UM–SJTU Joint Institute

Data Structures and Algorithms

(VE281)

Homework 1

Name: Ji Xingyou ID: 515370910197
Date: 26 September 2017

Contents

1 Theoretical Data

As is discussed in the class, we can get the following table summarizing the time complexity for each sorting algorithms.

	Worst case complexity	Average case complexity	In place	Stable
Bubble sort	$O(N^2)$	$O(N^2)$	Yes	Yes
Insertion sort	$O(N^2)$	$O(N^2)$	Yes	Yes
Selection sort	$O(N^2)$	$O(N^2)$	Yes	No
Merge sort	$O(NlogN)$	$O(NlogN)$	No	Yes
Quick sort extra	$O(N^2)$	$O(NlogN)$	No	No
Quick sort in place	$O(N^2)$	$O(NlogN)$	Yes	No

Table 1: Time complexity of comparison sorting

In this report, we will first implement all these six sorting algorithms, and then test the run time for each of them to see whether the above table makes sense.

The implementation of the algorithms is attached in the appendix.

2 Result Analysis

After finishing implementing the above six sorting algorithms, I wrote another program to test the run time of each algorithm. In this program, I set two clocks, noting the starting and finishing instance. To avoid uncertainty in the data, I wrote a while loop to run the program 10 times so that I could get the average value. There is one thing which requires extra carefulness. That is, we must ensure that every time inside the while loop, all the six sorting should meet the identical array. Otherwise, you will see that insertion sort will have a leading performance no matter how large the size is.

The array size I chose is 1, 10, 100, 1000, 5000, 10000, 50000, 100000.

The run time data for each sorting algorithms is listed in table 2.

Array size	Bubble sort	Insertion sort	Selection sort	Merge sort	Quick sort extra	Quick sort in place
1	1	0	0	0	0	1
10	1	0	0	5	2	2
100	34	11	20	40	20	11
1000	2744	880	1300	384	231	137
5000	72741	18552	27256	1706	1085	698
10000	308764	72484	107564	3951	2284	1404
50000	8705506	1802895	2661684	20023	12562	8597
100000	34518575	7144271	10326454	39702	25401	17999

Table 2: Run time of comparison sorting

The run time comparison is shown in figure 1. Notice that in the Matlab code, I used semilogy instead of plot command. The reason is that when using plot command, most data all gathered near the x-axis, which adds to the difficulty in observing.

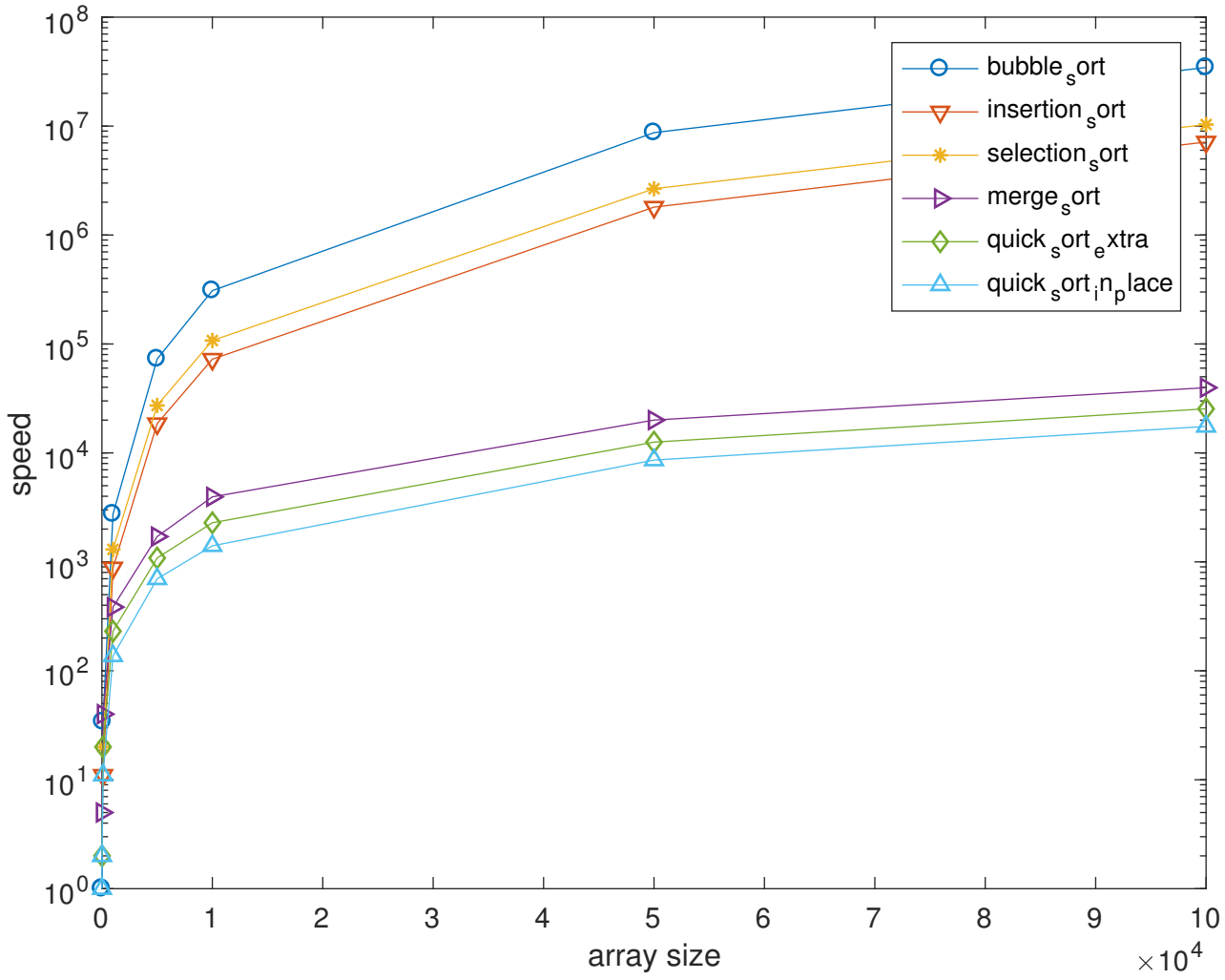

Figure 1: Run time comparison

Combining the table and figure above, we can conclude the following points.

1. For each sorting algorithm, the run time increases as the size of the array increases.

2. When the array size is small (less than 1000), bubble sort, insertion sort and selection sort have a better performance than merge sort, quick sort with extra space and quick sort in place.

3. When the array size becomes gradually larger (greater than 10000), merge sort, quick sort with extra space and quick sort in place have a much better performance than bubble sort, insertion sort and selection sort.

4. It can be inferred from the data and the figure that when the array size is very large (greater than 100000), bubble sort has the least performance, selection sort the second, insertion sort the third, merge sort the fourth, quick sort with extra space the fifth and quick sort in place the best.

From the conclusion listed above, we can see that it fits the time complexity shown in table 1, which means the algorithms make sense.

This inspires me that in future learning, when dealing with sorting, I should make a wise choice on which sorting algorithm to apply. For example, if there is no restriction on stablity and in place, when the array size is small, I should use insertion sort and selection sort. When the array size is large, I should choose merge sort, quick sort with extra space and quick sort in place.

3 Appendix

3.1 Sorting algorithms

```
1 //
2 //  main.cpp
3 //  project1
4 //
5 //  Created by            on 2017/9/11.
6 //  Copyright    2017             . All rights reserved.
7 //
8
9 #include <iostream>
```

```
10 #include <fstream>
11 #include <sstream>
12 #include <string>
13 #include <cstdlib>
14 #include <climits>
15 #include <ctime>
16 #include <cassert>
17
18 using namespace std;
19
20 void bubble_sort(int *arr,int lines);
21
22 void insertion_sort(int *arr,int lines);
23
24 void selection_sort(int *arr,int lines);
25
26 void merge(int *C,int *A,int len_A,int *B,int len_B,int lines);
27
28 void merge_sort(int *arr,int lines);
29
30 int quick_sort_partition_extra(int *arr,int left,int right,int lines);
31
32 void quick_sort_extra(int *arr,int left,int right,int size);
33
34 int quick_sort_partition_in_place(int *arr,int left,int right,int lines,int pivotat);
35
36 void quick_sort_in_place(int *arr,int left,int right,int size);
37 ////////////////////////////
```

```
38 ////////////////////////////
39 int main(int argc, const char * argv[])
40 {
41     int choice;
42     cin>>choice;
43     int lines;
44     cin>>lines;
45     int a[lines];
46     for(int i=0;i<lines;++i)
47     {
48         cin>>a[i];
49     }
50     int *arr=a;
51     switch(choice)
52     {
53         case 0:
54             bubble_sort(arr,lines);
55             break;
56         case 1:
57             insertion_sort(arr,lines);
58             break;
59         case 2:
60             selection_sort(arr,lines);
61             break;
62         case 3:
63             merge_sort(arr,lines);
64             break;
65         case 4:
```

```
66              quick_sort_extra(arr,0,lines-1,lines);
67              break;
68          case 5:
69              quick_sort_in_place(arr,0,lines-1,lines);
70              break;
71          default:
72              break;
73      }
74      for(int k=0;k<lines;++k)
75      {
76          cout<<arr[k]<<endl;
77      }
78      return 0;
79  }
80  ////////////////////////////
81  ////////////////////////////
82  void bubble_sort(int *arr,int lines)
83  {
84      for(int i=lines-1;i>0;--i)
85      {
86          for(int j=0;j<i;++j)
87          {
88              if(arr[j]>arr[j+1])
89              {
90                  swap(arr[j],arr[j+1]);
91              }
92          }
93      }
```

```
94  }
95  ////////////////////////////
96  ////////////////////////////
97  void insertion_sort(int *arr,int lines)
98  {
99      for(int i=0;i<=lines-1;++i)
100     {
101         int temp=arr[i];
102         int j=i;
103         while(j>=1)
104         {
105             if(arr[j-1]>temp)
106             {
107                 arr[j]=arr[j-1];
108                 --j;
109             }
110             else
111             {
112                 break;
113             }
114         }
115         arr[j]=temp;
116     }
117 }
118 ////////////////////////////
119 ////////////////////////////
120 void selection_sort(int *arr,int lines)
121 {
```

```
122     for(int i=0;i<lines-1;++i)
123     {
124         int index=i;
125         for(int j=i+1;j<lines;++j)
126         {
127             if(arr[j]<arr[index])
128             {
129                 index=j;
130             }
131         }
132         if(index!=i)
133         {
134             int tmp=arr[index];
135             arr[index]=arr[i];
136             arr[i]=tmp;
137         }
138     }
139 }
140 ////////////////////////////
141 ////////////////////////////
142 void merge(int *C,int *A,int len_A,int *B,int len_B,int lines)
143 {
144     int i=0,j=0,k=0;
145     while(i<len_A&&j<len_B)
146     {
147         if(A[i]<B[j])
148         {
149             C[k++]=A[i++];
```

```
150         }
151         else
152         {
153             C[k++]=B[j++];
154         }
155     }
156     while(i<len_A)
157     {
158         C[k++]=A[i++];
159     }
160     while(j<len_B)
161     {
162         C[k++]=B[j++];
163     }
164 }
165
166 void merge_sort(int *arr,int lines)
167 {
168     if(lines<=1)
169     {
170         return;
171     }
172     int mid=lines/2;
173     int *A=new int[mid];
174     int *B=new int[lines-mid];
175     for(int i=0;i<mid;++i)
176     {
177         A[i]=arr[i];
```

```
178     }
179     for(int i=mid;i<lines;++i)
180     {
181         B[i-mid]=arr[i];
182     }
183     merge_sort(A,mid);
184     merge_sort(B,lines-mid);
185     merge(arr,A,mid,B,lines-mid,lines);
186     delete [] A;
187     delete [] B;
188 }
189 ////////////////////////////
190 ////////////////////////////
191 int quick_sort_partition_extra(int *arr,int left,int right,int lines)
192 {
193     int *temp=new int[lines];
194     int l=0;
195     int r=lines-1;
196     for(int k=1;k<lines;++k)
197     {
198         if(arr[k]<arr[0])
199         {
200             temp[l]=arr[k];
201             l++;
202         }
203         else
204         {
205             temp[r]=arr[k];
```

```
206             r--;
207         }
208     }
209     temp[l]=arr[0];
210     for(int t=0;t<lines;++t)
211     {
212         arr[t]=temp[t];
213     }
214     delete [] temp;
215     return l;
216 }
217 
218 void quick_sort_extra(int *arr,int left,int right,int size)
219 {
220     int pivotat;
221     if(left>=right)
222     {
223         return;
224     }
225     pivotat=quick_sort_partition_extra(arr,left,right,size);
226     quick_sort_extra(arr,left,pivotat-1,pivotat-left);
227     quick_sort_extra(arr+pivotat+1,0,right-pivotat-1,right-pivotat);
228 }
229 ////////////////////////////
230 ////////////////////////////
231 int quick_sort_partition_in_place(int *arr,int left,int right,int lines,int pivotat)
232 {
233     swap(arr[0],arr[pivotat]);
```

```
234     int i=1;
235     int j=lines-1;
236     while(true)
237     {
238         while(i<lines-1&&arr[i]<arr[0])
239         {
240             ++i;
241         }
242         while(j>0&&arr[j]>=arr[0])
243         {
244             --j;
245         }
246         if(i<j)
247         {
248             swap(arr[i],arr[j]);
249         }
250         else
251         {
252             break;
253         }
254     }
255     swap(arr[0],arr[j]);
256     return j;
257 }
258
259 void quick_sort_in_place(int *arr,int left,int right,int size)
260 {
261     if(size==0)
```

```
262     {
263         return;
264     }
265     int pivotat=rand()%size;
266     if(left>=right)
267     {
268         return;
269     }
270     pivotat=quick_sort_partition_in_place(arr,left,right,size,pivotat);
271     quick_sort_in_place(arr,left,pivotat-1,pivotat-left);
272     quick_sort_in_place(arr+pivotat+1,0,right-pivotat-1,right-pivotat);
273 }
```

3.2 Run-time calculation

```
1  //
2  //  main.cpp
3  //  runtime_study
4  //
5  //  Created by            on 2017/9/23.
6  //  Copyright    2017             . All rights reserved.
7  //
8
9  #include <iostream>
10 #include <fstream>
11 #include <sstream>
12 #include <string>
13 #include <cstdlib>
14 #include <climits>
```

```
15 #include <ctime>
16 #include <cassert>
17
18 using namespace std;
19
20 void bubble_sort(int arr[],int lines);
21
22 void insertion_sort(int arr[],int lines);
23
24 void selection_sort(int arr[],int lines);
25
26 void merge(int *C,int *A,int len_A,int *B,int len_B,int lines);
27
28 void merge_sort(int arr[],int lines);
29
30 int quick_sort_partition_extra(int arr[],int left,int right,int lines);
31
32 void quick_sort_extra(int arr[],int left,int right,int size);
33
34 int quick_sort_partition_in_place(int arr[],int left,int right,int lines,int pivotat);
35
36 void quick_sort_in_place(int arr[],int left,int right,int size);
37 ////////////////////////////
38 ////////////////////////////
39 int main(int argc, const char * argv[])
40 {
41     int lines=50000;
42     long temp0=0;
```

```
43      long temp1=0;
44      long temp2=0;
45      long temp3=0;
46      long temp4=0;
47      long temp5=0;
48      clock_t start,finish;
49      cout<<"lines = "<<lines<<endl;
50      int arr[lines];
51      for(int i=0;i<lines;++i)
52      {
53          arr[i]=mrand48();
54      }
55      int brr[lines];
56      for(int j=0;j<lines;++j)
57      {
58          brr[j]=arr[j];
59      }
60      int k=0;
61      while(k<10)
62      {
63          ////////////////////////////
64          start=clock();
65          for(int a=0;a<lines;++a)
66          {
67              arr[a]=brr[a];
68          }
69          bubble_sort(arr,lines);
70          finish=clock();
```

```
71          long t0=finish-start;
72          temp0+=t0;
73          ////////////////////////////
74          start=clock();
75          for(int a=0;a<lines;++a)
76          {
77              arr[a]=brr[a];
78          }
79          insertion_sort(arr,lines);
80          finish=clock();
81          long t1=finish-start;
82          temp1+=t1;
83          ////////////////////////////
84          start=clock();
85          for(int a=0;a<lines;++a)
86          {
87              arr[a]=brr[a];
88          }
89          selection_sort(arr,lines);
90          finish=clock();
91          long t2=finish-start;
92          temp2+=t2;
93          ////////////////////////////
94          start=clock();
95          for(int a=0;a<lines;++a)
96          {
97              arr[a]=brr[a];
98          }
```

```
99          merge_sort(arr,lines);
100         finish=clock();
101         long t3=finish-start;
102         temp3+=t3;
103         ////////////////////////////
104         start=clock();
105         for(int a=0;a<lines;++a)
106         {
107             arr[a]=brr[a];
108         }
109         quick_sort_extra(arr,0,lines-1,lines);
110         finish=clock();
111         long t4=finish-start;
112         temp4+=t4;
113         ////////////////////////////
114         start=clock();
115         for(int a=0;a<lines;++a)
116         {
117             arr[a]=brr[a];
118         }
119         quick_sort_in_place(arr,0,lines-1,lines);
120         finish=clock();
121         long t5=finish-start;
122         temp5+=t5;
123         ////////////////////////////
124         k++;
125     }
126     long time0=temp0/10;
```

```
127     long time1=temp1/10;
128     long time2=temp2/10;
129     long time3=temp3/10;
130     long time4=temp4/10;
131     long time5=temp5/10;
132     cout<<time0<<endl;
133     cout<<time1<<endl;
134     cout<<time2<<endl;
135     cout<<time3<<endl;
136     cout<<time4<<endl;
137     cout<<time5<<endl;
138     return 0;
139 }
140 ////////////////////////////
141 ////////////////////////////
142 void bubble_sort(int *arr,int lines)
143 {
144     for(int i=lines-1;i>0;--i)
145     {
146         for(int j=0;j<i;++j)
147         {
148             if(arr[j]>arr[j+1])
149             {
150                 swap(arr[j],arr[j+1]);
151             }
152         }
153     }
154 }
```

```
155 ////////////////////////////
156 ////////////////////////////
157 void insertion_sort(int *arr,int lines)
158 {
159     for(int i=0;i<=lines-1;++i)
160     {
161         int temp=arr[i];
162         int j=i;
163         while(j>=1)
164         {
165             if(arr[j-1]>temp)
166             {
167                 arr[j]=arr[j-1];
168                 --j;
169             }
170             else
171             {
172                 break;
173             }
174         }
175         arr[j]=temp;
176     }
177 }
178 ////////////////////////////
179 ////////////////////////////
180 void selection_sort(int *arr,int lines)
181 {
182     for(int i=0;i<lines-1;++i)
```

```
183     {
184         int index=i;
185         for(int j=i+1;j<lines;++j)
186         {
187             if(arr[j]<arr[index])
188             {
189                 index=j;
190             }
191         }
192         if(index!=i)
193         {
194             int tmp=arr[index];
195             arr[index]=arr[i];
196             arr[i]=tmp;
197         }
198     }
199 }
200 ////////////////////////////
201 ////////////////////////////
202 void merge(int *C,int *A,int len_A,int *B,int len_B,int lines)
203 {
204     int i=0,j=0,k=0;
205     while(i<len_A&&j<len_B)
206     {
207         if(A[i]<B[j])
208         {
209             C[k++]=A[i++];
210         }
```

```
211         else
212         {
213             C[k++]=B[j++];
214         }
215     }
216     while(i<len_A)
217     {
218         C[k++]=A[i++];
219     }
220     while(j<len_B)
221     {
222         C[k++]=B[j++];
223     }
224 }
225
226 void merge_sort(int *arr,int lines)
227 {
228     if(lines<=1)
229     {
230         return;
231     }
232     int mid=lines/2;
233     int *A=new int[mid];
234     int *B=new int[lines-mid];
235     for(int i=0;i<mid;++i)
236     {
237         A[i]=arr[i];
238     }
```

```
239     for(int i=mid;i<lines;++i)
240     {
241         B[i-mid]=arr[i];
242     }
243     merge_sort(A,mid);
244     merge_sort(B,lines-mid);
245     merge(arr,A,mid,B,lines-mid,lines);
246     delete [] A;
247     delete [] B;
248 }
249 ////////////////////////////
250 ////////////////////////////
251 int quick_sort_partition_extra(int *arr,int left,int right,int lines)
252 {
253     int *temp=new int[lines];
254     int l=0;
255     int r=lines-1;
256     for(int k=1;k<lines;++k)
257     {
258         if(arr[k]<arr[0])
259         {
260             temp[l]=arr[k];
261             l++;
262         }
263         else
264         {
265             temp[r]=arr[k];
266             r--;
```

```
267         }
268     }
269     temp[l]=arr[0];
270     for(int t=0;t<lines;++t)
271     {
272         arr[t]=temp[t];
273     }
274     delete [] temp;
275     return l;
276 }
277 
278 void quick_sort_extra(int *arr,int left,int right,int size)
279 {
280     int pivotat;
281     if(left>=right)
282     {
283         return;
284     }
285     pivotat=quick_sort_partition_extra(arr,left,right,size);
286     quick_sort_extra(arr,left,pivotat-1,pivotat-left);
287     quick_sort_extra(arr+pivotat+1,0,right-pivotat-1,right-pivotat);
288 }
289 ////////////////////////////
290 ////////////////////////////
291 int quick_sort_partition_in_place(int *arr,int left,int right,int lines,int pivotat)
292 {
293     swap(arr[0],arr[pivotat]);
294     int i=1;
```

```
295     int j=lines-1;
296     while(true)
297     {
298         while(i<lines-1&&arr[i]<arr[0])
299         {
300             ++i;
301         }
302         while(j>0&&arr[j]>=arr[0])
303         {
304             --j;
305         }
306         if(i<j)
307         {
308             swap(arr[i],arr[j]);
309         }
310         else
311         {
312             break;
313         }
314     }
315     swap(arr[0],arr[j]);
316     return j;
317 }
318
319 void quick_sort_in_place(int *arr,int left,int right,int size)
320 {
321     if(size==0)
322     {
```

```
323         return;
324     }
325     int pivotat=rand()%size;
326     if(left>=right)
327     {
328         return;
329     }
330     pivotat=quick_sort_partition_in_place(arr,left,right,size,pivotat);
331     quick_sort_in_place(arr,left,pivotat-1,pivotat-left);
332     quick_sort_in_place(arr+pivotat+1,0,right-pivotat-1,right-pivotat);
333 }
```

3.3 Visualization

```
1  clear all;clc;
2  t0=[1 1 34 2744 72741 308764 8705506 34518575];
3  t1=[0 0 11 880 18552 72484 1802895 7144271];
4  t2=[0 0 20 1300 27256 107564 2661684 10326454];
5  t3=[0 5 40 384 1706 3951 20023 39702];
6  t4=[0 2 20 231 1085 2284 12562 25401];
7  t5=[1 2 11 137 698 1404 8597 17499];
8  size=[1 10 100 1000 5000 10000 50000 100000];
9  semilogy(size,t0,'o-',size,t1,'v-',size,t2,'*-',size,t3,'>-',size,t4,'d-',size,t5,'^-');
10 xlabel('array size');
11 ylabel('speed');
12 legend('bubble_sort','insertion_sort','selection_sort','merge_sort',
13 'quick_sort_extra','quick_sort_in_place');
```