EE3980 Algorithms

Homework 2. Heap Sort report

By 105061212 王家駿

1. Introduction

In this homework, we use the structure of homework 1 which includes four quadratic sorts, and add the **heap sort** function to the program. Then we compare the efficiency of heap sort with other quadratic sorts, with best-case, worst-case, and average-case analysis included. At the end, we also derive the time and space complexities of the sorting algorithm, and analyze the differences between them.

2. Implementation

The program structure is based on homework 1, and modified at several places:

- Use variable size strings by allocating memories based on the string length.
- Use pointers to character while swapping.
- Add functions to implement heap sort.
- Modified the sorting algorithms from 0-based to 1-based. ?

Since the implementation of the four quadratic algorithms were described in homework 1, we just derive the time/space complexities of them in best-cases, worst-cases, and average cases.

2.1. Selection Sort

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

The iteration at line 3 and 5 must go through to the end no matter how the elements of the array was arranged at the beginning. Thus, for the best-case, worst-case, and average-case, the program has to run through the outer loop and the inner loop, so the time complexities are all $O(n^2)$ for the three cases mentioned above.

For the space complexity, expect the array being sorted, we only need a temporary t to during the sorting. So the space complexity must be $O(n)$ (the array length).

Best-case time complexity: $O(n^2)$

Worst-case time complexity: $O(n^2)$

Average-case time complexity: $O(n^2)$

Space complexity: $O(n)$

2.2. Insertion Sort

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

The iteration at line 3 must go through to the end ,while the one at line 6 would stop at finding out an i such that $A[i] \leqq A[j]$. At the best-case, the iteration at line 6 stops at $i = j-1$, which indicates that the loop ends immediately. Thus, the time complexity of the best-case is $O(n)$. In the last two cases, the iteration at line 6 might stops at somewhere in the average-case and at the end of the iteration ($i = 1$). Thus, the time complexities of the worst-case and the average-case are both $O(n)$

For the space complexity, expect the array being sorted, we only need a temporary item to during the sorting. So the space complexity must be $O(n)$ (the array length).

Best-case time complexity: $O(n)$

Worst-case time complexity: $O(n^2)$

Average-case time complexity: $O(n^2)$

Space complexity: $O(n)$

2.3. Bubble Sort

```
1.  Algorithm BubbleSort(A, n)
2.  {
3.      for i:=1 to n-1 do{                     // find the smallest for A[i]
4.          for j:=n to i+1 step -1 do{
5.              if(A[j] < A[j-1]){              // swap A[j] and A[j-1]
6.                  t:=A[j];
7.                  A[j]:=A[j+1];
8.                  A[j+1]:=t;
9.              }
10.         }
11.     }
12. }
```

The iterations of the loops at line 3 and 4 go to the end, no matter how the elements of the array was arranged at the beginning. Thus, for the best-case, worst-case, and average-case, the program has to run through the outer loop and the inner loop, so the time complexities are all $O(n^2)$ for the three cases mentioned above.

The algorithm has to run through both the inner and the outer loop even when the array is well-sorted in the beginning, whose time complexity is still $O(n^2)$. To minimize the executing time of the best-case, we can add an if-else statement to check if the array is well-arranged after each outer loop iteration. For the best-case, the array is well-sorted in the beginning, and after the check the loop goes to the end soon. Since the outer loop only executes for one time, the time complexity can be reduced to $O(n)$. Is this correct?

For the space complexity, expect the array being sorted, we only need a temporary t to during the sorting (in-place sorting). So the space complexity must be $O(n)$ (the array length).

Best-case time complexity: $O(n^2)$ / $O(n)$ with additional statement

Worst-case time complexity: $O(n^2)$

Average-case time complexity: $O(n^2)$

Space complexity: $O(n)$

2.4. Shaker Sort

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

For the outer loop, the iteration goes to the end when $l > r$. Since l and r have step 1 and -1 separately at each iteration step, the loop ends when l and r meet at the middle of the array, which contributes the time complexity of $O(n)$. Then for the two inner loops, the iterations also have time complexities of $O(n)$ because they have to go through the array until l+1 or r-1. Thus, the time complexities are all $O(n^2)$ for the best-case, worst-case, and average-case.

For the space complexity, expect the array being sorted, we only need a temporary t to during the sorting (in-place sorting). So the space complexity must be $O(n)$ (the array length).

2.5. Heapify

```
Algorithm Heapify(A, root, n)
{
    t:=root; j:=root*2;
    while j<=n do{
        if(j < n && A[j] < A[j+1]){           // if rchild > lchild
            j:=j+1;                           // j is rchild
        }
        if(t > A[j]){                         // if root > children
            break;                            // done
        }
        else{
```

```
            A[j/2]:=A[j];                   // place child to parent
            j:=j*2;                         // j is child, keep finding
        }
    }
    A[j/2]:=t;                              // place the root node
}

```

What is this?

This function replaces the root node to rearrange the heap to the form of maxheap. We found the place where the root node should be inserted from the top, and keep tracing down the children. When tracing to the node j, if the value of the root node is larger than the children, we place the root node at node j to form a maxheap. Otherwise, we keep finding by assign j to the child node which is the larger until finding a right place for the root node.

The loop stops when j > n, and the j doubles itself after each iteration step. So the loop would execute at most $\log_2 n$ times. Thus, the time complexity of heapify once is $O(\log n)$.

2.6. Heap Sort

```
Algorithm HeapSort(A, n)
{
    for i:=n/2 to 1 step -1 do{
        Heapify(A, i, n);           // heapify all the subtrees
                                       to be a max heap
    }

    for i:=n to 2 step -1 do{       // repeat n-1 times
        t:=A[i];                    // swap the first and the last
```

```
        A[i]:=A[1];
        A[1]:=t;
        Heapify(A, 1, i-1);        // make A[1:i-1] be a max heap
    }
}
```

This algorithm rearranges the array in alphabetical order by using the heap sort. At first, we heapify the subtrees with the nodes which have at least one child to make the heap become a maxheap. At each iteration step, we swap the root node, whose value is the largest, with the last node of the array. Then we heapify the tree A[1:i-1], due to the last i nodes were already well-arranged, so the largest element would be placed at the top of the heap, and could be swapped for the next iteration. After the iteration goes to the end, the whole array is well-sorted.

For the two loops of this algorithm, the iterations go almost through the array. Thus, the time complexities are $O(n * \log n)$,which had timed the complexities of heapify, for the best-case, worst-case, and average-case.

For the space complexity, expect the array being sorted, we only need a temporary t to during the sorting (in-place sorting), and the heapify function is also a in-place function. So the space complexity must be $O(n)$ (the array length).

Best-case time complexity: $O(n * \log n)$

Worst-case time complexity: $O(n * \log n)$

Average-case time complexity: $O(n * \log n)$

Space complexity: $O(n)$

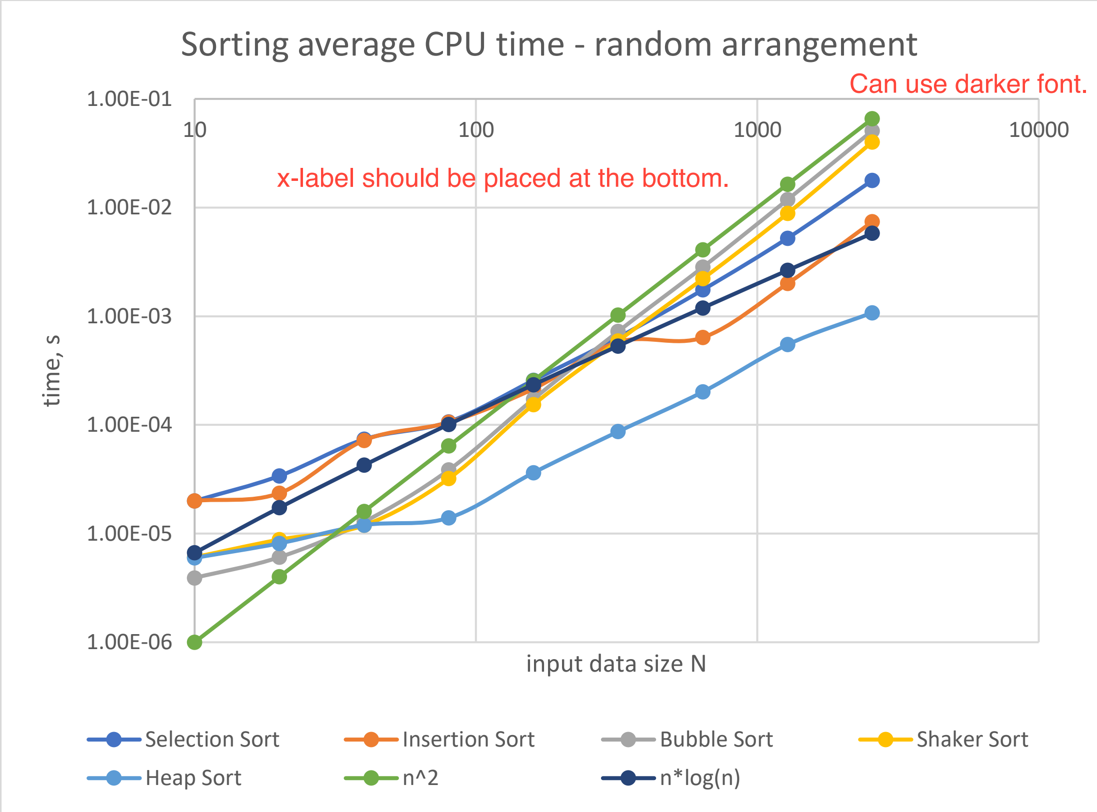
3. Executing results

What is this R?

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.

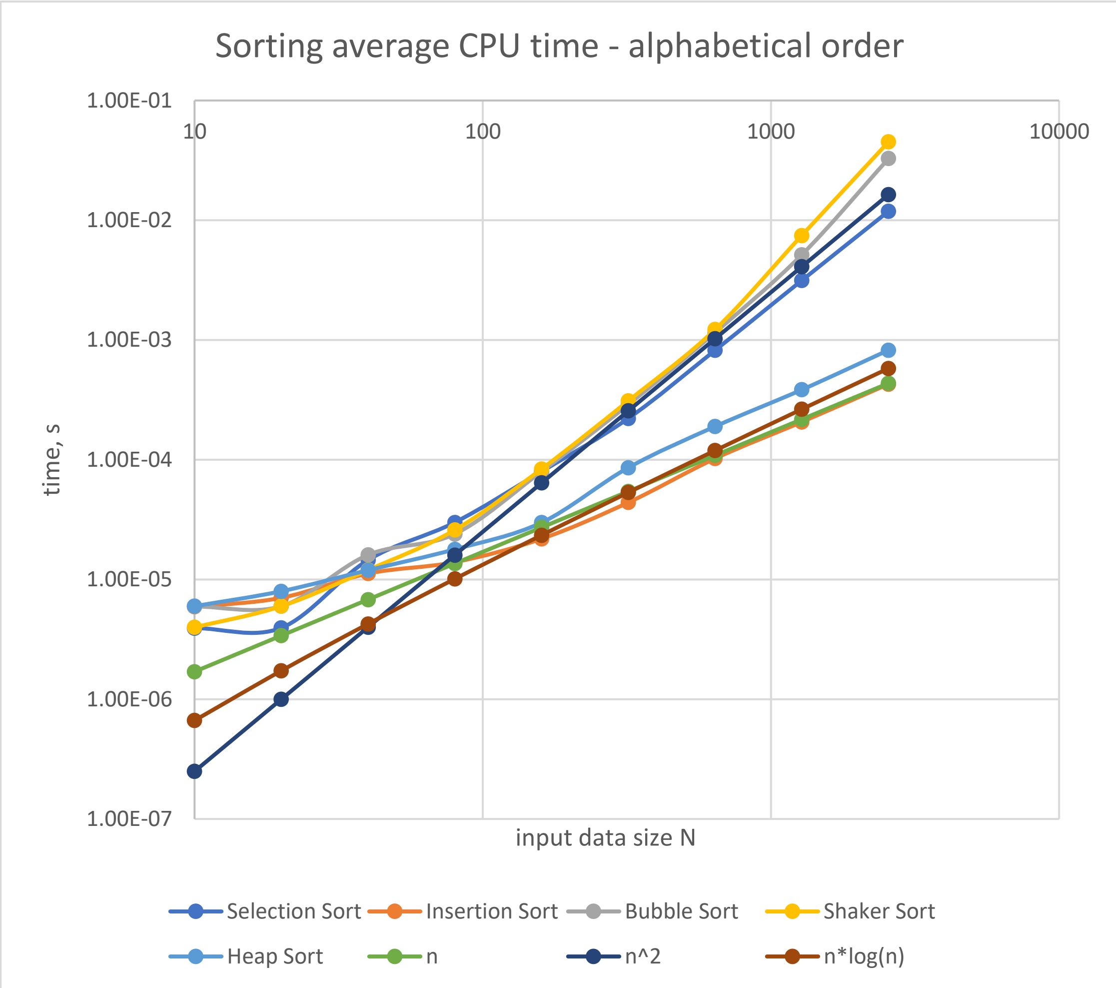
For the random arrangement input:

Data size (*N*)	**Selection Sort**	**Insertion Sort**	**Bubble Sort**	**Shaker Sort**	**Heap Sort**
10	19.95 μs	20.04 μs	3.912 μs	6.060 μs	5.976 μs
20	33.91 μs	23.54 μs	6.062 μs	8.804 μs	8.130 μs
40	73.73 μs	72.08 μs	12.83 μs	11.91 μs	11.97 μs
80	106.7 μs	105.7 μs	38.45 μs	32.04 μs	13.95 μs
160	257.4 μs	214.7 μs	172.8 μs	153.6 μs	36.32 μs
320	637.8 μs	579.2 μs	724.3 μs	592.6 μs	86.77 μs
640	1.751ms	637.6 μs	2.831ms	2.212ms	201.5 μs
1280	5.225ms	1.999ms	11.83ms	8.817ms	548.5 μs
2560	17.76ms	7.398ms	50.77ms	40.13ms	1.078ms



For the alphabetical order input:

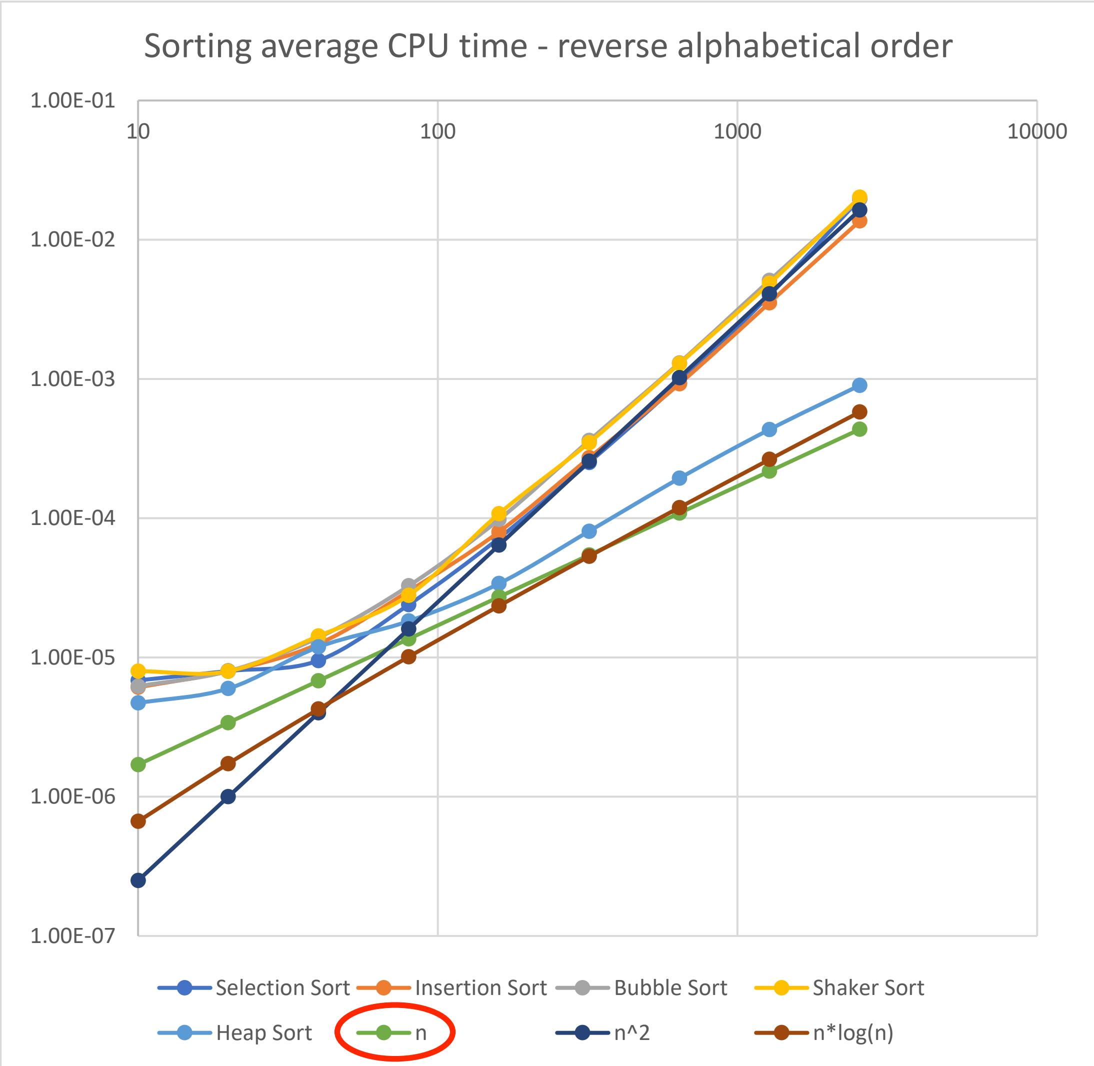
Data size (*N*)	**Selection Sort**	**Insertion Sort**	**Bubble Sort**	**Shaker Sort**	**Heap Sort**
10	3.930 μs	5.908 μs	5.956 μs	3.984 μs	5.986 μs
20	3.948 μs	7.028 μs	5.982 μs	5.984 μs	7.978 μs
40	14.53 μs	11.22 μs	16.04 μs	12.00 μs	11.97 μs
80	29.98 μs	13.96 μs	24.00 μs	25.91 μs	17.87 μs
160	79.79 μs	21.87 μs	79.79 μs	83.78 μs	29.87 μs
320	221.5 μs	43.95 μs	288.4 μs	310.2 μs	85.85 μs
640	820.2 μs	102.4 μs	1.154ms	1.222ms	189.5 μs
1280	3.144ms	206.9 μs	5.150ms	7.465ms	384.2 μs
2560	11.89ms	428.6 μs	32.74ms	45.15ms	823.4 μs



For the reverse alphabetical order input:

Data size (*N*)	**Selection Sort**	**Insertion Sort**	**Bubble Sort**	**Shaker Sort**	**Heap Sort**
10	6.854 μs	6.108 μs	6.220 μs	7.982 μs	4.712 μs
20	7.998 μs	7.980 μs	7.980 μs	7.962 μs	5.982 μs
40	9.510 μs	12.51 μs	13.89 μs	14.27 μs	11.90 μs
80	23.93 μs	29.94 μs	32.69 μs	28.00 μs	18.25 μs
160	71.87 μs	79.79 μs	97.73 μs	107.7 μs	33.91 μs
320	251.6 μs	271.4 μs	361.1 μs	349.1 μs	80.72 μs
640	939.6 μs	922.1 μs	1.305ms	1.287ms	193.5 μs

1280	4.001ms	3.531ms	5.105ms	4.857ms	434.1 μ s
2560	19.66ms	13.64ms	45.79ms	43.37ms	900.5 μ s



Is this needed?

4. Result analysis and conclusion

For the selection sort, we could observe that all of the three input orders lead to the time complexities of $O(n^2)$ from the graph. With a large number of data input, the average CPU runtime doesn't differ too much (from 11.89ms to 19.66ms). Thus, there exists a difference, but not too much, on CPU time consumed between the best-case and the worst-case.

For the insertion sort, we could observe that the time complexities are $O(n^2)$ for the average case and the worst-case, while it can reach the time complexity of $O(n)$ for the best case (the strings were arranged in alphabetical order at the first). Thus, there is a large scale of decreasing at the time consumed in the best-case.

For the bubble sort, we could observe that all of the three input orders lead to the time complexities of $O(n^2)$ from the graph. With a large number of data input, the algorithm would be slightly fast at the best-case (alphabetical order), but it is still slower than almost all the other algorithms.

For the shaker sort, we could observe that all of the three input orders lead to the time complexities of $O(n^2)$ from the graph, and it is also time-consumed as the bubble sort. However, with a large number of data input, the average CPU runtime doesn't differ too much (from 40.13ms to 45.15ms). The reason might be that the shaker sort finds the elements both from the beginning to the end and from the end to the beginning, so the order where the string arranged initially doesn't matter too much for the average CPU time.

For the heap sort, we could observe that all of the three input orders lead to the time complexities of $O(n * \log(n))$ from the graph. We could also find that the average CPU time measured doesn't differ too much.

When the heap sort operates, we place the last node of the unsorted tree to the root, and keep finding down where it should be placed during heapify. It is hard to minimize the time consumed in this step, because the last node we take is usually in the smaller half of the unsorted part according to the structure of max heap. Thus, the iteration almost has to go through the end of the loop to place the root with small value. In the end, it's difficult to figure out the exact arrangement for the best-case and the worst-case of heap sort. That's why the CPU time consumed doesn't differ too much at the three arrangement orders mentioned above.

The measured results are similar as we predicted before. For the average case and the worst-case, we can use the heap sort whose time complexity is $O(n * \log(n))$ to minimize the time consumed. And for the best-case, where the strings were already well-arranged, we can use the insertion sort to have the time complexity of $O(n)$.

Score: 79

o. See return.

[Writing] can elaborate more on heap property and heap sort.

[Writing] should be self-contained and clear.

[Results] plot can be improved.

[Analysis] what is the best-case complexity for bubble sort?

[Coding] 'A[i]' stores pointer to string, and need no malloc

[Coding] Comments need proper indentation as well.

hw02.c

```
/* EE3980 HW02 Heap Sort
 * 105061212, ¤®a @
 * 2019/03/13
 */

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

#define SORTING_ALGORITHM 0
/* modify the value to choose the sorting algorithm
 * 0: Selection Sort
 * 1: Insertion Sort
 * 2: Bubble Sort
 * 3: Shaker Sort
 * 4 or others: Heap Sort
 */

int N;                      // input size
char** data;                    // input data
char** A;                   // array to be sorted
int R = 500;                    // number of repetitions
int sorting_algorithm;              // type of sorting algorithm

void readInput(void);               // read all inputs
void printArray(char** A);          // print the content of array A
void copyArray(char** data, char** A);      // copy data to array A
double GetTime(void);               // get local time in seconds
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 ShakerSort(char** list, int n);        // in-place shaker sort
void HeapSort(char** list, int n);      // in-place heap sort
void Heapify(char** list, int root, int n); // rearrange to form max heap
void freeMemory(void);              // free all dynamic memories

int main(void)
{
    int i;                  // loop index
    double t;                   // for CPU time tracking

    readInput();                // read input data

    sorting_algorithm = SORTING_ALGORITHM;  // type of sorting algorithm

    t = GetTime();              // initialize time counter

```

```
    for(i = 0; i < R; i++){
    for (i = 0; i < R; i++) {
        copyArray(data, A);         // initialize array for sorting

        // execute sorting based on the algorithm chosen
        switch(sorting_algorithm){
            case 0:
                SelectionSort(A, N);
                break;
            case 1:
                InsertionSort(A, N);
                break;
            case 2:
                BubbleSort(A, N);
                break;
            case 3:
                ShakerSort(A, N);
                break;
            default:
                HeapSort(A, N);
        }

        if (i == 0) printArray(A);      // print sorted results
    }

    t = (GetTime() - t) / R;            // calculate CPU time
                        // per iteration
    printf("  CPU time = %e seconds\n",t);  // print out CPU time

    freeMemory();               // free dynamic memories

    return 0;
}

void readInput(void)                // read all inputs
{
    int i;                  // loop index
    char s[50];                 // temporary string for input

    scanf("%d",&N);             // read input size

    // allocate dynamic memories for the two arrays
    data = (char**)malloc(sizeof(char*) * (N+1));
    A = (char**)malloc(sizeof(char*) * (N+1));

    for(i = 1; i <= N; i++){
        scanf("%s",s);              // read input string to s

    //allocate dynamic memories based on the input string size
        data[i] = (char*)malloc(sizeof(char) * (strlen(s) + 1));
```

```
        A[i] = (char*)malloc(sizeof(char) * (strlen(s) + 1));
        A[i] can store pointer to string, and need no malloc

        strcpy(data[i], s);         // put the string s into data
    }
}

void printArray(char** A)           // print the content of array A
{
    int i;                  // loop index

    for(i = 1; i <= N; i++){
        printf("%d %s\n",i ,A[i]);      // print the index and words
                        // after sorted
    }

    // print the type of sorting algorithm
    switch(sorting_algorithm){
        case 0:
            printf("Selection sort:\n");
            break;
        case 1:
            printf("Insertion sort:\n");
            break;
        case 2:
            printf("Bubble sort:\n");
            break;
        case 3:
            printf("Shaker sort:\n");
            break;
        default:
            printf("Heap sort:\n");
    }

    printf("  N = %d\n",N);         // print the input size
}

void copyArray(char** data, char** A)       // copy data to array A
{
    int i;                  // loop index

    for(i = 1; i <= N; i++){
        A[i] = data[i];             // assign the pointer A[i] to
                        // point to the data array
    }
}

double GetTime(void)                // get local time in seconds
{
    struct timeval tv;              // time interval structure
```

```

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

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

void SelectionSort(char** list,int n)       // in-place selection sort
{
    int i, j, k;                  // loop index
    char* tp;                     // temporary pointer for swap

    for(i = 1; i <= n; i++){            // i runs through the array
    for (i = 1; i <= n; i++) {
        j = i;
        for(k = i+1; k <= n; k++){      // search for the smallest
                        // from list[i+1] to list[n]
            if(strcmp(list[k], list[j]) < 0){
                j = k;              // if found, remember it in j
            }
        }
    // swap list[i] and list[j] by using the pointer
        Comments should also be indented properly.
        tp = list[i];
        list[i] = list[j];
        list[j] = tp;
    }
}

void InsertionSort(char** list,int n)       // in-place insertion sort
{
    int i, j;                     // loop index
    char* tp;                     // temporary pointer for swap

    for(j = 2; j <= n; j++){            // j runs through the array
        tp = list[j];               // save content of list[j]
        i = j-1;
    // from list[j-1], find i for list[i] > tp
        Indentation.
        while(i >= 1 && strcmp(tp, list[i]) < 0){
            list[i+1] = list[i];
            i--;
        }
        list[i+1] = tp;             // place tp to the proper place
    }
}

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

```

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

void ShakerSort(char** list,int n)      // in-place shaker sort
{
    int j;                  // loop index
    int l = 1;                  // loop index
    int r = n;                  // loop index
    char* tp;                   // temporary pointer for swap

    while(l <= r){
        for(j = r; j >= l+1; j--){      // word exchange from r to l+1
            if(strcmp(list[j], list[j-1]) < 0){
        // swap list[j] and list[j-1]
                tp = list[j];
                list[j] = list[j-1];
                list[j-1] = tp;
            }
        }
        l++;

        for(j = l; j <= r-1; j++){      // word exchange from l to r-1
            if(strcmp(list[j], list[j+1]) > 0){
        // swap list[j] and list[j+1]
                tp = list[j];
                list[j] = list[j+1];
                list[j+1] = tp;
            }
        }
        r--;
    }
}

void HeapSort(char** list,int n)        // in-place heap sort
{
    int i;                  // loop index
    char* tp;                   // temporary pointer for swap

    // heapify all the subtrees and be a max heap
    for(i = n/2; i >= 1; i--){
```

```
        Heapify(list, i, n);
    }

    for(i = n; i >= 2; i--){            // repeat n-1 times
    //swap the first node and the last node
        tp = list[i];
        list[i] = list[1];
        list[1] = tp;

    // make list[1:i-1] be a max heap
        Heapify(list, 1, i-1);
    }
}

void Heapify(char** list, int root, int n)  // rearrange to form max heap
{
    char* tp = list[root];          // assign root value to tp
    int j;                  // loop index

    for(j = root*2; j <= n; j = j*2){       // j is the lchild of root and
                    // keeps finding its lchild
        if(j < n && strcmp(list[j], list[j+1]) < 0){
            j++;                // j is the rchild
                    // if rchild > lchild
        }
        if(strcmp(tp, list[j]) > 0){
            break;              // done if root > children
        }
        else{
            list[j/2] = list[j];        // place child node to parent
        }
    }

    list[j/2] = tp;             // put root to the proper place
}

void freeMemory(void)               // free all dynamic memories
{
    int i;                  // loop index

    for(i = 0; i <= N; i++){            // free memories of array data
        free(data[i]);
    }
    free(data);

    free(A);        // free memories of array A
}

```