Dynamic Programming: Smart Recursion[1]

This lecture marks the start of the module on *Dynamic Programming*. Dynamic Programming is an essential tool in the algorithm designer's repertoire. It solves problems that at first glance seem terribly difficult to solve. Dynamic Programming, unfortunately, is also something that some students have trouble understanding and using. I know, because I did. But it *is* a very simple idea, and once one gets used to it[2], kind of hard to muck up. We will begin DPs in earnest from next class, but today we explore the main idea behind dynamic programming: *recursing with memory* aka *bottom-up recursion* aka *smart recursion*.

1 Fibonacci Numbers: Recursion with Memory

Let us recall Fibonacci numbers.

$$F_1 = 1, F_2 = 1, \quad \forall n > 2, F_n = F_{n-1} + F_{n-2} \tag{1}$$

Here is a simple computational problem.

FIBONACCI
Input: A number n.
Output: The nth Fibonacci number, F_n.
Size: n.

Remark: *"Now hold on," I hear you cry, "we are back to handling numbers as input. Then if the input is n, shouldn't we consider the number of bits in n, that is $\log n$, as the size and not n?". Perfectly valid question. The answer lies in the output. The exercise below shows that the number of bits required to write F_n is $\Theta(n)$. This is the reason we are going to take n as the size.*

*But I **am** going to cheat a bit below – when I add two numbers, I am still going to* wrongly *assume they are $O(1)$ time operations. It is wrong because the number of bits in the number grows with n. A quiz will explore this more.*

✍

Exercise: *Prove that for any n, we have $2^{n/2} \leq F_n \leq 2^n$. Use induction.*

The definition (1) of Fibonacci numbers implies the following recursive algorithm.

```
procedure NaiveFib($n$):
    if $n \in \{1, 2\}$ then:
        return 1
    else:
        return NaiveFib($n-1$) + NaiveFib($n-2$)
```

[1]*Lecture notes by Deeparnab Chakrabarty. Last modified : 22nd Jan, 2024*
These have not gone through scrutiny and may contain errors. If you find any, or have any other comments, please email me at deeparnab@dartmouth.edu. Highly appreciated!

[2]Solving around 20 problems should do it

The above is a ***disastrous*** thing to do. However, for all the problems we will encounter for Dynamic Programming, if you have obtained a disastrous version of the algorithm as above, you are actually probably close to victory. That is, if we have been able to express your algorithm as a recursive algorithm like the one above, then the fix for the above algorithm will probably work for our recursive algorithm as well (and if it doesn't, then dynamic programming probably won't help). More precisely, we are going to see below how to smartly implement the recursion in (1). And this trick, almost always, will also *mechanically* work for the recursion you have obtained.

But before going further, why is this disastrous? What is the running time of NaiveFib? As warned before, I am assuming (wrongly) that the addition in Line 5 takes $O(1)$ time. Alternately, you can just think of $T(n)$ as the number of additions required to calculate the Fibonacci numbers. We see that the recurrence which governs the running time is

$$T(n) \leq T(n-1) + T(n-2) + O(1)$$

Even if we ignore the $O(1)$ in the RHS above, we see that the recurrence governing $T(n)$ is eerily similar to (1). And that is not good news – it says $T(n)$ can be as large as F_n which we know from the exercise above is $\geq 2^{n/2}$. Yikes!

First we observe *why* the above algorithm is so time consuming. To see this, let us draw out the "recursion tree" when NaiveFib is indeed called on a certain value. Figure 1 shows the case when called with $n = 6$. We see that many problems are solved *again and again*, recursively. There are two ways to fix this.

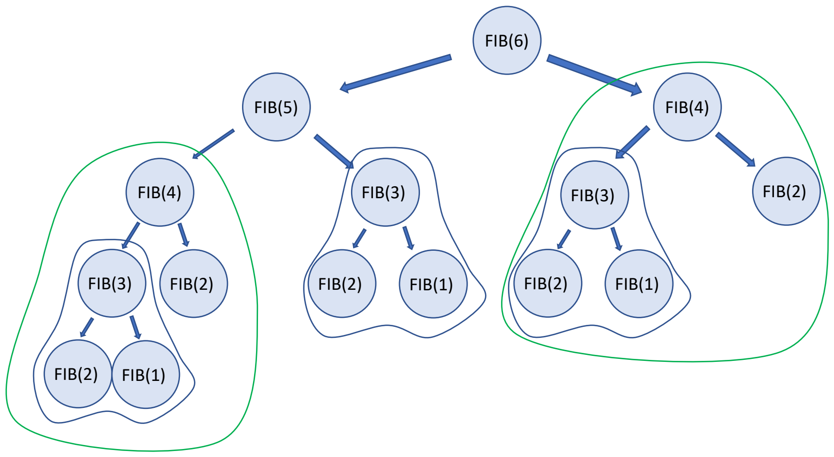

Figure 1: Call of NaiveFib on $n = 6$. The encircled parts of the tree show repeated computation.

Both involve the same principle. The idea is

If we remember the solutions to smaller subproblem',
Then there's no need to take the trouble to re-solve them.

Hey, that rhymed!

Implementation via Memoization.

```
1: Implement a “look-up table” $T$.
2: Define $T[1] \leftarrow 1$; $T[2] \leftarrow 1$.
3: procedure MEMOFIB($n$):
4:     if $T[n]$ is defined then:
5:         return $T[n]$
6:     else:
7:         $t \leftarrow$ MEMOFIB($n-1$) + MEMOFIB($n-2$)
8:         Set $T[n] \leftarrow t$
9:         return $t$
```

The *memoization* approach gets to the heart of the problem described above. It stores all previously computer Fibonacci numbers in a look-up table (dictionary) T. Different programming languages have different implementations of the look up table; we assume (perhaps wrongly) look ups take $O(1)$ time. Therefore, the running time of the above psuedocode is $O(n)$.

Bottom-Up Implementation: The Table method. This is the method which makes computation more *explicit* and will be what we will use throughout the course. The method described below seems a little wasteful implementation of the memoization idea; it has the benefit of (hopefully) being clearer.

We first observe that the *solution* to the problem FIB(n) depends on the *solutions* to the problems FIB($n-1$) and FIB($n-2$). Thus, if we stored these solutions in an array (sounds very much like a look-up table) $F[1:n]$, then the following picture shows the dependency of the various entries.

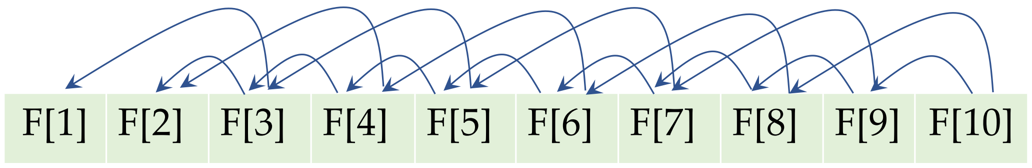

Figure 2: Dependency of the Fibonacci Array

Note two things: (a) The “graph” has no cycles. Otherwise, there will be a cyclic dependency and it doesn’t make sense. (b) There are “sinks” in this graph: points from which no arrows come out. These are the *base cases*; $F[1]$ and $F[2]$ don’t need any computation; they are both 1. Given this graph above, the algorithm to obtain the nth position $F[n]$ becomes clear: traverse it from the “sink” to $F[n]$. Here’s how.

```
1: procedure FIB($n$):
2:     Allocate space $F[1:n]$
3:     Set $F[1] \leftarrow 1$; $F[2] \leftarrow 1$.
4:     for $i = 3$ to $n$ do:
5:         $F[i] \leftarrow F[i-1] + F[i-2]$
6:         ▷ Note: the computation of $F[i]$ requires precisely the $F[j]$'s the $F[i]$ points to
7:     return $F[n]$
8:     ▷ Note: we have actually found all the $F[j]$ for $j \leq n$. The Table method often does more work than needed.
```

Remark: *Few comments. The table method solves a lot more; for instance, the table method will return all the F_j's for $1 \leq j \leq n$. The memoization method only solves what is needed. On the other hand, memoization includes recursion and implementing a look-up table. Although we have considered these to be $O(1)$-operations, in practice, depending on the system in which it is implemented, the running times can differ. All in all, the table method is more clear cut and easier to analyze. However, when faced in the "real world" (or your coding projects) do not forget memoization. In any case, if you do use memoization or bottom-up, the important thing to remember is to be correct.*

✍

Exercise: *Implement both the above methods, via memoization and the bottom up table method, in Python 3, and use the* `time()` *routine to see which is faster.*

2 Binomial Coefficients via Dynamic Programming

Let us begin with a *warm up* dynamic programming problem. But, let me use it as a sandbox to explain the DP process. The problem is the following: you take input two non-negative integers n and $0 \leq k \leq n$. Your goal is to output the *number* of distinct ways k things can be selected from n things. That is, the number of subsets of $\{1, 2, \ldots, n\}$ of size exactly k. If you remember your discrete math, the answer is $\binom{n}{k}$ whose "formula" is $\frac{n!}{k!(n-k)!}$. And so, you could use a program which evaluates the factorial and return this "quickly".

Exercise: *Hmm, how quickly? Even if we assume multiplying two numbers takes $O(1)$ time, how long will this way take?*

Let's solve this differently. First, let's understand that this answer depends on both n and k. So, let's call this answer $F(n,k)$ where F is a function that takes the two parameters At some point as you play with this question, one needs to hit upon the observation/idea, that $F(n,k)$ can be **recursively** evaluated. More precisely, $F(n,k)$ can be written as a function of some $F(m,j)$'s for "smaller" m's and j's. This step is, according to my opinion, the **most non-trivial** part in designing DP algorithms. They constitute "defining" the precise function you are looking for *and* figuring out the "recursive structure" governing this function. These two things go hand-in-hand.

Coming back to picking k things from n things. How do we figure this recursive substructure out? Here's one way to think about this. Imagine these n numbers lined up in a row. You need to pick k things from this row. Let's look at right-to-left[3]. Consider the number n. There are **two** choices here: either we pick it, or we don't pick it.

- Case 1: If we **don't** pick it, then all these collections of sets of size k must come from $\{1, 2, \ldots, n-1\}$. But wait...this is a *smaller subproblem*....this answer is precisely $F(n-1, k)$. And since this is smaller, I must just believe I have it already (perhaps by calling it recursively). We are **done** in this case.

- Case 2: Maybe we do pick item n in our set. Well, then the *rest* of the items are $(k-1)$ choices to be made...but all from the smaller set $\{1, 2, \ldots, n-1\}$. Whoa! That's another *smaller subproblem*...this answer is precisely $F(n-1, k-1)$. We are **done** in this case as well.

And if we are done in all the cases, then we are just **done**. We get the recurrence[4] (drumroll):

$$F(n, k) = F(n-1, k) + F(n-1, k-1) \tag{2}$$

Appreciate what we have done. We have piggybacked the problem of solving $F(n, k)$ onto the backs of $F(n-1, k)$ and $F(n-1, k-1)$. Next, we realize there was nothing special about "n" and "k". This would be true for any m and j. That is, $F(m, j) = F(m-1, j) + F(m-1, j-1)$....except when the numbers became too small. And so we come to the issue of **base cases**: small numbers for which we already know the answer.

How do we figure out the base cases? Well, let's look at the limits of m and j in $F(m, j)$: we see that (a) $0 \le m$,(b) $j \le m$, and (c) $0 \le j$. In the right hand side of (2) (with n replaced by m, and k replaced by j), we see that (a) is in danger if $m = 0, j = 0$, (b) is in danger if $m = j$, and (c) is in danger if $j = 0$.. Do we know the answers in these cases? Yes we do.

$$F(m, j) = 1 \quad \text{when } m = 0, j = 0 \qquad F(m, m) = 1 \ \text{ for all } m \quad F(m, 0) = 1 \quad \text{for all } m \tag{Base Cases}$$

Why was that: it's because, the number of subsets of size 0 for the empty set, or any set, is 1; the number of subsets of size m of $\{1, 2, \ldots, m\}$ is also 1.

The hard part is over. If we were to now succinctly write the solution, then we would proceed as follows.

a. **Definition.** For $0 \le m \le n$ and $0 \le j \le m$, *define* $F(m, j)$ to be the number of way j things can be chosen from m things. We are interested in $F(n, k)$.

b. **Base Case.** This is (Base Cases)

c. **Recurrence.** This is (2) written for general m and j. Rewriting it:

$$\text{For all } m \ge 1 \text{ and } 1 \le j \le m-1 \quad F(m, j) = F(m-1, j) + F(m-1, j-1) \tag{Recurrence}$$

At this point the **bad, naive** recursive algorithm should hopefully be clear to you.

[3]For reasons that will be clear in hindsight, it's always good to look right-to-left. Think of it this way: from n problem to smaller problem...you need to make n smaller...n is rightmost.

[4]This identity is often called Pascal's identity

```
procedure NaiveBinom($n, k$):
    if $k = 0$ or $n = k$ then:
        return 1.
    else:
        return NaiveBinom($n - 1, k$) + NaiveBinom($n - 1, k - 1$)
```

Again, if you write the running time of it using a recurrence inequality, you will see that the time taken is at least the value of the binomial coefficient. How large can they be? Do you remember? Can be pretty large. Rule of thumb $\binom{n}{n/2} \approx \frac{2^n}{\sqrt{n}}$. So no way this will compute for $n = 100$. But, we can fix this *just* like we fixed NaiveFib. There is one difference — the function above takes *two* parameters (n and k). And thus, our table needs to be *two dimensional*. Do you want to attempt this before reading on? It would be very instructive.

e. **Pseudocode.**

```
procedure Binom($n, k$):
    Allocate space $B[0 : n, 0 : k]$.
    Set $B[m, 0] = 1$ for all $0 \le m \le n$ ▷ Base case of $\binom{m}{0} = 1$ for all $m$.
    Set $B[m, m] = 1$ for all $0 \le m \le n$.▷ Base Case of $\binom{m}{m} = 1$ for all $m$.
    for $m = 1$ to $n$ do:
        for $j = 1$ to $m - 1$ do:
            $B[m, j] = B[m - 1, j] + B[m - 1, j - 1]$.
    ▷ Note: the computation of $B[m, j]$ requires $B[m - 1, j]$ and $B[m - 1, j - 1]$ and they have been computed before.
    return $B[n, k]$
    ▷ Note: we have actually found all the $B[m, j]$ for $m \le n, j \le k$. The Table method often does more work than needed.
```

f. **Running Time.** The running time and space of the above is $O(nk)$ time. Way better than the recursive algorithm and we can easily go for $n = 100$ and $k = 50$ (why don't you code it and see?)

Exercise: *How does it compare with the formula method?*

Let us end these notes with a *formal proof* of (Recurrence). Our reasoning behind that recurrence (which we mentioned at the top of previous page) is a little informal, and it is easy to informally argue wrong things (in a later lecture, I may try to pull a fast one). What I write below should also be pretty mechanical if your informal argument is clear. So here goes (ideally, this should come right after the recurrence, and that explains the enumeration).

d. **Formal Proof.** Let $\mathcal{F}$ denote the set of subsets of $\{1, 2, \ldots, m\}$ which are of size j. That is,

$$\mathcal{F}_{m,j} := \{S \subseteq \{1, 2, \ldots, m\} \ : \ |S| = j\}$$

By definition, $F(m, j) = |\mathcal{F}_{m,j}|$.

We partition $\mathcal{F}_{m,j}$ into $\mathcal{F}_0$ and $\mathcal{F}_1$ as follows (this formalizes the notion of picking m or not picking m).

$\mathcal{F}_0 := \{S \subseteq \{1,2,\ldots,m\} \quad : \quad |S| = j, \quad m \notin S\}$ and $\mathcal{F}_0 := \{S \subseteq \{1,2,\ldots,m\} \quad : \quad |S| = j, \quad m \in S\}$

Observe: $\mathcal{F}_0 \cap \mathcal{F}_1 = \emptyset$ (any element of $\mathcal{F}_0$ doesn't contain m and any element of $\mathcal{F}_1$ does contain m), and $\mathcal{F}(m,j) = \mathcal{F}_0 \cup \mathcal{F}_1$ (any subset of $\{1,2,\ldots,m\}$ either contains m or doesn't). So,

$$F(m,j) = |\mathcal{F}_0| + |\mathcal{F}_1|$$

Next observe two *bijections*. One is a simple one between $\mathcal{F}_0$ and $\mathcal{F}(m-1,j)$. Indeed, any set in $\mathcal{F}_0$ is **precisely** a subset of $\{1,2,\ldots,m-1\}$ and is of $|S| = j$. Thus, it belongs in $\mathcal{F}(m-1,j)$. The other is only a little more nontrivial. We claim there is a bijection between $\mathcal{F}_1$ and $\mathcal{F}(m-1,j-1)$: map any subset $S \in \mathcal{F}_1$ to $T := S \setminus \{m\}$ in $\mathcal{F}(m-1,j-1)$; T is indeed a subset of the first $m-1$ elements and is of size j. Similarly, given any $T \in \mathcal{F}(m-1,j-1)$ obtain an element in $\mathcal{F}_1$ by adding m to it, that is, $S := T \cup m$.

In sum, we get

$$|\mathcal{F}_0| = F(m-1,j) \quad \text{and} \quad |\mathcal{F}_1| = F(m-1,j-1)$$

which, when substituted in the previous equation gives us (Recurrence). This completes the formal proof of our informal argument.

Moving forward, we will solve many dynamic programming problems. For each of them, there will be these things in common.

a. **Definition.** The "function" that is at the core of the problem. This should be **precisely** defined, and it should be made clear which parameter we are interested in. We cannot overstate how important it is to be precise here.

b. **Base Cases.** The "small" parameters at which the function value is known.

c. **Recurrence.** How the function at a certain parameter can be derived from the function evaluated at "smaller" parameter.

d. **Reasoning/Formal Proof.** A reason as to why the above is true. A formal proof is appreciated, but it can be tedious at time. Nevertheless, it is the only foolproof method to be correct. (Indeed, in untimed assignments, I will need you to write this; in a timed exam a precise enough reasoning may suffice).

The above are the absolute beating heart of a dynamic programming problem. Once you have gotten the above, the battle is 80% won. But they all go hand-in-hand, and I think that is why getting comfortable with DP's takes time. Anyway, once you have this, one needs to implement the above "smartly".

e. **Implementation Pseudocode/Recovery Pseudocode.** This is the bottom-up representation of the above psuedocode. All you need to be careful about is when you are assigning value to a certain "cell", the other cells you are using are already filled up. Drawing a picture showing which cells depend on which may help.

   Sometimes, we would be a little smart with what we define F to be and then an extra "recovery pseudocode" may be required. For now, ignore this.

f. **Running Time.** You must tell what the running time of your code is.