Algorithm HW1

106061225 楊宇羲

Problem description:

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.

```
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:

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:

```
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]);
            }
        }
    }
}
```

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).

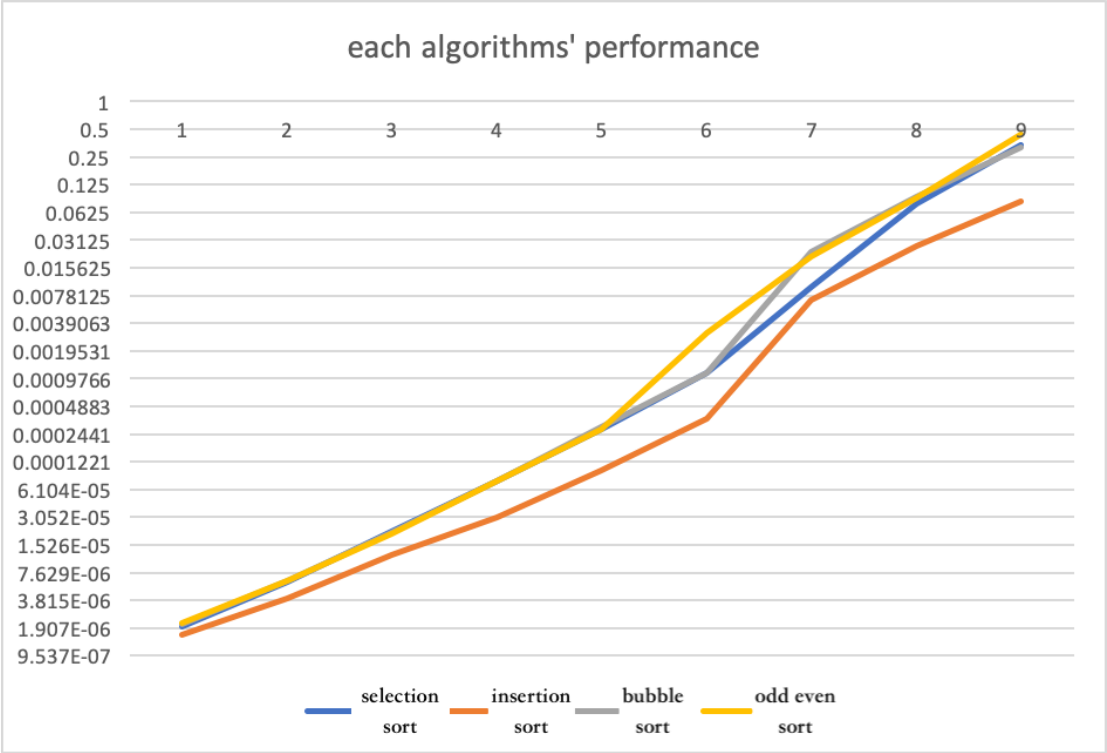
4. Odd Even sort:

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



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.