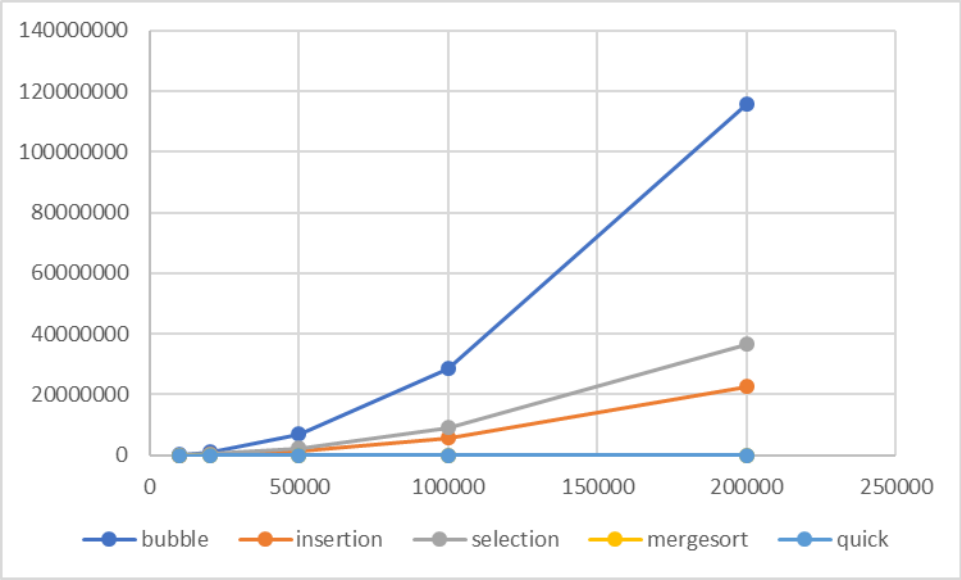
Lab03-SortingSelection

VE281 - Data Structures and Algorithms, Xiaofeng Gao, TA: Li Ma, Autumn 2019

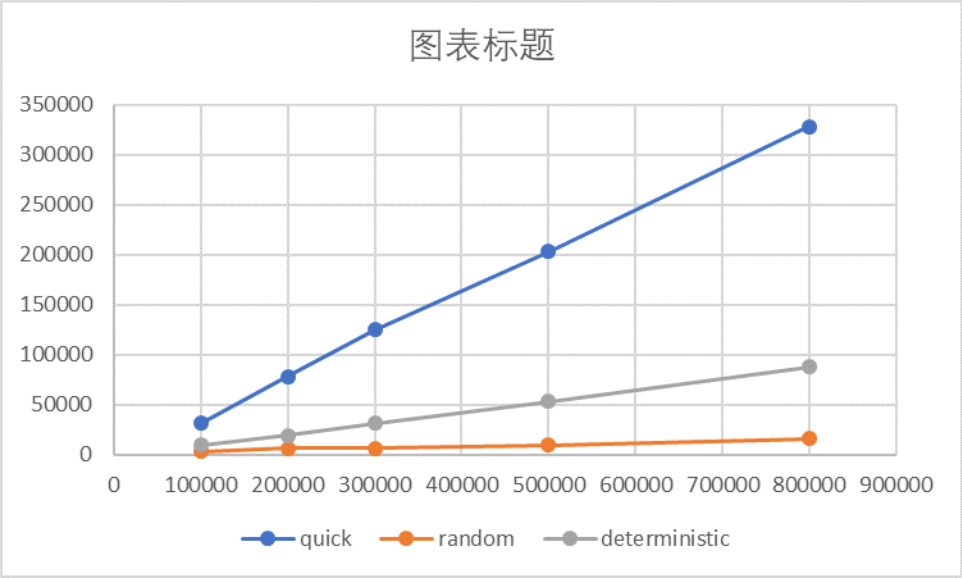
∗ Please upload your assignment to website. Contact webmaster for any questions.

∗ Name: Jintian Ge Student ID: 517021911142 Email: gejintian@sjtu.edu.cn

1. For the first part, the result is shown in the first picture. From the picture, we can see that bubble sort needs the most times to finish sorting, while quick sort and merge sort requires less with respect to the above three algorithm. This is because that bubble sort will go through all the array, while insertion and selection will break from the array earlier. However, their time complexity is still $O(n^2)$. As for quick sort and merge sort, they sacrifice space for time. Their time complexity is $O(n \log n)$. So, they are faster compared to the above three algorithms.



2. In this part, I run quick sort, random selection and deterministic selection. The result is shown in the following picture. For selection algorithm, I choose five locations(i), run the selection function for five times and calculate their average running time. The time complexity of selection algorithms are both $O(n)$. However, when n is not very large, $\log n$ is not very large, so the selection algorithms will not have the same advantage as what quick and merge sort have in the first part. However, we can still see that quick sort is much slower than selection algorithm. Between selection algorithms, though their time complexity are both $O(n)$, deterministic selection is only fast in a regular array. However, if the array is not regular(like random generated), deterministic selection will be slower than random selection because it needs more time to choose a pivot.



3. Here I want to discuss about the selection performance of deterministic selection. Theoretically speaking, deterministic selection should be faster than random selection, because it choose a "better" pivot. I discovered that random selection performance better than deterministic selection in irregular array. This is explained in part2. However, when I tested these algorithm, I tried to input an array with many same numbers. This made deterministic selection very slow, even much slower than bubble sort. I thought that this is because of the chosen of the pivot. If there is so many duplicates number, the pivot will be the same and in this case, it will always find the pivot and lost in the recursion part.

4. Which follows is the source code of my algorithm.

```
    #include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <cstdlib>
#include <climits>
#include <ctime>
#include <cassert>

using namespace std;

void swap(int &a, int &b);
void bubble(int *array, int n);
void insertion(int *array, int n);
void selection(int *array, int n);
void mergesort(int *array, int n);
void quick(int *array, int n);
int random(int *array, int n, int location);
int deterministic(int *array, int n, int location);
int partition(int *array, int left, int right);
int partit(int *array, int left, int right, int pivotat);


int main() {
```

```
25      int type;
26      int number;
27      cin>>type>>number;
28      if(type<5){
29          int a[number];
30          for(int i = 0; i < number; i++){
31              cin>>a[i];
32          }
33          switch(type){
34              case 0:
35              bubble(a,number);
36              break;
37              case 1:
38              insertion(a,number);
39              break;
40              case 2:
41              selection(a,number);
42              break;
43              case 3:
44              mergesort(a,number);
45              break;
46              case 4:
47              quick(a,number);
48              break;
49          }
50          for(int j = 0; j < number; j++){
51              cout<<a[j]<<endl;
52          }
53      }
54      else{
55          int location;
56          cin>>location;
57          int a[number];
58          for(int i = 0; i < number; i++){
59              cin>>a[i];
60          }
61          if(type == 5) cout<<"The order-"<<location<<" item is "<<
                random(a,number,location)<<endl;
62          else cout<<"The order-"<<location<<" item is "<<
                deterministic(a,number,location)<<endl;
63      }
64      return 0;
65  }
66
67
68  void swap(int &a, int &b){
69      int temp = a;
70      a = b;
71      b = temp;
72  }
```

```
73
74 void bubble(int *array, int n){
75     for(int i = n - 2; i >= 0; i--){
76         for(int j = 0; j <= i; j++){
77             if(array[j] > array[j+1]){
78                 swap(array[j], array[j+1]);
79             }
80         }
81     }
82 }
83
84 void insertion(int *array, int n){
85     for(int i = 1; i < n; i++){
86         int t = array[i];
87         int location = 0;
88         for(int j = i - 1; j >=0; j--){
89             if(array[j] > t){
90                 array[j + 1] = array[j];
91             }
92             else{
93                 location = j + 1;
94                 break;
95             }
96         }
97         array[location] = t;
98     }
99 }
100
101 void selection(int *array, int n){
102     for(int i = 0; i < n - 1; i ++){
103         int t = i;
104         for(int j = i + 1; j < n; j++){
105             if(array[t] > array[j]) t = j;
106         }
107         swap(array[i], array[t]);
108     }
109 }
110
111 void merge(int *a, int left, int mid, int right){
112     int i = left, j = mid + 1, k = 0;
113     int *c = new int[right - left + 1];
114     while(i <= mid && j <= right){
115         if(a[i] <= a[j]) c[k++] = a[i++];
116         else c[k++] = a[j++];
117     }
118     while(i <= mid) c[k++] = a[i++];
119     while(j <= right) c[k++] = a[j++];
120     for(int m = 0; m <= right - left; m++) a[left + m] = c[m];
121     delete[] c;
122 }
```

```
123 static void merge_helper(int *array, int left, int right){
124     if(left >= right) return;
125     int mid = (int)(left+right)/2;
126     merge_helper(array, left, mid);
127     merge_helper(array, mid + 1, right);
128     merge(array, left, mid, right);
129 }
130 void mergesort(int *array, int n){
131     merge_helper(array, 0, n - 1);
132 }
133 
134 int partition(int *array, int left, int right){
135     int i = left+1;
136     int j = right;
137     int p = left + rand()%(right - left + 1);
138     int pivot = array[p];
139     swap(array[left],array[p]);
140     if(i == j){
141         if(array[left]<=array[right]) return left;
142         else{
143             swap(array[left],array[right]);
144             return right;
145         }
146     }
147     while(i<j){
148         while(array[i] < pivot && i <= right){i++;}
149         while(array[j] >= pivot && j > left){j--;}
150         if(i < j) swap(array[i],array[j]);
151     }
152     swap(array[left],array[j]);
153     return j;
154 }
155 static void quick_helper(int *array, int left, int right){
156     int pivotat = 0;
157     if(left >= right) return;
158     pivotat = partition(array, left, right);
159     quick_helper(array, left, pivotat - 1);
160     quick_helper(array, pivotat + 1, right);
161 
162 }
163 void quick(int *array, int n){
164     quick_helper(array, 0, n - 1);
165 }
166 
167 
168 int random_helper(int *array, int left, int right, int location){
169     if(right - left == 0) return array[left];
170     int pivotat;
171     pivotat = partition(array, left, right);
172     if(pivotat == location) return array[pivotat];
```

```
173     if(pivotat > location) return random_helper(array, left,
            pivotat-1, location);
174     else return random_helper(array, pivotat+1, right, location);
175 }
176 int random(int *array, int n, int location){
177     return random_helper(array, 0, n - 1, location);
178 }
179
180
181 int partit(int *array, int left, int right, int pivotat){
182     int i = left + 1;
183     int j = right;
184     for(int m = left; m <= right; m++){
185         if(pivotat == array[m]){
186             swap(array[m], array[left]);
187             break;
188         }
189     }
190     if(i == j){
191         if(array[left]<=array[right]) return left;
192         else{
193             swap(array[left],array[right]);
194             return right;
195         }
196     }
197     while(i<j){
198         while(array[i] < pivotat && i <= right){i++;}
199         while(array[j] >= pivotat && j > left){j--;}
200         if(i < j) {
201             swap(array[i],array[j]);}
202     }
203     swap(array[left],array[j]);
204     return j;
205 }
206 int d_helper(int *array, int left, int right, int location){
207     if(right - left == 0) return array[left];
208     int n = right - left +1;
209     int number = n/5;
210     int medians[number];
211     int j;
212     if(n>=5){
213         int i;
214         for(i = 0; i < n - 4; i += 5){
215
216             selection(array+ left + i, 5);
217
218             medians[i/5] = array[left + i + 2];
219
220         }
221         int p = d_helper(medians, 0, number - 1, number/2);
```

```
222             j = partit(array, left, right,p);
223         }
224         else {j = partition(array, left, right);}
225         if(j == location) return array[j];
226         if(j > location) return d_helper(array, left, j - 1, location);
227         else return d_helper(array, j + 1, right, location);
228 }
229 int deterministic(int *array, int n, int location){
230         return d_helper(array, 0, n - 1, location);
231 }
```