Algorithm HW1

106061225 楊宇羲

Problem description:

Should explain this.

When it comes to sorting an array, there are several ways to approach the required outcome. However, there indeed exists a variety of performance among each algorithm, so today we are going to analyze 4 kinds of sorting algorithm, which are selection sort, insertion sort, bubble sort and odd even sort (shaker sort), respectively.

Approach:

There are 9 lists of strings for us to sort. With the input size ranging from 10 to 2560, we can examine whether the size of array will affect each algorithm's performance.

No screen dump, please.

```
double GetTime(void);                                   // get local time in seconds
void CopyArray(char **list, char** a, int n);           // copy array
void SelectionSort(char **list,int n);                  // in-place selection sort
void InsertionSort(char **list,int n);                  // in-place insertion sort
void BubbleSort(char **list,int n);                     // in-place bubble sort
void OddEvenSort(char **list,int n);                    // in-placr shaker sort
```

The above are all of the functions I used in this analysis. GetTime() is used to measure CPU time, we will call GetTime() function first to reset the timer before the array is being initialized to the original list, then each sorting method is carried out. After the sorting is finished we call the GetTime() function again to obtain finished time. The overall operation will be like this:

Can use pseudo codes to describe these operations.

1. Get the input array
2. Call GetTime(), start counting CPU time

(3 & 4 repeat 500 times)

3. Let the sorted array recover to the original one
4. Apply each sorting method.
5. Call GetTime() again, stop counting CPU time.

Analysis of each sorting algorithm:

1.Selection sort:

Algorithms need no double line-space.

```
algorithm SelectionSort (A, n){
    for i = 0 to n-2 do{
        min := i;
        for j = i+1 to n-1 do{
            if A[j] < A[min]
                min := j;
            }
            swap A[i] and A[min];
        }
    }
}
```

In this algorithm, the core concept is to find the smallest value in the unsorted part. We look up from the front, and whenever we found a minimum value at the back we put it to the last position of the sorted part, then look up the remaining unsorted part.

Hence, the total comparing time will be (n-1) + (n-2) + (n-3) + ⋯ + 1 = $\frac{n(n-1)}{2}$.

So, the time complexity in best, average and worst case all be O(n^2).

2. Insertion sort:

```
algorithm InsertionSort (A, n){
    for (i=1; i<n; i++){
        int item;
        int j;
        while (j>0 && A[j-1]>item) do{
            A[j] := A[j-1];
            j--;
        }
```

```
            a[j] := item;
        }
}
```

In insertion sorting, we will also create a sublist, however, this time we don't seek for the minimum value in the unsorted part, we choose the first index in the unsorted part and put it in the appropriate position in the sublist until the whole list is sorted.

In the best case, we only have to compare one time in each step, so the total step will be n times, which is O(n). However, in the worst case and average case, the total comparison is also $1 + 2 + 3 + \cdots + n = \frac{n(n-1)}{2}$.

3. Bubble sort:

```
algorithm BubbleSort(A,n){
    for (i=1 to n-2 step 1){
        for ( j=1 to n-2-i step 1){
            if (A[j]>A[j+1]){
                swap(A[j],A[j+1]);
            }
        }
    }
}
```

This is incorrect.

In this algorithm, differ from the previous algorithms, we look for the biggest value and put it to the end of the list, until all the elements are placed in the right position.

The worst case occurs if the list is in decreasing order, then we will have to swap $(n-1) + (n-2) + (n-3) + \cdots + 1 = \frac{n(n-1)}{2}$ time. On the other hand, we will perform 0

swap if the list is in increasing order. So the best case will be O(n) while the worst case is O(n^2) and average case also O(n^2).

4. Odd Even sort:

This is not correct.

Same as bubble sort, but we separate the list into even part and odd part. In every iteration, we will perform odd comparison and even comparison each for 1 time.

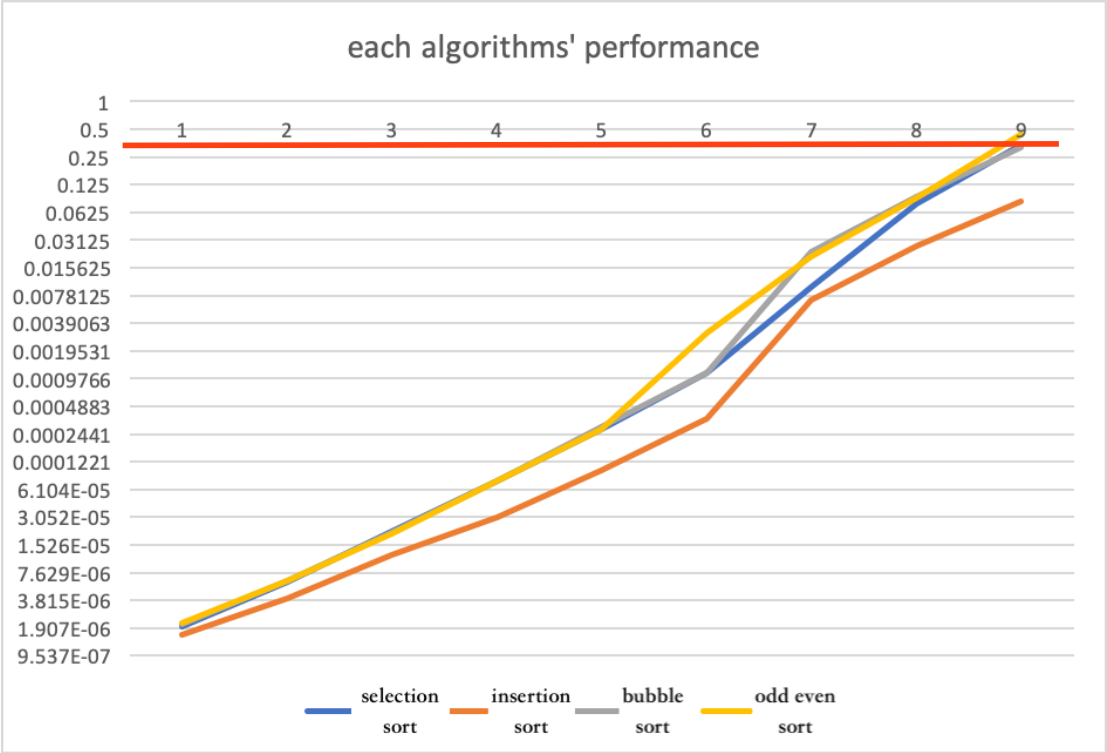
Its time complexity is same as bubble sort.

Result:

	selection sort	insertion sort	bubble sort	odd even sort
s1.dat	1.992226e-06	1.637936e-06	2.086163e-06	2.144337e-06
s2.dat	6.046295e-06	3.990173e-06	6.266117e-06	6.275654e-06
s3.dat	2.172804e-05	1.172781e-05	2.077389e-05	1.990604e-05
s4.dat	7.607794e-05	3.038406e-05	7.678175e-05	7.484818e-05
s5.dat	2.773137e-04	9.761572e-05	2.926617e-04	2.785401e-04
s6.dat	1.126614e-03	3.627863e-04	1.144978e-03	3.029098e-03
s7.dat	9.703258e-03	6.954372e-03	2.301722e-02	2.093242e-02
s8.dat	7.860320e-02	2.699897e-02	9.301401e-02	8.999784e-02
s9.dat	3.342732e-01	8.335954e-02	3.210775e-01	3.445004e-01

Unit: second

x- and y-label are needed.



y-axis unit: second (in log scale)

Observation:

Since this plot is in log scale so we can see meager difference between each algorithms' performance, however, insertion sort has excelling performances among other ones by about 50%, which is significant. I surmise that the difference is caused by the original order of the array instead of the size of array since every algorithm has an average case of O(n^2). In average case, although the 4 kinds of sorting methods have the same time complexity, but take bubble sort and insertion sort for instance, bubble sort will require more swapping than insertion sort, and this might be the main difference between them.

Conclusion:

In theory, insertion sorting will be less effective while the size of array grows bigger, however this phenomenon doesn't appear in this case. However, the overall running time tends to be correct since the size of array also grows exponentially. There surely exist some sorting methods with time complexity of O(NlogN) which is worth give it a try to find the difference in CPU timing.

```
$ gcc hw01.c
$ a.out < s1.dat
Selection Sort: N = 10 CPU = 8.621216e-07 seconds
Insertion Sort: N = 10 CPU = 8.358955e-07 seconds
Bubble Sort: N = 10 CPU = 8.420944e-07 seconds
OddEven Sort: N = 10 CPU = 8.959770e-07 seconds
1 anemometry
2 cates
3 cincture
4 homebuilder
5 preestablish
6 roccellaceae
7 seedbed
8 speedboat
9 synclinal
10 unamusing
```

Program has coding errors!

---

score: 30.0

- Report format
  - Report title should take 3 lines (Course title, HW title, ID and name)
  - Do not include screen dump in your report.
  - Algorithms and tables need no double line space.
- Introduction
  - Need to describe the problem you are solving
- Approach
  - Need to describe each algorithm clear (can use pseudo codes)
  - Can describe main function using pseudo codes
- Time/Space complexity
  - Time complexity can be derived using table approach.
  - Space complexity should be analyzed
- Results

  - Figures can be improved.

- Conclusion/observation
  - Can correlate your CPU times to the algorithm complexities

- Program format can be improved

hw01.c

```c
// EE3980 HW01 Quardratic Sorts
// 106061225, 楊宇羲
// 2020/3/10
   2020?

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/time.h>

double GetTime(void);                                   // get local time in seconds
void CopyArray(char **list, char** a, int n);           // copy array
void SelectionSort(char **list,int n);                  // in-place selection sort
  void SelectionSort(char **list, int n);                // in-place selection sort
void InsertionSort(char **list,int n);                  // in-place insertion sort
  void InsertionSort(char **list, int n);                // in-place insertion sort
void BubbleSort(char **list,int n);                     // in-place bubble sort
  void BubbleSort(char **list, int n);                   // in-place bubble sort
void OddEvenSort(char **list,int n);                    // in-placr shaker sort
  void OddEvenSort(char **list, int n);                  // in-placr shaker sort

int main(){
  int main(void)
  {

    int R = 500;                                        // time of repetitions
    int n;                                              // size of the array
    int i, j;                                           // for loop counting
    double t, t0, t1;                                   // for CPU time counting
    char **a = (char**)malloc(n*sizeof(char*));         // the original array
  n is not defined yet!
    char **list = (char**)malloc(n*sizeof(char*));      // the array being sorted
  n is not defined yet!
    char buffer[512];                                   // buffer for storing

    scanf("%d", &n);                                    // size of array

    for(i=0; i<n; i++){                                 // storing array to **a
    for (i = 0; i < n; i++) {                               // storing array to **a
```

```
30          scanf("%s", buffer);
31          a[i]=malloc(strlen(buffer+1));
            a[i] = malloc(strlen(buffer + 1));
            string length is not correct!
32          strcpy(a[i],buffer);
            strcpy(a[i], buffer);
33      }
34
35      // selection sort
36
37      t0 = GetTime();                                 // start counting time
38      for(i=0; i<R; i++){                             // repeat R times
        for (i = 0; i < R; i++) {                           // repeat R times
39          CopyArray(list,a,n);                        // call copy function
            CopyArray(list, a, n);                        // call copy function
40          SelectionSort(list,n);                      // start selection sort
            SelectionSort(list, n);                       // start selection sort
41      }
42      t1 = GetTime();                                 // stop counting time
43
44      t  = (t1-t0)/R;                                 // calculate CPU time
        t  = (t1 - t0) / R;                               // calculate CPU time
45      printf("Selection Sort: N = %d CPU = %e seconds\n", n, t );
        printf("Selection Sort: N = %d CPU = %e seconds\n", n, t);
46
47      // insertion sort
48
49      t0 = GetTime();                                 // start counting time
50      for(i=0; i<R; i++){                             // repeat R times
        for (i = 0; i < R; i++) {                           // repeat R times
51          CopyArray(list,a,n);                        // call copy function
            CopyArray(list, a, n);                        // call copy function
52          InsertionSort(list,n);                      // start insertion sort
            InsertionSort(list, n);                       // start insertion sort
53      }
54      t1 = GetTime();                                 // stop counting time
55
56      t  = (t1-t0)/R;                                 // calculate CPU time
        t  = (t1 - t0) / R;                               // calculate CPU time
57      printf("Insertion Sort: N = %d CPU = %e seconds\n", n, t );
        printf("Insertion Sort: N = %d CPU = %e seconds\n", n, t);
```

```
58
59      // bubble sort
60
61      t0 = GetTime();                                 // start counting time
62      for(i=0; i<R; i++){                             // repeat R times
        for (i = 0; i < R; i++) {                             // repeat R times
63          CopyArray(list,a,n);                        // call copy function
            CopyArray(list, a, n);                        // call copy function
64          BubbleSort(list,n);                         // start bubble sort
            BubbleSort(list, n);                         // start bubble sort
65      }
66      t1 = GetTime();                                 // stop counting time
67
68      t  = (t1-t0)/R;                                 // calculate CPU time
        t  = (t1 - t0) / R;                                 // calculate CPU time
69      printf("Bubble Sort: N = %d CPU = %e seconds\n", n, t );
        printf("Bubble Sort: N = %d CPU = %e seconds\n", n, t);
70
71      // odd even sort
72
73      t0 = GetTime();                                 // start counting time
74      for(i=0; i<R; i++){                             // repeat R times
        for (i = 0; i < R; i++) {                             // repeat R times
75          CopyArray(list,a,n);                        // call copy function
            CopyArray(list, a, n);                        // call copy function
76          OddEvenSort(list,n);                        // start odd even sort
            OddEvenSort(list, n);                         // start odd even sort
77      }
78      t1 = GetTime();                                 // stop counting time
79
80      t  = (t1-t0)/R;                                 // calculate CPU time
        t  = (t1 - t0) / R;                                 // calculate CPU time
81      printf("OddEven Sort: N = %d CPU = %e seconds\n", n, t );
        printf("OddEven Sort: N = %d CPU = %e seconds\n", n, t);
82
83      // print array
84
85      for(i=0; i<n; i++){                             // Display sorted array
        for (i = 0; i < n; i++) {                             // Display sorted array
86          printf("%d %s\n", i+1, list[i]);
            printf("%d %s\n", i + 1, list[i]);
```

```
87      }
88      return 0;
89  }
90
91  void SelectionSort(char **list,int n){
    void SelectionSort(char **list, int n)
    {
92
93      int i, j;                                   // for loop counting
94      char *temp;                                 // for swapping
95
96      for(i = 0; i < n; i++){                     // list[i] is standard
        for (i = 0; i < n; i++) {                     // list[i] is standard
97          for(j = i+1; j < n; j++){               // looking for smaller element
            for (j = i + 1; j < n; j++) {               // looking for smaller element
98              if(strcmp(list[i], list[j])>0){ // swap if latter index is smaller
                if (strcmp(list[i], list[j]) > 0) { // swap if latter index is smaller
99                  temp = list[i];
100                 list[i] = list[j];
101                 list[j] = temp;
102             }
103         }
104     }
105 }
106
107 void InsertionSort(char **list,int n){
    void InsertionSort(char **list, int n)
    {
108
109     int i;                                      // for loop counting
110     char *temp;                                 // for swapping
111
112     for(i = 1; i < n; i++){
        for (i = 1; i < n; i++) {
113         temp = list[i];                         // move to proper place
114         int j = i-1;
    Do not mix declarations with statements
115         while (strcmp(temp,list[j])<0 && j>0){  // find proper i
            while (strcmp(temp, list[j]) < 0 && j > 0) {  // find proper i
116             list[j+1] = list[j];                // move to upper place
                list[j + 1] = list[j];                // move to upper place
```

```
117             j--;
118         }
119         list[j+1] = temp;                   // replace temp
            list[j + 1] = temp;                   // replace temp
120     }
121 }
122
123 void BubbleSort(char **list,int n){
    void BubbleSort(char **list, int n)
    {
124
125     int i, j;
126     char *temp;
127
128     for(i = 0; i < n-1; i++){
        for (i = 0; i < n - 1; i++) {
129         for(j = 0; j < n-1-i; j++){             // sort from the last element
            for (j = 0; j < n - 1 - i; j++) {             // sort from the last element
130             if(strcmp(list[j], list[j+1])>0){   // compare to the right
                if (strcmp(list[j], list[j + 1]) > 0) {   // compare to the right
131                 temp = list[j];
132                 list[j] = list[j+1];
                    list[j] = list[j + 1];
133                 list[j+1] = temp;
                    list[j + 1] = temp;
134             }
135         }
136     }
137 }
138
139 void OddEvenSort(char** list,int n){
    void OddEvenSort(char** list, int n)
    {
140
141     int sorted=0;
        int sorted = 0;
142     int i;
143     char *temp;
144
145     while (!sorted){
        while (!sorted) {
```

```
146         sorted=1;
            sorted = 1;
147         for(i = 1; i <= n-2; i=i+2){            // sort even element
            for (i = 1; i <= n - 2; i = i + 2) {            // sort even element
148             if(strcmp(list[i], list[i+1])>0){   // apply bubble sort
                if (strcmp(list[i], list[i + 1]) > 0) {   // apply bubble sort
149                 temp = list[i];
150                 list[i] = list[i+1];
                    list[i] = list[i + 1];
151                 list[i+1] = temp;
                    list[i + 1] = temp;
152                 sorted = 0;
153             }
154         }
155         for(i = 0; i <= n-2; i=i+2){            // sort odd element
            for (i = 0; i <= n - 2; i = i + 2) {            // sort odd element
156             if(strcmp(list[i], list[i+1])>0){   // apply bubble sort
                if (strcmp(list[i], list[i + 1]) > 0) {   // apply bubble sort
157                 temp = list[i];
158                 list[i] = list[i+1];
                    list[i] = list[i + 1];
159                 list[i+1] = temp;
                    list[i + 1] = temp;
160                 sorted = 0;
161             }
162         }
163     }
164 }
165
166 void CopyArray(char** list, char** a, int n){
    void CopyArray(char** list, char** a, int n)
    {
167
168     int j;
169     for(j = 0; j < n; j++){
        for (j = 0; j < n; j++) {
170         list[j] = malloc(strlen(a[j]+1));   // dynamic storage
            list[j] = malloc(strlen(a[j] + 1));   // dynamic storage
171         strcpy(list[j], a[j]);              // use strcpy to copy
172     }
173 }
```

```
174
175 double GetTime(){
    double GetTime(void)
    {
176
177     struct timeval tv;
178
179     gettimeofday(&tv, NULL);
180     return tv.tv_sec + 1e-6 * tv.tv_usec;   // sec + micro sec
181 }
182
```