Lab03 Sorting and Selection

VE281 - Data Structures and Algorithms, Xiaofeng Gao, Autumn 2019

* Name:Jin Zhejian Student ID: 517370910167 Email: jinzhejian@outlook.com

1 Preparation

In order to show the relationship between runtime and array size better, I choose the array size 100, 500, 1000, 5000, 10000, 50000. I generate random numbers and put them in the array, and then save them in 6 .txt file. The source code is shown below.

```
#include <iostream>
#include <fstream>
#include <stdlib.h>
using namespace std;

int main(){
  fstream fso("data_50000.txt", ios::out);
  int size = 50000;
  int i;
  for(i = 0; i < size; ++i)
  fso << rand() << ' ';
  fso.close();

  fstream fsi("data_50000.txt", ios::in);
  int x[size];
  for(i = 0; i < size; ++i)
  fsi >> x[i];

  int j;
  for(i = 0; i < size; ++i)
    for(j = size-1; j > i; --j)
      if(x[j - 1] > x[j]){
      int t = x[j];
      x[j] = x[j - 1];
      x[j - 1] = t;
      }
  for(i = 0; i < size; ++i)
    cout << x[i] << '\t';
}

```

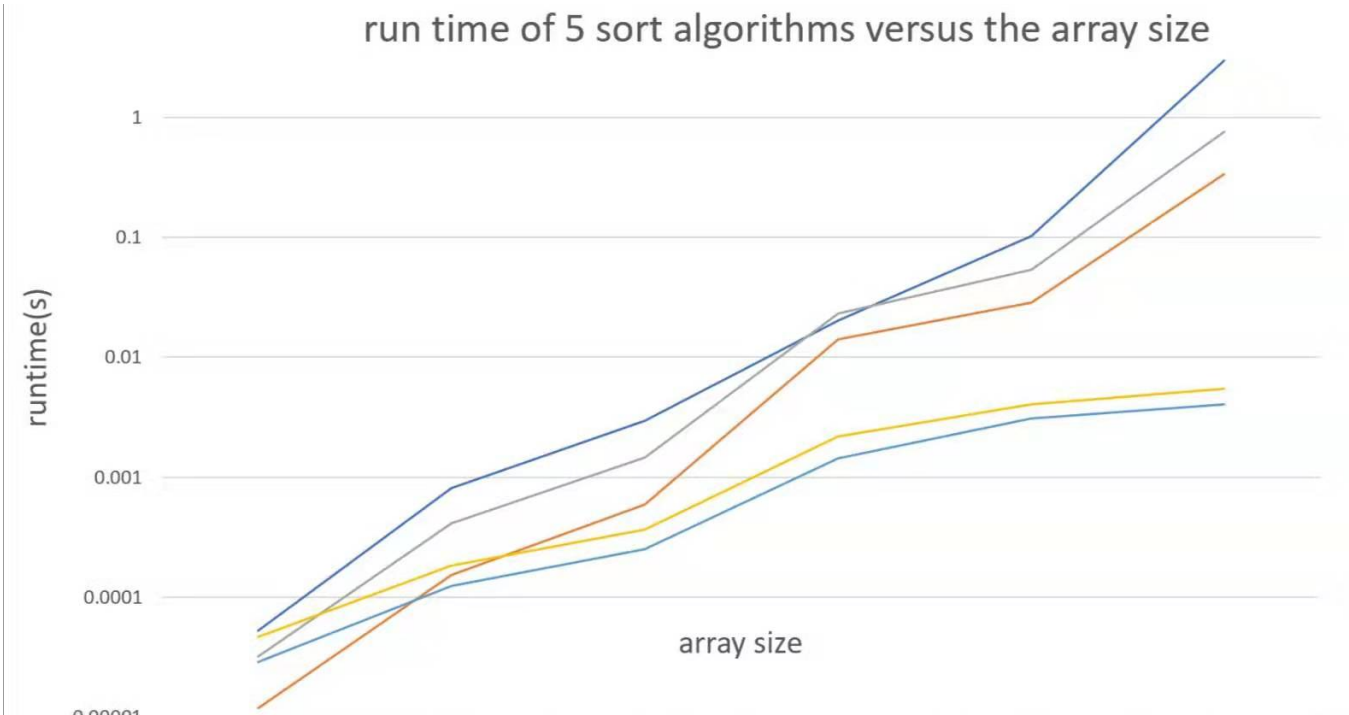
Listing 1: generate.cpp

In order to calculate the runtime of my realization more precisely, I do 20 times of the sort and calculate the average runtime. Also, for the selection algorithm, I choorse 20 different i numbers and calculate the average runtime.

2 Sort algorithms

In this lab, I studied 5 sort algorithms, they are: BubbleSort ,InsertionSort,SelectionSort MergeSort, and QuickSort. I chose the array size 100, 500, 1000, 5000, 10000, 50000, and I record each of the run times of the five sort algoritms.

The graph and my data are shown as follows:



	100	500	1000	5000	10000	50000
Bubble	0.000053	0.000812	0.002939	0.020332	0.103193	2.947915
Insertion	0.000012	0.000153	0.000595	0.014081	0.028535	0.336396
Selection	0.000032	0.000412	0.001453	0.023156	0.053947	0.752872
Merge	0.000047	0.000183	0.000366	0.002191	0.004083	0.005471
Quick	0.000029	0.000124	0.000251	0.001445	0.003117	0.004045

Figure 1: runtime of 5 sort algorithms versus the array size

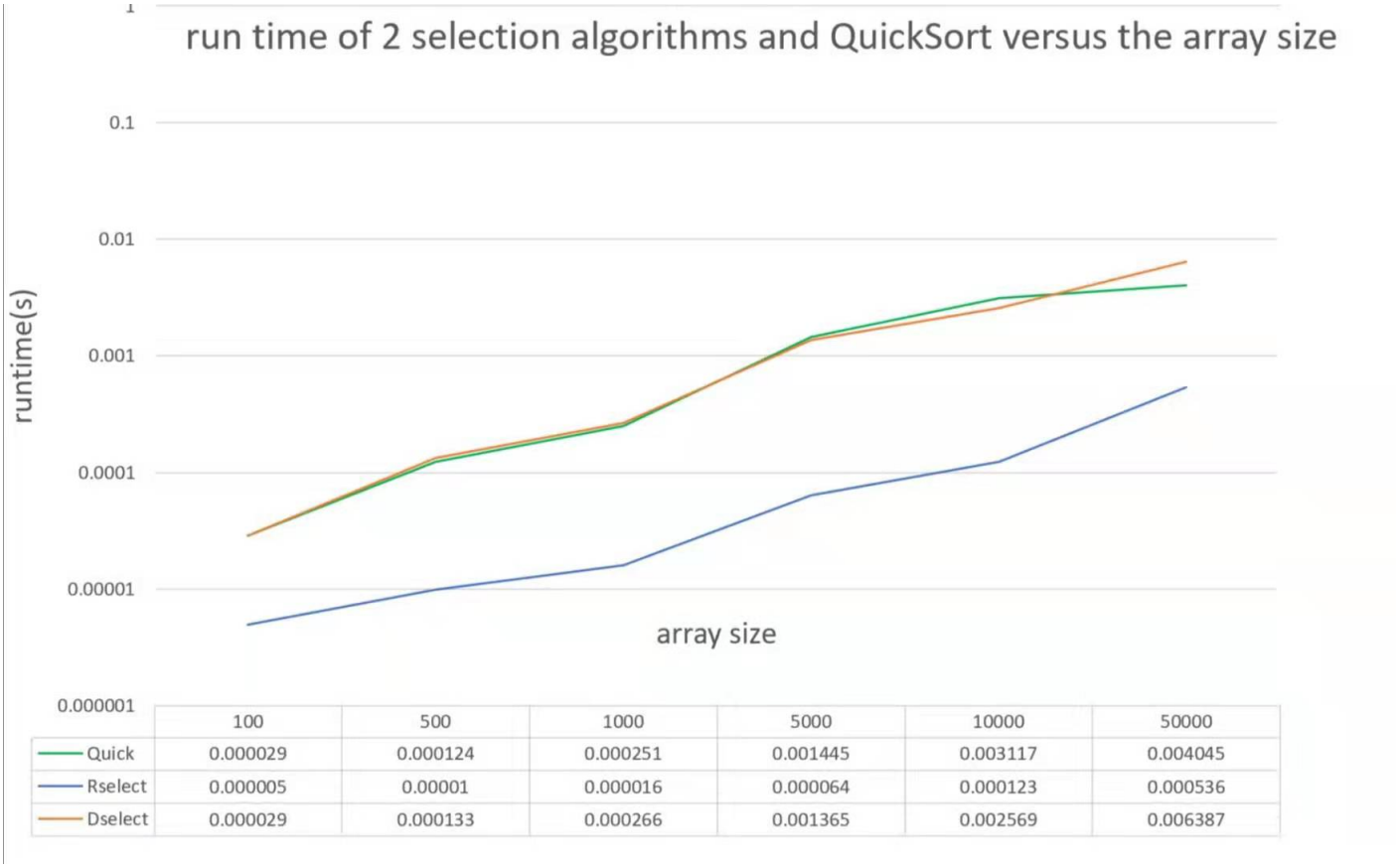
From the figure, we can find that the runtime of Merge Sort and Quick Sort is faster that of Bubble Sort, Insertion Sort, and Selection Sort.

In my realization, the Insertion Sort is several times faster than the Bubble Sort. One potential reason is that, in my code, in the while loop of Bubble Sort, the program need additionally to judge whether the Boolean variable sorted is true or false, which causes additional runtime. Of course, the Bubble Sort of my realization can be improved. Only needs to uncomment the commented part **if (sorted){break;}**. The runtime will be fasten observably.

For Merge Sort and Quick Sort, we can see from the figure that the two algorithms have the same time complexity, and Merge Sort is a little slower than the Quicksort. From my guess, the difference is due to the reason that Merge Sort uses Dynamic Memory in the heap, which is slower than the stack operations. This will be discussed further in the Selection algorithms part.

3 Selection algorithms

In this part, I compare the runtime of the two selection algorithms(random selection algorithm and deterministic selection algorithm) with the quick sort algorithm. The second figure indicates the runtime of 2 different selection algorithms and quick sort versus the array size, which is shown as follows:



	100	500	1000	5000	10000	50000
Quick	0.000029	0.000124	0.000251	0.001445	0.003117	0.004045
Rselect	0.000005	0.00001	0.000016	0.000064	0.000123	0.000536
Dselect	0.000029	0.000133	0.000266	0.001365	0.002569	0.006387

Figure 2: runtime of 2 selection algorithms and QuickSort versus the array size

The result does not satisfy our theoretical knowledge, although the graph shows the three algorithms have the same growing speed. Deterministic Selection is a faster version of Random Selection theoretically, however, in my realization, the Dselect is much faster than R select. My expatiation is that: Dselect and Quick Sort use **Dynamic Memory**, such as **int *C = new int[size]** in Dselect. It is done in the heap in the memory, whose memory management speed is slower than the stack, because it needs to dynamically allocate the memory when running the code.