EE3980 Algorithms

Homework 1. Quadratic Sorts report

By 105061212 王家駿

1. Introduction

In this homework, we demonstrate four different sorting algorithms: selection sort, insertion sort, bubble sort, and shaker sort. Then we sort a pile of strings into alphabetical order with these algorithms.

At first, we read all input data and store them into a two-dimensional array of characters. Then, we sort the strings into lexicographical order for R times, and track the CPU time before and after the sorting. So, we could get the average time for sorting by the CPU time difference divided by R. We finally print out the sorting results and average sorting time as the output.

2. Function description

2.1. Main function

In our main function, we mainly call the other functions to do specific tasks. To choose the sorting algorithm, we use a variable *sorting_algorithm*, which is defined as a symbolic constant before.

```
// execute sorting based on the algorithm chosen
        if(sorting_algorithm == 0){
            SelectionSort(A, N);
        }
        else if(sorting_algorithm == 1){
            InsertionSort(A, N);
        }
        else if(sorting_algorithm == 2){
            BubbleSort(A, N);
        }
        else{
            ShakerSort(A, N);
        }
```

2.2. readInput

In this function, we first read the input data size *N* for N different strings. Based on the data size, we allocate a two-dimensional array of characters, where the size of the first dimension is *N* and 50 for the second dimension, since most of the English words have less than 50 letters.

```
// allocate dynamic memories for input 2D array
    // set at most 50 characters in a word
    data = (char**)malloc(sizeof(char*) * N);
    for(i = 0; i < N; i++){
        data[i] = (char*)malloc(sizeof(char) * 50);
    }
```

We also allocate a two-dimensional array *A*, which is used for sorting, with the same method.

Then we read all the input data, and store them into array *data*.

```
for(i = 0; i < N; i++){
        scanf("%s",data[i]);            // read input words
    }
```

2.3. printArray

In this function, we print out the current data in array *A*. And based on the *sorting_algorithm* we defined, print out the algorithm type and the input data size *N*.

2.4. copyArray

In this function, we copy all contents of array *A* from *A* to *data* by using the **strcpy** function.

```
for(i = 0; i < N; i++){
        strcpy(A[i], data[i]);          // copy string to array A[i]
    }
```

2.5. GetTime

In the function, we obtain the current local CPU time with **<sys/time.h>** library. At first, we construct a structure *tv* with type of **timeval**. The timeval structure

contains two variables: *tv_sec* for current time recorded in seconds, and *tv_usec* for current time recorded in microseconds.

```
struct timeval
{
    long tv_sec;    // recorded time in seconds
    long tv_usec;   // recorded time in microseconds
}
```

Then we use the **gettimeofday** function to record current time into *tv*, and return the time data.

```
gettimeofday(&tv, NULL);            // write local time into tv

return tv.tv_sec + tv.tv_usec * 0.000001;   // return time with microsecond
```

2.6. SelectionSort

In this this function, we sort the strings in lexicographical order by **selection sort**. The algorithm starts from the index 1 of the array *list*, then goes through the array until the index *N*. At each step *i*, we find the smallest element from index *i*+1 to *N* and store the index in *j*, then swap the value of *list[i]* and *list[j]*, where we can make sure that the strings are well sorted from 1 to *i*. Thus, after the iteration goes to the end, the whole array would be well sorted.

The pointer *tp* in the function is allocated as one-dimensional array of characters, for the use of temporary when swapping the strings. It is also demonstrated in the other three sorting algorithms.

The time complexity of the algorithm is $O(n^2)$.

```
void SelectionSort(char** list,int n)       // in-place selection sort
{
    int i, j, k;                // loop index

    // allocate dynamic memories of string for swapping
    char* tp = (char*)malloc(sizeof(char) * 50);

    for(i = 0; i < n; i++){          // i runs through the array
        j = i;
```

```
        for(k = i+1; k < n; k++){        // search for the smallest
                                // from list[i+1] to list[n-1]
            if(strcmp(list[k], list[j]) < 0){
                j = k;              // if found, remember it in j
            }
        }
    // swap list[i] and list[j]
        strcpy(tp, list[i]);
        strcpy(list[i], list[j]);
        strcpy(list[j], tp);
    }

    free(tp);                   // free dynamic memories
}
```

2.7. InsertionSort

In this this function, we sort the strings in lexicographical order by **insertion sort**. The algorithm starts from the index 1 of the array *list*, then goes through the array until the index *N*. At each step *j*, we find an index *i* from *j*-1 to 1 such that *list[i] > list[j]*, and this is the place that *list[j]* should be placed, where we could make sure that all elements from 1 to *i* are smaller than *list[j]*, and the list is well sorted from 1 to *j*. Thus, after the iteration goes to the end, the whole array would be well sorted.

The time complexity of the algorithm is $O(n^2)$.

```
void InsertionSort(char** list,int n)       // in-place insertion sort
{
    int i, j;                   // loop index

    // allocate dynamic memories of string for swapping
    char* tp = (char*)malloc(sizeof(char) * 50);

    for(j = 1; j < n; j++){          // j runs through the array
        strcpy(tp, list[j]);            // save content of list[j] to tp
        i = j-1;

    // from list[j-1], find i for list[i] > tp
        while(i >= 0 && strcmp(tp, list[i]) < 0){
            strcpy(list[i+1], list[i]);
```

```
            i--;
        }
        strcpy(list[i+1], tp);          // place tp to the proper place
    }

    free(tp);                   // free dynamic memories
}
```

2.8. BubbleSort

In this this function, we sort the strings in lexicographical order by **bubble sort**. The algorithm starts from the index 1 of the array *list*, then goes through the array until the index *N*-1. At each step *i*, we start from *N* and check during the decrement until *i*+1. If the former one is bigger than the latter one, swap them so the smallest one form *i* to *N* would be placed to *i*. At the end of this step, the value from index 1 to *i* would be well ordered. Thus, after the iteration goes to the end, the whole array would be well sorted.

The time complexity of the algorithm is $O(n^2)$.

```
void BubbleSort(char** list,int n)      // in-place bubble sort
{
    int i, j;                   // loop index

    // allocate dynamic memories of string for swapping
    char* tp = (char*)malloc(sizeof(char) * 50);

    for(i = 0; i <= n-2; i++){           // i runs through the array
        for(j = n-1; j >= i+1; j--){     // j runs from n-1 to i+1
        // if list[j] < list[j-1], swap them
            if(strcmp(list[j], list[j-1]) < 0){
                strcpy(tp, list[j]);
                strcpy(list[j], list[j-1]);
                strcpy(list[j-1], tp);
            }
        }
    }

    free(tp);                   // free dynamic memories
}
```

2.9. ShakerSort

In this this function, we sort the strings in lexicographical order by **shaker sort**. Shaker sort is implemented based on bubble sort, but with two indexes: *l* starts form 1, and *r* starts from *N*. The iteration ends when the increment of *l* and decrement of *r* meet. From the left end, we sort the array incrementally while in decremental order from the right end. Thus, after the iteration goes to the end, the whole array would be well sorted.

The time complexity of the algorithm is $O(n^2)$.

```
1.  void ShakerSort(char** list,int n)      // in-place shaker sort
2.  {
3.      int j;                  // loop index
4.      int l = 0;                  // loop index
5.      int r = n-1;                // loop index
6.      // allocate dynamic memories of string for swapping
7.      char* tp = (char*)malloc(sizeof(char) * 50);
8.
9.      while(l <= r){
10.         for(j = r; j >= l+1; j--){       // word exchange from r to l
11.             if(strcmp(list[j], list[j-1]) < 0){
12.         // swap list[j] and list[j-1]
13.                 strcpy(tp, list[j]);
14.                 strcpy(list[j], list[j-1]);
15.                 strcpy(list[j-1], tp);
16.             }
17.         }
18.         l++;
19.
20.         for(j = l; j <= r-1; j++){       // word exchange from l to r
21.             if(strcmp(list[j], list[j+1]) > 0){
22.         // swap list[j] and list[j+1]
23.                 strcpy(tp, list[j]);
24.                 strcpy(list[j], list[j+1]);
25.                 strcpy(list[j+1], tp);
26.             }
27.         }
28.         r--;
```

```
29.     }
30.
31.     free(tp);                    // free dynamic memories
32. }
```

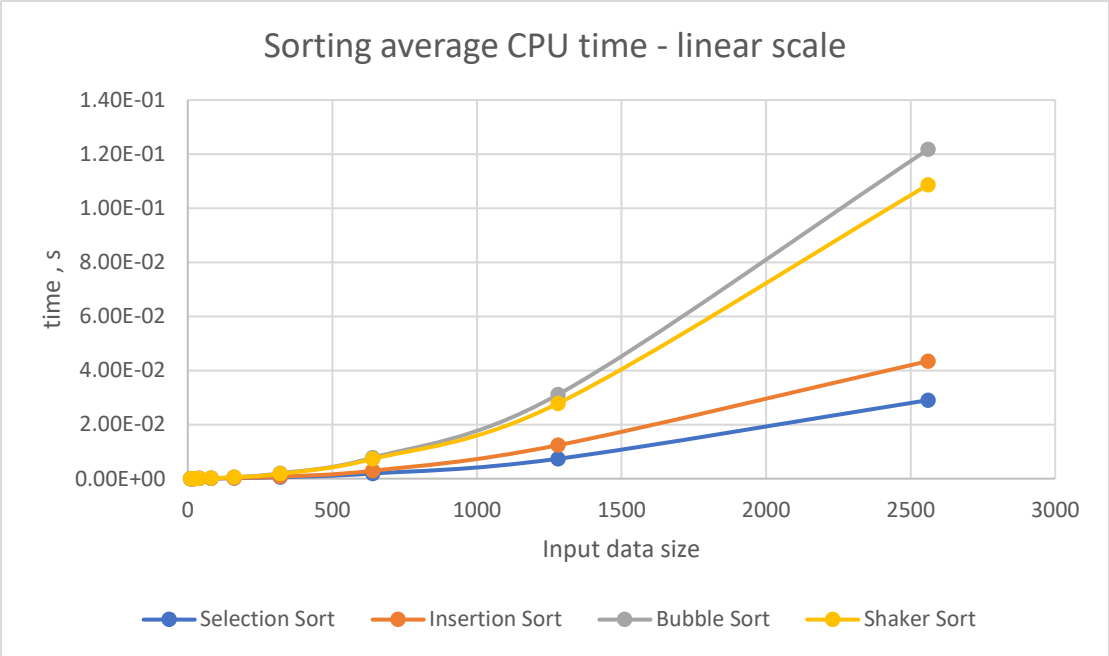
2.10. freeMemory

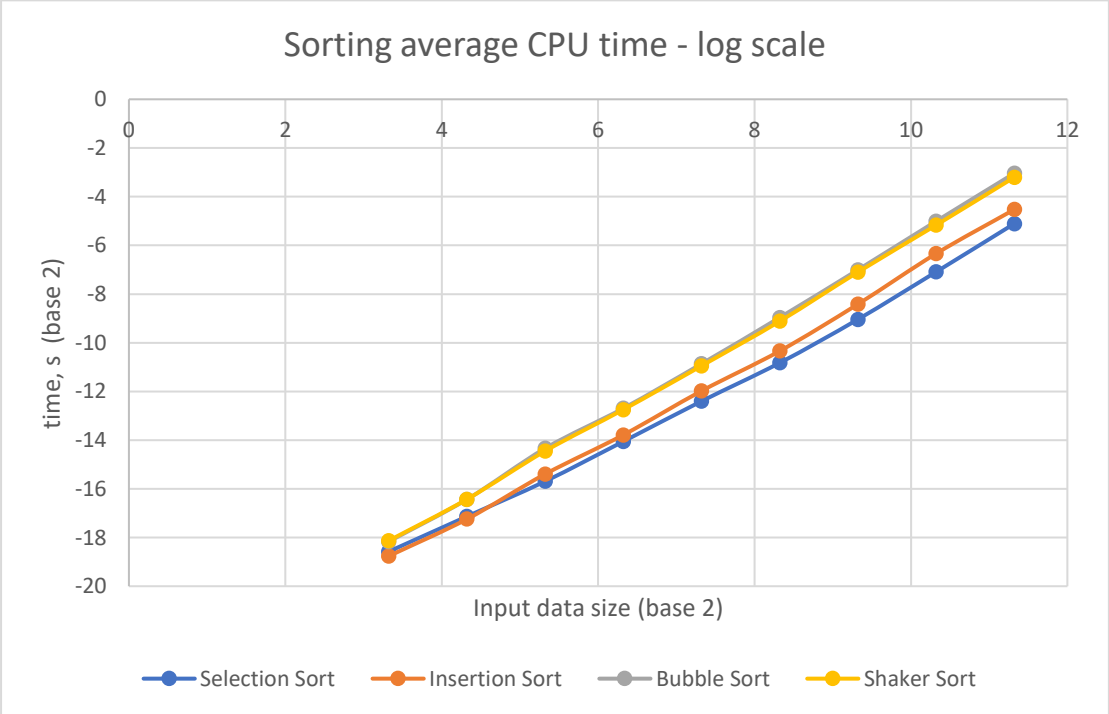
In this function, we release the dynamic memories that we allocated before, including array *data* and *A*.

3. Executing results

For R = 500, run the testing data from s1.dat to s9.dat with different input data size, and record the average CPU time used.

Data size (*N*)	Selection Sort	Insertion Sort	Bubble Sort	Shaker Sort
10	2.558 μ s	2.246 μ s	3.396 μ s	3.486 μ s
20	6.952 μ s	6.448 μ s	11.30 μ s	11.34 μ s
40	18.90 μ s	23.22 μ s	48.32 μ s	44.74 μ s
80	58.77 μ s	70.76 μ s	151.0 μ s	145.5 μ s
160	185.0 μ s	247.8 μ s	536.5 μ s	505.9 μ s
320	551.0 μ s	772.0 μ s	2.002 ms	1.802 ms
640	1.901 ms	2.932 ms	7.788 ms	7.300 ms
1280	7.342 ms	12.36 ms	31.07 ms	27.81 ms
2560	29.01 ms	43.44 ms	121.7 ms	108.6 ms





4. Result analysis and conclusion

From the graphs, we could find that the four algorithms have similar trends. As the input data size grows up, the average executing time increases by about n^2, and the fact matches our prediction – the time complexity of $O(n^2)$.

Though the four algorithms have the same time complexities, there exist large differences in the real executing time. The time consumed for large input data size

with these sorting algorithms is: bubble sort > shaker sort > insertion sort > selection sort, and the former two are especially inefficient than the latter two.

The result could be explained by the structure of the algorithm itself. For selection sort and insertion sort, there will be at most one string pair needed to be swapped in each iteration step. Yet, for bubble sort and shaker sort, there may be many swaps occurred in one iteration step. In each swapping, the strings might be copied for three times with temporary by the **strcpy** function, which might consume a lot of time during executing. Thus, a large difference between the algorithms may occur.

Although the time complexities are the same, the real executing times could differ a lot. Thus, it’s important to realize the details of the algorithms and properly choose the method for use.