Dynamic Programming (DP)

Using recursive solution properties to trade compute time with memory

Some thoughts about DP

“Once you understand it, dynamic programming is probably the easiest algorithm design to apply in practice. […] until you understand dynamic programming, it seems like magic”, S.S. Skiena

How to solve an optimisation problem ?

Core idea that leads to DP

Strategy 1 : Exhaustive search (enumeration)

- Guarantied to find the best solution

→ Global optimality

- Really slow : we have to enumerate all possible combinations

Strategy 2 : Greedy algorithm (heuristic)

- Based on local optimality with heuristics:

"take the best local decision at each step"

→ No guaranty for global optimality

- Usually efficient

What if we could have everything at once? -> idea behind DP

What is Dynamic Programming ?

Both:

- **Mathematical Optimization** Method
- Algorithmic **Paradigm**

What it means:

- <u>Link to math:</u> Kind of similar to **Induction Proof** , with the difference that the number of steps to perform is finite
- <u>Principle:</u> solve complicated problems by breaking them down into simpler sub-problems in a recursive manner.

Informatics vision: Divide & Conquer X Memoization

Divide & Conquer	• Break down the problem into smaller problems • Solve the subproblems • Combine the results
Memoization	• Identify the redundant subproblems to solve them only once

A (really) quick reminder on Divide & Conquer

Question: is 9 in this sorted list? 1 2 5 7 8 9 10 12

→ **Naive search:** look at every element from the start, O(n)

→ **Binary search:** split the search space in half at each iteration, O(log(n))

1 2 5 7 8 9 10 12

1 2 5 7 **8 9 10 12**

1 2 5 7 **8 9** 10 12

1 2 5 7 8 **9** 10 12 Found it !

How to choose the subproblems?

- The idea is to find how to recursively reach the easy problems, without losing optimality
- We want the result of the subproblems to be useful in order to compute the result of the greater one we're struggling with

Bellman's principle of optimality: if subproblems' result are optimal then their combination will be too

Bellman equations

Bellman equations are the rules to go from one subproblem to another.

- They need to go with Bellman’s principle of optimality

Try smally decreasing any problem parameter (number of element, size of a line, …)

The subproblems have to strictly decrease in size, leaning toward the base case

→ There cannot be any cycle !

- Don’t forget the base case, The recursion (induction) has to stop at some point !

DP - Why such a “bad” name ?

The term DP was invented by Richard Bellman in the 1940s - 1950s.

- Originally, DP describes problems related to dynamic processes where finding the optimal solution can be done by taking decision one after the other.

Bellman was working on optimization problems in a setting where funding often required work to **sound mathematically sophisticated yet obscure** to avoid interference from military bureaucracy.

How Bellman coined the term “Dynamic Programming”

“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". [...] What title, what name, could I choose? [...] Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also [...] 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. Thus, 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

Alternative names

- **Memoization-Based Programming:** Highlights the reuse of intermediate results to avoid redundant computation.
- **Subproblem Optimization Programming:** Emphasizes breaking problems into smaller subproblems and solving them optimally.
- **Overlapping Subproblems Method:** Highlights the key feature of DP where subproblems overlap, making memoization or tabulation effective.
- **Iterative Refinement Programming:** Describes the process of incrementally building up solutions in the bottom-up approach.
- **Recursive Optimization Programming:** Combines the recursive nature and optimization focus of DP.

A common example: the Fibonacci sequence

Rules (Bellman equations) :

$$F(0) = 0 \text{ , } F(1) = 1$$

$$F(n) = F(n - 1) + F(n - 2)$$

How to compute efficiently F(n) for a large n?

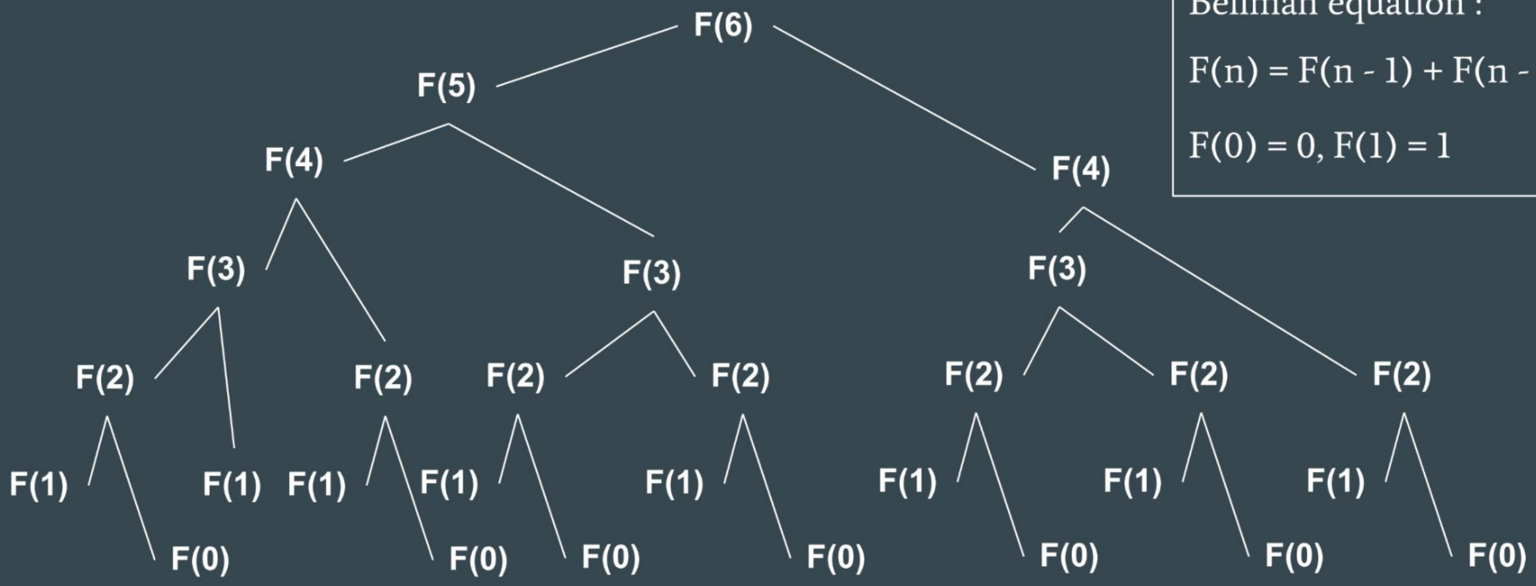
The Fibonacci sequence

Naive recursive algorithm :

```
def Fibonacci(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    else:
        return Fibonacci(n - 1) + Fibonacci(n - 2)

Fibonacci(n)
```

The Fibonacci sequence



Bellman equation :

$F(n) = F(n - 1) + F(n - 2)$

$F(0) = 0, F(1) = 1$

This algorithm runs in O(2^n), there is probably a better way to do this...

Memoization: how to trade space for time

Keep track of which subproblems have already been solved

- Keep those subproblems’ result in memory
- If one shows up again, take its result out of memory instead of computing it again

An efficient algorithm for computing the Fibonacci sequence

```
def Fibonacci(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    if memo[n] != -1:
        return memo[n]
    else:
        result = Fibonacci(n - 1) + Fibonacci(n - 2)
        memo[n] = result
        return result

memo = [-1] * (n + 1)
Fibonnaci(n)
```

An efficient algorithm for computing the Fibonacci sequence

```
def Fibonacci(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    if memo[n] != -1:
        return memo[n]
    else:
        result = Fibonacci(n - 1) + Fibonacci(n - 2)
        memo[n] = result
        return result

memo = [-1] * (n + 1)
Fibonnaci(n)
```

```
<Base case>
<reuse already computed
subproblems if possible>
<Bellman equations>
<store result (memoize)>
```

Sequence of calls with this algorithm

Bellman equation :

$$F(n) = F(n-1) + F(n-2)$$

$$F(0) = 0,\ F(1) = 1$$

We've achieved a O(n) algorithm !

Bottom Up vs Top Down

Bottom up

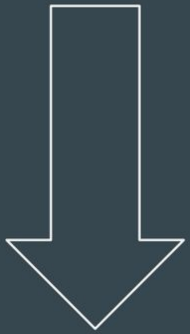
- Iterative version
- Build subproblems and grow toward the global one
- A bit faster, better for memory concerns

Top down

- Recursive version
- Decrease from the initial problem to the base cases
- Easy to implement once you have the Bellman equation

Initial problem

Base cases



Initial problem

Base cases

Bottom Up vs Top Down : Fibonacci sequence

Bottom Up

```
fibo = [0] * (n + 1)
fibo[0] = 0
fibo[1] = 1
for k in range(2, n + 1):
    fibo[k] = fibo[k - 1]
            + fibo[k - 2]
```

Top Down

```
def Fibonacci(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    if memo[n] != -1:
        return memo[n]
    else:
        result = Fibonacci(n - 1)
               + Fibonacci(n - 2)
        memo[n] = result
        return result

memo = [-1] * (n + 1)
Fibonnaci(n)
```

Quick recap : What do we need ?

- Find a way to split the initial problem into subproblems that are easier to solve
- Find the Bellman equation to jump from one problem to its subproblems easily
- Think about the base cases you can consider !
- Don’t forget to memoize
- Enjoy being a wizard of algorithms!

Next lectures :

- How are you going to store the subproblems’ results? Choose your data structures wisely !
- Common DP problems
- Take a step back : how to quickly evaluate if DP is necessary / will be fast enough?

Credits

Slides: Arthur Tondereau, Louis Sugy, Onyr (Florian Rascoussier) for INSAlgo

The algorithm design manual, Steven S.Skiena

Wikipedia of course!