HW01 title?

Algorithm EE398000 hw01

106061146 陳兆廷

1. Introduction:

The analysis and comparisons of the four sorting algorithms: selection sort, insertion sort, bubble sort and shaker sort.

The goal of the algorithms is to sort a list of vocabulary into alphabetical order.

The input of the algorithms is a list of vocabulary, which is in a form of 2d array. The output of the algorithms is a list of sorted, alphabetically ordered vocabulary.

2. Approach:

a. Analysis:

Analysis of **Selection Sort**:

i. **Intro:** Find minimum value, put it in the front of array.

ii. **Proof:** Each iteration in the ith iterations, it brings the smallest value among the n-1 items to the front of the array. Therefore, the algorithm finds the values in ascending order, undoubtly.

iii. Space complexity:

Fixed part: int: (i, j, k, n), *char: (tmp)

Variable part: *char (A) x n

Total: n (*char) + 4

iv. Time complexity:

Statement	s/e	Freq.	Total
Algorithm SelectionSort(**A, n**)	0	0	0
{	0	0	0
for i := 1 **to** n **do** {	n+1	1	n+1
j := i;	1	n	n
for k := i+1 **to** n **do** {	n+1	n	n(n+1)
if (A[k] < A[j]) then j = k	2 or 1	n(n+1)	n(n)*2or1
}	0	0	0
tmp = list[i]; list[i] = list[j]; list[j] = tmp;	3	n	3n
}	0	0	0
}	0	0	0
Total			$3n^2$ +4n+0

*since aveg of i is n/2, I calculate T(P) with i = n/2.

Best case: $2n^2$ +4n+2 (O(n^2)).

Worst Case: $3n^2$ +4n+2 (O(n^2)).

Average Case: O(n^2).

Therefore, the time complexity of Selection sort is O(n^2).

Analysis of **Insertion Sort**:

i. **Intro:** Compare with the minimum value, if true, swap, put it in the front of array.

ii. **Proof:** Like Selection sort, but Insertion sort gradually inserts the items from 2 to n into array in the front, in the right order.

iii. **Space complexity:**

Fixed part: int: (i, j, n), *char: (tmp)

Variable part: *char (A) x n

Total: n (*char) + 3

iv. **Time complexity:**

Statement	s/e	Freq.	Total
Algorithm InsertionSort(**A, n**)	0	0	0
{	0	0	0
for j := 2 **to** n **do** {	n-1	1	n-2
tmp := A[j];	1	n-2	n-2
i := j - 1;	1	n-2	n-2
while ((i >= 1) **and** (tmp < A[i])) **do** {	n (*)	n-2	n^2-2n
A[i + 1] := A[i]; i := i-1;	2	(n-2)(c)n	(2n-4)cn
}	0	(n-2)(c)n	0
A[i + 1] = tmp;	1	n-2	n-2
}	0	0	0
}	0	0	0
Total			(1+2c)n^2

*Assume >=, <, and are all performed. (I recall that if i<1, computer won't bother do the rest computation.)

What is c?

Best case: if c = 0 or 1, it is mostly sorted, it won't do anything. (O(1))

Worst case: if c ~= n, it is almost decreasing, it is ~ (1+2c)n^2.(O(n^2))

Average case: O(n^2) since most of the case require sorting.

Therefore, the time complexity of Selection sort is O(n^2).

Analysis of **Bubble Sort**:

i. **Intro:** Swap whenever it's in the wrong order.

ii. **Proof:** It swaps whenever two neighbors are not in the right order. From ith iteration in 1 to n-1 iterations, it does the aforementioned behavior from n to i+1. This way, no single thing is in the wrong order.

iii. **Space complexity:**

Fixed part: int: (i, j, n), *char: (tmp)

Variable part: *char (A) x n

Total: n (*char) + 3

iv. **Time complexity:**

Statement	s/e	Freq.	Total

Algorithm BubbleSort(A, n)	0	0	0
{	0	0	0
for i := 1 to n - 1 do {	n	1	n
for j := n to i + 1 step -1 do {	n	n-1	n^2-n
if (A[j] < A[j – 1]) {	1	n-1	n-1
tmp = A[j]; A[j] = A[j-1]; A[j-1] = tmp;	3	c(n-1)	3c(n-1)
}	0	0	0
}	0	0	0
}	0	0	0
}	0	0	0
Total			n^2+…

Best case: if c = 0 or 1, the 2nd iteration does nothing, then it is (O(n)).

Worst case: if c ~= n, it is almost decreasing, it is ~ n^2+…(O(n^2))

Average case: O(n^2) since most of the case require sorting.

Therefore, the time complexity of Bubble sort is O(n^2).

Analysis of **Shaker Sort**:

i. **Intro:** Swap whenever it’s in the wrong order but goes from both ends repeatedly.

ii. **Proof:** Like Bubble sort, but it reduces the 1 to n cycle to half of it, and repeat the swapping from n to i+1 again, from i to n-1. Same as Bubble sort, no single thing is in the wrong order.

iii. Space complexity:

Fixed part: int: (j, l, r, n), *char: (tmp)

Variable part: *char (A) x n

Total: n (*char) + 4

iv. Time complexity:

Statement	s/e	Freq.	Total
Algorithm ShakerSort(A, n)	0	0	0
{	0	0	0
l := 1; r := n;	2	1	2
while l <= r do {	n+1	1	n+1
for j := r to l + 1 step -1 do {	n	n	n^2
if (A[j] < A[j – 1]) {	c	n	cn
tmp = A[j]; A[j] = A[j-1]; A[j-1] = tmp;	3	n	3n
}	0	n	0
}	0	n	0
l = l + 1;	1	n	1n
for j := l to r - 1 do {	n	n	n^2
if (A[j] > A[j + 1]) {	c	n	cn
tmp = A[j]; A[j] = A[j+1]; A[j+1] = tmp;	3	n	3n
}	0	n	0
}	0	n	0
r := r – 1;	1	n	1n
}	0	0	0
}	0	0	0
Total			2n^2…

Best case: if c = 0 or 1, 2nd iterations do nothing, then it is (O(n)).

Worst case: if c ~= 1/2n, it is almost decreasing, it is(O(n^2)).

Average case: O(n^2) since most of the case require sorting, but it goes 2 times slower than bubble sort.

Therefore, the time complexity of Shaker sort is O(n^2).

Comparing Table:

	Selection	Insertion	Bubble	Shaker
Space	n (*char) + 4	n (*char) + 3	n (*char) + 3	n (*char) + 4
Time	**O(n^2)**	**O(n^2)**	**O(n^2)**	**O(n^2)**

Prediction: Shaker > Bubble ? Insertion > Selection Prediction of what?

The reason of Insertion > Selection is because the comparing step is lesser in Insertion sort. Shaker sort reduces the swapping steps by comparing from 2 ends.

The speed of the algorithms depends on whether writing data(swapping) or going through more steps is faster. Based on my learning in Computer Architecture, writing data is way slower.

Therefore, **my prediction is: Insertion > Selection > Shaker > Bubble**.

b. Implementation: (hw01.c on NTHUEE workstation, gcc 4.1.2)

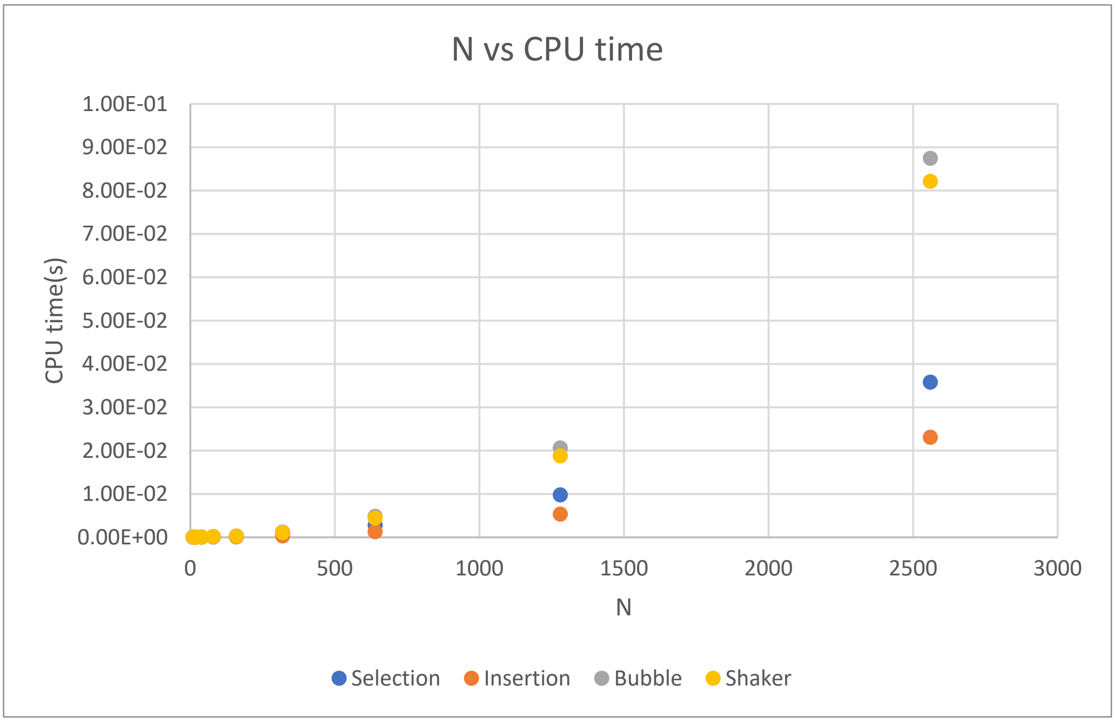
3. Result:

a. Result Table: CPU time? Unit?

N	Selection	Insertion	Bubble	Shaker
10	9.90E-07	7.30E-07	1.11E-06	1.12E-06
20	6.79E-06	5.09E-06	1.00E-05	9.99E-06
40	2.33E-05	1.44E-05	3.96E-05	4.04E-05
80	8.60E-05	5.27E-05	1.64E-04	1.64E-04
160	1.33E-04	7.21E-05	2.77E-04	2.65E-04
320	1.14E-03	3.03E-04	1.13E-03	1.07E-03
640	2.76E-03	1.24E-03	4.81E-03	4.39E-03
1280	9.75E-03	5.32E-03	2.06E-02	1.87E-02
2560	3.58E-02	2.31E-02	8.74E-02	8.21E-02

b. Observation:

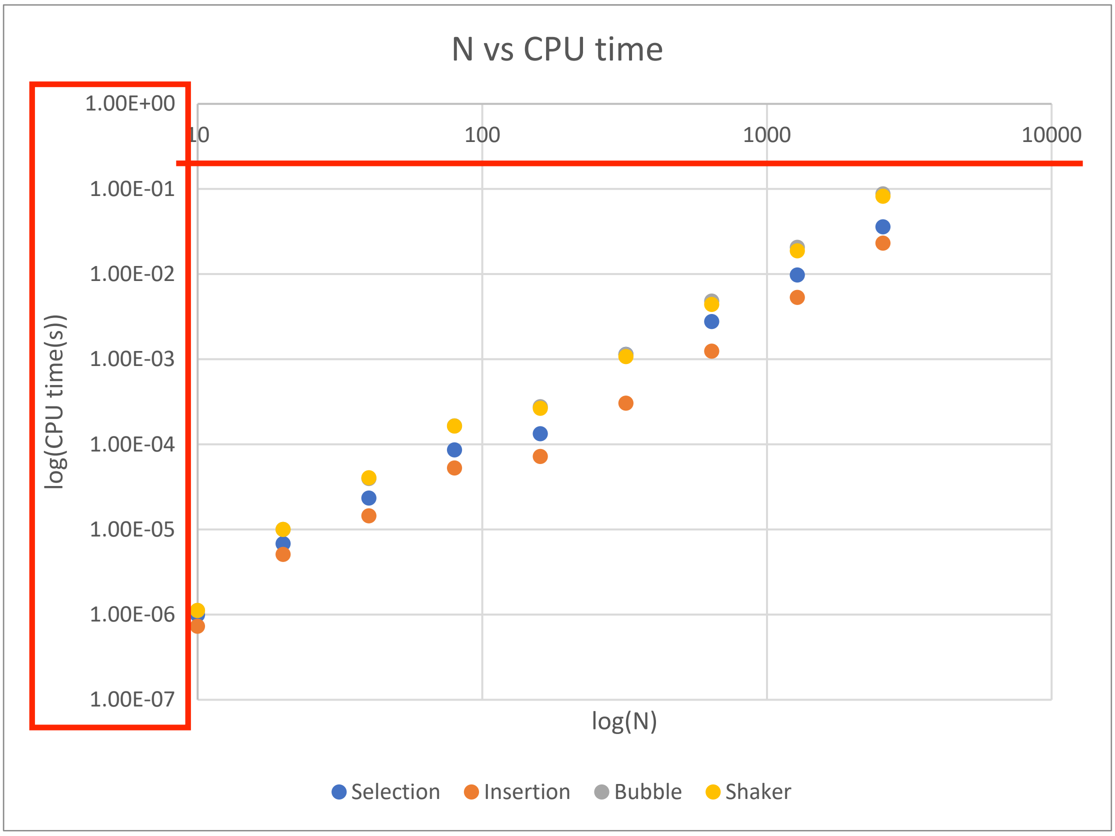
- N vs CPU time chart:



Speed: Insertion > Selection > Shaker > Bubble.

It seems that the result meets by prediction. Furthermore, the time complexities of the four algorithms are O(n^2), same as those that I've calculated.

- Log(N) vs log(CPU time) chart:



How?

When y-axis is logged, the graph looks linear. It indicates that they are O(n^2).

Logged x-axis is just for good spacing between different Ns.

*The N=10 & N=40 case's bubble sort is faster than shaker sort. They might just be the special case when N is small.

c. Conclusion:

   i. Experimented Speed: **Insertion > Selection > Shaker > Bubble**.
   ii. Time Complexity: **Insertion = Selection = Shaker = Bubble = O(n^2)**.
   iii. Best Case: **Insertion > Shaker = Bubble > Selection.**

   Is this correct?

   iv. Space Complexity: **Insertion = Selection = Shaker = Bubble = theta(1)**
   v. **Swapping (data writing) is much slower than comparing.**

hw01.c

```
// EE3980 HW01 Quadratic Sorts
// 106061146, Jhao-Ting, Chen
// 2020/03/11

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

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

void readInput(void);                           // read all inputs
void printArray(char **A);                      // print the content of 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

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

    readInput();                            // read input data

    t = GetTime();                          // initialize time counter
    for (i = 0; i < R; i++) {
        copyArray(data, A);                 // initialize array for sorting
        SelectionSort(A, N);                // execute selection sort
//      InsertionSort(A, N);                // execute insertion sort
        //      InsertionSort(A, N);                // execute insertion sort
//      BubbleSort(A, N);                   // execute bubble sort
        //      BubbleSort(A, N);                   // execute bubble sort
//      ShakerSort(A, N);                   // execute shaker sort
        //      ShakerSort(A, N);                   // execute shaker sort
        if (i == 0) printArray(A);          // print sorted results
    }
    t = (GetTime() - t) / R;                // calculate CPU time / iteration
    printf("Selection sort:\n  N = %d\n  CPU time = %e\n", N, t);// result
    printf("Selection sort:\n  N = %d\n  CPU time = %e\n", N, t);  // result
//  printf("Insertion sort:\n  N = %d\n  CPU time = %e\n", N, t);// result
    //  printf("Insertion sort:\n  N = %d\n  CPU time = %e\n", N, t);// result
//  printf("Bubble sort:\n  N = %d\n  CPU time = %e\n", N, t);// result
```

```
    //  printf("Bubble sort:\n  N = %d\n  CPU time = %e\n", N, t);// result
//  printf("Shaker sort:\n  N = %d\n  CPU time = %e\n", N, t);// result
    //  printf("Shaker sort:\n  N = %d\n  CPU time = %e\n", N, t);// result

    return 0;
}

void readInput(void)                               // read all inputs
{
    scanf("%d", &N);                               // read N for # of words
    int i;                                         // for looping
Do not mix declarations with statements

    data = (char **)calloc(N, sizeof(char *));  // initialize data size (N)
    A = (char **)calloc(N, sizeof(char *));     // initialize A size (N)

    for (i = 0; i < N; i++){
    for (i = 0; i < N; i++) {
        data[i] = (char *)calloc(50, sizeof(char)); // set each row 50 chars
        A[i] = (char *)calloc(50, sizeof(char));    // for data & A
        scanf("%s", data[i]);                    // store each word in data
    }
}

void printArray(char **A)        // print the content of array A
{
    int i;                                  // for looping
    for (i = 0; i < N; i++){
    for (i = 0; i < N; i++) {
        printf("%s\n", A[i]);               // print each string
    }

}

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

    for (i = 0; i < N; i++){
    for (i = 0; i < N; i++) {
        A[i] = data[i];                     // copy from data to A
    }
}

double GetTime(void)                        // demonstration code from 1.1.3
{
    struct timeval tv;

    gettimeofday(&tv, NULL);
    return tv.tv_sec + 1e-6 * tv.tv_usec;
```

```
}

void SelectionSort(char **list, int n) // in-place selection sort
{
    int i, j, k;                        // for looping
    char *tmp = (char *)calloc(50, sizeof(char));   // initialize tmp size
    for (i = 0; i < N; i++){                // for every A[i]
    for (i = 0; i < N; i++) {                // for every A[i]
        j = i;                              // Initialize j to i
        for (k = i + 1; k < N; k++){    // search for smallest in A[i+1 : n]
        for (k = i + 1; k < N; k++) {    // search for smallest in A[i+1 : n]
            if (strcmp(list[k], list[j]) < 0){  // found, store in j
            if (strcmp(list[k], list[j]) < 0) {  // found, store in j
                j = k;
            }
        }
        tmp = list[i];                      // swap A[i] & A[j]
        list[i] = list[j];
        list[j] = tmp;
    }
}

void InsertionSort(char **list, int n) // in-place insertion sort
{
    int i, j;
    char *tmp = (char *)calloc(50, sizeof(char));   // initialize tmp size
    for (j = 1; j < N; j++){                // Assume A[0: j-1] already sorted
    for (j = 1; j < N; j++) {                // Assume A[0: j-1] already sorted
        tmp = list[j];                      // Move A[j] to its proper place
        i = j - 1;                          // Init i to be j-1
        while ((i >= 0) && (strcmp(tmp, list[i]) < 0)){//Find i for A[i]<=A[j]
        while ((i >= 0) && (strcmp(tmp, list[i]) < 0)) { // Find i for A[i]<=A[j
]
            list[i + 1] = list[i];          // Move A[i] up by one position
            i = i - 1;
        }
        A[i + 1] = tmp;                     // Move A[j] to A[i+1]
    }
}

void BubbleSort(char **list, int n) // in-place bubble sort
{
    int i, j;
    char *tmp = (char *)calloc(50, sizeof(char));   // Initialize tmp size
    for (i = 0; i < N - 1; i++){            // Find the smallest item for A[i]
    for (i = 0; i < N - 1; i++) {            // Find the smallest item for A[i]
        for (j = N - 1; j > i; j--){
        for (j = N - 1; j > i; j--) {
            if (strcmp(list[j], list[j-1]) < 0){    // swap A[j] & A[j-1]
            if (strcmp(list[j], list[j - 1]) < 0) {    // swap A[j] & A[j-1]
```

```
                tmp = list[j];
                list[j] = list[j-1];
                list[j] = list[j - 1];
                list[j-1] = tmp;
                list[j - 1] = tmp;
            }
        }
    }
}

void ShakerSort(char **list, int n)  // in-place shaker sort
{
    int l = 0, r = N-1, j;
    char *tmp = (char *)calloc(50, sizeof(char));   // Initialize tmp size
    while (l <= r) {
        for (j = r; j >= l+1; j--){     // Element exchange from r down to l
        for (j = r; j >= l + 1; j--) {     // Element exchange from r down to l
            if (strcmp(list[j], list[j-1]) < 0){    // swap A[j] & A[j-1]
            if (strcmp(list[j], list[j - 1]) < 0) {    // swap A[j] & A[j-1]
                tmp = list[j];
                list[j] = list[j-1];
                list[j] = list[j - 1];
                list[j-1] = tmp;
                list[j - 1] = tmp;
            }
        }
        l++;
        for (j = l; j <= r-1; j++){     // Element exchange from l to r
        for (j = l; j <= r - 1; j++) {     // Element exchange from l to r
            if (strcmp(list[j], list[j+1]) > 0){    // swap A[j] and A[j+1]
            if (strcmp(list[j], list[j + 1]) > 0) {    // swap A[j] and A[j+1]
                tmp = list[j];
                list[j] = list[j+1];
                list[j] = list[j + 1];
                list[j+1] = tmp;
                list[j + 1] = tmp;
            }
        }
        r--;
    }
}
```

[Program Format] can be improved.
[Introduction] of the problem can be more clear.
[CPU time] measurement method should be described clearly.
[Space] complexity O(1)?
[Figures] can be improved.
[Observation] can be strengthened.
[Report] uses 12 pt fonts, single column format and double line-space.
[Writing] needs to be logical and consistent.

Score: 74