EE3980 Algorithms

HW2 Heap Sort

104061212 馮立俞

2017/3/18

I. Introduction

Continuing HW01, in which we analyzed the efficiency of **Insertion Sort**, **Selection Sort**, and **Bubble Sort**. In this assignment, another sorting algorithm, **Heap Sort** is implemented. Additionally, we'll explore how efficiency change for each algorithm when they encounter best case or worst case situations.

II. Approach & Analysis

Heap Sort

```
1.  Algorithm HeapSort(A, n)  // Sort A[1 : n] into nondecreasing order.
2.  {
3.          for i: = ⌊n / 2⌋ to 1 step- 1 do // Init A[1 : n] to be a max heap.
4.               Heapify(A, i, n);
5.          for i: = n to 2 step- 1 do {              // Move maximum to the end.
6.                  t: = A[i];A[i] = A[1];A[1] = t; // Then make A[1 : i-1] a max
7.                                                  //heap.
8.                  Heapify(A, 1, i- 1);
9.          }
10. }
```

```
1.  Algorithm Heapify(A, i, n) // To maintain max heap property for the tree with
                               // root A[i ]. The size of A is n.
2.  {
3.          j: = 2× i;item: = A[i];done: = false; // A[2 × i ] is the lchild.
4.          while ((j _ n) and(not done))
5.              do { // A[2 × i + 1] is the rchild.
6.                  if ((j < n) and(A[j] < A[j + 1])) then j: = j + 1;
7.                  if (item > A[j]) then done: = true;
8.                        // If larger than children, done.
9.                  else { // Otherwise, continue.
10.                     A[⌊j / 2⌋]: = A[j];
11.                     j: = 2× j;
12.                 }
13.             }
14.         A[⌊j / 2⌋]: = item;
15. }
```

To analyze **Heap Sort**'s efficiency, we need to look at **Heapify**.

The **Heapify** function has $O(\log n)$ in worst case, and $\Omega(1)$ in best case.

The worst case occurs when the root should make its way to the leaf level, i.e. from level 1 to level $\log_2 n$.

The best case, on the contrary, occurs when the root is already the maximum of the given tree. Only initialization and one comparison is needed in such case.

Therefore, in the process of **Heap Sort**, the best case occurs when the to-be-heapified array is already a max heap, for which its roots barely need to move. ($\Theta(n)$)And the worst case occurs when the input is a min-heap. Almost all members would move up along the heap in the first loop of **HeapSort**

Function.($\Theta(n \log n)$)

Recap: **Insertion Sort**

```
1.  void InsertionSort(char ** list, int n) {
2.      int i, j;
3.      char * temp;
4.      for (j = 1; j < n; j++) {
5.          temp = list[j];
6.          i = j - 1;
7.          while ((i >= 0) && (strcmp(temp, list[i]) < 0)) {
8.              list[i + 1] = list[i];
9.              i--;
10.         }
11.         list[i + 1] = temp;
12.     }
13. }
```

We can obviously see that from the while loop that the best case for **Insertion Sort** happens when the initial sequence is (non-decreasingly in this case) ordered.

Selection Sort and **Bubble Sort** performs $\frac{n(n-1)}{2}$ times of operation regardless of the input. There's no best or worst case for them. Though a ordered sequence could save some swapping for **Bubble Sort**.

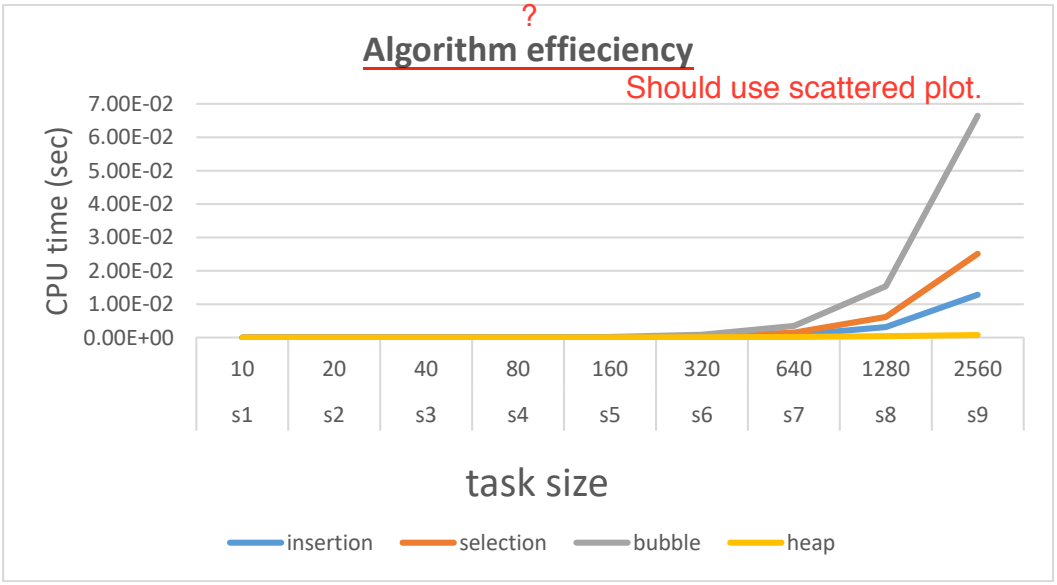
Nonetheless, their space complexity are all $\Theta(n)$.

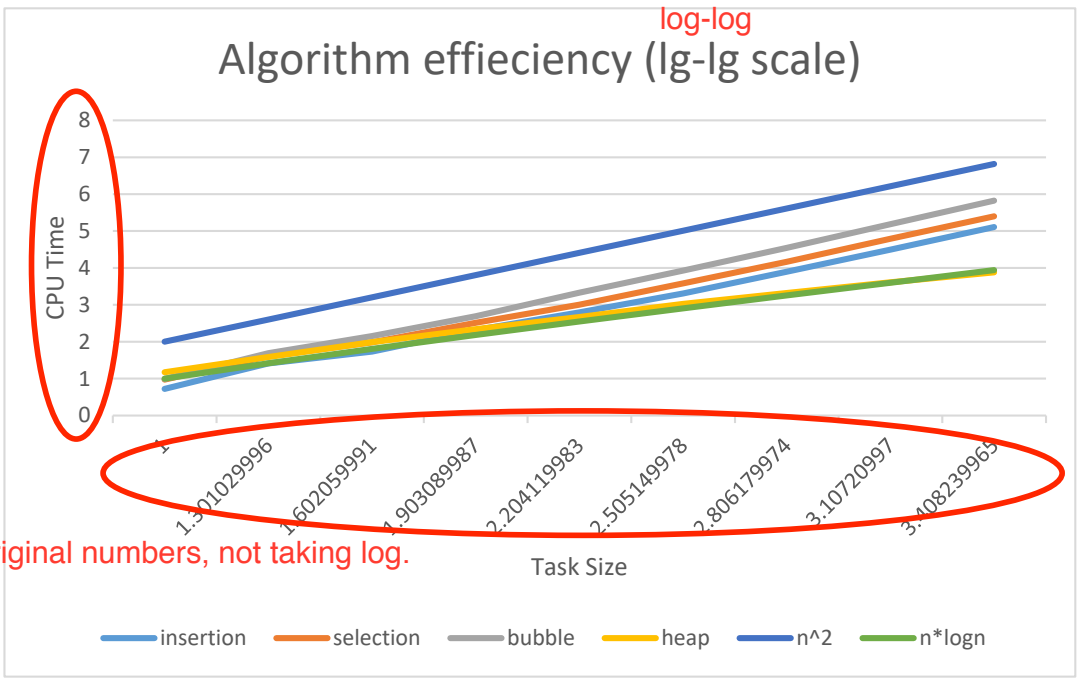
III. Results & Analysis

What is this table?

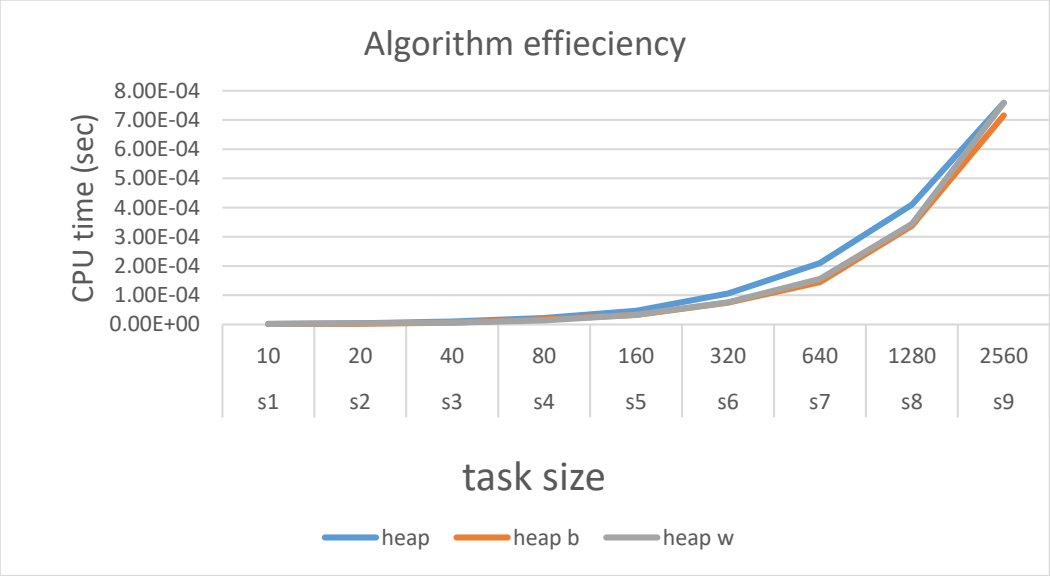
Task Size	insertion	selection	bubble	heap
10	5.28E-07	9.38E-07	9.78E-07	1.48E-06
20	2.57E-06	4.06E-06	4.87E-06	3.84E-06
40	5.37E-06	9.60E-06	1.43E-05	9.70E-06
80	2.07E-05	3.27E-05	4.90E-05	2.18E-05
160	6.23E-05	1.01E-04	2.14E-04	4.67E-05
320	2.04E-04	3.79E-04	8.51E-04	1.06E-04
640	7.90E-04	1.47E-03	3.51E-03	2.10E-04
1280	3.15E-03	6.18E-03	1.54E-02	4.10E-04
2560	1.28E-02	2.51E-02	6.65E-02	7.59E-04

?





Heap Sort, unlike the other three, grows with nlogn.



Best/ Worst case difference wasn't observed clearly though.

```c
/*EE3980 HW02 Heap Sort
 *Li-Yu Feng 104061212
 *Date: 2018/3/18
 */


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

void SelectionSort(char **list,int n);
void InsertionSort(char **list,int n);
void BubbleSort(char **list,int n);
void Heapify(char **list, int i, int n);
void HeapSort(char **list,int n);
void MaxHeapGen(char **list, int n);
void MinHeapGen(char **list, int n);
void MinHeapify(char **list,int i, 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;            //
            }
        }
    }
```

Can add comments to explain each function's purpose.

Comments?

```
51 }
52
53
54 void SelectionSort(char **list,int n){
55     int i,j,k;
56     char *temp;
57
58     for(i = 0; i < n; i++){
59         j = i;
60         for(k = i+1;k < n; k++){
61             if( strcmp(list[k],list[j]) < 0 )
62                 j = k;
63         }
64         temp = list[i];         //
65         list[i] = list[j];      //
66         list[j] = temp;         //swapping the remaining smallest(j) with i
67     }
68 }
69
70 void Heapify(char **list, int i, int n){
71     int j = i*2;
72     char *temp = list[i-1];
73     bool done = false;
74
75     while(j<=n && !done){
76         if(j<n && strcmp(list[j-1],list[j+1-1]) < 0) j++;
77         if(strcmp(temp,list[j-1] ) > 0 ) done = true;
78         else{
79             list[j/2-1] = list[j-1];
80             j *= 2;
81         }
82     }
83     list[j/2-1] = temp;
84 }
85
86 void HeapSort(char **list,int n){
87     char *temp;
88     int i;
89
90     for( i = n/2 ; i>0 ; i--)
91         {Heapify(list,i,n);}
92     for(i = n; i > 1; i-- ){
93         temp = list[i-1];
94         list[i-1] = list[0];
95         list[0] = temp;
96         Heapify(list,1,i-1);
97     }
98 }
99
100 void MaxHeapGen(char **list, int n){
```

```
101     int i;
102     for( i = n/2 ; i>0 ; i--)
103         {Heapify(list,i,n);}
104
105 }
106
107 void MinHeapGen(char **list, int n){
108     int i;
109     for( i = n/2 ; i>0 ; i--)
110         {MinHeapify(list,i,n);}
111
112 }
113
114 void MinHeapify(char **list,int i, int n){
115     int j = i*2;
116     char *temp = list[i-1];
117     bool done = false;
118
119     while(j<=n && !done){
120         if(j<n && strcmp(list[j-1],list[j+1-1]) > 0) j++;
121         if(strcmp(temp,list[j-1] ) < 0 ) done = true;
122         else{
123             list[j/2-1] = list[j-1];
124             j *= 2;
125         }
126     }
127     list[j/2-1] = temp;
128
129 }
130
131
132 double GetTime(void)
133 {
134     struct timeval tv;
135     gettimeofday(&tv,NULL);
136     return tv.tv_sec+1e-6*tv.tv_usec;
137 }
138
139 int main()
140 {
141     int i,j;
142     int Nwords;
143     double t1,t2;
144     char **words,**A,**OrderedCase,**ReversedCase,**MaxHeap,**MinHeap;
145     int flag = 4;   //(1) insertion sort (2) selection sort (3) bubble sort (4)
     heap sort.
146
147
148
149
```

```
150     scanf("%d", &Nwords);
151     words = (char**)malloc(Nwords * sizeof(char*));     //
152     for(i = 0; i < Nwords; i++)                         //
153         words[i] = (char *)malloc(sizeof(char*));        //
154     A = (char**)malloc(Nwords * sizeof(char*));         //
155     OrderedCase = (char**)malloc(Nwords * sizeof(char*));   //
156     ReversedCase = (char**)malloc(Nwords * sizeof(char*));  //
157     MaxHeap = (char**)malloc(Nwords * sizeof(char*));       //
158     MinHeap= (char**)malloc(Nwords * sizeof(char*));        //
159     for(i = 0; i < Nwords; i++)                         //
160         A[i] = (char *)malloc(sizeof(char*));           //
161
162     for(i = 0; i < Nwords ; i++){                       //
163         scanf("%s", words[i]);                          //scan words
164     }
165     OrderedCase = words;
166     HeapSort(OrderedCase,Nwords);
167     for(i = 0;i < Nwords; i++)
168         ReversedCase[i] = OrderedCase[Nwords-1-i];
169
170     MaxHeap = words;                //
171     MaxHeapGen(MaxHeap,Nwords);     //
172     MinHeap = words;                //
173     MinHeapGen(MinHeap,Nwords);     // Generrate best/worst case for Heapsort
174
175     //Perform sorting
176     t1 = GetTime();
177     for(i = 0; i<500; i++){
178         A  = words;//OrderedCase;//MaxHeap
179         switch(flag){
180             case 1:
181                 InsertionSort(A,Nwords); break;
182             case 2:
183                 SelectionSort(A,Nwords); break;
184             case 3:
185                 BubbleSort(A,Nwords);   break;
186             case 4:
187                 HeapSort(A,Nwords); break;
188             default:
189                 return 0;
190         }
191         if(i==0){
192             for(j = 0; j < Nwords ; j++)
193                 printf("%d %s\n",j+1,A[j]);
194         }
195     }
196     t2 = GetTime();
197     switch(flag){
198         case 1:
199             printf("insertion sort:\n");break;
```

Too small!

Too small!

MaxHeap=MinHeap=words!

```
200         case 2:
201             printf("selection sort:\n");break;
202         case 3:
203             printf("bubble sort:\n");break;
204         default:
205             printf("heap sort:\n");break;
206     }
207     printf("N=%d\nCPU time = %.5g seconds\n", Nwords, (t2-t1)/500.0);
208
209
210     free(words);    //avoid memory leakage
211     return 0;
212 }
213
214
215
```

Score: 57

[Best-case] and worst-case input patterns need to be described.

[Best-case] and worst-case analysis and measurements should be performed for SelectionSort, InsertionSort and BubbleSort.

[Tables and plots] can be more accurately presented.

[lines 153, 160] strings $words[i]$ and $A[i]$ are too small.

[lines 170-173] $MaxHeap = MinHeap = words$.

[line 178] $A = words$ and thus sorting is not done on the original array.