EE3980 Algorithms

HW1 Quadratic Sorts

104061212 馮立俞
201//3/11

Introduction

In this homework, we're required to perform various quadratic sorting algorithms (**Insertion Sort**, **Selection Sort** and **Bubble Sort**) on files of different amount of strings. We'll then record the CPU time used by each algorithm, and see how problem size affects program running time for them.

Approach and pre-analysis

We will record the CPU time needed to perform each sorting algorithm for 500 times, then take the average to find out how much time a single algorithm spent.

Insertion Sort

Can use pseudo code.

```
void InsertionSort(char ** list, int n) {
    int i, j;
    char * temp;
    for (j = 1; j < n; j++) {
        temp = list[j];
        i = j - 1;
        while ((i >= 0) && (strcmp(temp, list[i]) < 0)) {
            list[i + 1] = list[i];
            i--;
        }
        list[i + 1] = temp;
    }
}
```

In each round, selection sort compare i-th member with the already-sorted members. Then place it in a proper position such that all members ahead are bigger, and all members behind smaller. After n rounds, every member is bigger than its successors, and smaller than its predecessors. The algorithm is correct.

In the i-th round of the outer loop, the inner loop has to perform 1 ~ i-1 times of comparison, depending on how big the i-th element is. Thus in the best case, this algorithm can achieve 1+1+1+...+1 = n i.e. $\Theta(n)$ complexity. The worst case could be 1 + 2 + ... +(n-1) = $\frac{n(n-1)}{2}$, which has $\Theta(n^2)$ complexity. On the average, we expect the inner loop to do $\frac{i}{2}$ times, and this would also result in $\Theta(n^2)$ complexity.

Selection Sort

```c
void SelectionSort(char * * list, int n) {
    int i, j, k;
    char * temp;
    for (i = 0; i < n; i++) {
        j = i;
        for (k = i + 1; k < n; k++) {
            if (strcmp(list[k], list[j]) < 0) j = k;
        }
        temp = list[i];     //
        list[i] = list[j]; //
        list[j] = temp;    //swapping the remaining smallest(j) with i
    }
}
```

In the i-th round, insertion guarantees that i-th smallest element to be placed at i-th position. After finishing the n-th round, the strings are well sorted.

Assume the outer loop has loop index i, the inner has to perform (n-1)−i times of ‘`if`’ statement, which has time complexity of O(1).Therefore, as i increase from 0, 1 to n-2, the inner loop will execute (n-1) + (n-2) +...+ 1 times. The algorithm will be obviously of $\Theta(n^2)$ complexity since the sum of the series is $\frac{n(n-1)}{2}$

Bubble Sort

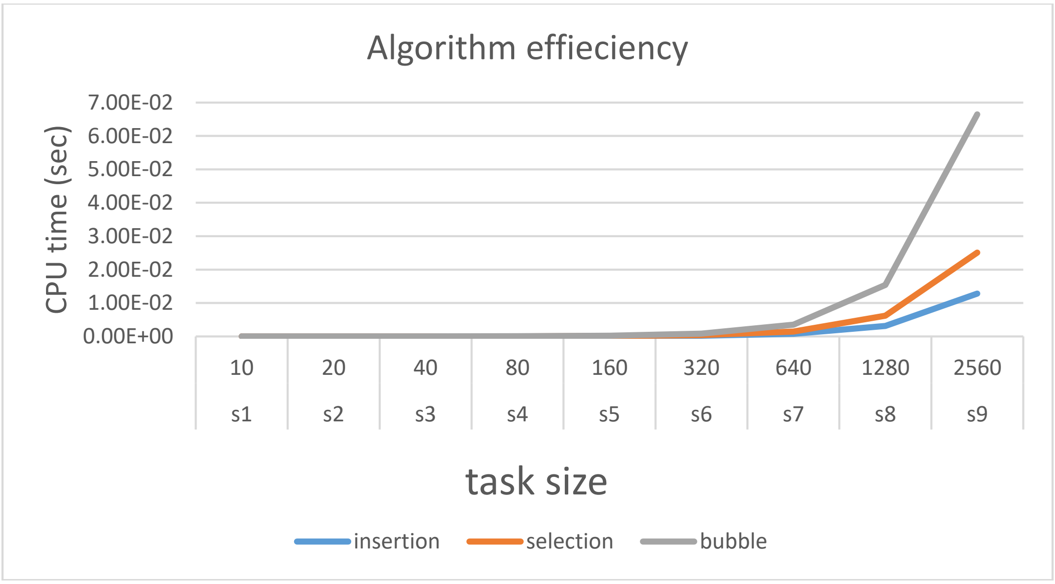
```c
void BubbleSort(char * * list, int n) {
    int i, j;
    char * temp;
    for (i = 0; i < n - 1; i++) {
        for (j = n - 1; j > i; j--) {
            if (strcmp(list[j], list[j - 1]) < 0) {
                temp = list[j]; // swapping
                list[j] = list[j - 1]; //
                list[j - 1] = temp; //
            }
        }
    }
}
```

Similar to selection sort, bubble sort places n-th element in i-th round.

Assume the outer loop has loop index i, the inner loop has to perform $n - 1 - i$ times of operation of O(1) time complexity. Thus the overall amount of operation executed is (n-1) + (n-2) + ... + 1 = $\frac{n(n-1)}{2}$. We can then conclude that bubble sort's time complexity is $\Theta(n^2)$

Results and analysis

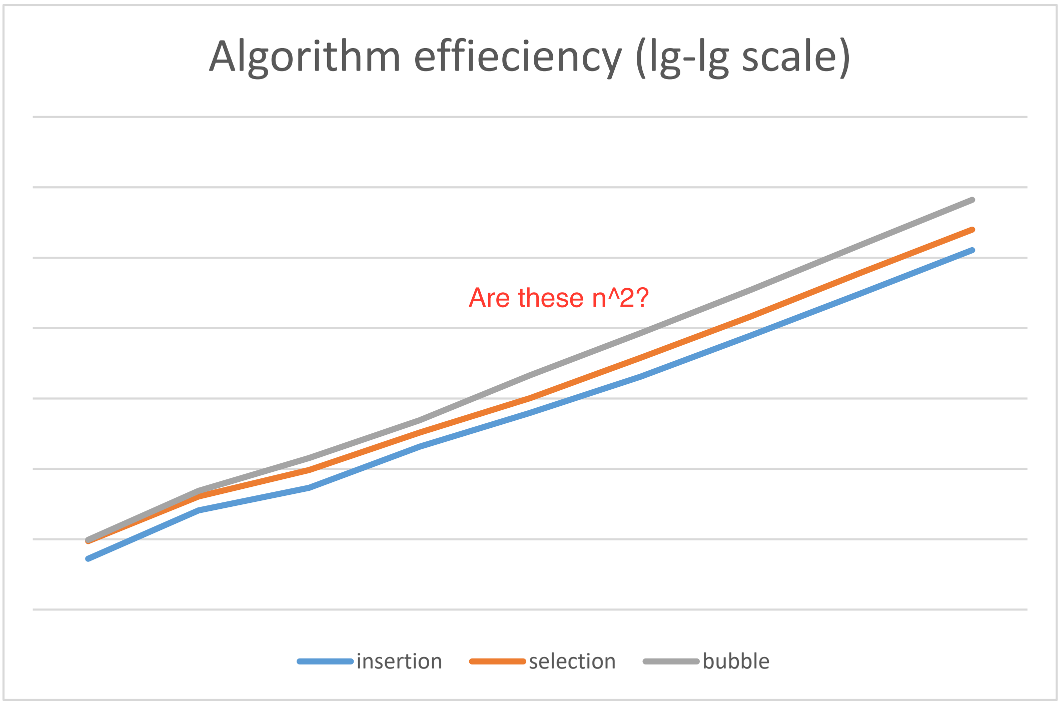
Though all of the three share the same complexity. Insertion sort performed most efficiently in most cases. Selection sort is second-best. Bubble sort behaves poorly in terms of speed. Their actual comparison is shown on the graph below.



If we look further into these three algorithms, above results shouldn't be surprising. Insertion sort, unlike other two, doesn't compare or swap current string with every string in the already-sorted group. On the other hand, Selection sort beats bubble sort since it only performs one swapping in each round.

However, though having different efficiency, if we view the graph in log-log scale, we can clearly see that three of the curves are all linear, which indicates polynomial complexity. ?

x-axis label? y-axis label? unit?



In sum, all of the three are valid sorting algorithms of $\Theta(n^2)$ time complexity. It's also noteworthy that depending on the efficiency of doing swapping / comparing. Sometimes selection sort may outperform insertion sort due to its fewer swapping need.

```c
/*EE3980 HW01 Quadratic Sorts
 *Li-Yu Feng 104061212
 *Date: 2018/3/7
 */


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

void SelectionSort(char **list,int n);
void InsertionSort(char **list,int n);
void BubbleSort(char **list,int n);
double GetTime(void);

void InsertionSort(char **list,int n){
    int i,j;
    char *temp;

    for(j = 1; j < n; j++){
        temp = list[j];
        i = j-1;
        while((i>=0) && (strcmp(temp,list[i]) < 0) ){
            list[i+1] = list[i];
            i--;
        }
        list[i+1] = temp;
    }
}

void BubbleSort(char **list,int n){
    int i,j;
    char *temp;

    for (i = 0; i < n-1; i++){
        for(j = n-1; j > i; j--){
            if( strcmp(list[j],list[j-1]) < 0 ){
               temp = list[j];              // swapping
               list[j] = list[j-1];         //
               list[j-1] = temp;            //
            }
        }
    }
}


void SelectionSort(char **list,int n){
    int i,j,k;
    char *temp;

    for(i = 0; i < n; i++){
        j = i;
        for(k = i+1;k < n; k++){
            if( strcmp(list[k],list[j]) < 0 )
                j = k;
        }
        temp = list[i];         //
        list[i] = list[j];      //
        list[j] = temp;         //swapping the remaining smallest(j) with i
    }
}


double GetTime(void)
{
    struct timeval tv;
    gettimeofday(&tv,NULL);
    return tv.tv_sec+1e-6*tv.tv_usec;
}

int main()
{
    int i,j;
    int Nwords;
    double t1,t2;
    char **words,**A;

    scanf("%d", &Nwords);
    words = (char**)malloc(Nwords * sizeof(char*));     //
    for(i = 0; i < Nwords; i++)                         //
        words[i] = (char *)malloc(sizeof(char*));       //  Maybe too small!
    A = (char**)malloc(Nwords * sizeof(char*));         //
    for(i = 0; i < Nwords; i++)                         //
        A[i] = (char *)malloc(sizeof(char*));           //

    for(i = 0; i < Nwords ; i++){                       //
        scanf("%s", words[i]);                          //scan words
    }

//Perform insertion sort
    t1 = GetTime();
    for(i = 0; i<500; i++){
        memcpy(A,words,Nwords * sizeof(char*));  No need for this
        InsertionSort(A, Nwords);
        if(i==0){
            for(j = 0; j < Nwords ; j++)
                printf("%d %s\n",j+1,A[j]);
        }
    }
    t2 = GetTime();
    printf("%s:\nN=%d\nCPU time = %.5g seconds\n", "insertion sort",Nwords, (t2-t1)/500.0);

//Perform selection sort
```

Only one sort evaluation is needed for one run.

```

    t1 = GetTime();
    for(i = 0; i<500; i++){
        memcpy(A,words,Nwords * sizeof(char*));

        SelectionSort(A, Nwords);
        if(i==0  && Nwords <=20  ){
            for(j = 0; j < Nwords ; j++)
                printf("%d %s\n",j+1,A[j]);
        }
    }
   t2 = GetTime();
   printf("%s:\nN=%d\nCPU time = %.5g seconds\n", "selection sort",Nwords, (t2-t1)/500.0);

//Perform bubble sort
    t1 = GetTime();
    for(i = 0; i<500; i++){
        memcpy(A,words,Nwords * sizeof(char*));

        BubbleSort(A, Nwords);
        if(i==0 && Nwords <= 20){
            for(j = 0; j < Nwords ; j++)
                printf("%d %s\n",j+1,A[j]);
        }
    }
    t2 = GetTime();
    printf("%s:\nN=%d\nCPU time = %.5g seconds\n", "bubble sort",Nwords, (t2-t1)/500.0);
    for(j = 0; j < Nwords ; j++)
                printf("%d %s\n",j+1,words[j]);
```

What is this for?

```

    free(words);    //avoid memory leakage
    free(A);
    return 0;
}
```

Score: 76

o. Can use pseudo to explain you algorithms

o. Can tabulate your CPU times

o. How do you know your CPU times correlate to $O(n^2)$

[Figure 2] should be clearly presented.

o. What are the space complexity?

o. Array $A[i]$ might be too small for long words.

[line 94] *memcpy* is not needed.
- A can be a simple array of $char*$ not $A[1:n][8]$.

o. The assignment requires one algorithm to be executed for each run.