Recursion (II)

1 Memoization and dynamic programming

Recursive code tends to be simple and clear, but sometimes it is highly inefficient. Think of the method that we wrote before to calculate the nth Fibonacci number:

```
public static int fib(int n) {
    if ((n == 1) || (n == 2)) {
      return 1;
    } else {
        int result = fib(n-1) + fib(n-2);
        return result;
    }
}
```

This methods works fine for small numbers (up to 30), but then it becomes extremely expensive to compute and takes a very long to provide an answer, as shown in Figure 1. Why is this?

To understand what is happening here, let's see what the method is doing for $n = 4$. As n is not 1, we have to calculate `fib(3)` and `fib(2)` and then add them up. First we calculate the former, `fib(3)`, so we calculate `fib(2)` and `fib(1)` and add them up; they are both 1, so `fib(3)` is 2. Then we move on to calculate `fib(2)`... but wait a second... we are calculating `fib(2)` again!

The higher the number, the more calculations we are repeating unnecessarily (see Figure 2). For every new number of Fibonacci, we are roughly duplicating the number of calculations needed with respect to the preceding number. This leads to exponential growth, which is exactly the behaviour observed in Figure 1.

Not all is lost for recursive computation, however, because we can very easily keep track of the values we have already calculated. If we store these value in memory for later use we do not need to calculate them again. We can use an array for this purpose, created the first time the method is called. The resulting method could look like this:

```
// arrays are 0-based, so F(1) is stored at position 0, etc
private int[] precalculated = null;

public int fib(int n) {
    if (precalculated == null) {
```

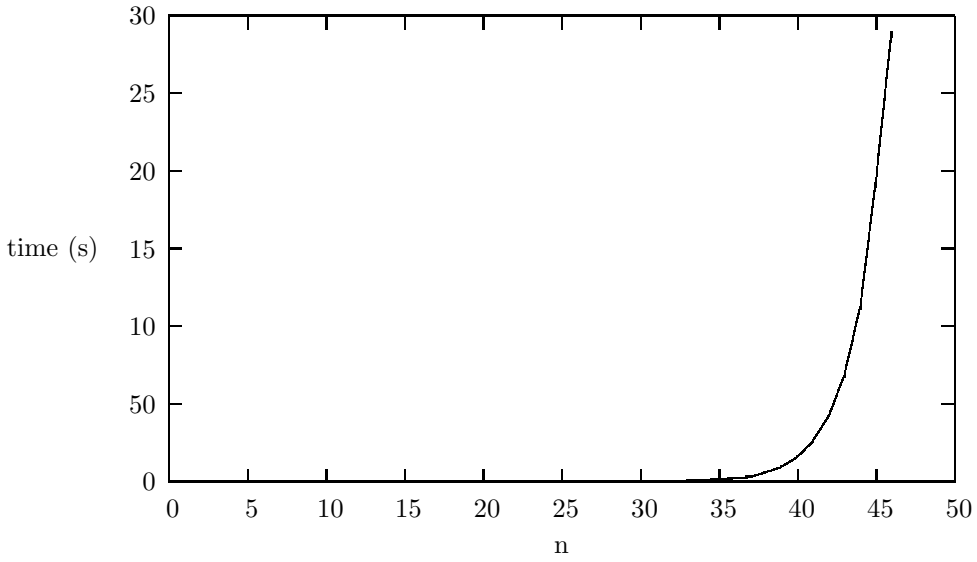

Figure 1: Time used by the method fib(int) depending on the input parameter

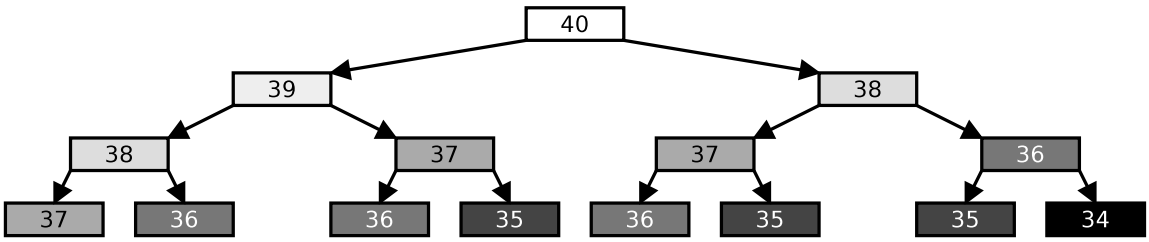

Figure 2: Tree of recursive computations for the Fibonacci numbers. It can be observed that there are many repetitions.

```
        precalculated = new int[n];
        for (int i = 0; i < precalculated.length; i++) {
             precalculated[i] = -1; // to indicate "not calculated yet"
        }
    }
    if ((n == 1) || (n == 2)) {
      return 1;
    } else {
        if (precalculated[n-1] != -1) {
            return precalculated[n-1];
        } else {
            int result = fib(n-1) + fib(n-2);
            precalculated[n-1] = result;
            return result;
        }
    }
}
// ...additional code would go here...
```

This is kind of OK, but the clarity of this code could be greatly improved with just two small tricks: first, move the initialisation of the array to another method; second, remove the first `if` by including the base cases (1 and 2) in the array at initialisation. The new code would look like this:

```
// arrays are 0-based, so F(1) is stored at position 0, etc
private int[] precalculated = null;

public int fib(int n) {
    if (precalculated == null) {
        initPrecalculatedArray(n);
    }
    if (precalculated[n-1] != -1) {
        return precalculated[n-1];
    } else {
        int result = fib(n-1) + fib(n-2);
        precalculated[n-1] = result;
        return result;
    }
}

private void initPrecalculatedArray(int size) {
   precalculated = new int[size];
   for (int i = 0; i < precalculated.length; i++) {
        precalculated[i] = -1; // to indicate "not calculated yet"
   }
   precalculated[0] = 1; // F(1)
   precalculated[1] = 1; // F(2)
}
```

```
// ...additional code would go here...
```

Now the method `fib(int)` is much clearer. If the array of precalculated values is not initialised (i.e. it is `null`), we initialise it. Then we start calculating smaller Fibonacci numbers and storing them in the array. Before we do any calculation, we check the array; if it contains the precalculated value, we just use that number. There are no repeated calculations in this case. This version of the method uses a couple of milliseconds for finding a Fibonacci number, even for larger numbers.

Storing intermediate variables for reusing them is a simple technique called *memoization* (not to be confused with memorization), sometimes written memo-ization.

The approach of solving a problem by dividing it into smaller sub-problems, and combining the results of the smaller instances to find the solution of the bigger one (memoizing intermediate results to prevent redundant re-computation), is called *dynamic programming*[1].

2 Divide-and-conquer

Some years ago, a guy called Julius Caesar conquered the Gaul in just eight years; a remarkable feat at the time. He did not conquer the whole Gaul in one go, but rather divided the problem (conquering the Gaul) into smaller subproblems (conquering each small tribe) and then solved the smaller problems to get to the final solution.

We can use the same strategy in our programs. We can divide a problem into smaller subproblems that we can attack more easily; once we have the solution for the smaller subproblems, we can integrate those solutions to get a general solution. This is called a divide-and-conquer strategy and fits nicely with recursive approaches.

2.1 More examples from the real world

Another example of divide-and-conquer in the real world (that, like the Julius Caesar's example, seemed more interesting in the days before email and twitter) is the postal mail system. The postal mail does not deliver each letter individually, because that would be too costly. Instead, the letters are roughly organised into zones, and then sent in big lots to the zone offices to be delivered. Zone offices do not deliver letters individually either, but assign them to areas or postal codes, and send them in big batches to the right local post office. The local post office is then able to deliver all letters in their area to their final destinations.

In the days of the internet, the Domain Name System is another example of recursive divide-and-conquer. When you look for the server at "www.bbk.ac.uk", you start by asking the owner of the ".uk" domain. It does not have the address of every single machine in the UK, but it knows which DNS servers are responsible of the top-level subdomains (e.g. co, ac, etc), so it can redirect you to the "ac" responsible. This server does not know every single machine in British universities, but it knows who is responsible of each subdomain (e.g. bbk, ox, cam, etc), and will send you to the server responsible of "bbk". This server can tell you what is the address of "www.bbk.ac.uk" so that you can access the homepage of Birkbeck.

[1] It is important to note that (somewhat confusingly) the term "dynamic programming" has nothing to do with writing software, and actually *predates* the appearance of computers and software by many years. Originally, the term programming in 'dynamic programming' referred to "program" in the sense of a military schedule for training not in the sense of a list of instructions to be processed by a computer.

As you can see, dividing the problems of postal mail delivery and domain name resolution into smaller subproblems makes them more manageable. You have also observed that the smaller subproblems are basically smaller versions of the big problem. This is the type of situation in which recursive approaches come naturally.

2.2 Programming example: depth and size of a tree

A divide-and-conquer strategy is defined by three steps:

1. Division of the big problem into two or more subproblems.
2. Finding the solution of the smaller subproblems.
3. Integration of the subsolution into a solution to the big problem.

We have already seen an example of a divide-and-conquer strategy. When we calculated the depth of a binary tree, we did not need to iterate through all the nodes in the tree counting steps as we traversed the tree up and down. Instead, we divided the problem into two smaller subproblems: finding the size of the left subtree and the right subtree. Once we found the solution of both subproblems, the integration involved taking the maximum depth and add one to it.

A related problem is finding the size of a tree, i.e. the number of elements it contains. First, we divide the problem into two smaller subproblems: finding the size of the left and right subtrees. Once we get the solution to those problems, we can combine them by adding them up plus one (for the root). Look at the following example with additional comments marking the steps:

```
public int size() {
    // Step 1: Division of the problem into size-of-left and size-of-right
    int leftSize = 0
    if (left != null) {
        leftSize = left.size();
    }
    int rightSize = 0;
    if (right != null) {
        rightSize = right.size();
    }
    // Step 2: Integration of both subproblems
    int result = 1 + leftSize + rightSize;
    return result;
}
```