CS 344: Design and Analysis of Computer Algorithms **Rutgers: Spring 2022**

Lecture 9

February 15, 2022

Instructor: Sepehr Assadi

1 Introduction to Dynamic Programming

We will introduce *dynamic programming* in this lecture, an extremely useful and popular algorithmic technique for **speeding up computation of recursive algorithms**.

Our running example throughout this introduction is computing *Fibonacci numbers*, the sequence of numbers: $1, 1, 2, 3, 5, 8, 13, 21, 34, \cdots$ (do you see the pattern?) The pattern in this sequence is that (except for the first two numbers), each number is obtained by summing the two numbers before it in the sequence. More formally, we can define the n-th Fibonacci number, denoted by F_n, as follows:

$$F_n = \begin{cases} 1 & \text{if } n = 1 \text{ or } n = 2 \\ F_{n-1} + F_{n-2} & \text{otherwise.} \end{cases}$$

A Recursive Algorithm

Considering Fibonacci numbers are defined recursively, we can perhaps compute F_n easily also using the same recursive formula:

`RecFibo`(n) :

1. If $n = 1$ or $n = 2$, return 1.
2. Otherwise, return `RecFibo`$(n-1)$ + `RecFibo`$(n-2)$.

The correctness of this algorithm is very easy to see. But what can be said about its running time?

Let us write a recurrence for the runtime. We use $T(n)$ to denote the runtime[1] of `RecFibo`(n). Hence, $T(n) = T(n-1) + T(n-2) + O(1)$. Note that we can freely use equality here instead of inequality because again $T(n-1)$ and $T(n-2)$ is precisely the runtime of `RecFibo`$(n-1)$ and `RecFibo`$(n-2)$ not some upper bound that may not be equality in general.

While we generally did not solve recurrences like the above one, one thing should be clear from this recurrence: $T(n)$ behaves very similarly to F_n itself and in fact $T(n) \geq F_n$. You may have seen in previous courses that $F_n \approx \phi^n$ where $\phi = (\sqrt{5}+1)/2 \approx 1.61\cdots$ is the so-called golden ratio. For our purpose, it suffices to say that $T(n) = \Omega(c^n)$ for some constant $c > 1$.

As we know, exponential functions grow very fast and thus the runtime of the above algorithm becomes *super slow* even for very small values of n. But why this recursive algorithm, which sounds so natural for this problem, runs this slow? The answer to this question turns out to be simple and has to do with the fact that the above algorithm is "forgetful" – we can see this by examining the chain of recursive calls in the algorithm in Figure 1 for the simple case when $n = 6$.

[1] We omit the word 'worst case' because there is only one possible input for n not a whole set (unlike sorting arrays of length n) that we may need to take a worst case over.

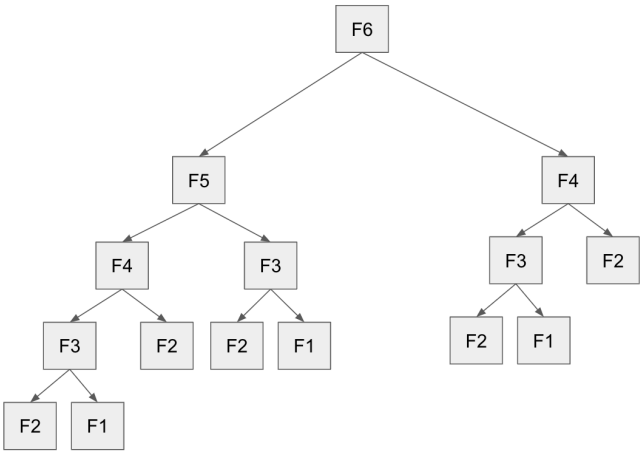

Figure 1: The recursion tree for `RecFibo`(6), i.e., for $n = 6$.

The problem is that the algorithm again and again computes the same value of `RecFibo`(i) for different choices of i as can be seen in Figure 1.

This is indeed problematic: why do we have to recompute the same function again and again? The answer is well we do not have to! We just have to be slightly more careful.

Memoization

We can speed up the computation of this recursive algorithm significantly by simply storing the computed answers in a table and make sure we do not recompute them again and again by consulting this table. Specifically, we can do as follows. We first pick an array $F[1 : n]$ and initialize it so that all its entries are written as 'undefined'. We then run the following algorithm:

`MemFibo`(n) :

1. If $F[n] \neq$ 'undefined', return $F[n]$.
2. If $n = 1$ or $n = 2$, let $F[n] = 1$.
3. Otherwise, let $F[n] =$ `MemFibo`$(n-1) +$ `MemFibo`$(n-2)$.
4. Return $F[n]$.

The logic of this algorithm is exactly the same as the logic of `RecFibo` and thus again there is nothing new to prove about the correctness of this algorithm. But what can we say about the runtime of this algorithm? Let us first check the recursion tree for this algorithm in Figure 2. In this new recursion tree, we are not going to recompute an already computed solution and thus for every value of i, there is only one entry for `MemFibo`(i) in this tree.

Analyzing the runtime of `MemFibo`(n) in general is also easy. There are in total n *subproblems* (or recursive calls) when computing the function `MemFibo`(n) (they are `MemFibo`(i) for $1 \leq i \leq n$). Each recursive call takes $O(1)$ time *on its own* ignoring the time needed that takes inside each inner recursion. Since we only compute each recursive call once, the total runtime is then $O(n)$. If you noticed, this is the same exact principle we used before when writing the recursion tree for each recurrence and computed the value of the tree – simply compute the total time spent on each call to the function and add them up to obtain the final runtime.

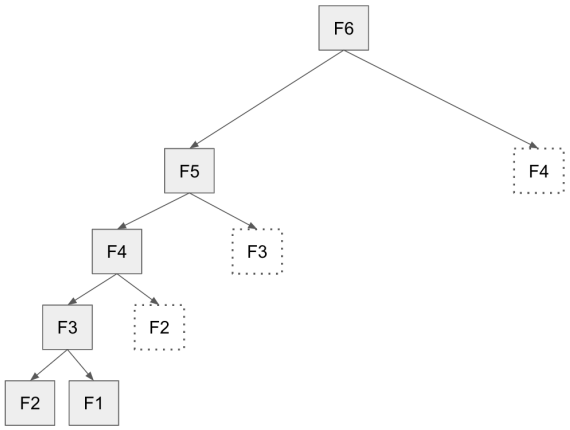

Figure 2: The recursion tree for `MemFibo`(6), i.e., for $n = 6$.

From an algorithmic point of view, we are already done! We obtained a fast algorithm for computing F_n as we wanted to. However, it is sometimes worth stating this algorithm in a different manner without involving the recursive calls. In other words, instead of relying on recursions to do the job for us when computing `MemFibo`(n), we can explicitly fill out the needed values in the table $F[n]$. The main benefit of this approach is that it makes understanding the runtime of the algorithm crystal clear (it also has some benefit in terms of reducing the runtime by some constant factor but we can ignore that for the purpose of this course).

Bottom-Up Dynamic Programming

We now replace the recursive calls in `MemFibo` with an *explicit* way of filling the table F it uses directly in a bottom-up fashion, i.e., by starting from the bottom of recursion tree (the base cases) and going to top (the final value we are interested in). The algorithm is as follows:

`DynamicFibo`(n):

1. Let $F[1 : n]$ be an empty array.
2. Let $F[1] = 1$ and $F[2] = 1$.
3. For $i = 3$ to n: let $F[i] = F[i-1] + F[i-2]$.
4. Return $F[n]$.

Again, you can verify (and prove) that the values in table F is exactly the same in both `DynamicFibo` and `MemFibo`. And again, this will imply the correctness of this new algorithm. As for the runtime? Well the algorithm consists of a single for-loop iteration with $\Theta(n)$ iterations, each involving $O(1)$ time, thus the total runtime is $O(n)$.

Concluding Remarks

Congratulations! We finished our first dynamic programming algorithm. All other dynamic programming algorithms we will see in this course follow the same exact pattern:

- Write a *recursive* formula for the problem we want to solve.
- Compute the recursive formula using a recursive function.

- Use a table to store the value of the recursive function (on each subproblem) whenever we compute it once. After this, *never* recompute that value and simply return the value from the stored table.
- To perform the step above, we can either use the memoization technique or the iterative approach by explicitly filling out this table. For the purpose of this course, **there is absolutely no difference between these two approaches**: both are considered valid dynamic programming solutions (even though for traditional reasons they are named differently) and both will result in the *asymptotically* same worst case runtime. The decision for using each of the two is ultimately up to you (and may vary from problem to problem) as each one has their own benefits and drawbacks. Generally speaking, memoization seems to be the more straightforward approach (at least to your Instructor) but is harder for analyzing the runtime, while bottom-up dynamic programming requires you to think carefully how the values of the function are obtained recursively from smaller subproblems (although this is generally not a hard task) but then the runtime analysis becomes very easy and straightforward.

  So to sum up this part, you can *always* use either the memoization or the bottom-up dynamic programming (the iterative approach for filling the table), whenever you decide, or asked, to write a dynamic programming algorithm for a problem.

As we saw so far, dynamic programming is simply a way of *speeding up* recursive algorithms by not recomputing the same value again and again. This avoiding re-computation part is rather mechanical and once you see a couple of examples, you will learn how to do this easily. The main part of the approach is however in designing a suitable *recursive formula/function* for the problem at hand and prove its correctness (even though for Fibonacci numbers this part was rather trivial). This is the part which we will focus on in the remainder of this lecture.