Practical 1: Analysis of Algorithms

Background
There used to be an attitude that still persists today that as long as your code runs and gives the correct output, nothing else matters. In the past if your code ran rather slowly the easy solution was to add more processing power. Ultimately it comes down to which solution (building a better algorithm or adding more computational resources) is the best option given the requirements of your project. If you just need to make something work for a small scale project then focusing too much on optimizing your algorithm is not worthwhile.

However, in a world where algorithms increasingly need to perform effectively with endless amounts of data or on limited hardware, the ability to measure and monitor the performance of alternative algorithmic solutions becomes a very attractive skill. And if you want to work for a top computer company (i.e. Facebook, Google, Tesla) then this is a prerequisite skill.

What am I doing today?
Today's practical focuses on 3 things:

1. Counting basic instructions of programs
2. Translating a pseudocode multiplication program in to Java
3. Timing the clock performance of a program

Instructions
Try all the questions. Ask for help from the demonstrators if you get stuck.
Solutions will be posted afterward.

***Grading: Remember** if you complete the practical, add the code to your GitHub repo which needs to be submitted at the end of the course **for an extra 5%**

****NOTE:** we will demo how to get set up with your GitHub repo in Practical 3

1. Algorithms for Multiplication

Russian Peasant's algorithm

The *Russian Peasant's Algorithm* is an algorithm for multiplication that uses doubling, halving, and addition. This was an algorithm or a tool that was used before computers by people to multiply two numbers. One advantage it possesses over the standard method of multiplication that is taught in many schools (i.e. the Standard Algorithm) is that you do not need to have previously memorized multiplication tables to use the algorithm. In practice, the Russian Peasant's Algorithm was likely calculated with the aid of small stones or beads to represent the units.

Read more here:
https://en.wikipedia.org/wiki/Ancient_Egyptian_multiplication#Russian_peasant_multiplication

The Russian Peasant's Algorithm Instructions:

The basic idea is that to multiply x by y, we can compute instead:

While (number1 != 0){

If number1 is even, then add nothing

If number1 is odd, add number 2 to the running total

number1/2

number2*2

}

Multiply 238 x 13

For example, to multiply 238 (number 1) by 13 (number 2) , the smaller of the numbers (to reduce the number of steps), 13, is written on the right and the larger on the left (the order does not matter, it's a matter of convenience). The left number is progressively halved (discarding any remainder) and the right one doubled, until the left number is 1. Then you add all of the number2s together in rows where the number 1 is odd (see below).

13	**238**	**13**	**238**
6 (remainder discarded)	**476**	~~6~~	~~476~~
3	**952**	**3**	**952**
1 (remainder discarded)	**1904**	**1**	+ **1904**
			3094

Exercise:
Your first exercise is to use the Russian Peasant's Algorithm to multiply **by hand**:

68 x 139

Half (throw away remainder)	**Double**
~~68~~	~~139~~
~~34~~	~~278~~
17	556
~~8~~	~~1112~~
~~4~~	~~2224~~
~~2~~	~~4448~~
1	8896
Total	9452

2. Counting Instructions

When analysing the performance of an algorithm we can assign a fixed and equivalent amount of time to the following operations:

- Assigning a value to a variable
- Calling a method
- Performing an arithmetic operation
- Comparing two numbers
- Indexing into an array
- Following an object reference

- Returning from a method

We call these individual “units of processing” **UNIT TIME**. We imagine an abstract computer that take one UNIT of TIME of each primitive operation above. This is the first step in assessing how our algorithm’s performance changes as the input grows WITHOUT having to consider specific software or hardware details.

Exercise

Count the instructions below in the psuedocode algorithm below and add them up to see how this algorithm performs.

arrayMax function	Finds the biggest integer in an array
Input: an array A of N integers	
Output: maximum element of A	
arrayMax(A, n) {	1
currentMax = A[0]	2
For (i=0; i < A.length; i++){	3N (1 + 2N + N - 1)
If (A[i] > currentMax) then	2N
currentMax = A[i]	2N
}	
Return currentMax	1
End function	
Total: T(N)	7N + 4 which is O(n)

3. Implementing the Russian Peasant's algorithm in Java (using ints/ longs) and verify its correctness.

Take the pseudocode below and translate into a basic java algorithm that implements the Russian Peasant's algorithm:

3. Try your best to write this code yourself (no cutting and pasting or searching the web for solutions) - this is the best way to learn. If you get stuck as a demonstrator for assistance.
4. Your algorithm can take input either through hard-coded integers or through prompt input.
5. Test your algorithm on several input integers and verify the correctness of the output using a calculator (or by hand).

English-style Pseudocode:
Let the two given numbers be 'a' and 'b'.
1) Initialize result 'res' as 0.
2) Do following while 'b' is greater than 0
  a) If 'b' is odd, add 'a' to 'res'
  b) Double 'a' and halve 'b'
3) Return 'res'.

***Java-style* Pseudocode for Russian algorithm**

```
RussianMultiply(n, m){

        accumulator = 0
        while (n!= 0){
        If n is even, then add nothing
        If n is odd, accumulator +=  m
        n1/2
        m*2
        }


}
```

4. Time the performance of the algorithm with various input numbers and plot the results

1. To get a very rough idea of how your algorithm performs, add code to time how it performs multiplying numbers of different size. Java.lang.System.currentTimeMillis() will be of use here.

2. You can use this graph to get your started if you like.

Import java.lang.system.currentTimeMillis()
This method returns the difference, measured in milliseconds, between the current time and midnight, January 1, 1970 UTC(coordinated universal time)
Final long elapsedTime = System.currentTimeMillis()

Sample code you can use to time how your algorithm performs with different size input:

```
public static void main(String[] args)
{
final long startTime = System.currentTimeMillis();
myFunction(N):
final long elapsedTime = System.currentTimeMillis() - startTime;
System.out.println("the time taken " + elapsedTime);
}
```

Unfortunately, due to the simplistic nature of the Russian Peasant's algorithm, it is almost impossible to measure the running time on a modern system. Though the algorithm does run in O(log n) time, a typically measurable result, there are some specific details relating to the algorithm which influence the actual maximum and minimum runtimes possible using this algorithm. The 'n', in the above O(log n) characterisation of the algorithm, represents the position of the most significant set bit (1) in the input integers. However, most modern systems support at most 64-bit integers, meaning that there is a hard ceiling on how large the input can be. In absolute terms, the loop which is the most computationally expensive part of the algorithm (see code highlighted in red below), is only iterated over between 0 and 8 times for a 64-bit integer. Given these tiny differences, it is almost impossible to measure the difference in running time, as the loop itself contains only three very basic arithmetic instructions. Therefore, the algorithm effectively runs in constant time given the limited constraints of the test environment.

```
public static long russianPeasantAlgorithm(long a, long b) throws Exception {
        long result = 0;                // this red section is iterated over between 0 and 8 times.
        while(b > 0){                   // on modern systems, this is effectively constant-time,
                if((b % 2) == 1){       // because the loop contains very simple instructions
                        result += a;
                }
                b /= 2;
                a *= 2;
        }
        return result;
}
```