CS 310

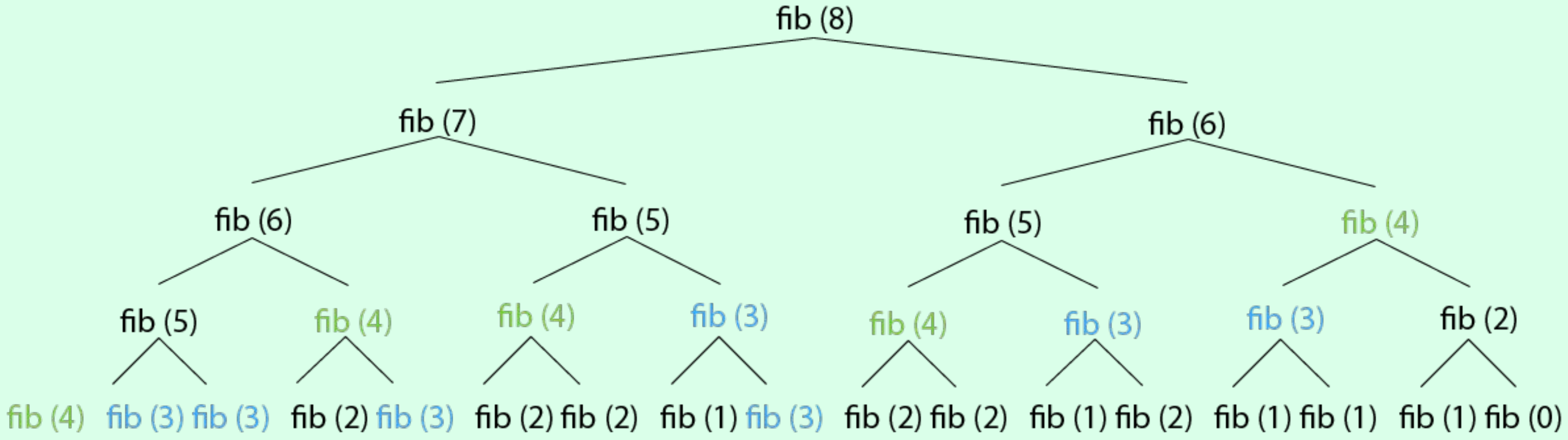
ONE CHEESY TORTILLA CHIP...
ONE CHEESY TORTILLA CHIP...
TWO CHEESY TORTILLA CHIPS...
THREE CHEESY TORTILLA CHIPS...
FIVE CHEESY TORTILLA CHIPS...
EIGHT CHEESY TORTILLA CHIPS...
MATH GEEKS SHOULDN'T BE ALLOWED ANYWHERE NEAR CERTAIN FOODS.
WHAT'S WRONG WITH FIBONACHOS?
THERE'RE ONLY 12 LEFT...NOW WHAT?
www.foxtrot.com
©2009 Bill Amend / Dist. by Universal Press Syndicate
FOXTROT © 2009 BILL AMEND. REPRINTED WITH PERMISSION OF UNIVERSAL UCLICK. ALL RIGHTS RESERVED.

Memoization - Trade space for time

```
/* initialization */
for j= 1 to n
   Memo[j] = -1
Memo[0] = 1
Memo[1] = 1

fib(n){
   if (Memo[n] < 0)
      Memo[n] = fib(n-1) + fib (n-2)
   return Memo[n];
}
```

Fibonacci Series


fib (8)
fib (7)
fib (6)
fib (6)
fib (5)
fib (5)
fib (4)
fib (5)
fib (4)
fib (4)
fib (3)
fib (4)
fib (3)
fib (3)
fib (2)
fib (4) fib (3) fib (3) fib (2) fib (3) fib (2) fib (2) fib (1) fib (3) fib (2) fib (2) fib (1) fib (2) fib (1) fib (1) fib (1) fib (0)

Memoization - Trade space for time

```
/* initialization */
for j= 1 to n
   Memo[j] = -1
Memo[0] = 1
Memo[1] = 1

fib(n){
   if (Memo[n] < 0)
      Memo[n] = fib(n-1) + fib (n-2)
   return Memo[n];
}
```

Running time: O(n)

Memoization - Trade space for time

```
/* initialization */
for j= 1 to n
   Memo[j] = -1
Memo[0] = 1
Memo[1] = 1

fib(n){
   if (Memo[n] < 0)
      Memo[n] = fib(n-1) + fib (n-2)
   return Memo[n];
}
```

Running time:
*Number of entries in the memo * time to fill each entry*

Memoization - Trade space for time

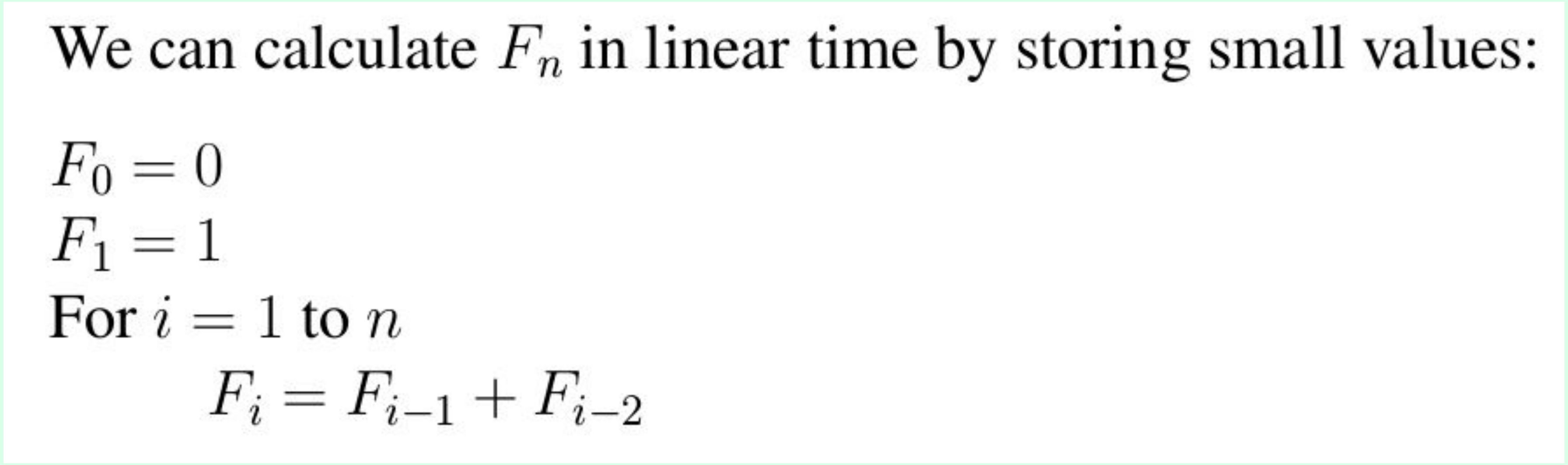
```
/* initialization */
for j= 1 to n
   Memo[j] = -1
Memo[0] = 1
Memo[1] = 1

fib(n){
   if (Memo[n] < 0)
      Memo[n] = fib(n-1) + fib (n-2)
   return Memo[n];
}
```

Running time: O(n)

*Number of entries in the memo * time to fill each entry*

Aside: Fibonacci iterative solution - Bottom up



We can calculate F_n in linear time by storing small values:

```
$F_0 = 0$
$F_1 = 1$
For $i = 1$ to $n$
        $F_i = F_{i-1} + F_{i-2}$
```

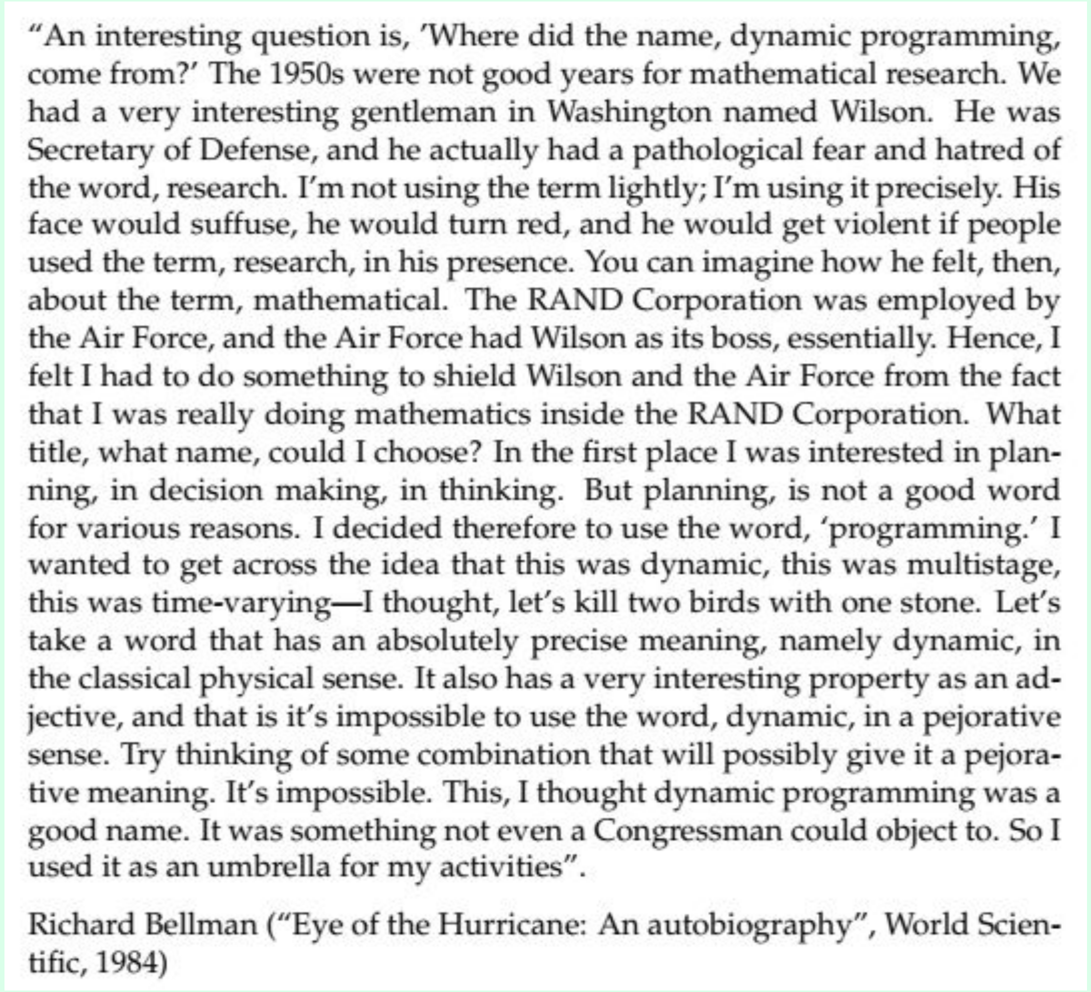
Running time: O(n)

Aside (optional)

Faster algorithms for computing a ***single term*** in the series (e.g. *Fib*(*n*))

- https://www.nayuki.io/page/fast-fibonacci-algorithms

Dynamic Programming



"An interesting question is, 'Where did the name, dynamic programming, come from?' The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word, research. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term, research, in his presence. You can imagine how he felt, then, about the term, mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word, 'programming.' I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying—I thought, let's kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word, dynamic, in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. This, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities".

Richard Bellman ("Eye of the Hurricane: An autobiography", World Scientific, 1984)

Dynamic Programming

Powerful algorithmic technique that has a **very broad applicability.**

Dynamic Programming

Dynamic programming applies to optimization problems that have the following two characteristics:

- optimal substructure and
- overlapping subproblems

Optimal Substructure

- A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to subproblems.

Dynamic programming is a technique for efficiently computing recurrences by storing partial results.

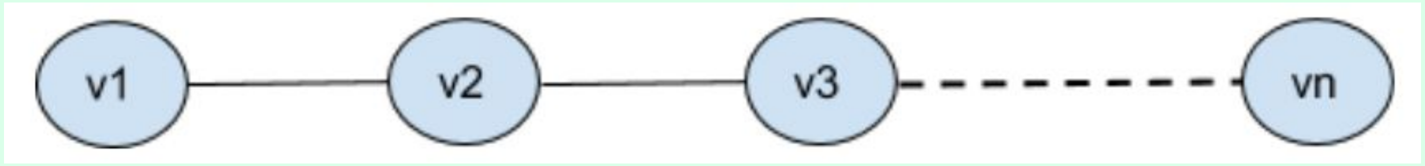
Dynamic programming

We need to specify

- An **ordering** on the subproblems, and
- A recurrence relation that shows how to solve a subproblem given the answers to “smaller” subproblems that appear earlier in the ordering.

An example

Undirected graph G(V, E) where the vertices are connected in a **chain** as shown below.



The vertices represent chemicals and the edges between them represent interactions between pair of chemicals. **Each of the chemicals have a price (price$_i$) in rupees.** You have to pack a subset of chemicals in **one** box such that the **total price is maximized**. Chemicals that interact with each other cannot be placed together in the box.

Basis: `f(0)=0, f(1)=`$\texttt{price}_1$

Recurrence: `f(n) = max(` $\texttt{price}_n$`+f(n-2), f(n-1) )`

Running time?

Determine running time through:

Number of entries in the memo * time to fill each entry

Dynamic programming

- There is an **ordering** on the subproblems, and
- A recurrence relation that shows how to solve a subproblem given the answers to “smaller” subproblems that appear earlier in the ordering.
- In dynamic programming the **DAG** is implicit. Its nodes are the subproblems we define, and its edges are the dependencies between the subproblems.

Running time?

Determine running time through DAG

Running time?

There are n subproblems and constant time to compute each subproblem.

Time complexity: O(n)