COMP20012 Tutorial 2: Algorithms and their Complexity – The questions

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'.)

COMP20012 Tutorial 2: Algorithms and their Complexity – Tutors notes

Tutors: There is rather a lot of material here but much of it is presented in the lectures. Try to get through questions 1 and 2.

This material here is in the course textbook, Chapter 5 and 7 of *Data Structures and Problem Solving using Java*, by Mark Allen Weiss, 2002.

David

Aim: This is to reinforce the basics of complexity measures which are described and discussed in the lectures. It consists of pen-and-paper exercises as well as a requirement for the group to formulate the basic concepts in their own words.

For the former, some of the key aspects are measures that are independent of the way that an algorithm is implemented, ie measures intrinsic to the algorithm itself. For time complexity, these are counting suitable operations - these operations themselves should be of constant complexity in terms of the algorithm's input size and should reflect where the real work is concentrated in an algorithm. The best case etc are in terms of the range of actual inputs for each input size.

The pen-and-paper exercises ask the students not only to calculate a few of these measures for familiar algorithms, but also what forms of approximation are appropriate in these calculations. They illustrate how small changes in algorithms can have a considerable effect on the complexity, eg from exponential to linear!

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'.

Answer

(1) Multiplication of two $N \times N$ matrices, is $c_{i,k} = \Sigma_j a_{i,j} \times b_{j,k}$, that is for each of the N^2 results $c_{i,k}$, we perform N multiplications, and $N-1$ additions, so in total N^3 multiplications of elements and $N^2(N-1)$ additions of elements.

The intention is that the power is calculated as $(\ldots(A \times A) \times A) \times A) \ldots \times A)$. Each multiplication of an $N \times N$ matrix requires, using the standard formula for matrix multiplication, $O(N^3)$ multiplications of matrix elements. So the overall complexity is $O(N^4)$.

Strassen's algorithm for matrix multiplication allows us to reduce this to $O(N \times N^{2.81})$ Moreover, we can accumulate the exponential in faster ways e.g. $A^4 =$ let $S = A \times A$ in $S \times S$, requiring only two multiplications. You may want to hint at these improved algorithms, and further improvements based on repeatedly splitting the sequence of multiplications and saving intermediate values.

(2) A long multiplication of an M-digit number a with an N-digit number b requires $M \times N$ multiplications of digits. Now M is approximately $\log_{10} a$ and N is approximately $\log_{10} b$. The number of additions is harder - there may be carry digits. Without carry digits it takes $(M-1) \times (N-1)$ digit additions. Anyway, both the number of multiplications and of additions is quadratic, $O(MN)$.

There are faster methods of integer multiplication, suitable for large integers, based upon the Discrete (or Fast) Fourier Transform. These are $O(N \log(N))$ for integers of N digits. The overheads of this method make it unsuitable for small integers,

(3) If the number of digits in the numerator is N and in the denominator is D, then the number of multiplications of digits is $O(N \times D)$, so the performance is quadratic. This takes a bit of seeing and an example is perhaps worthwhile.

```
      288
   -------
12 ) 3456
     24
     --
     105
      96
     ---
       96
       96
       --
        0
```

At each stage we invent a digit of the result, multiply each digit of the denominator by this, perform a subtraction and then move along one digit of the numerator. So for each digit of the numerator, we perform D multiplications. The invention of the digits of the result can, of course, be performed with at most $9D$ digit multiplications if necessary.

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 100-th 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'.)

Answer

Here is a Fibonnacci algorithm based upon storing the two previous items, calculating the current item then updating the previous to move along one step in the sequence. This is a simple example of so-called *dynamic programming* – preventing the recalculation of results in a recursive definition by storing the required intermediate results.

```
static int fib(int n)
  { int x, y;
    x= 0; y= 1;
    for (int i = 1; i <= n; i++)
       { x = x + y; y = x - y; }
    return x; }
```

Note: The subtraction present here is simply to assign the original value of x to y (we could have used another variable to store x temporarily).

For the complexity of the recursive algorithm: If C_N is the complexity of calculating the N-th item (counting the number of additions) then the definition of the Fibonacci sequence gives:

$$C_N = C_{N-1} + C_{N-2} + 1$$

(where the 1 is the one addition at top level and the other two terms are the number of additions in the recursive calls). Thus the complexity grows just as the terms of the sequence grow, i.e. $C_N = O(r^N)$, and the computation is *exponential*. More accurately, the solution of the

recurrence relation is $C_N = f(N+2) + f(N+1) - 1$, where $f(N)$ is the N-th Fibonacci number, as may be verified by substitution.

It may be worth explaining how quickly this rises and how inefficient and infeasible this becomes.

The other algorithms are all *linear*, $O(N)$.