

Universidade Federal de São Carlos - UFSCar
Departamento de Computação - DC
CEP 13565-905, Rod. Washington Luiz, s/n, São Carlos, SP

Programação Dinâmica - Parte 1

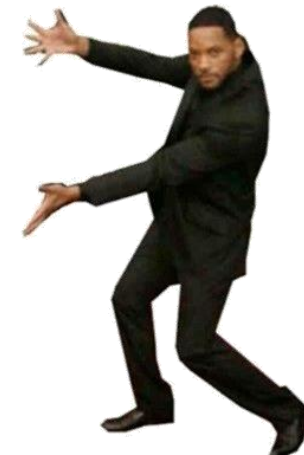
Prof. Dr. Alan Demétrius Baria Valejo

CCO-00.2.01 - Projeto e Análise de Algoritmos (*Design And Analysis Of Algorithms*)
1001525 - Projeto e Análise de Algoritmos - Turma A

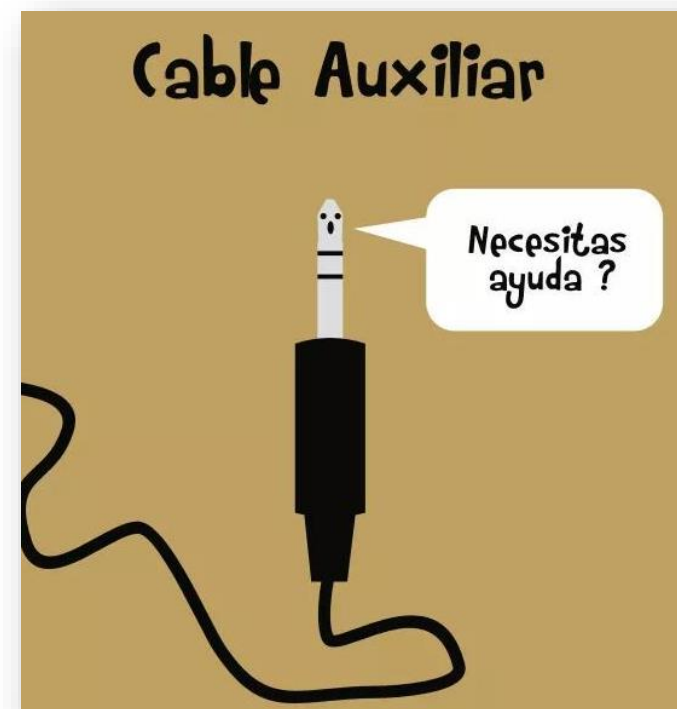
- Motivação
- Conceitos básicos
- Problema do troco

“NÃO EXISTE NADA PERFEITO NESSE MUNDO”

- Primeiro ensinamento
 - Programação dinâmica não é perfeita



- Segundo ensinamento
 - Além de ser utilizado como técnica base para solução de muitos problemas difíceis, programação dinâmica também é muito utilizado como uma técnica auxiliar para outros algoritmos, portanto, é possível que você já tenha visto ou estudado programação dinâmica em algum momento e, provavelmente, você não tenha se atentado.



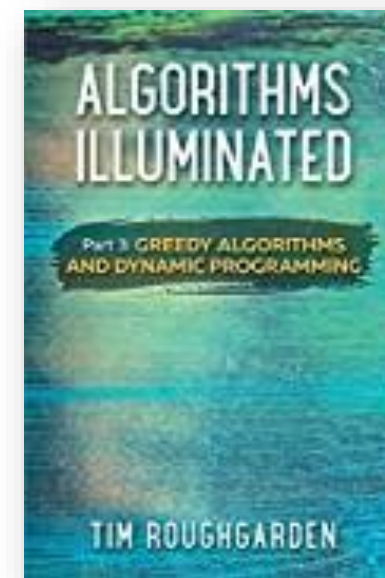
- Terceiro ensinamento
 - A primeira vez que você se depara com programação dinâmica pode parecer difícil, porém, após se deparar com esse conceito algumas vezes, irá se tornar mais simples

Pep Talk

- It is totally normal to feel confused the first time you see dynamic programming. Confusion should not discourage you. It does not represent an intellectual failure on your part, only an opportunity to get even smarter.

Conversa estimulante

- É totalmente normal sentir-se confuso na primeira vez que vê a programação dinâmica. A confusão não deve desencorajá-lo. Não representa uma falha intelectual de sua parte, apenas uma oportunidade de ficar ainda mais inteligente.



Tim Roughgarden - Algorithms Illuminated

- Quarto ensinamento
 - Lembram de algoritmos gulosos?
 - Todos os algoritmos pareciam iguais
 - Mesmo algoritmos mais complexos, como o Kruskal, tinha a mesma estrutura e escondiam sua complexidade em ordenações ou tipo abstrato de dados.
 - Programação dinâmica segue a mesma linha, quase todos os problemas possuem uma solução cuja estrutura algorítmica é bem parecida

- Quarto ensinamento
 - Lembram de algoritmos gulosos?
 - Todos os algoritmos pareciam iguais
 - Mesmo algoritmos mais complexos, como o Kruskal, tinha a mesma estrutura e escondiam sua complexidade em ordenações ou tipo abstrato de dados.
 - Programação dinâmica segue a mesma linha, quase todos os problemas possuem uma solução cuja estrutura algorítmica é bem parecida



Mentira



Fala logo o que é PD e para de enrolar

- Uma estratégia ótima apresenta a propriedade segundo a qual, despeito das decisões tomadas para se atingir um estado particular num certo estágio, as decisões restantes a partir deste estado devem constituir uma estratégia ótima.



- Uma estratégia ótima apresenta a propriedade segundo a qual, despeito das decisões tomadas para se atingir um estado particular num certo estágio, as decisões restantes a partir deste estado devem constituir uma estratégia ótima.

Resumindo, se eu tiver uma resposta ótima para estados anteriores eu consigo uma resposta ótima para o meu estado corrente.

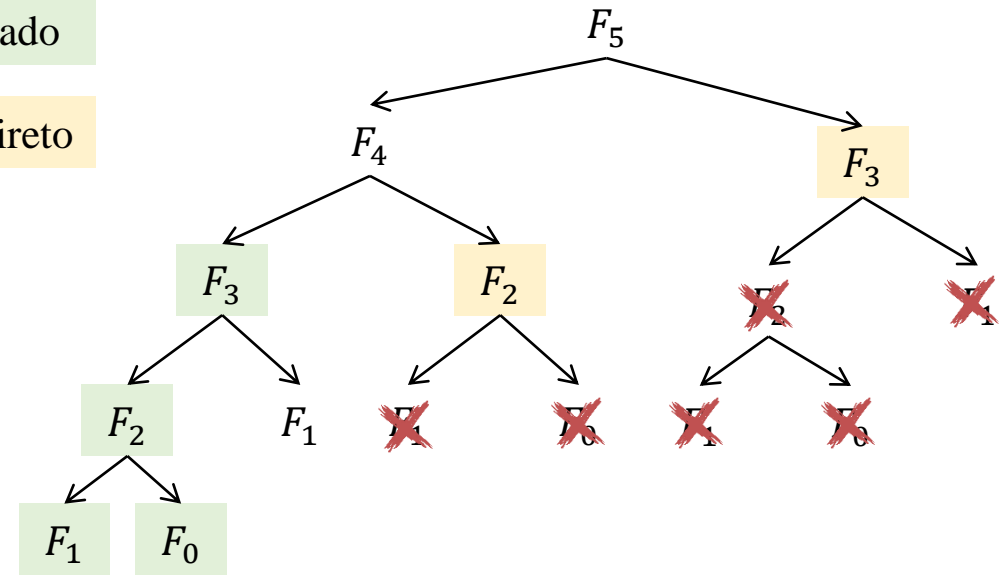


- A programação dinâmica consiste em:
 - Resolver subproblemas
 - *Memoizar*
- Uma vez que a resposta de um problema depende de subproblemas para os quais você já calculou a resposta, não é necessário calculá-las novamente

- A programação dinâmica consiste em:
 - Resolver subproblemas
 - *Memoizar*
- Uma vez que a resposta de um problema depende de outras, calculá-las novamente

memorizado

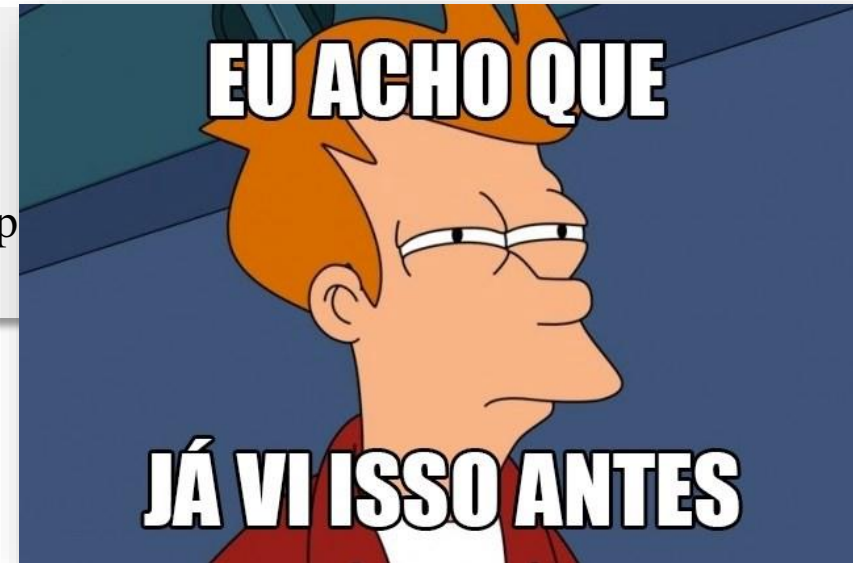
cálculo direto



essário

Opa, pera ai, eu já isso em algum lugar!!!!

- A programação dinâmica consiste em:
 - Resolver subproblemas
 - *Memoizar*
 - Uma vez que a resposta de um problema depende de subproblemas p
calculá-las novamente
- é necessário



- A programação dinâmica consiste em:
 - Resolver subproblemas
 - *Memoizar*
- Uma vez que a resposta de um problema depende de subproblemas para os quais você já calculou a resposta, não é necessário calculá-las novamente

Simples, certo?





- A sequência de Fibonacci é uma sequência de números inteiros, começando normalmente por 0 e 1, na qual cada termo subsequente corresponde à soma dos dois anteriores
- Matemático italiano Leonardo de Pisa ou Leonardo Fibonacci

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, ...

- A sequência de Fibonacci é infinita, portanto, o ideal é que você defina um valor objetivo

$$F_n \begin{cases} 1 & \text{se } n = 1 \text{ ou } n = 2, \\ F_{n-1} + F_{n-2} & \text{caso contrário.} \end{cases}$$



- *Top-down (memoization)*
 - Parte-se da solução geral ótima que se deseja encontrar e, então, analisa-se quais subproblemas são necessários resolver até que se chegue em um subproblema com resolução trivial
 - Utiliza-se soluções recursivas
 - No Fibonacci, começamos pelo n -ésimo número e, recursivamente, ir calculando os valores de $fib(n - 1)$, $fib(n - 2)$, ...

$$fib(n) \rightarrow fib(n - 1) \rightarrow fib(n - 2) \rightarrow \dots \rightarrow fib(2) \rightarrow fib(1) \rightarrow fib(0)$$

- *Top-down (memoization)*
 - Parte-se da solução geral ótima que se deseja encontrar e, então, analisa-se quais subproblemas são necessários resolver até que se chegue em um subproblema com resolução trivial
 - Utiliza-se soluções recursivas
 - No Fibonacci, começamos pelo n -ésimo número e, recursivamente, ir calculando os valores de $fib(n - 1)$, $fib(n - 2)$, ...

$$fib(n) \rightarrow fib(n - 1) \rightarrow fib(n - 2) \rightarrow \dots \rightarrow fib(2) \rightarrow fib(1) \rightarrow fib(0)$$

- *Top-down (memoization)*
 - Parte-se da solução geral ótima que se deseja encontrar e, então, analisa-se quais subproblemas são necessários resolver até que se chegue em um subproblema com resolução trivial
 - **Utiliza-se soluções recursivas**
 - No Fibonacci, começamos pelo n -ésimo número e, recursivamente, ir calculando os valores de $fib(n - 1)$, $fib(n - 2)$, ...

$$fib(n) \rightarrow fib(n - 1) \rightarrow fib(n - 2) \rightarrow \dots \rightarrow fib(2) \rightarrow fib(1) \rightarrow fib(0)$$

- *Top-down (memoization)*
 - Parte-se da solução geral ótima que se deseja encontrar e, então, analisa-se quais subproblemas são necessários resolver até que se chegue em um subproblema com resolução trivial
 - Utiliza-se soluções recursivas
 - No Fibonacci, começamos pelo n -ésimo número e, recursivamente, ir calculando os valores de $fib(n - 1)$, $fib(n - 2)$, ...

$fib(n) \rightarrow fib(n - 1) \rightarrow fib(n - 2) \rightarrow \dots \rightarrow fib(2) \rightarrow fib(1) \rightarrow fib(0)$

```
def fibonacciTopDown(n, table = {}):  
    if n in {0, 1}:  
        return n  
    try:  
        return table[n]  
    except:  
        table[n] = fibonacciTopDown(n-1) + fibonacciTopDown(n-2)  
        return table[n]
```

} Solução trivial

} Busca na tabela de *memoização*

} Armazena os resultados

Dica Python: isso funciona, mesmo não passando como parâmetro!!!

```
def fibonacciTopDown(n, table = {}):  
    if n in {0, 1}:  
        return n  
    try:  
        return table[n]  
    except:  
        table[n] = fibonacciTopDown(n-1) + fibonacciTopDown(n-2)  
        return table[n]
```

} Solução trivial

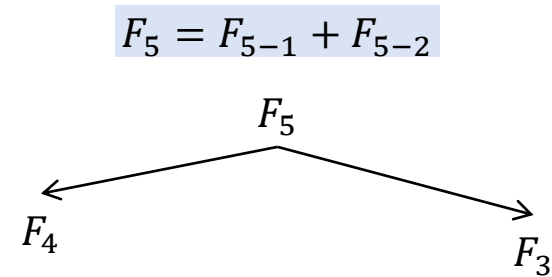
} Busca na tabela de *memoização*

} Armazena os resultados

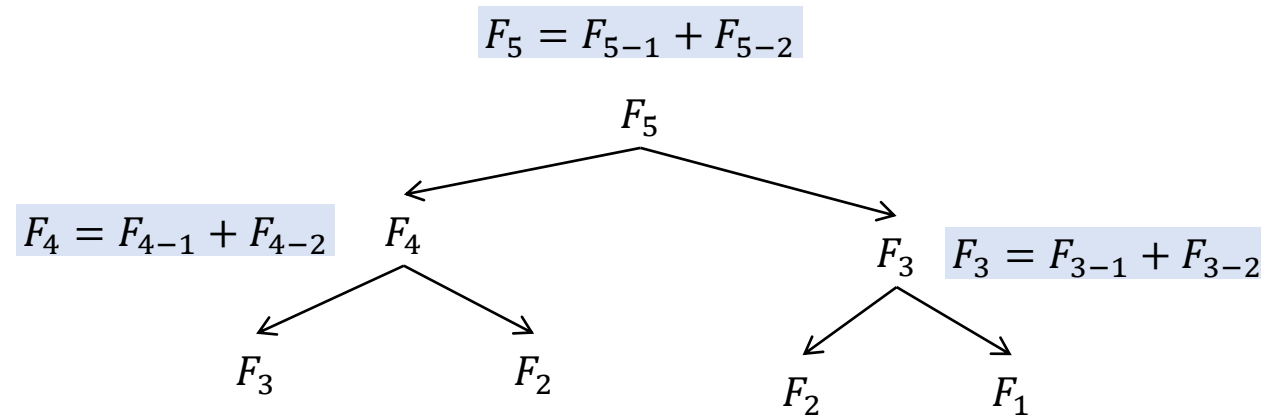
- Como funciona essa função?
- Suponha F_5

F_5

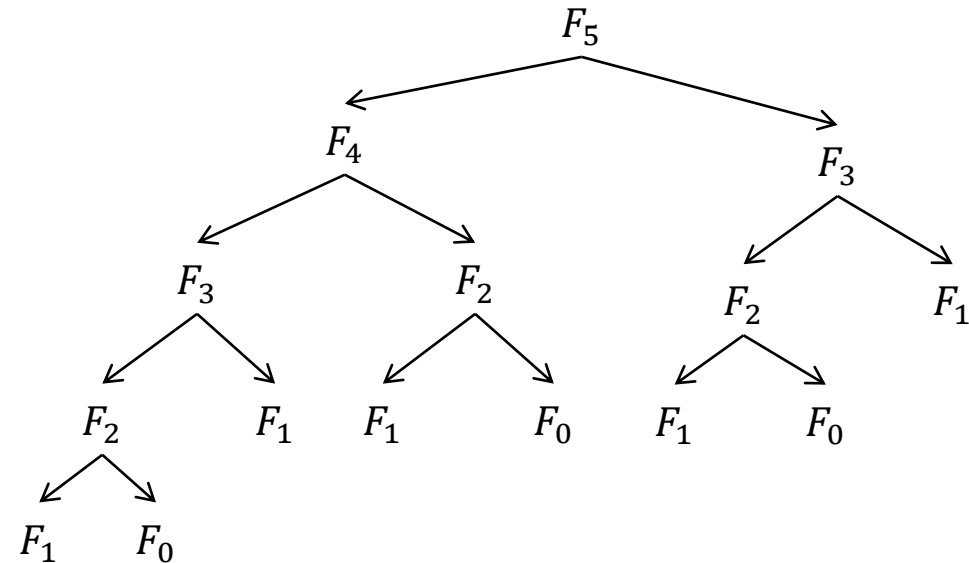
- Como funciona essa função?
- Suponha F_5



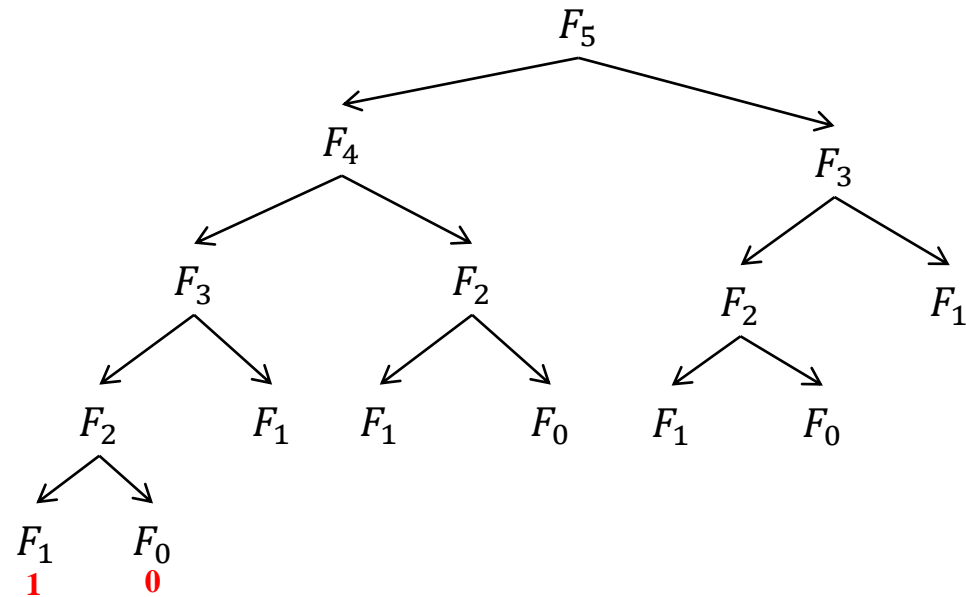
- Como funciona essa função?
- Suponha F_5



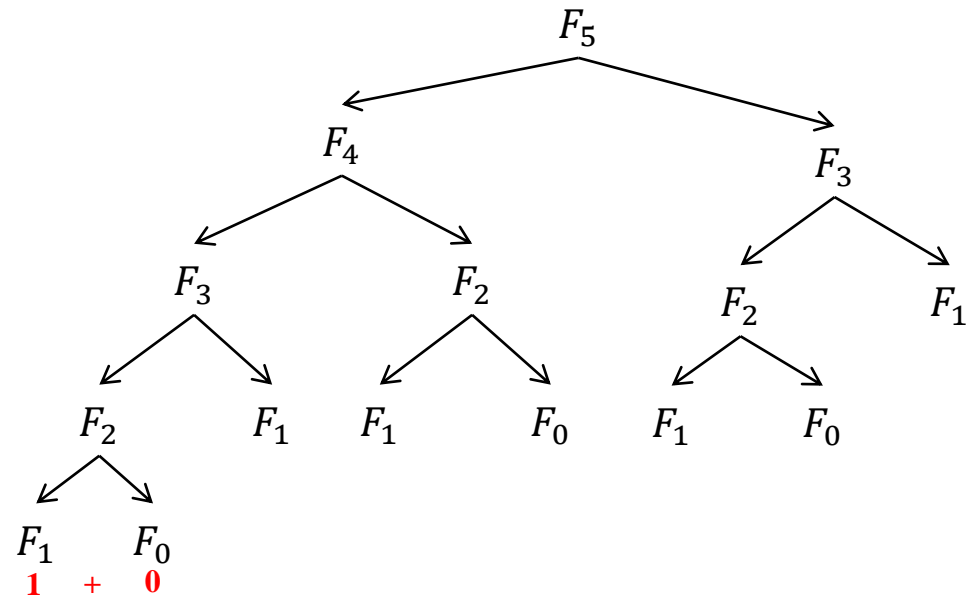
- Como funciona essa função?
- Suponha F_5



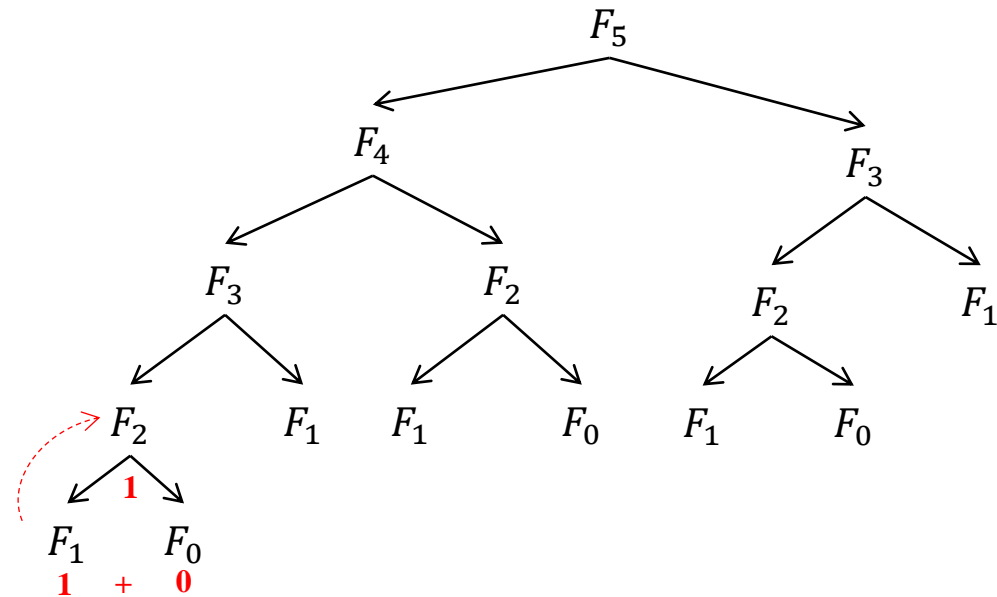
- Como funciona essa função?
- Suponha F_5



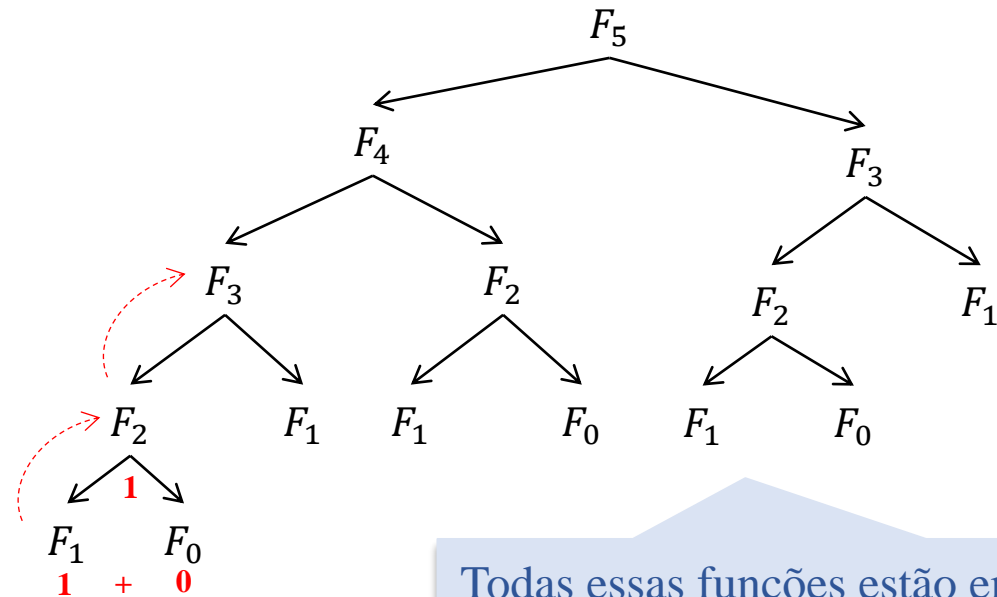
- Como funciona essa função?
- Suponha F_5



- Como funciona essa função?
- Suponha F_5

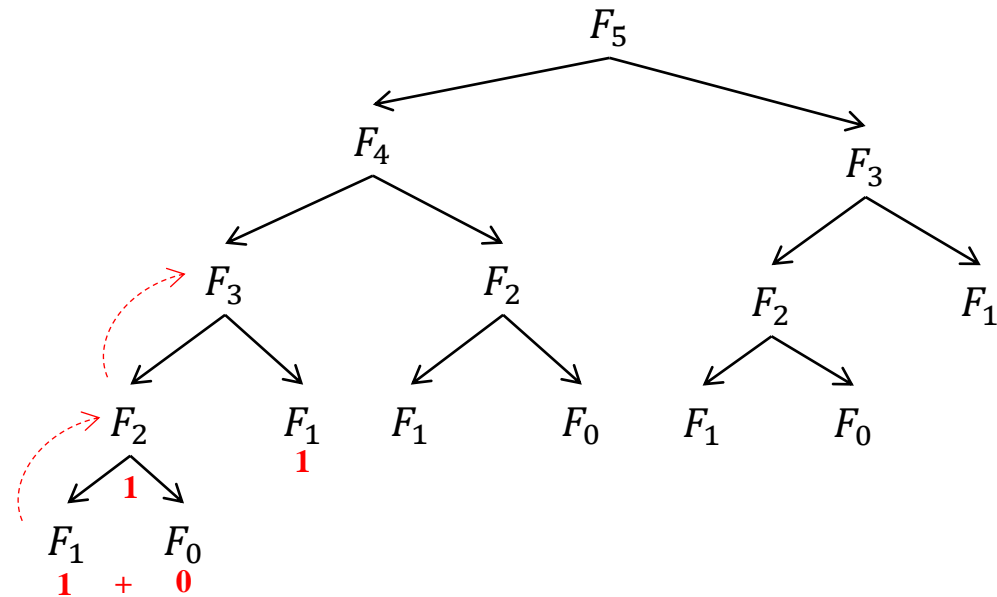


- Como funciona essa função?
- Suponha F_5

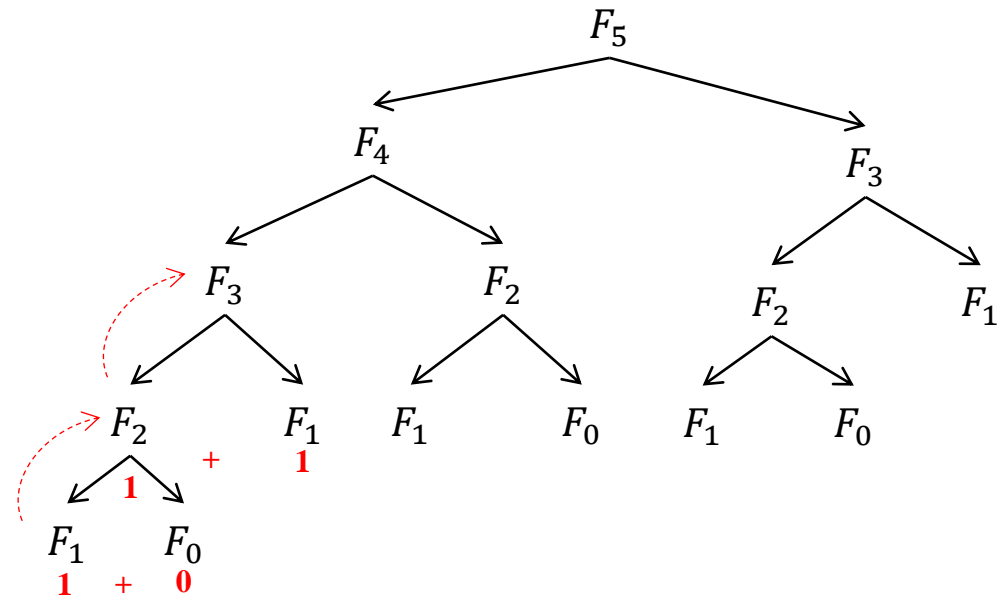


Todas essas funções estão empilhadas na memória e o compilador ou interpretador não sabe que elas já foram calculadas

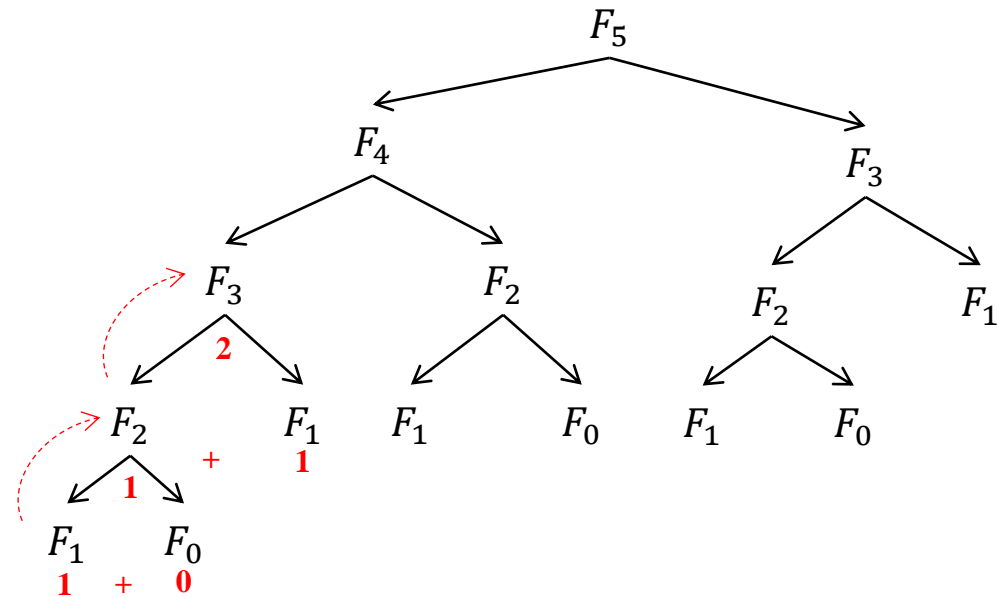
- Como funciona essa função?
- Suponha F_5



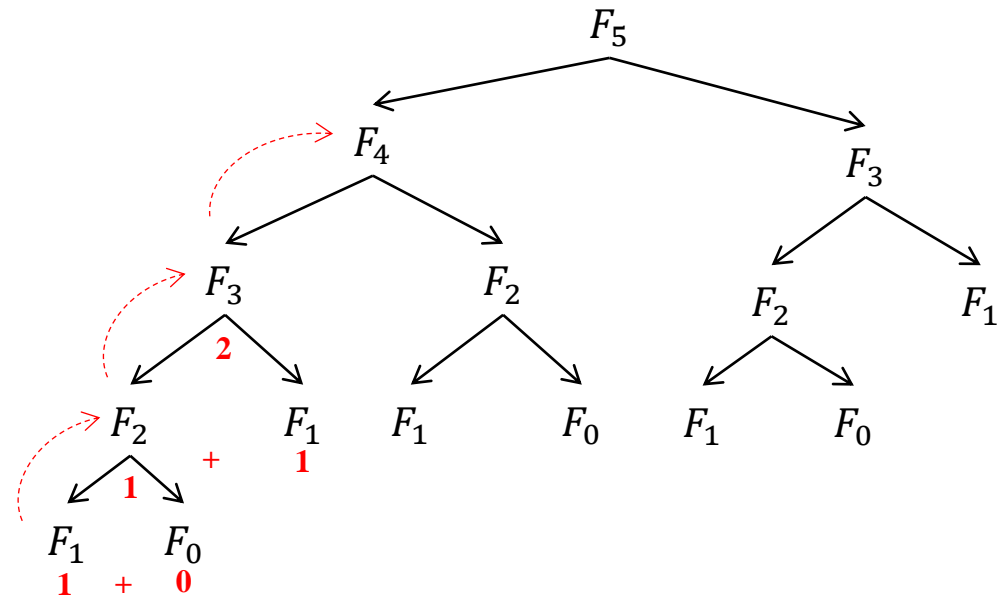
- Como funciona essa função?
- Suponha F_5



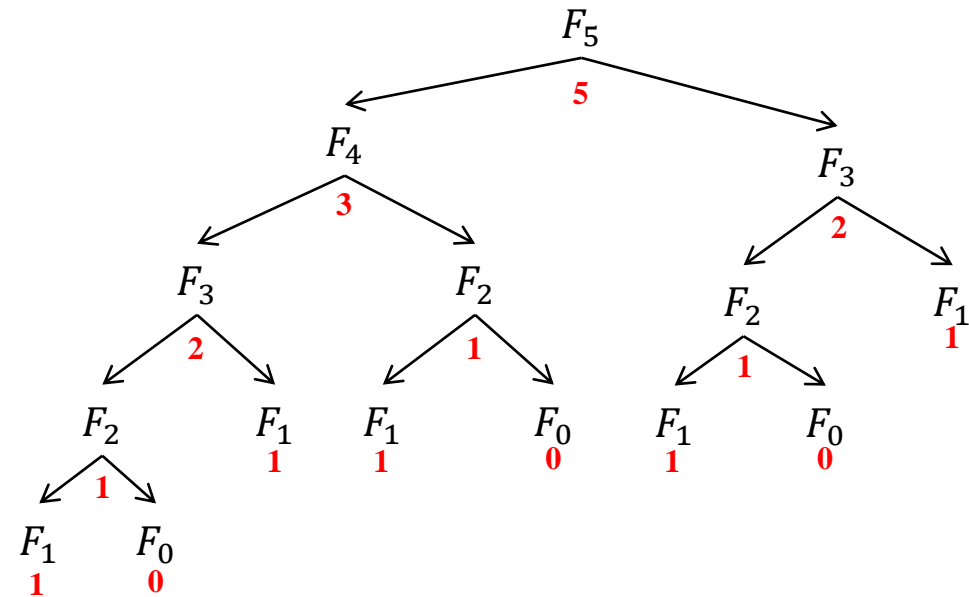
- Como funciona essa função?
- Suponha F_5



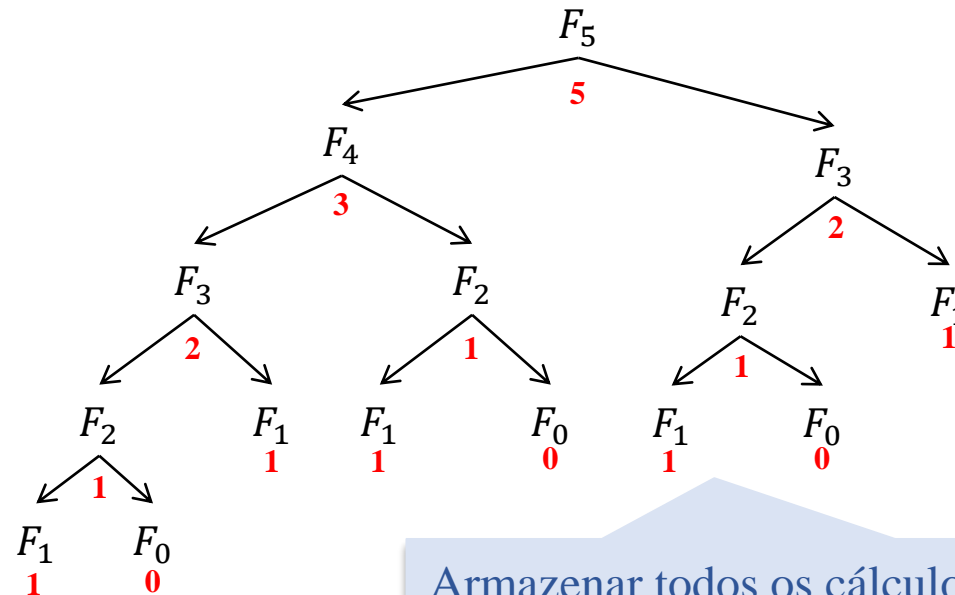
- Como funciona essa função?
- Suponha F_5



- Como funciona essa função?
- Suponha F_5



- Como funciona essa função?
- Suponha F_5

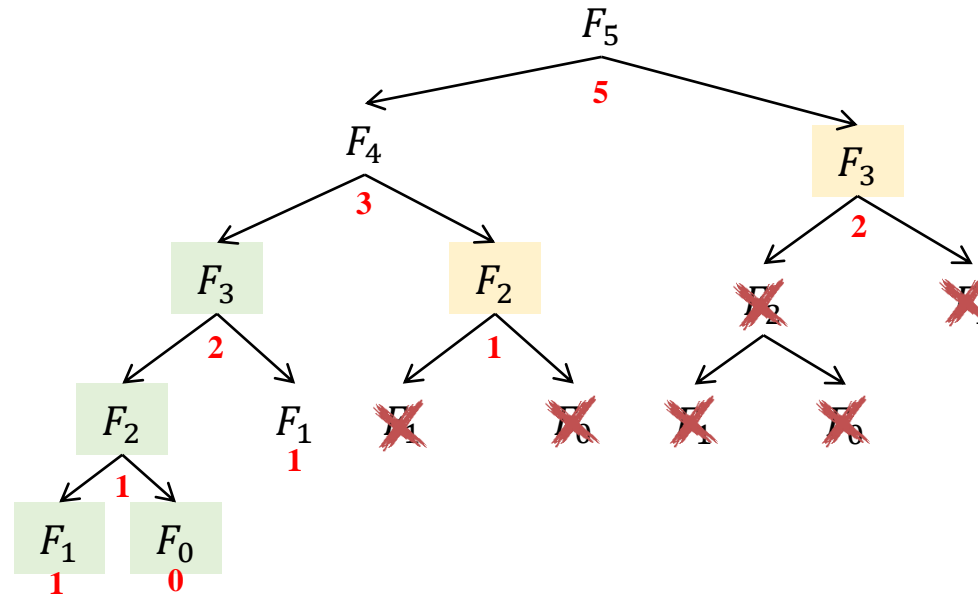


Armazenar todos os cálculos em um *cache* e quando eu encontrar um ramo previamente calculado eu posso utilizar esse resultado

- Como funciona essa função?
- Suponha F_5

memorizado

cálculo direto



- *Bottom-up (tabulation)*
 - Parte-se da solução trivial e desenvolvem-se os cálculos até encontrar a solução ótima do problema
 - Utiliza-se soluções iterativas
 - Calcula-se os subproblemas menores e aumenta-se a complexidade com o passar do tempo
 - No Fibonacci, começamos calculando $fib(1)$, depois $fib(2)$, $fib(3)$, e assim por diante até $fib(n)$
 - Observe que, nesta abordagem, nós sabemos que, na n -ésima iteração, $fib(n - 1)$, já foi resolvido, logo não precisamos verificá-la novamente, como na abordagem *top-down*. Portanto, em alguns casos, a *memoização* fica implícita

$$fib(0) \rightarrow fib(1) \rightarrow fib(2) \rightarrow \dots \rightarrow fib(n - 2) \rightarrow fib(n - 1) \rightarrow fib(n)$$

- *Bottom-up (tabulation)*
 - Parte-se da solução trivial e desenvolvem-se os cálculos até encontrar a solução ótima do problema
 - Utiliza-se soluções iterativas
 - Calcula-se os subproblemas menores e aumenta-se a complexidade com o passar do tempo
 - No Fibonacci, começamos calculando $fib(1)$, depois $fib(2)$, $fib(3)$, e assim por diante até $fib(n)$
 - Observe que, nesta abordagem, nós sabemos que, na n -ésima iteração, $fib(n - 1)$, já foi resolvido, logo não precisamos verificá-la novamente, como na abordagem *top-down*. Portanto, em alguns casos, a *memoização* fica implícita

$$fib(0) \rightarrow fib(1) \rightarrow fib(2) \rightarrow \dots \rightarrow fib(n - 2) \rightarrow fib(n - 1) \rightarrow fib(n)$$

- *Bottom-up (tabulation)*
 - Parte-se da solução trivial e desenvolvem-se os cálculos até encontrar a solução ótima do problema
 - **Utiliza-se soluções iterativas**
 - Calcula-se os subproblemas menores e aumenta-se a complexidade com o passar do tempo
 - No Fibonacci, começamos calculando $fib(1)$, depois $fib(2)$, $fib(3)$, e assim por diante até $fib(n)$
 - Observe que, nesta abordagem, nós sabemos que, na n -ésima iteração, $fib(n - 1)$, já foi resolvido, logo não precisamos verificá-la novamente, como na abordagem *top-down*. Portanto, em alguns casos, a *memoização* fica implícita

$$fib(0) \rightarrow fib(1) \rightarrow fib(2) \rightarrow \dots \rightarrow fib(n - 2) \rightarrow fib(n - 1) \rightarrow fib(n)$$

- *Bottom-up (tabulation)*
 - Parte-se da solução trivial e desenvolvem-se os cálculos até encontrar a solução ótima do problema
 - Utiliza-se soluções iterativas
 - **Calcula-se os subproblemas menores e aumenta-se a complexidade com o passar do tempo**
 - No Fibonacci, começamos calculando $fib(1)$, depois $fib(2)$, $fib(3)$, e assim por diante até $fib(n)$
 - Observe que, nesta abordagem, nós sabemos que, na n -ésima iteração, $fib(n - 1)$, já foi resolvido, logo não precisamos verifica-la novamente, como na abordagem *top-down*. Portanto, em alguns casos, a *memoização* fica implícita

$$fib(0) \rightarrow fib(1) \rightarrow fib(2) \rightarrow \dots \rightarrow fib(n - 2) \rightarrow fib(n - 1) \rightarrow fib(n)$$

- *Bottom-up (tabulation)*
 - Parte-se da solução trivial e desenvolvem-se os cálculos até encontrar a solução ótima do problema
 - Utiliza-se soluções iterativas
 - Calcula-se os subproblemas menores e aumenta-se a complexidade com o passar do tempo
 - No Fibonacci, começamos calculando *fib(1)*, depois *fib(2)*, *fib(3)*, e assim por diante até *fib(n)*
 - Observe que, nesta abordagem, nós sabemos que, na *n*-ésima iteração, *fib(n - 1)*, já foi resolvido, logo não precisamos verificá-la novamente, como na abordagem *top-down*. Portanto, em alguns casos, a *memoização* fica implícita

$$fib(0) \rightarrow fib(1) \rightarrow fib(2) \rightarrow \dots \rightarrow fib(n - 2) \rightarrow fib(n - 1) \rightarrow fib(n)$$

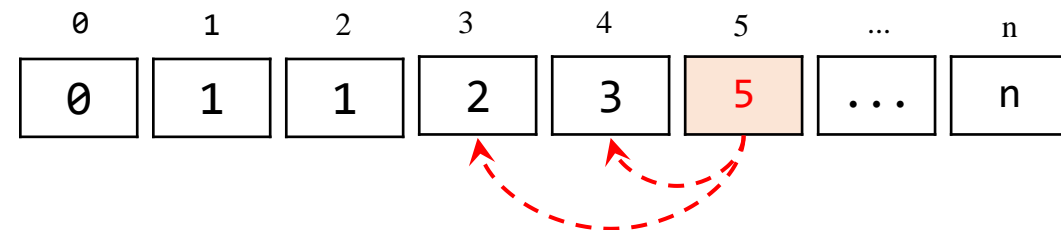
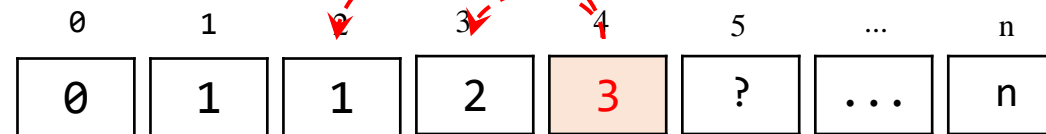
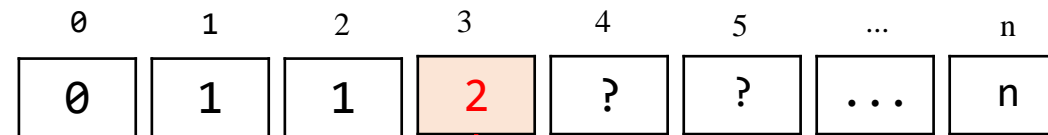
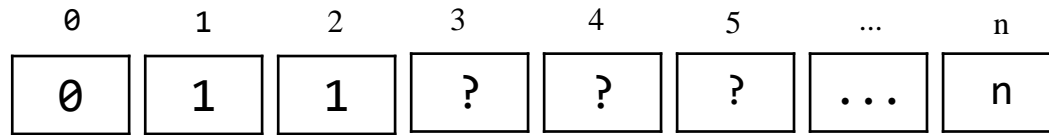
- *Bottom-up (tabulation)*
 - Parte-se da solução trivial e desenvolvem-se os cálculos até encontrar a solução ótima do problema
 - Utiliza-se soluções iterativas
 - Calcula-se os subproblemas menores e aumenta-se a complexidade com o passar do tempo
 - No Fibonacci, começamos calculando $fib(1)$, depois $fib(2)$, $fib(3)$, e assim por diante até $fib(n)$
 - Observe que, nesta abordagem, nós sabemos que, na n -ésima iteração, $fib(n - 1)$, já foi resolvido, logo não precisamos verificá-la novamente, como na abordagem *top-down*. Portanto, em alguns casos, a *memoização* fica implícita

$$fib(0) \rightarrow fib(1) \rightarrow fib(2) \rightarrow \dots \rightarrow fib(n - 2) \rightarrow fib(n - 1) \rightarrow fib(n)$$

```
def fibonacciBottomUp(n, table = {}):  
    table[0] = 0  
    table[1] = 1  
    for cont in range(2, n + 1):  
        table[cont] = table[cont - 1] + table[cont - 2]  
    return table[n]
```

Solução trivial

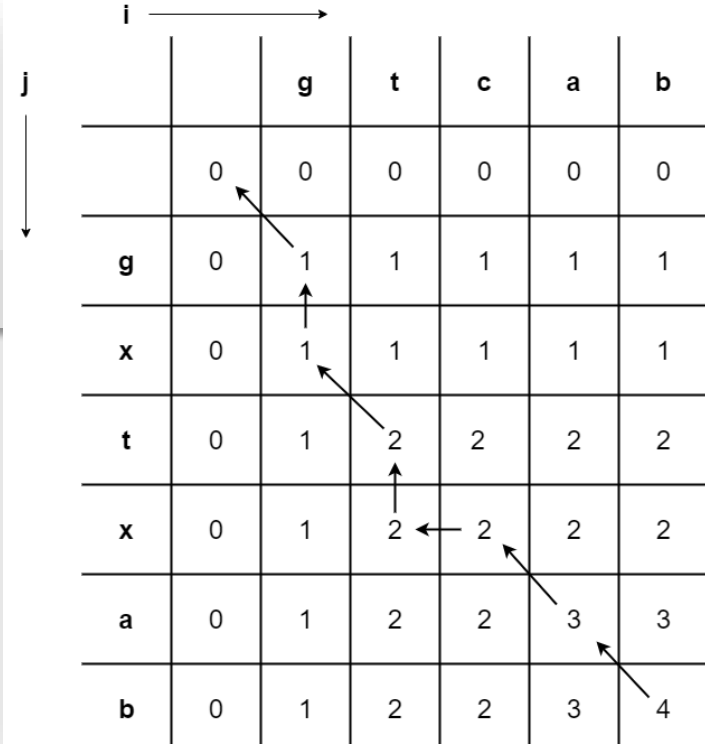
Busca na tabela de memoização



Isso é programação dinâmica?

```
def fibonacci(n):  
    if n in {0, 1}:  
        return(n)  
    fn_2 = 0  
    fn_1 = 1  
    for i in range(2, n + 1):  
        fn = fn_1 + fn_2  
        fn_2 = fn_1  
        fn_1 = fn  
    return(fn)
```

- As respostas obtidas são armazenadas em uma **matriz de memoização**



		i →					
j ↓			g	t	c	a	b
		0	0	0	0	0	0
	g	0	1	1	1	1	1
	x	0	1	1	1	1	1
	t	0	1	2	2	2	2
	x	0	1	2	2	2	2
	a	0	1	2	2	3	3
	b	0	1	2	2	3	4

- As respostas obtidas são armazenadas em uma **matriz de memoização**

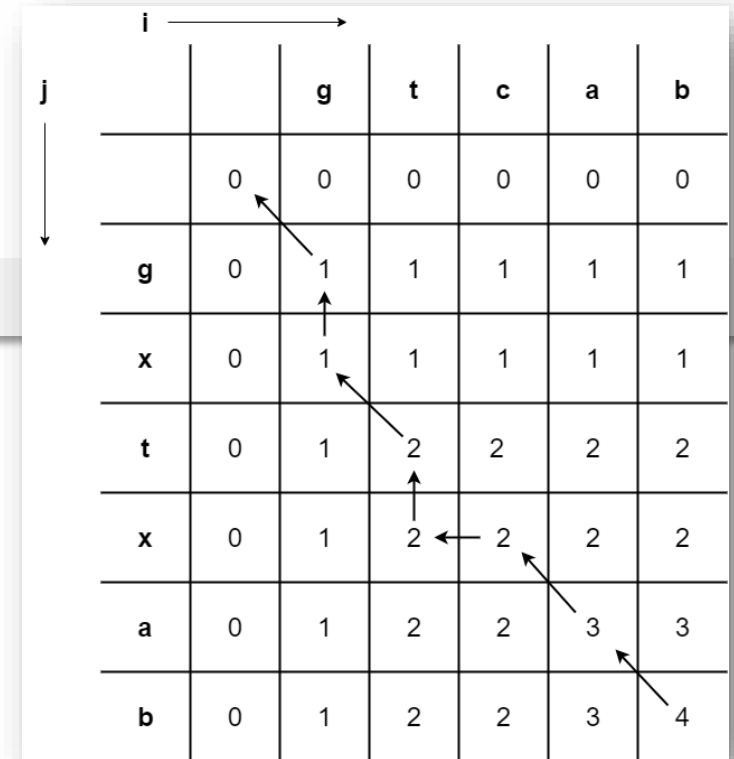
Em alguns casos temos um **vetor de memoização**

		i →					
j ↓			g	t	c	a	b
		0	0	0	0	0	0
	g	0	1	1	1	1	1
	x	0	1	1	1	1	1
	t	0	1	2	2	2	2
	x	0	1	2	2	2	2
	a	0	1	2	2	3	3
	b	0	1	2	2	3	4

- As respostas obtidas são armazenadas em uma **matriz de memoização**

Em alguns casos temos um **vetor de memoização**

Na verdade a **matriz de memorização** é compactada em um **vetor de memoização**



		i →					
j ↓			g	t	c	a	b
		0	0	0	0	0	0
	g	0	1	1	1	1	1
	x	0	1	1	1	1	1
	t	0	1	2	2	2	2
	x	0	1	2	2	2	2
	a	0	1	2	2	3	3
	b	0	1	2	2	3	4

Porque usamos essas estruturas de dados?

- ???????

- As respostas obtidas são armazenadas em uma **matriz de memoização**

Em alguns casos temos um **vetor de memoização**

Na verdade a **matriz de memorização** é compactada em um **vetor de memoização**

		i →					
j ↓			g	t	c	a	b
		0	0	0	0	0	0
	g	0	1	1	1	1	1
	x	0	1	1	1	1	1
	t	0	1	2	2	2	2
	x	0	1	2	2	2	2
	a	0	1	2	2	3	3
	b	0	1	2	2	3	4

Porque usamos essas estruturas de dados?

- **Acesso direto: Tempo constante $O(1)$**

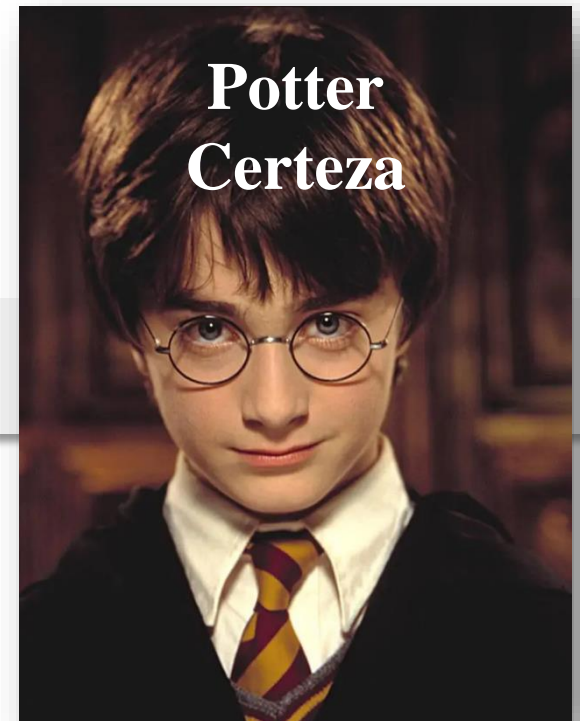
- As respostas obtidas são armazenadas em uma **matriz de memoização**

Em alguns casos temos um **vetor de memoização**

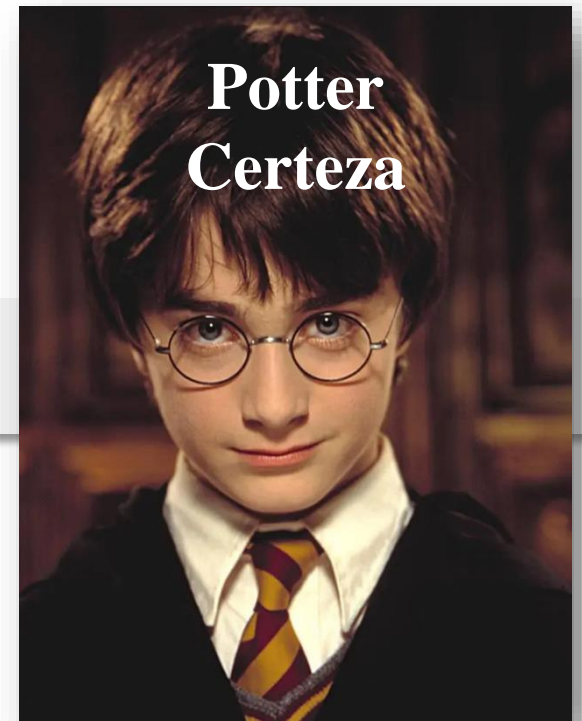
Na verdade a **matriz de memorização** é compactada em um **vetor de memoização**

		i →					
j ↓			g	t	c	a	b
		0	0	0	0	0	0
	g	0	1	1	1	1	1
	x	0	1	1	1	1	1
	t	0	1	2	2	2	2
	x	0	1	2	2	2	2
	a	0	1	2	2	3	3
	b	0	1	2	2	3	4

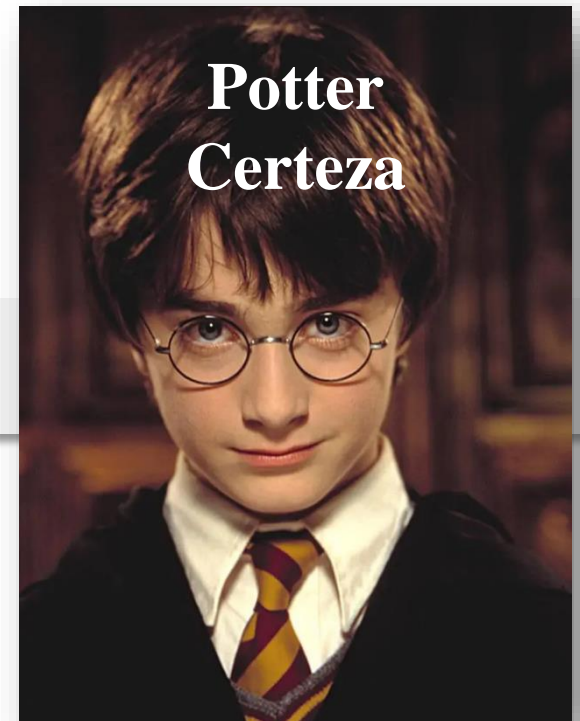
- No geral, existem as duas soluções *bottom-up* e *top-down* para um mesmo problema
- Com certeza a mais utilizada é a *bottom-up*, porque?



- No geral, existem as duas soluções *bottom-up* e *top-down* para um mesmo problema
- Com certeza a mais utilizada é a *bottom-up*, porque?



- No geral, existem as duas soluções *bottom-up* e *top-down* para um mesmo problema
- Com certeza a mais utilizada é a *bottom-up*, porque?



- No geral, existem as duas soluções *bottom-up* e *top-down* para um mesmo problema
- Com certeza a mais utilizada é a *bottom-up*, porque?

É mais complexo de se perceber a solução, porém, não usar recursão é mais simples para o pensamento “de um programador” e torna mais simples analisar a complexidade.



- Usualmente, apresentamos a **solução** por **Programação Dinâmica** por meio de uma **relação de recorrência**
- Temos dois cenários gerais
 - Problemas naturalmente definidos como uma relação de recorrência, como o Fibonacci
 - Problemas possíveis de serem modelados usando relação de recorrência, mesmo que a definição não seja naturalmente uma recorrência

- Usualmente, apresentamos a **solução** por **Programação Dinâmica** por meio de uma **relação de recorrência**
- Temos dois cenários gerais
 - Problemas naturalmente definidos como uma relação de recorrência, como o Fibonacci
 - Problemas possíveis de serem modelados usando relação de recorrência, mesmo que a definição não seja naturalmente uma recorrência

$$F_n \begin{cases} 1 & \text{se } n = 1 \text{ ou } n = 2, \\ F_{n-1} + F_{n-2} & \text{caso contrário.} \end{cases}$$

- Usualmente, apresentamos a **solução** por **Programação Dinâmica** por meio de uma **relação de recorrência**
- **Temos dois cenários gerais**
 - Problemas naturalmente definidos como uma relação de recorrência, como o Fibonacci
 - Problemas possíveis de serem modelados usando relação de recorrência, mesmo que a definição não seja naturalmente uma recorrência

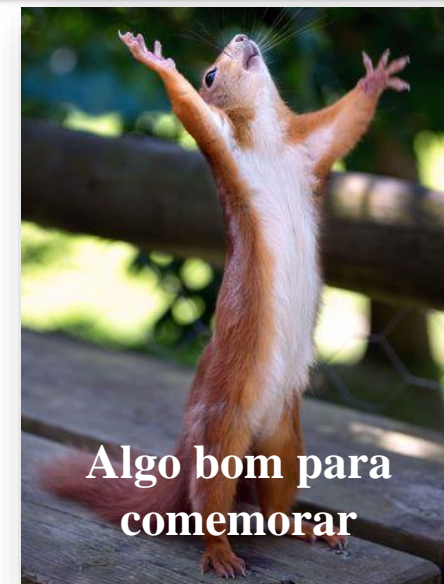


Mas não se desespere!!!

Nem sempre é fácil visualizar ou modelar uma recorrência, porém, uma vez definida, a implementação é simples.

- Usualmente, apresentamos a **solução** por **Programação Dinâmica** por meio de uma **relação de recorrência**
- Temos dois cenários gerais
 - Problemas naturalmente definidos como uma relação de recorrência, como o Fibonacci
 - Problemas possíveis de serem modelados usando relação de recorrência, mesmo que a definição não seja naturalmente uma recorrência

Vejam o lado bom, já vimos recorrência



- Programação Dinâmica – Parte 2

Obrigado



Dúvidas

Email: alanvalejo@ufscar.br

Acessar o fórum no Moodle