



ASSIGNMENT 5 — SOLUTION

GENERAL INFORMATION

This assignment is made public on Friday, March 7, 2003. The assignment is due on

Friday, March 14, 1 PM.

Hand in your assignment to the teaching assistant running your lab session.

The first page of your (written) assignment has to contain at least the following information:

- the course name (Introduction to Programming - Concepts and Tools)
- your name and your student number
- name and student number of the fellow student if you submit in pairs
- assignment number

Please staple your assignment!

You will get back the graded assignment one week after submission deadline.

QUESTIONS

1. (*Big-Oh*)

- You are given the information that the running time of one algorithm is $O(n^2)$, and that the running time of another algorithm is $O(n \log(n))$. What can you say about the relative performance of the algorithms?
- You are given the information that the running time of one algorithm is n^2 , and that the running time of another algorithm is $O(n \log(n))$. What can you say about the relative performance of the algorithms?
- You are given the information that the running time of one algorithm is n^2 , and that the running time of another algorithm is $n \log(n)$. What can you say about the relative performance of the algorithms?

SOLUTION

- The only thing that can be said is that for large problem sets the first algorithm takes more time than the second. More precisely, we say that for large n (n larger than some fixed constant N) the second algorithm is faster than the first. However, we cannot say what that N is.
- The same as above holds. Even though we know the complexity of one algorithm precisely, we only know the complexity of the second up to a constant factor.
- For $n \geq 1$ the second algorithm will be faster than the first, since in that case $\log(n) < n$, thus $n \log(n) < n^2$.

2. (Sorting)

(From Peter Sestoft's notes.)

- (a) Manually execute selection sort on the array

0	1	2	3	4	5
35	62	28	50	11	45

Show the values of i and `least` and the contents of the array, in each iteration of the outer loop *after* the `swap` method call.

- (b) Manually execute quicksort on the array above. Show the values of l, r, i, j and x and the array's contents at appropriate steps of the execution.

SOLUTION

For the first I used the following program:

```
class SestoftSel{

    public static void main(String[] args){

        int[] a = {35, 62, 28, 50, 11, 45};

        selsort(a);
    }

    static void selsort(int[] a){
        for(int i =0; i<a.length; i++)
        {
            int least = i;
            for(int j =i+1; j<a.length; j++)
            {
                if(a[j]<a[least])
                    least = j;
            }
            swap(a,i,least);
            // print out information
            System.out.print("i: " + i);
            System.out.print("\tleast: " + least);
            System.out.print("\tarray: ");
            print(a);
            System.out.println();
        }
    }

    static void swap(int[] a, int s, int t){
        int temp = a[s];
        a[s] = a[t];
        a[t] = temp;
        return;
    }

    static void print(int[] a){
        for(int i = 0; i<a.length; i++)
            System.out.print(a[i] + "\t");
        return;
    }
}
```

which gives the following:

```
$java SestofteSel
i: 0   least: 4   array: 11   62   28   50   35   45
i: 1   least: 2   array: 11   28   62   50   35   45
i: 2   least: 4   array: 11   28   35   50   62   45
i: 3   least: 5   array: 11   28   35   45   62   50
i: 4   least: 5   array: 11   28   35   45   50   62
i: 5   least: 5   array: 11   28   35   45   50   62
$
```

Replacing the method `selsort` by `quicksort` below,

```
static void qsort(int[] a, int l, int r){
    if(l < r){
        int i = l;
        int j = r;
        int x = a[(i+j)/2];
        do{
            while(a[i]<x) i++;
            while(a[j]>x) j--;
            if(i<=j)
                {
                    swap(a,i,j);

                    i++;
                    j--;
                }
            // print out information
            System.out.print("l: " + l);
            System.out.print("\tr: " + r);
            System.out.print("\ti: " + i);
            System.out.print("\tj: " + j);
            System.out.print("\tx: " + x);
            System.out.print("\tarray: ");
            print(a);
            System.out.println();
        }while(i<=j);
        qsort(a,l,j);
        qsort(a,i,r);
    }
}
```

gives the following:

```
l: 0 r: 5 i: 1 j: 3 x: 28 array: 11 62 28 50 35 45
l: 0 r: 5 i: 2 j: 1 x: 28 array: 11 28 62 50 35 45
l: 0 r: 1 i: 1 j: -1 x: 11 array: 11 28 62 50 35 45
l: 2 r: 5 i: 3 j: 4 x: 50 array: 11 28 45 50 35 62
l: 2 r: 5 i: 4 j: 3 x: 50 array: 11 28 45 35 50 62
l: 2 r: 3 i: 3 j: 2 x: 45 array: 11 28 35 45 50 62
l: 4 r: 5 i: 5 j: 3 x: 50 array: 11 28 35 45 50 62
```

3. (Time needed to solve problems)

In this question you are supposed to get a feel that good algorithms matter by calculating, for various problem sets, the time needed to perform the task.

For example, a machine performing 10^6 operation per second, will use for a linear algorithm on a data set of size 10^9 about 1000 seconds, which is between 15 and 20 minutes. Using a $n \log(n)$ algorithms the same task takes about 8 hours, while the task will never finish if the algorithm is quadratic (to be precise, it will take more than 30000 years).

You are supposed to write a program that takes as line arguments one of the flags `-l`, `-n`, and `-q` for linear, $n \log(n)$, or quadratic time, the performance rate of the machine in operations per seconds, and the size of the data set, and will output in years, days, hours, and seconds, the time that it will take to perform that task.

The program might be called as follows

```
java Time -q 1000000 1000000000
```

and should output something like

```
Quadratic:
Operations: 1000000      Problem Size: 1000000000
Time: 31709 year(s), 289 day(s), 1 hour(s), and 2800 second(s)
```

To do so we will break down the task:

- (a) Write a method

```
long linear(long, long)
```

that will take as first argument the number of operations (per second), and as second argument the problem size. The method should return the number of seconds that it takes to perform a task of *problem size* with a *linear time* algorithm with bound n on a machine which performs *number of operations* per second.

- (b) Write similar methods

```
long nlogn(long, long)
```

and

```
long quadratic(long, long)
```

calculating the time needed if the algorithm is $n \log(n)$ or n^2 (quadratic).

Note 1: The natural logarithm \ln (with base the mathematical constant e) can be accessed through `Math.log(double)`. Its return type is also double. To calculate the logarithm with base 2 note that $\log(n) = \ln(n)/\ln(2)$.

Note 2: Some of the calculations in the two methods will bring us out of range of `long` integers. You should therefore do the calculations (inside the methods) with double values, and then cast to `long` before you return the result.

- (c) Express time in seconds as years, days, hours, and seconds. To do so write a method

```
int [] convert(long)
```

that will take as argument a time in seconds, and will return an array of length 4, where cell 0 contains the years, cell 1 the days, cell 2 the hours, and cell 3 the seconds. You may assume that a year has 365 days.

- (d) Write your main method. To convert a string holding a `long` integer number into a `long` number use the method

```
static long parseLong(String s){
    return Long.parseLong(s);
}
```

To get the flag character `l`, `n`, or `q` use the method

```

    static char parseSecondChar(String s){
        return s.charAt(1);
    }

```

Finally, use a switch statement depending on that flag to determine the running time.

- (e) Use your program to calculate the time needed on a machine that performs 1 million operations per second on a problem set of size 1 million, if the algorithm is linear, $n \log(n)$, or quadratic.
- (f) The same as above, this time with a problem set of size 1 billion (10^9).
- (g) Do similar calculations (problem sets of size one million and one billion, algorithms of linear, $n \log(n)$, and quadratic complexity) for machines that perform 10^9 and 10^{12} operations per second.

Submit your program, and the answers you found in the last three questions.

SOLUTION

One solution is the following program:

```

import tio.*;

class Time{

    public static void main(String[] args){

        long ops = parseLong(args[1]);
        long problem = parseLong(args[2]);
        char complex = parseSecondChar(args[0]);

        switch(complex){
            case 'l':
                System.out.println("Linear:");
                printline(ops,problem,convert(linear(ops,problem)));
                System.out.println();
                break;
            case 'n':
                System.out.println("nlog(n):");
                printline(ops,problem,convert(nlogn(ops,problem)));
                System.out.println();
                break;
            case 'q':
                System.out.println("Quadratic:");
                printline(ops,problem,convert(quadratic(ops,problem)));
                System.out.println();
                break;
            default:
                System.out.println("You didn't specify correctly the complexity using -l, -n, -q");
        }
    }

    static long linear(long opsPerSecond, long problemSize){
        return problemSize/opsPerSecond;
    }

    static long nlogn(long opsPerSecond, long problemSize){
        double t = problemSize * Math.log(problemSize)/Math.log(2);
        return (long)(t/opsPerSecond);
    }
}

```

```

static long quadratic(long opsPerSecond, long problemSize){
    double t = problemSize;
    return (long)((t*t)/opsPerSecond);
}

static long[] convert(long timeInSecs){
    long[] t = new long[4];

    long r = timeInSecs;
    /* The variable r is not needed, one could use timeInSecs instead.
    */

    t[3] = (long) (r%3600);
    r = r/3600;
    t[2] = (long)(r%24);
    r = r/24;
    t[1] = (long)(r%365);
    t[0] = (long)(r/365);
    return t;
}

static void printline(long opsPerSecond, long problemSize, long[] time){
    System.out.print("Operations: " + opsPerSecond);
    System.out.println("\tProblem Size: " + problemSize);
    System.out.print("Time: " + time[0] + " year(s), ");
    System.out.print(time[1] + " day(s), ");
    System.out.print(time[2] + " hour(s), and ");
    System.out.print(time[3] + " second(s)");
    return;
}

static long parseLong(String s){
    return Long.parseLong(s);
}

static int parseInt(String s){
    return Integer.parseInt(s);
}

static char parseSecondChar(String s){
    return s.charAt(1);
}
}

```

4. (Sorting arrays of numbers)

In this question we will measure the performance of sorting algorithms. We will create arrays of size 0, 500, 1000, ..., 10000, fill them with random numbers, sort them, and measure how long the sorting took.

- (a) Write a method of signature

```
void fill(int[], int)
```

that will fill the array passed to the method with randomly generated integer numbers in the range from 0 (included) to the integer number passed to the method (excluded). Recall that random numbers in the range from 0 to $n - 1$ can be generated using `(int)(Math.random()*n)`

- (b) Write a main method that will do the following: Within a loop you should create arrays of size 0, 500, 1000, ..., 10000, fill them with random numbers between 0 and 10000, sort them using selection sort, and type a message on the screen that the array of size n got sorted. The code for selection sort can be downloaded from the homepage.)
- (c) Modify your program to actually print out the time (in milli-seconds) that it takes to sort your arrays. (Information on how to do this follows below.) The output of your program might look as follows:

```
$ java TimedSorting
0 cells sorted in 0 milliseconds
500 cells sorted in 11 milliseconds
1000 cells sorted in 13 milliseconds
1500 cells sorted in 22 milliseconds
2000 cells sorted in 45 milliseconds
2500 cells sorted in 65 milliseconds
3000 cells sorted in 91 milliseconds
3500 cells sorted in 121 milliseconds
4000 cells sorted in 157 milliseconds
4500 cells sorted in 200 milliseconds
5000 cells sorted in 240 milliseconds
5500 cells sorted in 289 milliseconds
6000 cells sorted in 342 milliseconds
6500 cells sorted in 403 milliseconds
7000 cells sorted in 461 milliseconds
7500 cells sorted in 535 milliseconds
8000 cells sorted in 603 milliseconds
8500 cells sorted in 680 milliseconds
9000 cells sorted in 765 milliseconds
9500 cells sorted in 849 milliseconds
10000 cells sorted in 939 milliseconds
$
```

Submit the program listing, one output of your program, and a plot showing the graphical relationship between the size of the array and the time needed to sort the data.

- (d) Do the same as in the question above, this time using quicksort to sort the contents of the array. (The code for quicksort can be downloaded from the homepage.)

To solve the last two tasks you need to copy the file `StopWatch.java` from the homepage into the folder holding your program files. The file provides access to a very primitive stop-watch that you can use to do your measurements. A stop-watch is created using the command

```
StopWatch sw = new StopWatch();
```

Once a stop-watch is created it starts running (and will never stop). The stop-watch comes with two functions:

```
sw.reset();
```

will reset the watch to 0, and

```
sw.getTime();
```

will return the current time in milli-seconds as a long value.

Thus, to use the watch create it at the beginning of your program, set it to 0 when you want to start using it, and read off the current time and store that value or print it on the screen once you want to do your measurement.

For example, the following little program asks the user to input some integer data, and types out in milli-seconds how long it took the user to input the data — not very useful, but it shows the use of the stop-watch. Note carefully where we reset the watch.

```
/* Program to print the time it takes the user to input data.
 * Uses Stopwatch class.
 *
 * Carsten Butz, September 2002
 */
import tio.*;

class StopUserInput{

    public static void main(String[] args){
        int input=0;
        Stopwatch sw = new Stopwatch();

        while(input != -1){
            sw.reset();
            System.out.println("Please enter a number and press return.");
            input = Console.in.readInt();
            System.out.print("It took you " + sw.getTime());
            System.out.println(" millisecond(s) to enter the data.");
        }
    }
}
```

SOLUTION

```
class TimedSorting{

    public static void main(String[] args){

        Stopwatch c = new Stopwatch();

        int i=0;
        final int MAX = 10000; // max size of the array to be sorted
        final int STEPS = 500; // increase in size
        int[] arr;           // array
        long timeTaken = 0;   // long to hold the time taken

        while(i<=MAX){
            arr = new int[i];           // create the array
            fillArray(arr,MAX);         // fill the array with data

            c.reset();                  // set stop watch to 0 (and start running)
            selsort(arr);               // sort the array
            timeTaken = c.getTime();     // read the stop watch and store the time found
            System.out.println(i + " cells sorted in " + timeTaken + " milliseconds");
            i = i+STEPS;                // prepare for next loop
        }
    }

    static void fillArray(int[] a, int m){
        for(int i = 0; i<a.length; i++)
            a[i] = (int)(Math.random()*m);
    }

    static void selsort(int[] a){
        for(int i =0; i<a.length; i++)
        {
            int least = i;
            for(int j =i+1; j<a.length; j++)
            {
                if(a[j]<a[least])
                    least = j;
            }
            swap(a,i,least);
        }
    }

    static void swap(int[] a, int s, int t)
    {
        int temp = a[s]; a[s] = a[t]; a[t] = temp;
    }
}
```