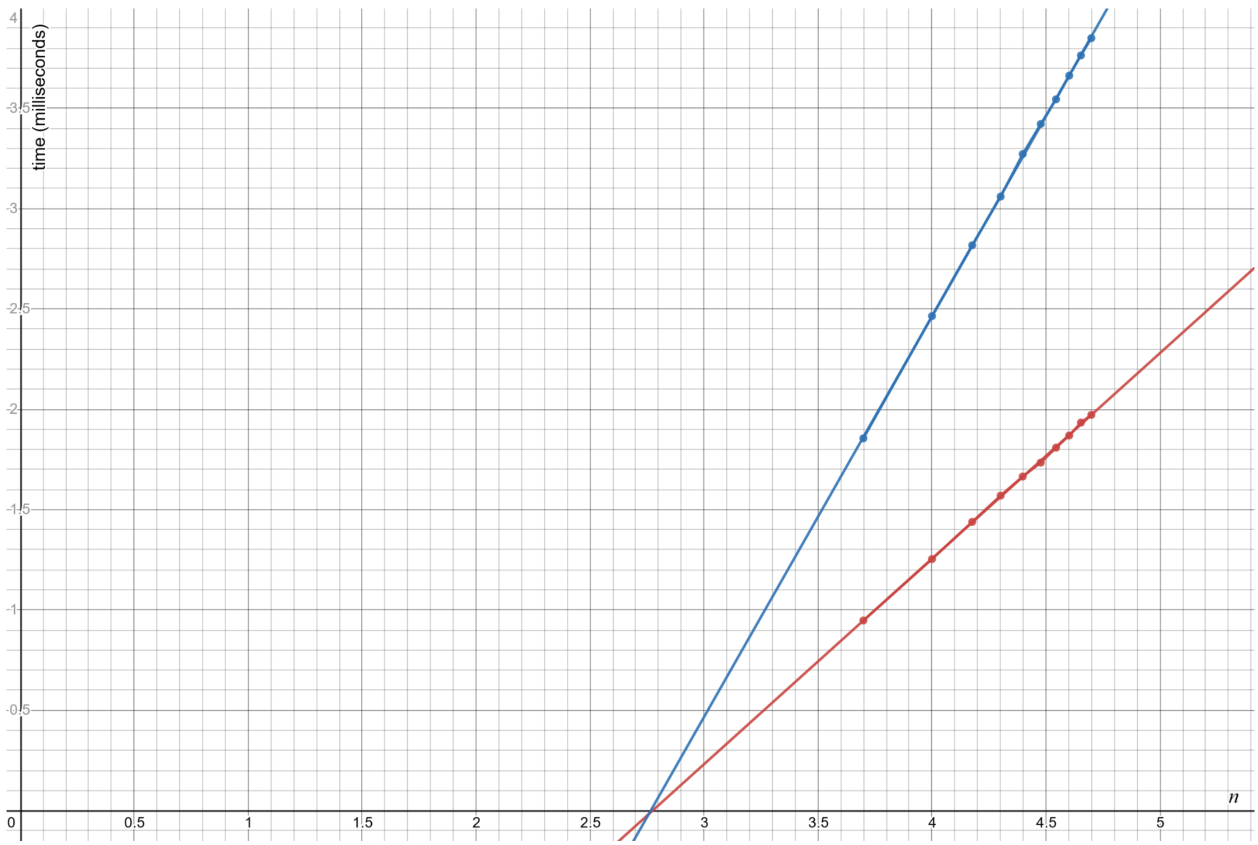


Running Time for Merge Sort



c) To plot the curves in one graph, I used log-log scale in both axes so that we can see both graphs more clearly. Slope of the blue line is *1.99597* and slope of red is *1.02439.*

Log-Log Plot of Running Times



d) To predict running times for size 500000, we can use best fit curves from the graphs.

- Merge Sort : $T(n) = 1.89376\, x\, 10^{-3}\, n - 1.08733$

$$T(500,000) = 1.89376\, x\, 10^{-3}\, x\, 500000 - 1.08733$$
$$T(500,000) = 945.79\, milliseconds \cong 1\, second$$

- Insertion Sort : $T(n) = 2.86058\, x\, 10^{-6}\, x\ n^{2}$

$$T(500,000) = 2.86058\, x\, 10^{-6}\, x\, 500000^{2}$$
$$T(500,000) = 715145\, milliseconds \cong 12\, minutes\, !$$

I actually checked both algorithms with size 500000 on *flip*. Here is the result.

```
Insertion Sort
   Size (n)       Time (ms)
    500000          743258

Merge Sort
   Size (n)       Time (ms)
    500000          1012.23
flip3 ~/cs325 213$
```