COMP20012 Tutorial 2: Algorithms and their Complexity

This is to occupy one tutorial session. It covers material in Chapter 5 in the course textbook (*Data Structures and Problem Solving using Java*, by Mark Allen Weiss, 2002). You should attempt the questions, writing out full answers, before the tutorial. You may need to consult the course textbook to help with some of the material.

Question 1.

Be prepared to explain to your tutor the following concepts:

1. Measures of the performance of an algorithm. Time and space complexity.
2. Worst-case, average-case and best-case complexity.

Question 2.

Describe the performance of the following algorithms using suitable measures of complexity. You should make clear what operations you are counting, what is the worst-case that you are considering, (and, perhaps, average-case and best-case, where appropriate). Consider also space complexity.

1. Multiplication of two $N \times N$ matrices. Also, the N-th power of an $N \times N$ matrix, using repeated matrix multiplication,
2. Long multiplication of integers (i.e. the standard digitwise multiplication of integers),
3. Integer division (i.e. an integer result, plus remainder) using 'long division'.

Question 3.

The definition of the Fibonacci sequence is as follows:

$$f(0) = 0, \quad f(1) = 1, \quad f(n+2) = f(n+1) + f(n).$$

This could be interpreted as a recursive algorithm for computing the Fibonacci sequence. Here is an alternative algorithm:

```
static int fib(int n)
{
  int [] FibSeq = new int[n+1];
  FibSeq[0]= 0; FibSeq[1]= 1;
  for (int i= 2; i <= n; i++)
    FibSeq[i] = FibSeq[i-1] + FibSeq[i-2];
  return FibSeq[n];
}
```

This creates an array containing the whole sequence up to the nth item. Yet only the two immediate predecessors are required to calculate a value in the sequence. Use this idea to recast this algorithm, removing the array and using instead two additional variables.

What is the (time) complexity of the original recursive definition as a means of computing items in the Fibonacci sequence? Hint: Consider how many addition operations are used at each unfolding of the recursion, and consider the size of the tree of recursive calls. Use the fact that the Fibonacci sequence $f(n)$ is of order r^n where r is the golden ratio, $r = 1.61803\ldots$.

What does this result say about the efficient use of recursion?

What is the (time) complexity of the above algorithm in Java and of your improved version? (This is a simple example of a general technique: saving the results of sufficient recursive calls to enable a result to be computed without recomputing intermediate values. It is sometimes called 'memoization'.)