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.

```
1.  // execute sorting based on the algorithm chosen
2.          if(sorting_algorithm == 0){
3.              SelectionSort(A, N);
4.          }
5.          else if(sorting_algorithm == 1){
6.              InsertionSort(A, N);
7.          }
8.          else if(sorting_algorithm == 2){
9.              BubbleSort(A, N);
10.         }
11.         else{
12.             ShakerSort(A, N);
13.         }
```

Can use switch instead of cascaded-if.

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 $O(n^2)$. How?

Use pseudo code.

```
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--;
```

```
    }

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

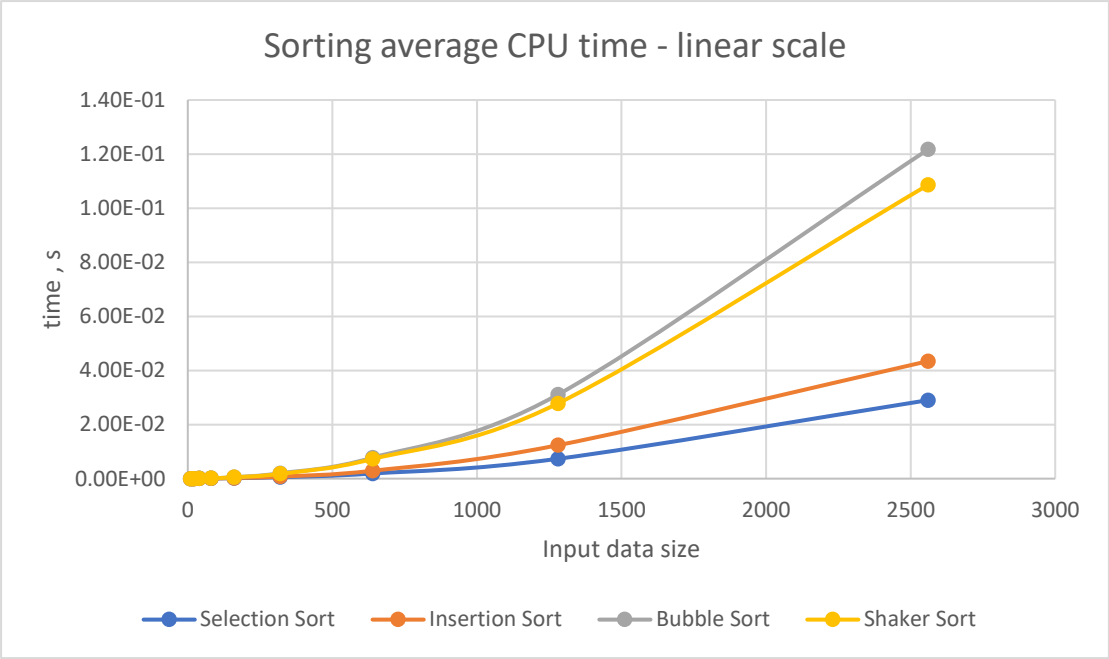
2.10. freeMemory

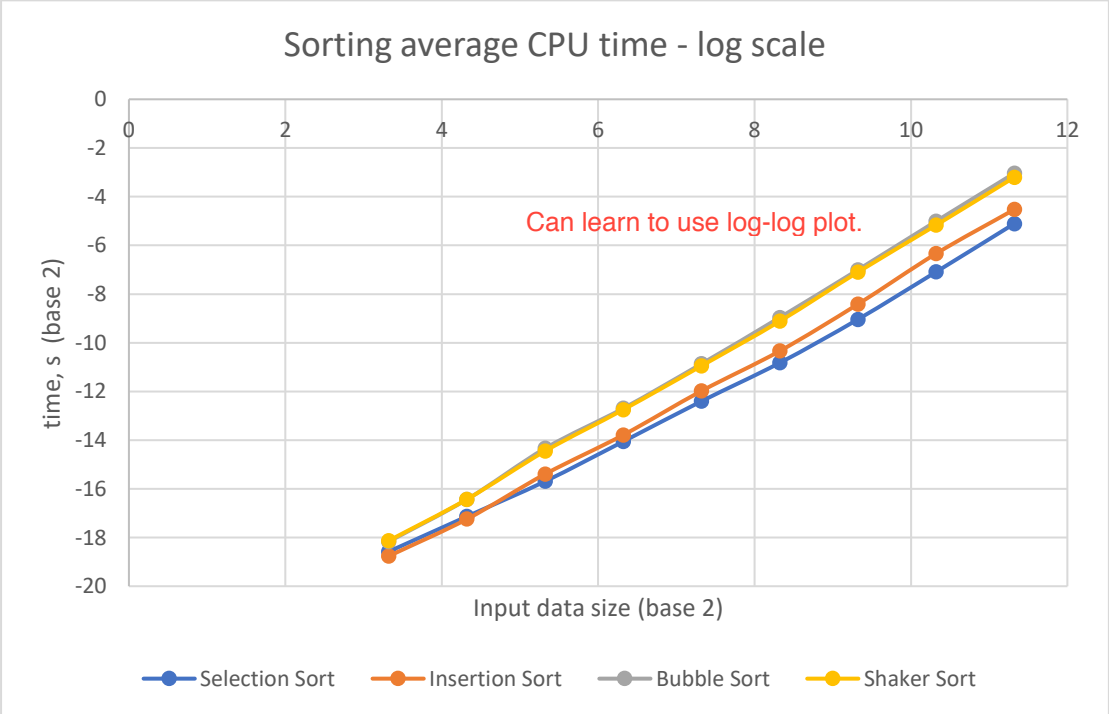
In this function, we release the dynamic memories that we allocated before, including array *data* and *A*.

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 $O(n^2)$. How?

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. Is this correct?

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.

Score: 74

———-

o. See return.

[Writing] Should use double space report format.

[Approach] Should explain your algorithms in pseudo codes.

[Analysis] Should derive time complexity for each algorithm.

[Analysis] Should derive space complexity for each algorithm.

[Results] Can correlate CPU times to analytical complexity.

[Coding] Should use variable size strings.

[Coding] Use pointer for swapping.

[Coding] 'strcpy(data[i], data[j])' can have 'bus error'.

hw01.c

```
1  /* EE3980 HW01 Quadratic Sorts
2   * 105061212, 王家駿
3   * 2019/03/04
4   */
5
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <string.h>
9  #include <sys/time.h>
10
11 #define SORTING_ALGORITHM 0
12 /* modify the value to choose the sorting algorithm
13  * 0: Selection Sort
14  * 1: Insertion Sort
15  * 2: Bubble Sort
16  * 3 or others: Shaker Sort
17  */
18
19 int N;                      // input size
20 char** data;                    // input data
21 char** A;                   // array to be sorted
22 int R = 500;                    // number of repetitions
23 int sorting_algorithm;             // type of sorting algorithm
24
25 void readInput(void);              // read all inputs
26 void printArray(char** A);         // print the content of array A
27 void copyArray(char** data, char** A);      // copy data to array A
28 double GetTime(void);              // get local time in seconds
29 void SelectionSort(char** list,int n);      // in-place selection sort
30 void InsertionSort(char** list,int n);      // in-place insertion sort
31 void BubbleSort(char** list,int n);     // in-place bubble sort
32 void ShakerSort(char** list,int n);     // in-place shaker sort
33 void freeMemory(void);              // free all dynamic memories
34
35 int main(void)
36 {
37     int i;                  // loop index
38     double t;                   // for CPU time tracking
39
40     readInput();                // read input data
41
42     sorting_algorithm = SORTING_ALGORITHM;  // type of sorting algorithm
43
44     t = GetTime();              // initialize time counter
45
46     for(i = 0; i < R; i++){
47         copyArray(data, A);         // initialize array for sorting
48
```

```
49      // execute sorting based on the algorithm chosen
50          if(sorting_algorithm == 0){
51              SelectionSort(A, N);
52          }
53          else if(sorting_algorithm == 1){
54              InsertionSort(A, N);
55          }
56          else if(sorting_algorithm == 2){
57              BubbleSort(A, N);
58          }
59          else{
60              ShakerSort(A, N);
61          }
62
63          if (i == 0) printArray(A);      // print sorted results
64      }
65
66      t = (GetTime() - t) / R;            // calculate CPU time
67                          // per iteration
68      printf("  CPU time = %e seconds\n",t);  // print out CPU time
69
70      freeMemory();               // free dynamic memories
71
72      return 0;
73  }
74
75  void readInput(void)                // read all inputs
76  {
77      int i;                  // loop index
78
79      scanf("%d",&N);             // read input size
80
81      // allocate dynamic memories for input 2D array
82      // set at most 50 characters in a word
83      data = (char**)malloc(sizeof(char*) * N);
84      for(i = 0; i < N; i++){
85          data[i] = (char*)malloc(sizeof(char) * 50);
            Can use variable size array.
86      }
87
88      // allocate dynamic memories for the copied input data
89      A = (char**)malloc(sizeof(char*) * N);
90      for(i = 0; i < N; i++){
            This loop is not needed.
91          A[i] = (char*)malloc(sizeof(char) * 50);
92      }
93
94      for(i = 0; i < N; i++){
95          scanf("%s",data[i]);            // read input words
96      }
```

```
 97 }
 98
 99 void printArray(char** A)           // print the content of array A
100 {
101     int i;                  // loop index
102
103     for(i = 0; i < N; i++){
104         printf("%d %s\n",i+1 ,A[i]);        // print the index and words
105                         // after sorted
106     }
107
108     // print the type of sorting algorithm
109     if(sorting_algorithm == 0){
110         printf("Selection sort:\n");
111     }
112     else if(sorting_algorithm == 1){
113         printf("Insertion sort:\n");
114     }
115     else if(sorting_algorithm == 2){
116         printf("Bubble sort:\n");
117     }
118     else{
119         printf("Shaker sort:\n");
120     }
121
122     printf("  N = %d\n",N);         // print the input size
123 }
124
125 void copyArray(char** data, char** A)       // copy data to array A
126 {
127     int i;                  // loop index
128
        this function can be more efficient.
129     for(i = 0; i < N; i++){
130         strcpy(A[i], data[i]);          // copy string to array A[i]
131     }
132 }
133
134 double GetTime(void)                // get local time in seconds
135 {
136     struct timeval tv;              // time interval structure
137
138     gettimeofday(&tv, NULL);            // write local time into tv
139
140     return tv.tv_sec + tv.tv_usec * 0.000001;   // return time with microsecond
141 }
142
143 void SelectionSort(char** list,int n)       // in-place selection sort
144 {
145     int i, j, k;                // loop index
```

```
146
147     // allocate dynamic memories of string for swapping
148     char* tp = (char*)malloc(sizeof(char) * 50);
        Is this really needed?
149
150     for(i = 0; i < n; i++){         // i runs through the array
151         j = i;
152         for(k = i+1; k < n; k++){       // search for the smallest
153                         // from list[i+1] to list[n-1]
154             if(strcmp(list[k], list[j]) < 0){
155                 j = k;              // if found, remember it in j
156             }
157         }
158     // swap list[i] and list[j]
159         strcpy(tp, list[i]);
160         strcpy(list[i], list[j]);
            This can cause segmentation fault if i = j
161         strcpy(list[j], tp);
162     }
163
164     free(tp);                   // free dynamic memories
165 }
166
167 void InsertionSort(char** list,int n)       // in-place insertion sort
168 {
169     int i, j;                   // loop index
170
171     // allocate dynamic memories of string for swapping
172     char* tp = (char*)malloc(sizeof(char) * 50);
173
174     for(j = 1; j < n; j++){         // j runs through the array
175         strcpy(tp, list[j]);            // save content of list[j] to tp
176         i = j-1;
177
178     // from list[j-1], find i for list[i] > tp
179         while(i >= 0 && strcmp(tp, list[i]) < 0){
180             strcpy(list[i+1], list[i]);
181             i--;
182         }
183         strcpy(list[i+1], tp);          // place tp to the proper place
184     }
185
186     free(tp);                   // free dynamic memories
187 }
188
189 void BubbleSort(char** list,int n)      // in-place bubble sort
190 {
191     int i, j;                   // loop index
192
193     // allocate dynamic memories of string for swapping
```

```
194     char* tp = (char*)malloc(sizeof(char) * 50);
195 
196     for(i = 0; i <= n-2; i++){          // i runs through the array
197         for(j = n-1; j >= i+1; j--){        // j runs from n-1 to i+1
198         // if list[j] < list[j-1], swap them
199             if(strcmp(list[j], list[j-1]) < 0){
200                 strcpy(tp, list[j]);
201                 strcpy(list[j], list[j-1]);
202                 strcpy(list[j-1], tp);
203             }
204         }
205     }
206 
207     free(tp);                   // free dynamic memories
208 }
209 
210 void ShakerSort(char** list,int n)      // in-place shaker sort
211 {
212     int j;                  // loop index
213     int l = 0;                  // loop index
214     int r = n-1;                // loop index
215     // allocate dynamic memories of string for swapping
216     char* tp = (char*)malloc(sizeof(char) * 50);
217 
218     while(l <= r){
219         for(j = r; j >= l+1; j--){      // word exchange from r to l
220             if(strcmp(list[j], list[j-1]) < 0){
221         // swap list[j] and list[j-1]
222                 strcpy(tp, list[j]);
223                 strcpy(list[j], list[j-1]);
224                 strcpy(list[j-1], tp);
225             }
226         }
227         l++;
228 
229         for(j = l; j <= r-1; j++){      // word exchange from l to r
230             if(strcmp(list[j], list[j+1]) > 0){
231         // swap list[j] and list[j+1]
232                 strcpy(tp, list[j]);
233                 strcpy(list[j], list[j+1]);
234                 strcpy(list[j+1], tp);
235             }
236         }
237         r--;
238     }
239 
240     free(tp);                   // free dynamic memories
241 }
242 
243 void freeMemory(void)               // free all dynamic memories
```

```
244 {
245     int i;                  // loop index
246 
247     for(i = 0; i < N; i++){         // free memories of array data
248         free(data[i]);
249     }
250     free(data);
251 
252     for(i = 0; i < N; i++){         // free memories of array A
253         free(A[i]);
254     }
255     free(A);
256 }
257 
```