Artificial Intelligence

Paradigm

G. Guérard

Department of Nouvelles Energies
Ecole Supérieur d'Ingénieurs Léonard de Vinci

Lecture 2

Outline

Overview

A recursive function **calls itself** *directly* or *indirectly*.

It is a programming tool, based on a non-intuitive mode of thinking.

Recursion form the base to another paradigm:
DIVIDE-AND-CONQUER.

Overview

ITERATION:

- **Uses** repetition structures (*for, while or do...while*)
- **Repetition through** explicitly use of repetition structure
- **Terminates when** loop-continuation conditions fail
- **Controls repetition** by using a counter.

RECURSION:

- **Uses** selection structures (*if, if...else or switch*)
- **Repetition through** repeated method calls
- **Terminates when** base cases are satisfied
- **Controls repetition** by dividing problem into simpler one.

Stack

To understand how recursion works, it helps to visualize what's going on. To help visualize, we will use a common concept called the STACK.

Stack

A stack basically operates like a container with priority on inside objects.

It has only two operations:

- PUSH: you can push something onto the stack.
- POP: you can pop something off the top of the stack.

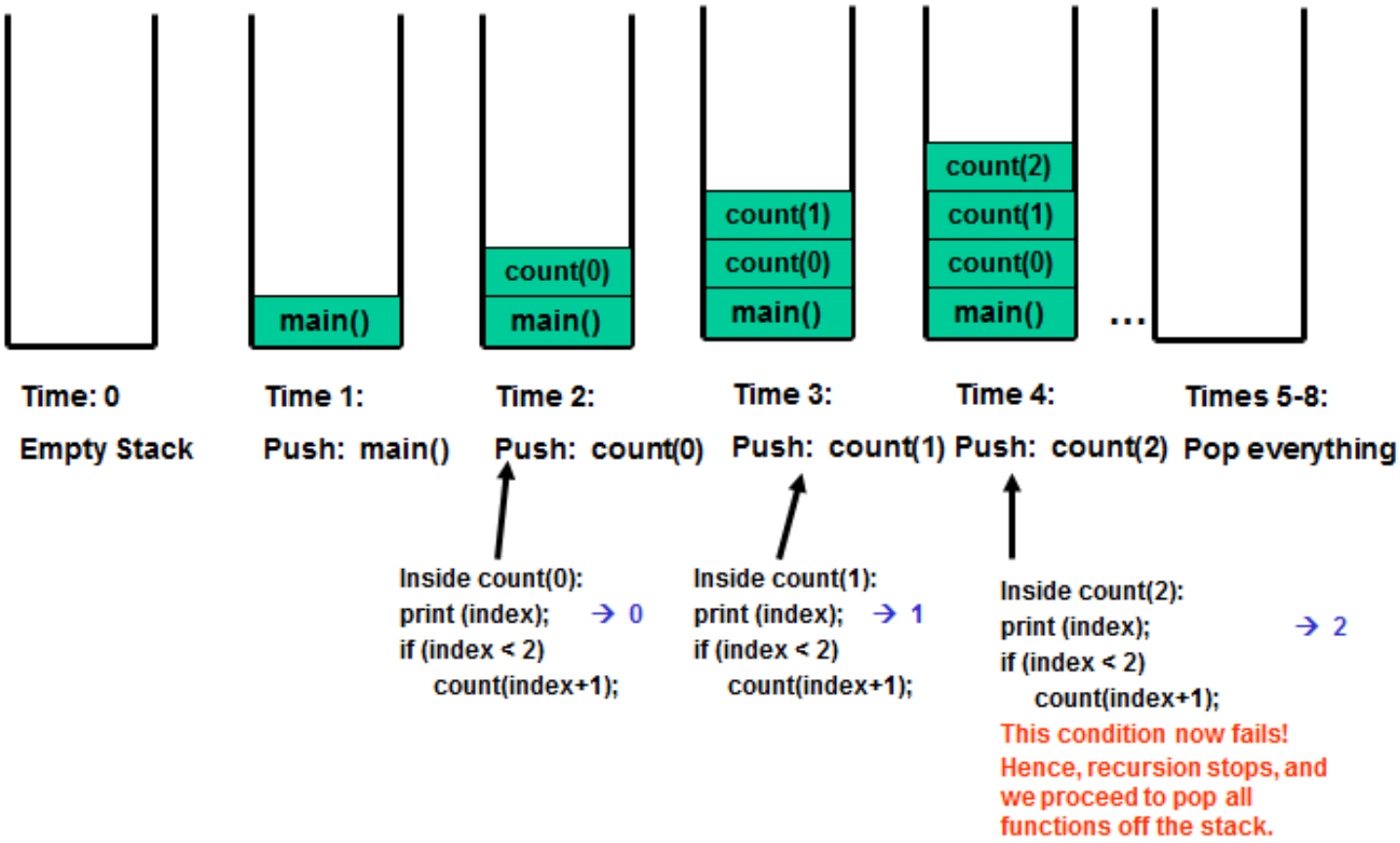
Stacks and Methods

When you run a program, the computer creates a stack for you.

- Each time you **invoke** a method, the method is placed on top of the stack (PUSH).
- When the method **returns** or exits, the method is popped off the stack (POP).
- If a method calls itself recursively, you just push another copy of the method onto the stack.

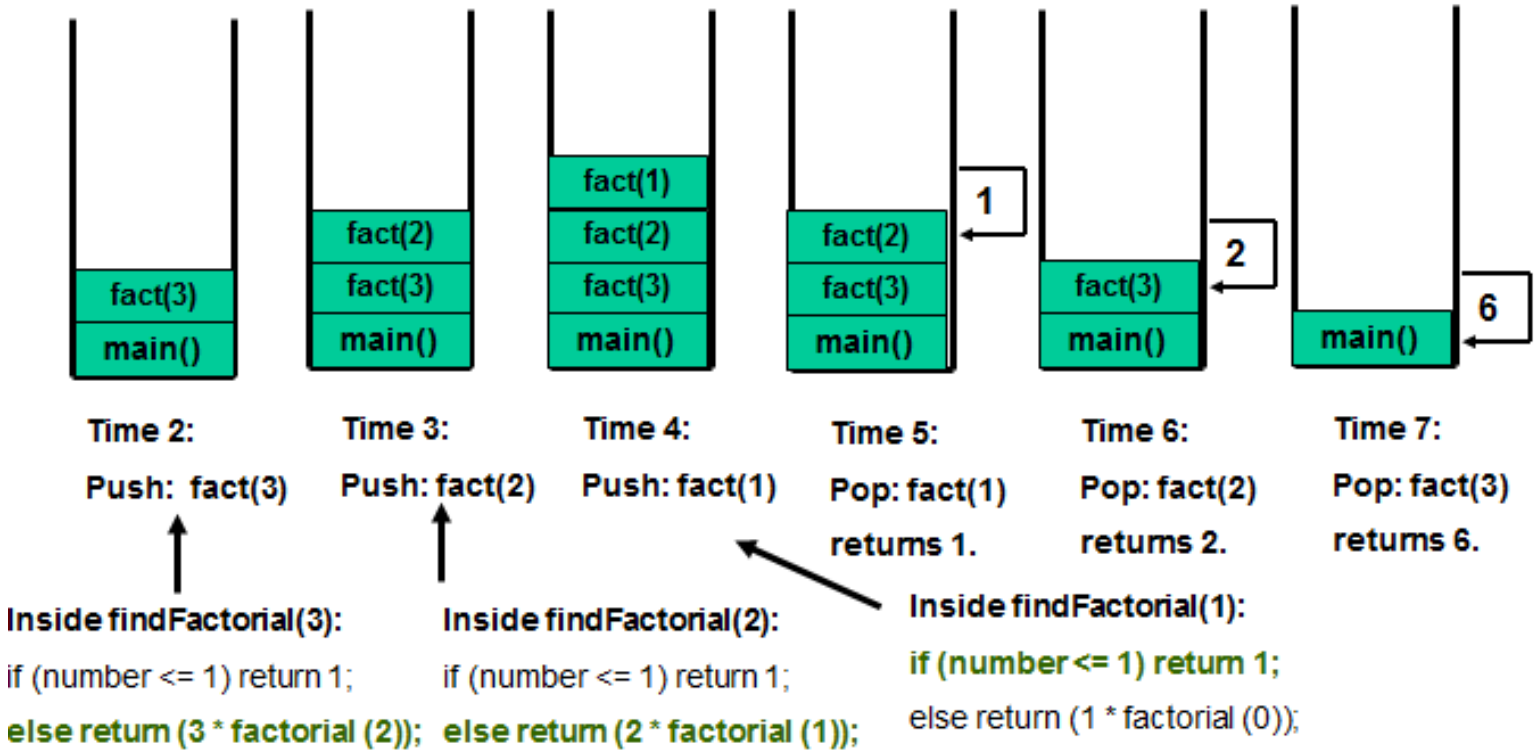
Example

Consider *Void Count(int index)* a recursion function: *if(index < 2) Count(index+1)*



Another Example

Consider *Int Factorial(int number)* a recursion function: *if ((number == 1) || (number == 0)) return 1; else return (number * Factorial (number-1))*



Pro. and Con.

1. More overhead than iteration;
2. More memory intensive than iteration;
3. Can also be solved iteratively;
4. Often can be implemented with only a few lines of code.

Introduction

DIVIDE-AND-CONQUER is not a trick. It is a very useful general purpose tool for designing efficient algorithms.
It follows those steps:

1. **Divide:** *divide a given problem into subproblems (of approximately equal size)*
2. **Conquer:** *solve each subproblem directly or recursively*
3. **Combine:** *and combine the solutions of the subproblems into a global solution.*

Algorithm

The *general structure* of an algorithm designed by using divide and conquer is:

$divide\_and\_conquer(P(n))$
if $n \leq n_c$ **then** $\langle$ solve directly $P(n)$ $\rangle$
else
    $\langle$ divide $P(n)$ in k subproblems $P_1(n_1), \ldots P_k(n_k)$ $\rangle$
    for $i \leftarrow 1, k$
        $divide\_and\_conquer(P_i(n_i))$
    endfor $\langle$ combine the results $\rangle$
endif

Mergesort

The algorithm sorts an array of size N by splitting it into two parts of almost equal size, recursively sorting each of them, and then merging the two sorted subarrays back together into a fully sorted list in $O(N)$ time (comparing in order both array into a single array).

Mergesort(A, i, j) : Sort $A[i \ldots j]$

If $(i \neq j)$

{

Mergesort $\left(A, i, \lfloor \frac{i+j}{2} \rfloor\right)$

Mergesort $\left(A, 1 + \lfloor \frac{i+j}{2} \rfloor, j\right)$

Merge the two sorted lists

$A\left[i \ldots \lfloor \frac{i+j}{2} \rfloor\right]$ and $A\left[1 + \lfloor \frac{i+j}{2} \rfloor, j\right]$

and return complete sorted list

}

Complexity

$M\left(\frac{j-i}{2}\right)$

$M\left(\frac{j-i}{2}\right)$

$O(j-i)$

Mergesort

The algorithm sorts an array of size N by splitting it into two parts of almost equal size, recursively sorting each of them, and then merging the two sorted subarrays back together into a fully sorted list in $O(N)$ time.

The running time of the algorithm satisfies the Master theorem:

$$\forall N > 1,\ M(N) \leq 2M(N/2) + O(N)$$

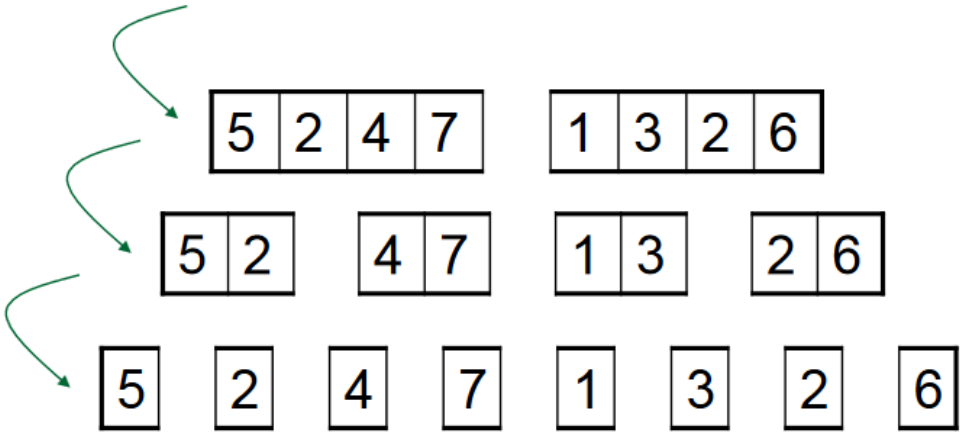
wich we implies

$$M(N) = O(N\ log\ N).$$

Mergesort

Divide

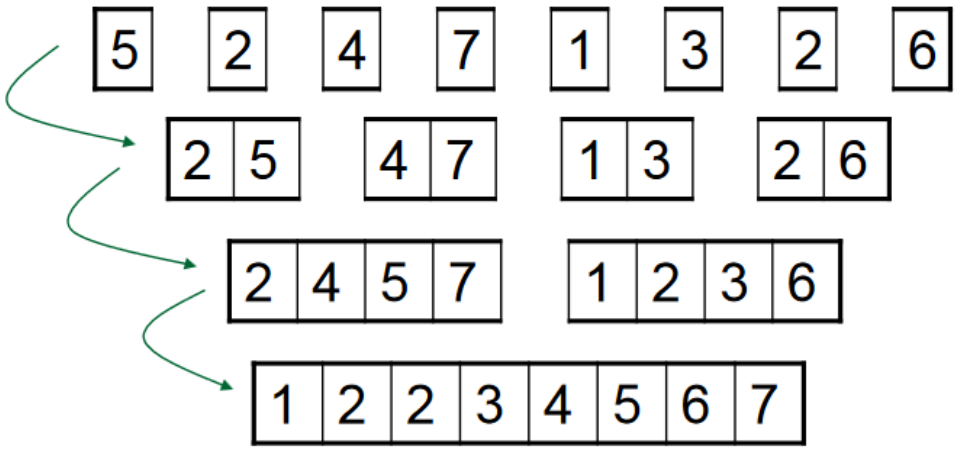
$log(n)$ divisions to split an array of size n into elements.



Mergesort

Conquer

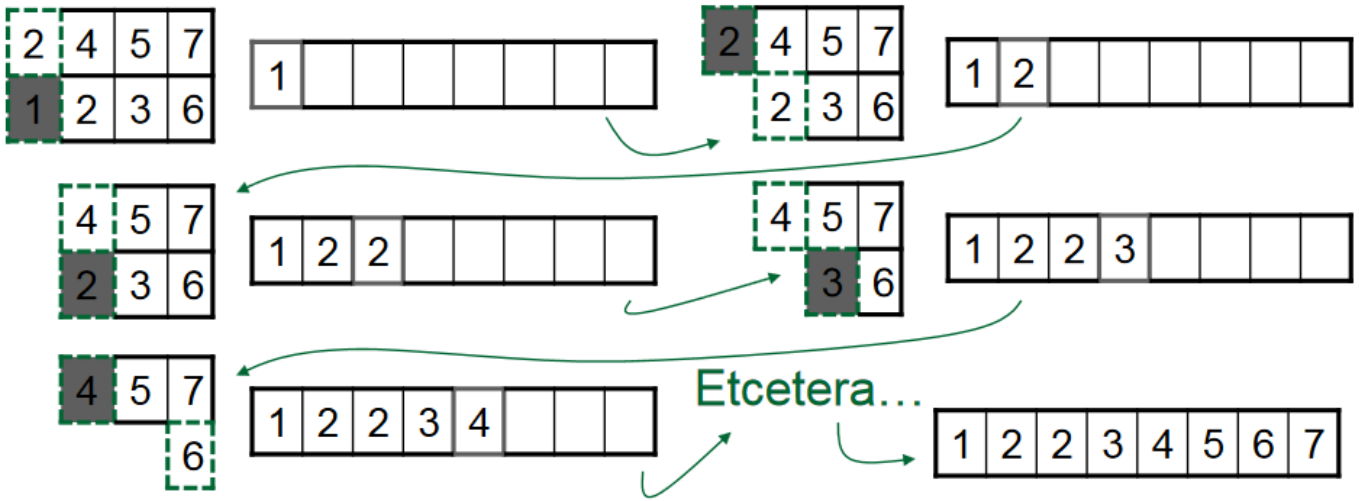
$log(n)$ iterations, each iterations takes $O(n)$ time, for a total time $O(n\ log\ n)$



Mergesort

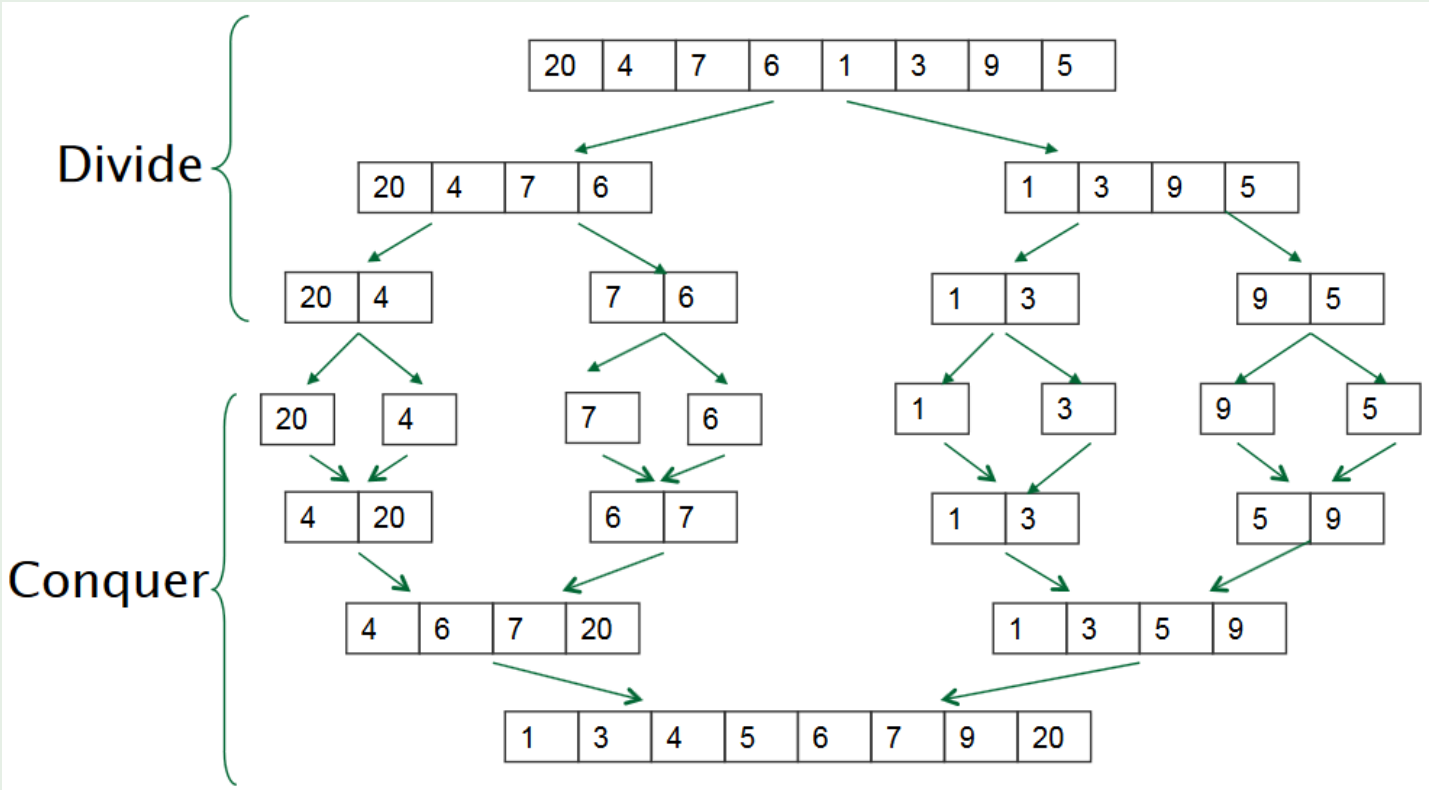
Combine

Two sorted arrays can be merged in linear time into a sorted array.



Mergesort

Example



Overview

DYNAMIC PROGRAMMING is a powerfull algorithmic design technique. Large class of seemingly exponential problems have a polynomial solution via dynamic programming, particularly for optimization problems.

The main difference between greedy, D&C and DP programs are:

- **Greedy:** *build up a solution incrementally, optimizing some local criterion.*
- **Divide-&-conquer:** *break up a problem into independent subproblems, solve each one and combine solution.*
- **Dynamic programming:** *break up a problem into a series of overlapping subproblems, and build up solutions to larger and larger subproblems.*

Four steps of Dynamic programming

1. Define subproblems and characterize the structure of an optimal solution (OPTIMAL SUBSTRUCTURE).
2. Recursively define the value of an optimal solution (RECURSIVE FORMULATION).
3. TOP-DOWN: Recurse and memoize; or BOTTOM-UP: Compute the value of an optimal solution using an array/table.
4. Construct an OPTIMAL SOLUTION from the computed information.

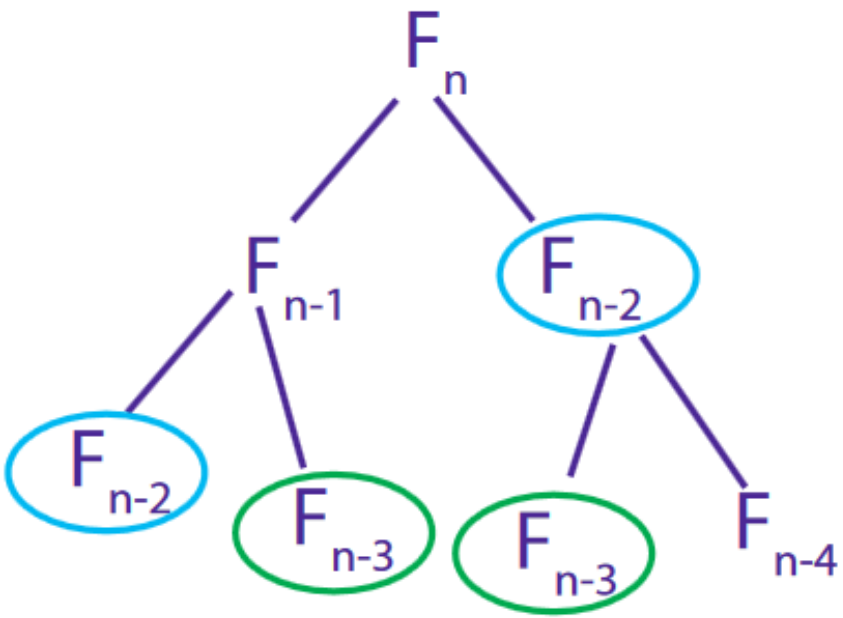
Example

Fib(n): *if* $n \leq 2$ *return 1; else return Fib(n-1)+Fib(n-2);*

Running time: $M(n) = M(n-1) + M(n-2) + O(1) \geq 2M(n-2) + O(1) \geq 2^{n/2}$.

An exponential running time is bad for this kind of problem.

We could memoize some inner solution to compute the problem.



Top-Down Dynamic Program

```
memo = { }
fib(n):
        if n in memo: return memo[n]
        else: if n ≤ 2 : f = 1
              else: f = fib(n − 1) + fib(n − 2)
              memo[n] = f
              return f
```

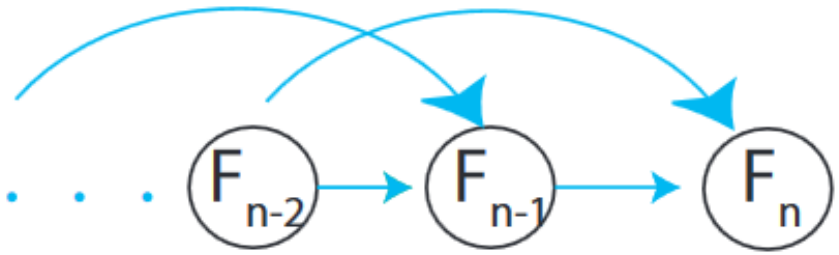
In this program, fib(k) only recurses first time called, for any k. Thus, there are only n nonmemoized calls. Each memoized calls are free, in $O(1)$. Top-Down memoize and re-use solutions to subproblems that help solve problem.

Bottom-Up Dynamic Program

```
fib = {}
for k in [1, 2, ..., n]:
    if k ≤ 2: f = 1
    else: f = fib[k − 1] + fib[k − 2]
    fib[k] = f
return fib[n]
```

$\Theta(1)$ (inner block) — $\Theta(n)$ (outer block)

Bottom-up dynamic program construct the solution from the last subproblems to the problem itself. We have just to remenber the last two fibs. Bottom-up dynamic programs follow the same scheme.



Optimal Substructure

Definition

A problem have OPTIMAL SUBSTRUCTURE when the optimal solution of *a problem contains in itself solutions for subproblems of the same type*.

If a problem presents this characteristic, we say that it respects the optimality principle.

Overlapping Substructure

Definition

A problem is said to have OVERLAPPING SUBPROBLEMS if the *problem can be broken down into subproblems which are reused several times or a recursive algorithm for the problem solves the same subproblem over and over* rather than always generating new subproblems (for example by memoization).

See Fibonacci example.

Four steps of Dynamic programming v2

1. Characterize the optimal solution of the problem.
    1. Understand the problem
    2. Verify if a brute force algorithm is enough (optional)
    3. Generalize the problem
    4. Divide the problem in subproblems of the same type
    5. Verify if the problems obeys the optimality principle and overlapping subproblems.
       If the problem presents these two characteristics, we know that dynamic programming is applicable.

Four steps of Dynamic programming v2

2. Recursively define the optimal solution, by using optimal solutions of subproblems
    1. Recursively define the optimal solution value, exactly and with rigour, from the solutions of subproblems of the same type
    2. Imagine that the values of optimal solutions are already available when we need them
    3. Mathematically define the recursion

Four steps of Dynamic programming v2

3. Compute the solutions of all subproblems: top-down
   1. Use the recursive function directly obtained from the definition of the solution and keep a table with the results already computed
   2. When we need to access a value for the first time we need to compute it, and from then on we just need to see the already computed result.

Four steps of Dynamic programming v2

3. Compute the solutions of all subproblems: bottom-up
   1. Find the order in which the subproblems are needed, from the smaller subproblem until we reach the global problem and implement, using a table
   2. Usually this order is the inverse to the normal order of the recursive function that solves the problem

Four steps of Dynamic programming v2

4. Reconstruct the optimal solution, based on the computed values
    1. Directly from the subproblems table
    2. OR New table that stores the decisions in each step

Matches

- There are n matches
- In each play you can choose to remove 1, 3, or 8 matches
- Whoever removes the last stones, wins the game.

GIVEN THE NUMBER OF INITIAL STONES, CAN THE PLAYER THAT STARTS TO PLAY GUARANTEE A WIN?

Matches

1. Characterize the optimal solution of the problem.

- In BRUTE FORCE algorithm, there are 3^k possible games.
- Let $win(i)$ be a boolean value representing if we can win when there are i matches:
  - $win(1)$, $win(3)$,$win(8)$ are true
  - For the other cases:
    - if your play goes make the game go to winning position, then our opponent can force your defeat.
    - Therefore, our position is a winning position if we can get to a losing position.
    - If all possible movements lead to a winning position, then your position is a losing one.

Matches

2 Recursively define the optimal solution.

- $win(0) = false$
- $win(i) = \begin{cases} true & if\,(win(i-1) = false \,||\, win(i-3) = false \,||\, win(i-8) = false \\ false & otherwise \end{cases}$

Matches

3 Compute the solutions of all subproblems: bottom-up

- For $i \leftarrow 0$ to n do
  - if(($i \geq 1$ && $win(i-1) = false$) || ($i \geq 3$ && $win(i-3) = false$) || ($i \geq 8$ && $win(i-8) = false$))
  - then $win(i) \leftarrow true$
  - else $win(i) \leftarrow false$

i	0	1	2	3	4	5	6	7	8	9	10	11	12
win(i)	f	t	f	t	f	t	f	t	t	t	t	f	t

Fundamental knowledge

You have to know before the tutorial:

1. *Recursion:*
   1. *Principle;*
   2. *Stack: how it works;*
   3. *Divide-and-Conquer paradigm: three steps and general structure;*
   4. *Understand the mergesort algorithm.*
2. *Dynamic programming:*
   1. *Principle;*
   2. *Dynamic programming paradigm: four steps and bottom-up recursion (see v2);*
   3. *Optimal substructure;*
   4. *Overlapping subproblems.*