EE 3980 Algorithms

Homework 1. Quadratic Sorts Report

105061110 周柏宇

2020/3/13

1. Introduction

In this homework, we implemented 4 sorting algorithms: selection sort, insertion sort, bubble sort and shaker sort. To measure the performance, we run each of the algorithm to sort different number of English words in-place and in ascending order for 500 times to calculate the average CPU run time. Finally, we print out the sorted list, name of sorting algorithm, size of inputs and the average execution times.

2. Analysis & Implementation

2.1 Selection Sort

```
1.  Algorithm SelectionSort(A, n)
2.  {
3.      for i := 1 to n do {
4.          j := i;                              // initialize j to be i
5.          for k := i + 1 to n do  // find the smallest in A[i + 1 : n]
6.              if (A[k] < A[j]) then j := k;   //found, record the index
7.          t := A[i]; A[i] := A[j]; A[j] := t; // swap A[i] and A[j]
8.      }
9.  }
```

In selection sort, we first search for the smallest element in A[i+1, n] and switch the smallest element found with A[i]. As i increase, we gradually finish the sort.

increases

The outer loop will execute n times and the inner loop will execute n – i times.

$$\sum_{i=1}^{n} n - i = n^2 - \frac{n(n+1)}{2} = \frac{n(n-1)}{2}$$

Therefore, the selection sort has time complexity $O(n^2)$ and space complexity $O(1)$ since there is no extra memory required relating to the input size.

2.2 Insertion Sort

```
1.  Algorithm InsertionSort(A, n)
2.  {
3.      for j := 2 to n do {     // assume A[1 : j - 1] already sorted
4.          item := A[j];        // move A[j] to its proper place
5.          i := j - 1;          // initialize i to be j - 1
6.          while ((i >= 1) and (item < A[i])) do {
7.                                   // find i such that A[i] <= A[j]
8.              A[i + 1] := A[i];    // move A[i] up by one position
9.              i := i - 1;
10.         }
11.         A[i + 1] := item;        // move A[j] to A[i + 1]
12.     }
13. }
```

In insertion sort, we start at the second element of A and compare it with the elements to its left (smaller index). If the elements to its left is bigger than A[j], then move it to the right. When the while loop stops, it means we either encountered a element smaller than A[j] or A[j] is the smallest among the elements to its left. We fill A[i + 1] with A[j].

The outer loop executes n – 1 times and the while loop has the worst case of j – 1 times execution.

$$\sum_{j=2}^{n} j - 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

which is $O(n^2)$ in time complexity and $O(1)$ in space complexity.

2.3 Bubble Sort

```
Algorithm BubbleSort(A, n)
{
    for i := 1 to n - 1 do {         // find the smallest item for A[i]
        for j := n to i + 1 step -1 do {
            if (A[j] < A[j - 1]) {   // swap A[j] and A[j - 1]
                t := A[j]; A[j] := A[j - 1]; A[j - 1] := t;
            }
        }
    }
}
```

In bubble sort, we keep comparing the contiguous elements starting from the right and move the smaller one to the left. After the inner loop is done, we can make sure the smallest element in A[i : n] has moved to A[i].

The outer loop executes n – 1 times and the inner loop executes n – i times. The calculation is similar to selection sort, which gives the bubble sort $O(n^2)$ time complexity and $O(1)$ space complexity.

2.4 Shaker Sort

```
Algorithm ShakerSort(A, n)
{
    l := 1; r := n;
    while l <= r do {
        for j := r to l + 1 step -1 do { // element exchange from r to l
            if (A[j] < A[j - 1]) {       // swap A[j] and A[j - 1]
                t := A[j]; A[j] := A[j - 1]; A[j - 1] := t;
            }
        }
        l := l + 1;
        for j := l to r - 1 do {         // element exchange from l to r
            if (A[j] > A[j + 1]) {       // swap A[j] and A[j + 1]
                t := A[j]; A[j] := A[j + 1]; A[j + 1] := t;
            }
        }
        r := r - 1;
    }
```

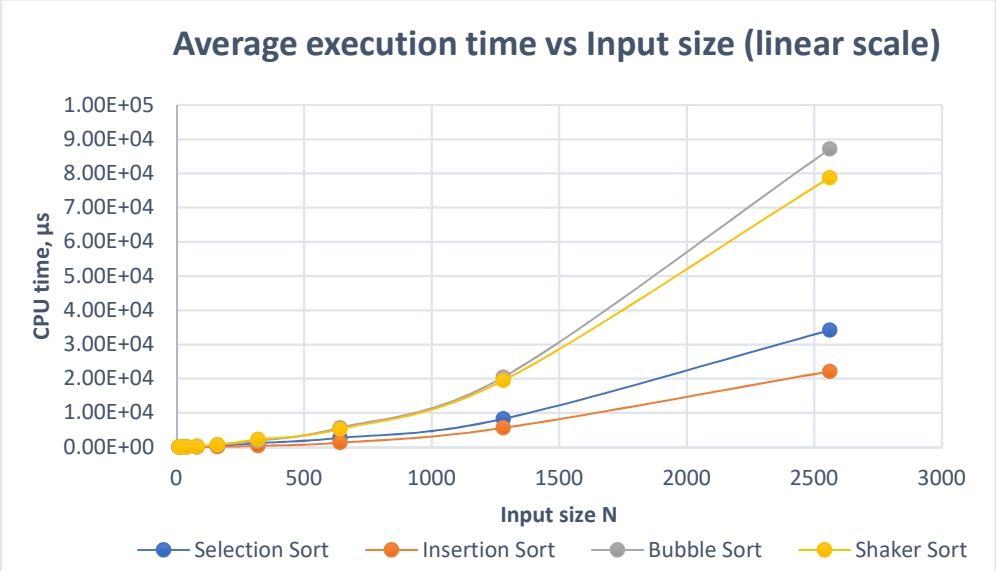
```
18. }
```

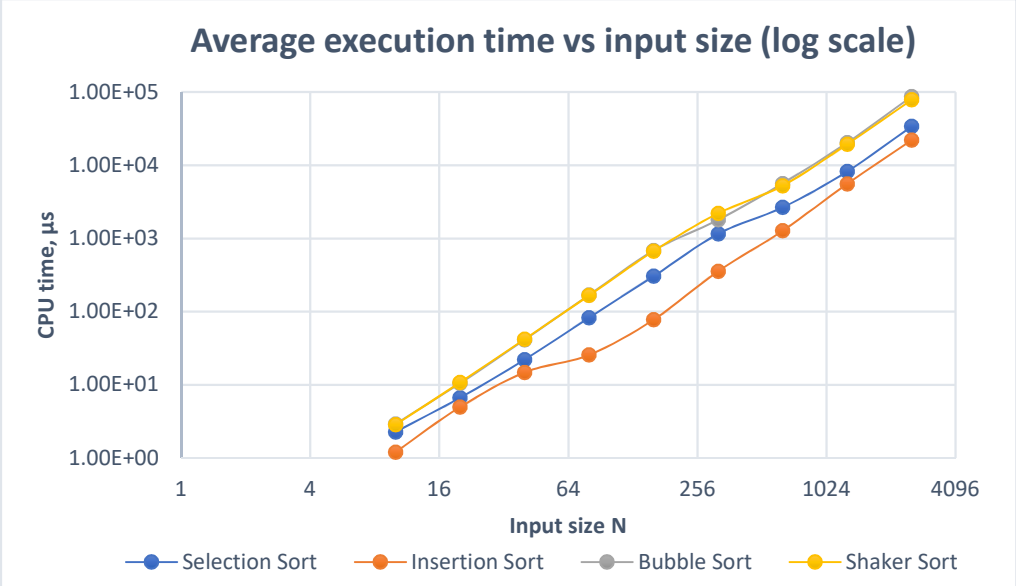
The shaker sort is very similar to the bubble sort. Inside the while loop it performs two bubble sorts – one from the right and one from the left. After each bubble sort, the unsorted area reduces by one. Although under some cases it can be more efficient than bubble sort, for the worst case shaker sort has the same time and space complexity as bubble sort – $O(n^2)$ and $O(1)$ respectively.

3. Result and Observation

Input size N	Selection Sort	Insertion Sort	Bubble Sort	Shaker Sort
10	2.28357	1.21212	2.92778	2.85769
20	6.68621	4.98009	10.5400	10.8159
40	22.1720	14.8222	41.3857	42.2621
80	82.8619	25.6281	169.110	166.694
160	305.348	78.0401	684.662	673.282
320	1154.24	357.700	1798.58	2205.04
640	2660.69	1275.13	5642.05	5257.94
1280	8227.41	5581.26	20420.3	19429.4
2560	34145.1	22112.4	87123.1	78771.4

Table 1. Average execution time [μ s] vs input data size





Although the 4 algorithms have same time complexity i.e. the quadratic trend, the actual run time varies. For the bubble sort and shaker sort, since they perform similar operations but only in different orders, we expect them to have similar performance over random inputs. But inside the loop they may need to do multiple times of swapping, it contributes to longer execution time comparing to selection sort and insertion sort, which only do at most two assignments. As for insertion sort, the inner loop has a chance to exit early compared to that of selection sort, which has to run n – i times deterministically. Therefore, we can infer that insertion sort will execute faster than selection sort for our implementation.

hw01.c

```
1 // EE3980 HW01 Quadratic Sorts
2 // 105061110, 周柏宇
3 // 2020/03/12
4
5 #include <stdio.h>
6 #include <stdlib.h>
7 #include <string.h>
8 #include <sys/time.h>
9 #define METHOD 2                    // algorithm index
10                                    // 0: selection sort 1: insertion sort
11                                    // 2: bubble sort    3: shaker sort
12 int N;                             // input size
13 char **data;                       // input data
14 char **A;                          // array to be sorted
15 int R = 500;                       // number of repetitions
16 char algNames[][10] = {            // names of sorting algorithms available
17     "Selection", "Insertion", "Bubble", "Shaker"
18 };
19 void (*Sort[4])(char **list, int n);// to store pointer of sorting algorithms
   void (*Sort[4])(char **list, int n); // to store pointer of sorting algorithms
20
21 void readInput(void);                   // read all inputs
22 void printArray(char **A);              // print the content of array A
23 void copyArray(char **data, char **A);  // copy data to array A
24 double GetTime(void);                   // get local time in seconds
25 void SelectionSort(char **list, int n); // in-place selection sort
26 void InsertionSort(char **list, int n); // in-place insertion sort
27 void BubbleSort(char **list, int n);    // in-place bubble sort
28 void ShakerSort(char **list, int n);    // in-place shaker sort
29 void swap(char **a, char **b);          // swap two words
30 void freeMemory(char **list, int n);    // free allocated memory
31
32 int main(void)
33 {
34     int i;                                  // index
35     double t;                               // local time
36
37     readInput();                            // store inputs in array data
38     t = GetTime();                          // get local time
39     A = (char **)malloc(sizeof(char *) * N);// allocate memory for copying
       A = (char **)malloc(sizeof(char *) * N);  // allocate memory for copying
40     Sort[0] = SelectionSort;                // store function pointer in array
41     Sort[1] = InsertionSort;
42     Sort[2] = BubbleSort;
43     Sort[3] = ShakerSort;
44     for (i = 0; i < R; i++) {
45         copyArray(data, A);                 // copy the data to array A
46         (*Sort[METHOD])(A, N);              // execute the sorting algorithm
```

```
47          if (i == 0) {                       // if it is the first execution
48              printArray(A);                  // print out the sorted array
49          }
50      }
51      t = (GetTime() - t) / R;                // calculate the average run time
52      printf("%s Sort:\n", algNames[METHOD]); // print out the algorithm name
53      printf("  N = %d\n", N);                // print out the input size
54      printf("  CPU time = %.5e seconds\n", t);   // print out average CPU time
55      freeMemory(data, N);                    // free array data
56      free(A);                                // free array A
57
58      return 0;
59  }
60
61  void readInput(void)            // read all inputs
62  {
63      int i;                      // index
64      char tmpWord[1000];         // store input temporarily
65
66      scanf("%d", &N);            // input number of entries
67      data = (char **)malloc(sizeof(char *) * N); // allocate memory for pointers
68      for (i = 0; i < N; i++) {
69          scanf("%s", tmpWord);   // input a word
70          // allocate memory just enough to fit the word
71          data[i] = (char *)malloc(sizeof(char) * (strlen(tmpWord) + 1));
72          strcpy(data[i], tmpWord);   // transfer the input to array
73      }
74  }
75
76  void printArray(char **A)               // print the content of array A
77  {
78      int i;                              // index
79
80      for (i = 0; i < N; i++) {
81          printf("%d %s\n", i + 1, A[i]); // print the index and array content
82      }
83  }
84
85  void copyArray(char **data, char **A)   // copy data to array A
86  {
87      int i;                              // index
88
89      for (i = 0; i < N; i++) {
90          A[i] = data[i];                 // copy content from array data to A
91      }
92  }
93
94  double GetTime(void)                        // get local time in seconds
95  {
96      struct timeval tv;                      // variable to store time
```

```
 97
 98     gettimeofday(&tv, NULL);                // get local time
 99     return tv.tv_sec + 1e-6 * tv.tv_usec;   // return local time in seconds
100 }
101
102 void SelectionSort(char **list, int n)  // in-place selection sort
103 {
104     int i, j, k;                        // index
105
106     for (i = 0; i < n; i++) {
107         j = i;                          // initialize j with i
108         for (k = i + 1; k < n; k++) {
109             if (strcmp(list[k], list[j]) < 0) { // if word at k is smaller
110                 j = k;                  // store the index at j
111             }
112         }
113         swap(&list[i], &list[j]);       // swap the words at index i and j
114     }
115 }
116
117 void InsertionSort(char **list, int n)  // in-place insertion sort
118 {
119     int j, i;                           // index
120     char *tmp;                          // temporary char pointer
121
122     for (j = 1; j < n; j++) {       // assume list[0 : j - 1] already sorted
123         tmp = list[j];              // copy the word at index j
124         i = j - 1;                  // initialize i with j - 1
125         // repeat until list[i] is smaller
126         while ((i >= 0) && (strcmp(tmp, list[i]) < 0)) {
127             list[i + 1] = list[i];  // fill the previous word with current word
128             i--;                    // move on to the next word
129         }
130         list[i + 1] = tmp;          // fill the word list[j] at index i + 1
131     }
132 }
133
134 void BubbleSort(char **list, int n)             // in-place bubble sort
135 {
136     int i, j;                                   // index
137
138     for (i = 0; i < n - 1; i++) {
139         for (j = n - 1; j > i; j--) {           // list[0 : i - 1] is sorted
140             if (strcmp(list[j], list[j - 1]) < 0) { // if right word is smaller
141                 swap(&list[j], &list[j - 1]);   // swap the right and left word
142             }
143         }
144     }
145 }
146
```

```
147 void ShakerSort(char **list, int n) // in-place shaker sort
148 {
149     int j;                          // index
150     int l = 0;                      // left bound
151     int r = n - 1;                  // right bound
152 
153     while (l <= r) {                // while there are elements between bounds
154         for (j = r; j > l; j--) {   // move the smallest word to the left
155             if (strcmp(list[j], list[j - 1]) < 0) { // if right word is smaller
156                 swap(&list[j], &list[j - 1]);   // swap the right and left word
157             }
158         }
159         l++;                        // close the bound on the left
160         for (j = l; j < r; j++) {   // move the biggest word to the right
161             if (strcmp(list[j], list[j + 1]) > 0) { // if left word is bigger
162                 swap(&list[j], &list[j + 1]);   // swap the right and left word
163             }
164         }
165         r--;                        // close the bound on the right
166     }
167 }
168 
169 void swap(char **a, char **b)   // swap two words
170 {
171     char *tmp;                  // temporary char pointer
172 
173     tmp = *a;                   // copy word at address a
174     *a = *b;                    // change word at a to word at b
175     *b = tmp;                   // change word at b to original word at a
176 }
177 
178 void freeMemory(char **list, int n) // free allocated memory
179 {
180     int i;                          // index
181 
182     for (i = 0; i < n; i++) {
183         free(list[i]);              // free the memory that stores the words
184     }
185     free(list);                     // free the memory that stores the pointers
186 }
```

[Format] can be improved.
[Introduction] of the problem can be more clear.
[CPU time] measurement method should be described clearly.
[Time] complexity should be analyzed using table or counting method.
[Space] complexity O(1)?
[Report] uses 12 pt fonts, single column format and double line-space.

Score: 84