Memoization

Table of contents

Introduction

To develop a recursive algorithm or solution, we first have to define the problem (and the recursive definition, often the hard part), and then implement it (often the easy part). This is called the **top-down** solution, since the recursion open up from the top until it reaches the base case(s) at the bottom.

Once we have a recursive algorithm, the next step is to see if there are **redundancies** in the computation — that is, if the same values are being computed multiple times. If so, we can benefit from **memoizing** the recursion. And in that case, we can create a memoized version and see what savings we get in terms of the running time.

Recursion has certain overhead cost, that may be minimized by transforming the memoized recursion into an **iterative** solution. And, finally, we see if there are further improvements that we can make to improve the time and space complexity.

The steps are as follows:

1. write down the recursion,
2. implement the recursive solution,
3. memoize it,
4. transform into an iterative solution, and finally
5. make further improvements.

Back to Table of contents

Example using the Fibonacci sequence

To see this in action, let's take Fibonacci numbers as an example. Fortunately for us, the mathematicians have already defined the problem for us – the Fibonacci numbers are `‹ 0, 1, 1, 2, 3, 5, 8, 13, ... ›` (some define it without the leading `0`, which is ok too). Each number, except for the first two, is nothing but the sum of the previous two numbers. The first two are by definition `0` and `1`. These two facts give us the recursive definition to compute the n^{th} Fibonacci number for some $n \geq 0$. This process, by the way, can be quite complex (and exciting, as you'll see next semester in Algorithms), and that's why we've chosen this easy example.

Let's go through the 5 steps below.

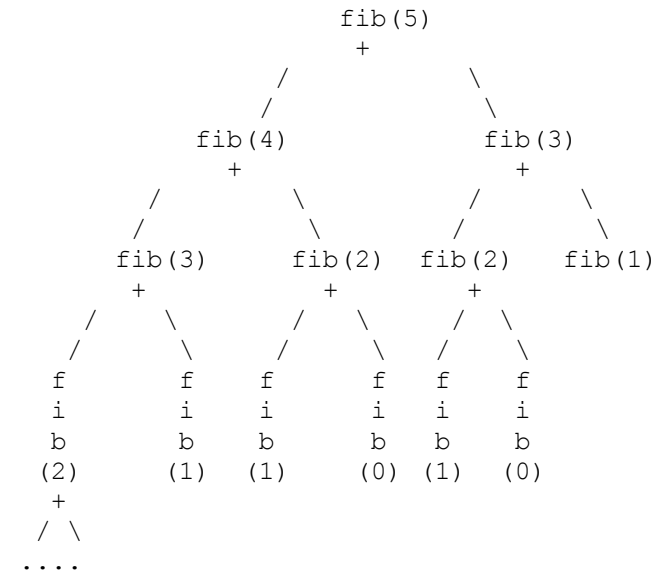
Step 1: Write or formulate the recursive definition of the n^{th} Fibonacci number (defined only for $n \geq 0$).

$$
fib(n) = \begin{cases} n & \text{if } n < 2 \\ fib(n-1) + fib(n-2) & n \geq 2 \end{cases}
$$

Step 2: Write the recursive implementation. This usually follows directly from the recursive definition.

```
/**
 * Returns the n'th Fibonacci number.
 * @return the n'th Fibonacci number.
 * @throws IllegalArgumentException if n < 0.
 */
static int fib(int n) {
    if (n < 2)                  // handle base conditions.
        return n;
    else                        // handle recursive cases.
        return fib(n - 1) + fib(n - 2);
}
```

Step 3: Memoize the recursion
Would this recursion benefit from memoization? Well, let's see by "unrolling" the recursion `fib(5)` a few levels:



Now you should notice something very interesting — we're computing the same fibonacci number quite a few times. We're computing `fib(2)` **3** times and `fib(3)` **2** times. Is there any

reason why we couldn't simply save the result after computing it the first time, and re-using it each time it's needed afterward? This is the primary motivation for memoization – to avoid re-computing overlapping subproblems. In this example, `fib(2)` and `fib(3)` are **overlapping subproblems** that occur independently in different contexts.

Memoization certainly looks like a good candidate for this particular recursion, so we'll go ahead and memoize it. Of course, the first question is how we save the results of these overlapping subproblems, that we want to re-use later on.

The basic idea is very simple — before computing the i^{th} fibonacci number, first check if that has already been solved; if so, just look up the answer and return; if not, compute it and save it before returning the value. We can modify our `fib` method accordingly (using some pseudocode for now). Since fibonacci is a function of `1` integer parameter, the easiest is to use a `1`-dimensional array or table with a capacity of `n + 1` (we need to store `fib(0) ... fib(n)`, which requires `n + 1` slots) to store the intermediate results.

What is the cost of memoizing a recursion? It's the space needed to store the intermediate results. Unless there are overlapping subproblems, memoizing a recursion will not buy you anything at all, and in fact, cost you more space for nothing!

Remember this — memoization trades space for time.

Let's try out our first memoized version. (`M_fib` below stands for "memoized fibonacci").

```
static int M_fib(int n) {
    // Assume that we have some "global" array (also called a table)
    // F with n+1 capacity. Why can F not be a local variable?
    if (n < 2)
        return n;
    else {
        // Compute (and save) if it's not already computed.
        if (F[n] is empty)                // <<<< NOTE PSEUDOCODE!
            F[n] = M_fib(n - 1) + M_fib(n - 2);

        // Now just return the computed (and saved) value.
        return F[n];
    }
}
```

This is all the we need to avoid redundant computations of overlapping subproblems. The first time `fib(3)` is called, it'll compute the value and save the answer in `F[3]`, and then subsequent calls would simply return `F[3]` without doing any work at all!

There are a few details we've left out at this point:

1. Where would we create this "table" to store the results?
2. How can we initialize each element of `F` to indicate that the value has not been computed/saved yet? (Note the "is empty" in the code above).

Let's take these one at a time.

1. The "fib" method is a function of a single parameter — n, so if we wanted to save the intermediate results, all we need is an array that goes from `0 ... n` (i.e., of `n + 1` capacity). For more complex problems, we'll need much more sophisticated containers (the parameter may not be simple integers, which can be used to index into an array for example). Again, we've picked an easy example, but the basic process is the same for any other problem.

   Since the local variables within a method are created afresh each time the method is called, `F` cannot be a local array. We can either use an instance variable within an object, or create an array in the caller of fib(4), and then pass the array to fib (in which case we'll have to modify fib to have another parameter of `type int[]`).

2. We need to use a sentinel which will indicate that the value has not been computed. Since it's array of `int`s, we can't use `null` (which is the sentinel used to indicate the absence of an object). However, we know that the factorial of any number is a non-negative integer, so we can use any negative number as the sentinel. Let's choose `-1`.

So, let's have an array `F` of `n+1` capacity that holds all the values of the intermediate results we need to compute the n^{th} Fibonacci number. We can have a **wrapper** method, and which creates this array or table, initializes the table and passes it onto `M_fib` as a parameter.

First, the wrapper method called `fib`, which basically sets up the table for `M_fib`, and calls it on the user's behalf.

```
static int fib(int n) {
    // Create and initialize the table.
    int[] F = new int[n + 1];
    for (int i = 0; i <= n; i++)
        F[i] = -1;                          // No solution saved in F[i] yet.

    // Now we can call M-Fib with this extra parameter "F".
    return M_fib(n, F);
}

static int M_fib(int n, int[] F) {
    // The table "F" is being passed as a parameter. The alternative is
    // to use a global variable.
    if (n < 2)
        return n;
    else {
        // Compute (and save) if it's not already computed.
        if (F[n] == -1)
            F[n] = M_fib(n - 1) + M_fib(n - 2);

        // Now just return the computed (and saved) value.
        return F[n];
    }
}
```

To compute the 5^{th} fibonacci number, we simply call `fib(5)`, which in turn calls `M_fib(5, F)` to compute and return the value.

Now that we have a memoized fibonacci, the next question is to see if we can improve the space overhead of memoization.

Step 4: Convert the recursion to iteration – the bottom-up solution.

To compute the 5^{th} fibonacci number, we wait for 4^{th} and 3^{th}, which in turn wait for 2^{nd}, and so on until the base cases of `n = 0` and `n = 1` return the values which move up the recursion stack. Other than the `n = 0` and `n = 1` base cases, the first value that is actually computed and saved is `n = 2`, and then `n = 3`, and then `n = 4` and finally `n = 5`. Then why not simply compute the solutions for `n = 2, 3, 4, 5` by iterating (using the base cases of course), and fill in the table from left to right? This is called the **bottom-up** solution since the recursion tree starts at the bottom (the base cases) and works its way up to the top of the tree (the initial call). The bottom-up technique is more popularly known as **dynamic programming**, a topic that we will spend quite a bit of time on next semester!

```
static int fib(int n) {
    int[] F = new int[n + 1];   // The table to store computed values.

    F[0] = 0;                   // The base case for n = 0.
    F[1] = 1;                   // The base case for n = 1.

    for (int i = 2; i <= n; i++) {
        F[i] = F[i - 1] + F[i - 2];
    }

    return F[n];
}
```

You should convince yourself that this is indeed a solution to the problem, only using iteration instead of memoized recursion. Also, that it solves each subproblem (e.g., `fib(3)` and `fib(2)`) **exactly once**, and **re-uses the saved answer**.

This one avoids the overhead of recursion by using iteration, so tends to run much faster.

Can we improve this any further?

Step 5: improving the space-requirement in the bottom-up version

The n^{th} Fibonacci number depends only on the `(n - 1)` and `(n - 2)` Fibonacci numbers. However, we are storing ALL the intermediate results from `2 ... n - 1` Fibonacci numbers before computing the n^{th} one. What if we simply store the last two? In that case, instead of having a `n + 1` array, we need just **two** instance variables (or an array with `2` elements). Here's what the answer may look like.

```
static int fib(int n) {
```

```
    int f_2 = 0;                // The n-2 value.
    int f_1 = 1;                // The n-1 value.

    // The result - initialize to n (why? So that it works when n is 0 or 1).
    int f = n;

    for (int i = 2; i <= n; i++) {
        f = f_1 + f_2;
        // Now update the f_1 and f_2 for the next iteration (if any).
        f_2 = f_1;
        f_1 = f;
    }
    return f;
}
```

Back to Table of contents

Summary

We have just gone through the 5 steps of taking a recursive problem, writing a recursive solution, memoizing that recursion, transforming that into an iterative solution and then squeezing some more performance out of it. There are many questions left to be asked of course:

- Given a problem, how do we develop a recursive solution for it?
- What is the performance of the recursive solution?
- Given a recursive solution, would it benefit from memoization? In other words, are there overlapping subproblems?
- How would you transform a recursive solution into an iterative one?

Back to Table of contents