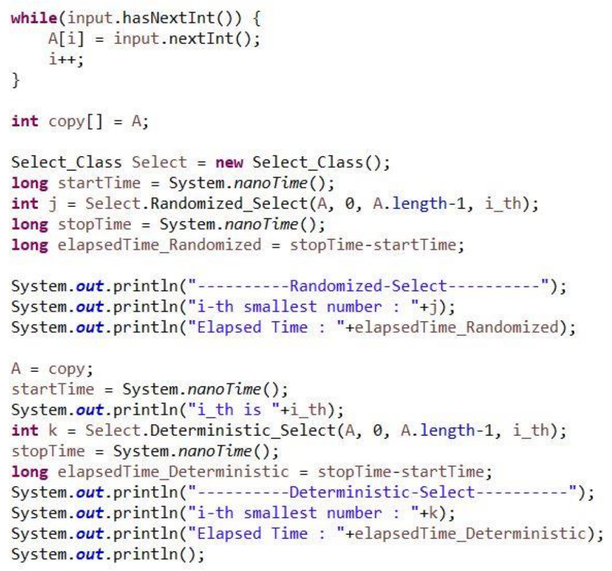
Deterministic Selection and Randomized Selection by Java

현지훈(2016-14251 컴퓨터공학부)

1. Actual time usage of the two algorithms.

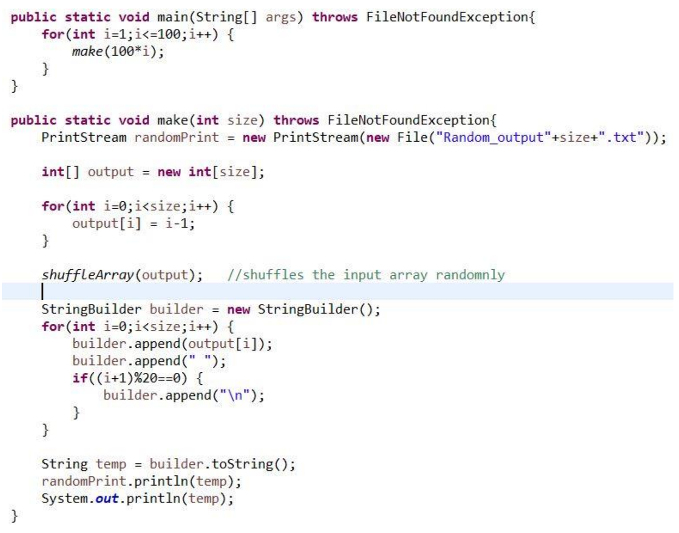
In order to calculate the elapsed time, I used the System.nanoTime() method in java, as in figure 2. I printed the result time in not only the console, but also in a .txt file to compute the ration of the constants.

For the input files, I made a hundred input files, whose sizes varies from 100 to 10,000, with intervals of 100. In each input file, the range of the integers are from -1 to (100*i-2), mixed randomly. (See figure 1 for better understanding.).



```
while(input.hasNextInt()) {
    A[i] = input.nextInt();
    i++;
}

int copy[] = A;

Select_Class Select = new Select_Class();
long startTime = System.nanoTime();
int j = Select.Randomized_Select(A, 0, A.length-1, i_th);
long stopTime = System.nanoTime();
long elapsedTime_Randomized = stopTime-startTime;

System.out.println("----------Randomized-Select----------");
System.out.println("i-th smallest number : "+j);
System.out.println("Elapsed Time : "+elapsedTime_Randomized);

A = copy;
startTime = System.nanoTime();
System.out.println("i_th is "+i_th);
int k = Select.Deterministic_Select(A, 0, A.length-1, i_th);
stopTime = System.nanoTime();
long elapsedTime_Deterministic = stopTime-startTime;
System.out.println("----------Deterministic-Select----------");
System.out.println("i-th smallest number : "+k);
System.out.println("Elapsed Time : "+elapsedTime_Deterministic);
System.out.println();
```

Figure 2:How I measured the elapsed time for each algorithm



```
public static void main(String[] args) throws FileNotFoundException{
    for(int i=1;i<=100;i++) {
        make(100*i);
    }
}

public static void make(int size) throws FileNotFoundException{
    PrintStream randomPrint = new PrintStream(new File("Random_output"+size+".txt"));

    int[] output = new int[size];

    for(int i=0;i<size;i++) {
        output[i] = i-1;
    }

    shuffleArray(output);   //shuffles the input array randomnly

    StringBuilder builder = new StringBuilder();
    for(int i=0;i<size;i++) {
        builder.append(output[i]);
        builder.append(" ");
        if((i+1)%20==0) {
            builder.append("\n");
        }
    }

    String temp = builder.toString();
    randomPrint.println(temp);
    System.out.println(temp);
}
```

Figure 1:How the random input files were made

For each algorithm, I recorded the respective elapsed times which differed as the input size changed. I conducted this for all input files, which means I ran this program with 100 iterations. Then by using excel, the constants hidden in the asymptotic time complexities were able to be calculated.

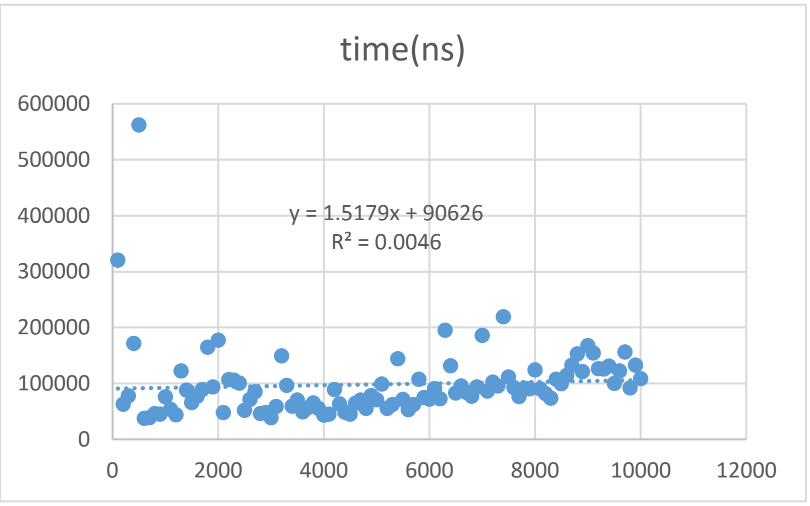


Figure 3:Randomized Selection

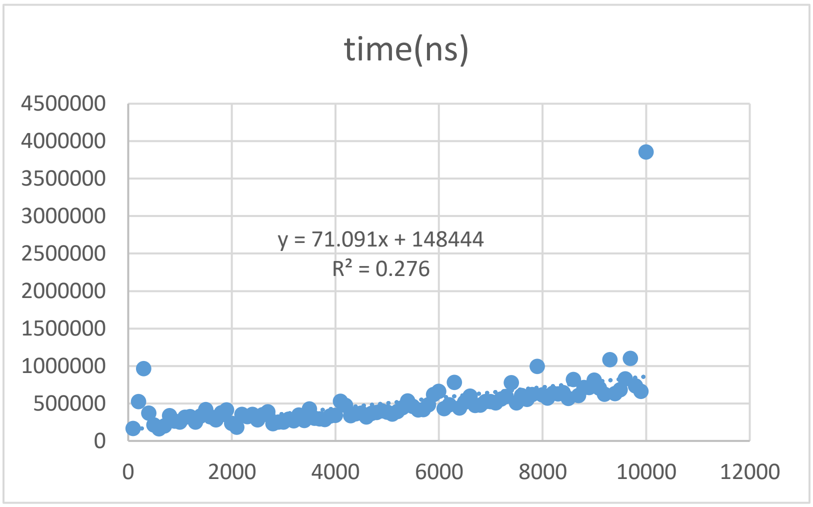
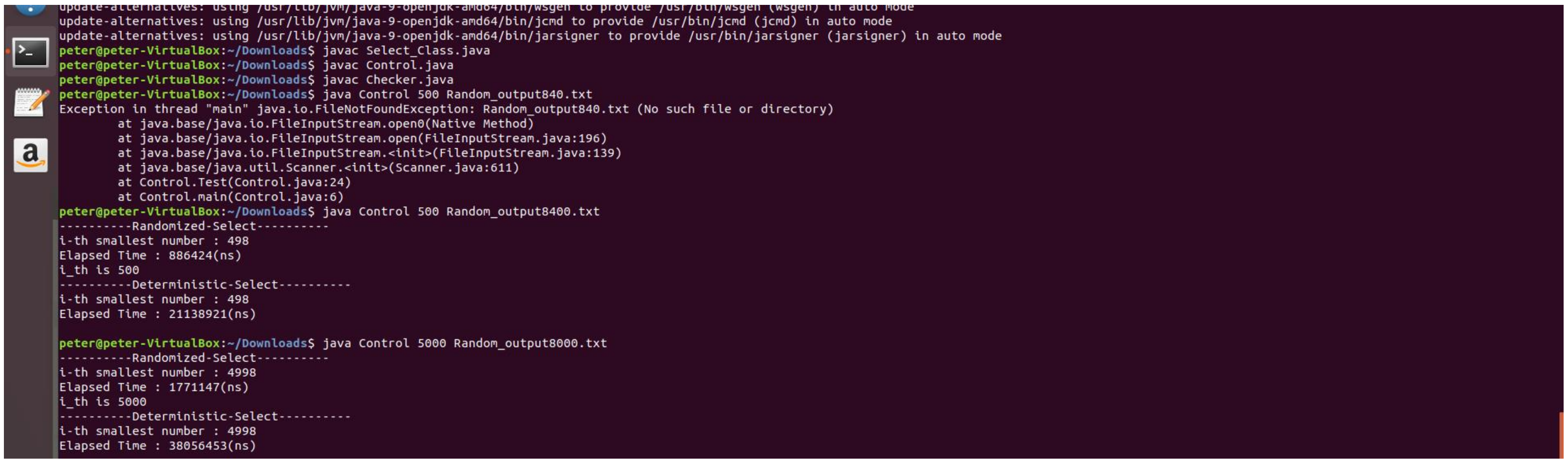


Figure 4:Deterministic Selection

Figure 3 and 4 shows the result analyzed via excel. The x-axis represents the input size, whereas the y axis represents the elapsed time. The result of linear aggression on these data gave me the constants hidden in asymptotic time complexities.

Thus, Deterministic Selection's coefficient was 71.091, whereas the randomized selection's coefficient was 1.5179. The ratio between these two numbers would be **46.835** (Deterministic : Randomized), which means that the randomized selection algorithm was mostly faster than the deterministic selection algorithm. I expect this happened because the overhead that was needed to calculate the pivot ("median of the medians") of the Deterministic Selection was much higher than that of the Randomized Selection method.

2. Example running of the two algorithms within the Linux environment.



```
update-alternatives: using /usr/lib/jvm/java-9-openjdk-amd64/bin/wsgen to provide /usr/bin/wsgen (wsgen) in auto mode
update-alternatives: using /usr/lib/jvm/java-9-openjdk-amd64/bin/jcmd to provide /usr/bin/jcmd (jcmd) in auto mode
update-alternatives: using /usr/lib/jvm/java-9-openjdk-amd64/bin/jarsigner to provide /usr/bin/jarsigner (jarsigner) in auto mode
peter@peter-VirtualBox:~/Downloads$ javac Select_Class.java
peter@peter-VirtualBox:~/Downloads$ javac Control.java
peter@peter-VirtualBox:~/Downloads$ javac Checker.java
peter@peter-VirtualBox:~/Downloads$ java Control 500 Random_output840.txt
Exception in thread "main" java.io.FileNotFoundException: Random_output840.txt (No such file or directory)
        at java.base/java.io.FileInputStream.open0(Native Method)
        at java.base/java.io.FileInputStream.open(FileInputStream.java:196)
        at java.base/java.io.FileInputStream.<init>(FileInputStream.java:139)
        at java.base/java.util.Scanner.<init>(Scanner.java:611)
        at Control.Test(Control.java:24)
        at Control.main(Control.java:6)
peter@peter-VirtualBox:~/Downloads$ java Control 500 Random_output8400.txt
----------Randomized-Select---------
i-th smallest number : 498
Elapsed Time : 886424(ns)
i_th is 500
----------Deterministic-Select----------
i-th smallest number : 498
Elapsed Time : 21138921(ns)

peter@peter-VirtualBox:~/Downloads$ java Control 5000 Random_output8000.txt
----------Randomized-Select---------
i-th smallest number : 4998
Elapsed Time : 1771147(ns)
i_th is 5000
----------Deterministic-Select----------
i-th smallest number : 4998
Elapsed Time : 38056453(ns)
```

Figure 3:Example running of my program

There are 3 .java files and 100 inputs within the attached. In order to run java files in linux environment, you have to install jdk(or jre) beforehand. After jdk is installed, we have to make class files from the java files.

Make a class file with the following command.

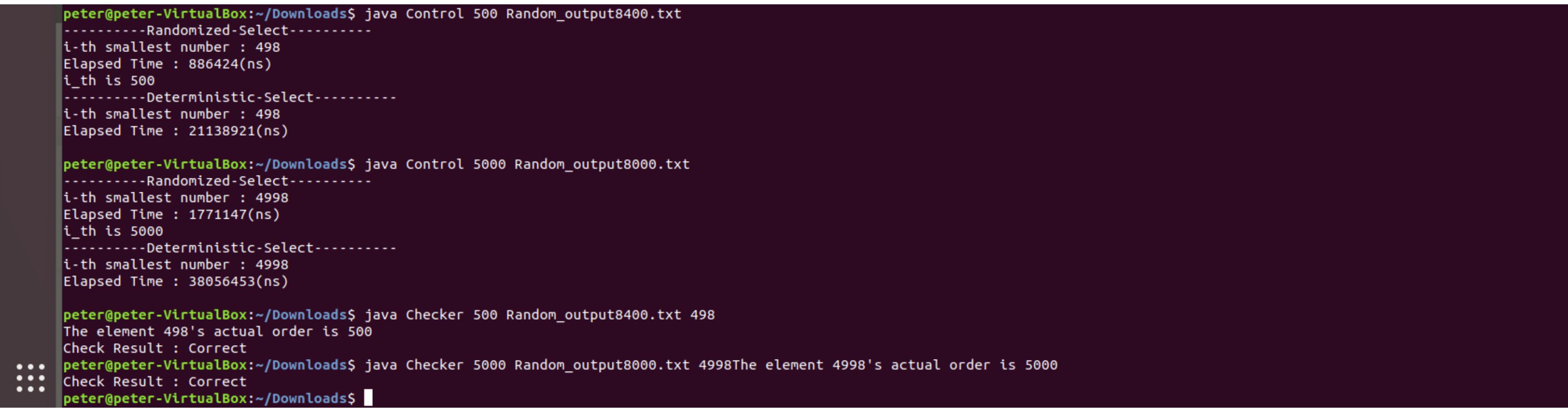
javac (java file name).java

After classes are formed, run the code with the command

java Control (i) (file)

Input ‘i’ means that you should find the i-th smallest element. Input ‘file’ is the file name. In my first example, I made my program find the 500th smallest element from “Random_output8400.txt”, and in my second example, I made my program find the 5000th smallest element from “Random_output8000.txt”.

3. How to run my checker program



```
peter@peter-VirtualBox:~/Downloads$ java Control 500 Random_output8400.txt
----------Randomized-Select---------
i-th smallest number : 498
Elapsed Time : 886424(ns)
i_th is 500
----------Deterministic-Select---------
i-th smallest number : 498
Elapsed Time : 21138921(ns)

peter@peter-VirtualBox:~/Downloads$ java Control 5000 Random_output8000.txt
----------Randomized-Select---------
i-th smallest number : 4998
Elapsed Time : 1771147(ns)
i_th is 5000
----------Deterministic-Select---------
i-th smallest number : 4998
Elapsed Time : 38056453(ns)

peter@peter-VirtualBox:~/Downloads$ java Checker 500 Random_output8400.txt 498
The element 498's actual order is 500
Check Result : Correct
peter@peter-VirtualBox:~/Downloads$ java Checker 5000 Random_output8000.txt 4998The element 4998's actual order is 5000
Check Result : Correct
peter@peter-VirtualBox:~/Downloads$
```

Figure 4:How to run my checker program

My checker program is implemented in my Checker.java file. After making a class from this file, type the following statement.

java Checker (i) (file) (i-th element)

The input ‘i’ and ‘file’ is same as before. The ‘i-th element’ input is the result obtained from the Select algorithms. In the example above, it shows that the resulted 498 and 4998 from the Selection algorithm was correct.

How my checker program works is quite simple. After the program reads in the i-th element provided from the selection algorithm, it just reads in the whole file’s number array, and compares it one by one sequentially. If the i-th element is bigger than or equal to the compared element, then the ‘order’ variable, which is set as 0 initially, is added by a value of 1. If else, it just moves on to the next array element. The moving on stops when tie i-th element matches the compared element. Because this program just sequentially compares the input elements, and has no iteration, the checker program for selection always runs in linear time.