SIN5013 - Análise de Algoritmos e Estruturas de Dados

Algoritmos gulosos, tentativa e erro, e programação dinâmica

Prof. Flávio Luiz Coutinho

Problema do troco

- Solução gulosa: não garante a solução ótima (!)

Problema do troco

- Solução gulosa: não garante a solução ótima (!)

- Solução tentativa e erro: complexidade exponencial (!!!)

Problema do troco

- Solução gulosa: não garante a solução ótima (!)

- Solução tentativa e erro: complexidade exponencial (!!!)

- Haveria alguma alternativa que:

Problema do troco

- Solução gulosa: não garante a solução ótima (!)

- Solução tentativa e erro: complexidade exponencial (!!!)

- Haveria alguma alternativa que:
    - garantisse a solução ótima?

Problema do troco

- Solução gulosa: não garante a solução ótima (!)

- Solução tentativa e erro: complexidade exponencial (!!!)

- Haveria alguma alternativa que:
    - garantisse a solução ótima?
    - tivesse complexidade menor do que exponencial?

Problema do troco

- Solução gulosa: não garante a solução ótima (!)

- Solução tentativa e erro: complexidade exponencial (!!!)

- Haveria alguma alternativa que:
    - garantisse a solução ótima?
    - tivesse complexidade menor do que exponencial?
    - SIM! Com programação dinâmica

Programação dinâmica

Técnica de programação em que:

- um problema é quebrado em subproblemas que são resolvidos recursivamente.

Programação dinâmica

Técnica de programação em que:

- um problema é quebrado em subproblemas que são resolvidos recursivamente.

Mas além disso:

- há sobreposição e repetição de subproblemas

Programação dinâmica

Técnica de programação em que:

- um problema é quebrado em subproblemas que são resolvidos recursivamente.

Mas além disso:

- há sobreposição e repetição de subproblemas.
- o problema deve ter **subestrutura ótima:**

Programação dinâmica

Técnica de programação em que:

- um problema é quebrado em subproblemas que são resolvidos recursivamente.

Mas além disso:

- há sobreposição e repetição de subproblemas.
- o problema deve ter **subestrutura ótima:** uma solução ótima para o problema pode ser construída a partir das soluções ótimas dos subproblemas.

Programação dinâmica

Duas abordagens:

- *Bottom-up*: resolve-se os subproblemas menores primeiro, e a partir das soluções, os subproblemas maiores vão sendo resolvidos, de forma iterativa. Durante este processo, costuma-se gerar uma tabela com os resultados parciais.

Programação dinâmica

Solução *bottom-up* para o problema do troco, com n = 6 e v = { 4, 3, 1}:

```
S(0)   = {}
```

```
S(1)   = { 1 } + S(0)   = { 1 }
```

```
S(2)   = { 1 } + S(1)   = { 1, 1 }
```

Programação dinâmica

Solução *bottom-up* para o problema do troco, com n = 6 e v = { 4, 3, 1}:

```
S(3)    = { 1 } + S(2)    = { 1, 1, 1 }
        = { 3 } + S(0)    = { 3 }


S(4)    = { 1 } + S(3)    = { 1, 3 }
        = { 3 } + S(1)    = { 3, 1 }
        = { 4 } + S(0)    = { 4 }
```

Programação dinâmica

Solução *bottom-up* para o problema do troco, com n = 6 e v = { 4, 3, 1}:

```
S(5)   = { 1 } + S(4)   = { 1, 4 }
       = { 3 } + S(2)   = { 3, 1, 1 }
       = { 4 } + S(1)   = { 4, 1 }

S(6)   = { 1 } + S(5)   = { 1, 1, 4 }
       = { 3 } + S(3)   = { 3, 3 }
       = { 4 } + S(2)   = { 4, 1, 1 }
```

Programação dinâmica

Duas abordagens:

- *Top-down*: a forma como costumamos resolver um problema recursivamente. Para resolver um problema maior, geramos instâncias menores do mesmo tipo de problema, que são resolvidas recursivamente. A partir das soluções das instâncias menores, determinamos a solução do problema inicial. Um fator crucial envolve armazenar os resultados dos subproblemas já computados em algum tipo de estrutura (tabela), para que possam ser reutilizados quando algum subproblema precisar ser resolvido novamente.

Programação dinâmica

Solução *top-down* para o problema do troco, com n = 6 e v = { 4, 3, 1}:

```
S(6)   = { 1 } + S(5)   = { 1, 1, 4 }
       = { 3 } + S(3)   = { 3, 3 }
       = { 4 } + S(2)   = { 4, 1, 1 }
```

* Sempre que o resultado de um subproblema é calculado pela primeira vez, guardamos esse resultado em uma tabela para reaproveitá-lo mais tarde.

Um outro exemplo: sequência de Fibonacci

Sequência: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

Outra forma de definir a sequência:

$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ F_n = F_{n-1} + F_{n-2} \end{cases}$$

Um outro exemplo: sequência de Fibonacci

Duas implementações:

- iterativa (*bottom-up*)
- recursiva (*top-down*)

- simulação e considerações

Um outro exemplo: sequência de Fibonacci

f(4)

f(4)

STACK

Um outro exemplo: sequência de Fibonacci

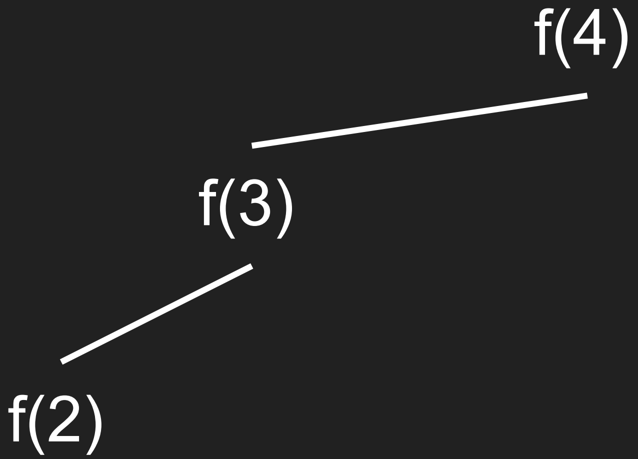
f(4)

f(3)

f(3)
f(4)

STACK

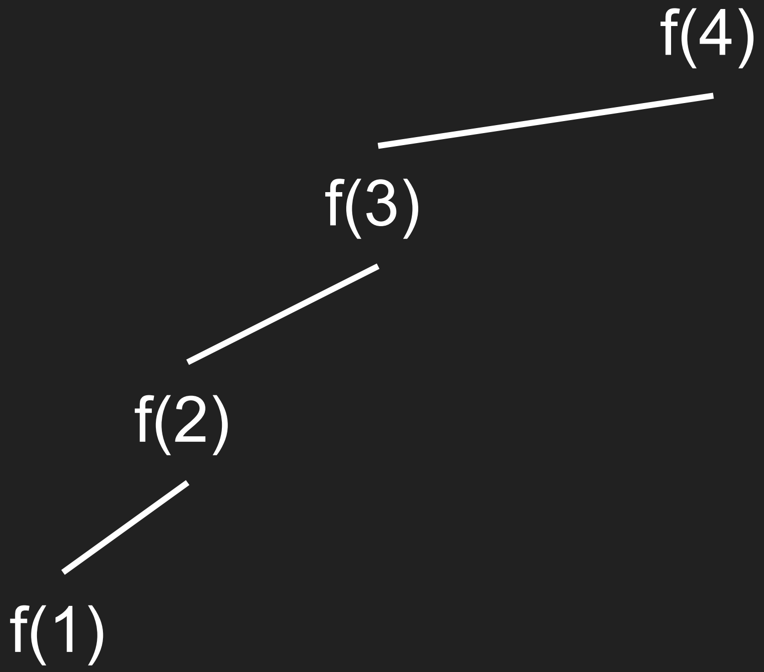
Um outro exemplo: sequência de Fibonacci



f(2)
f(3)
f(4)

STACK

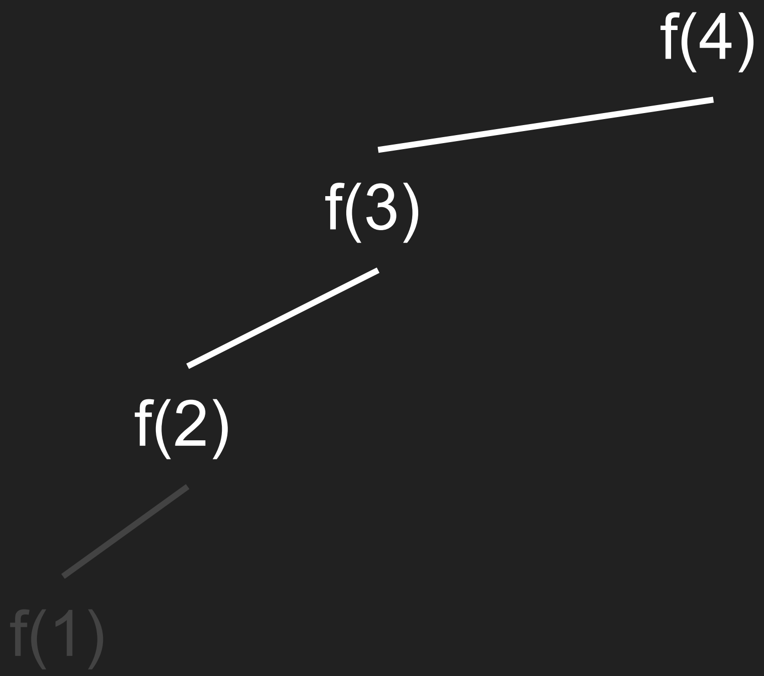
Um outro exemplo: sequência de Fibonacci



f(1)
f(2)
f(3)
f(4)

STACK

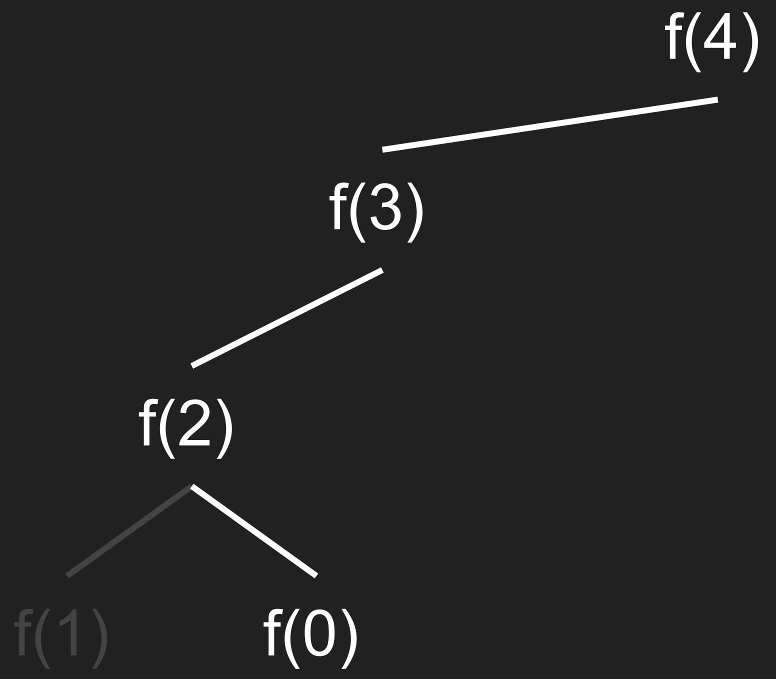
Um outro exemplo: sequência de Fibonacci



f(2)
f(3)
f(4)

STACK

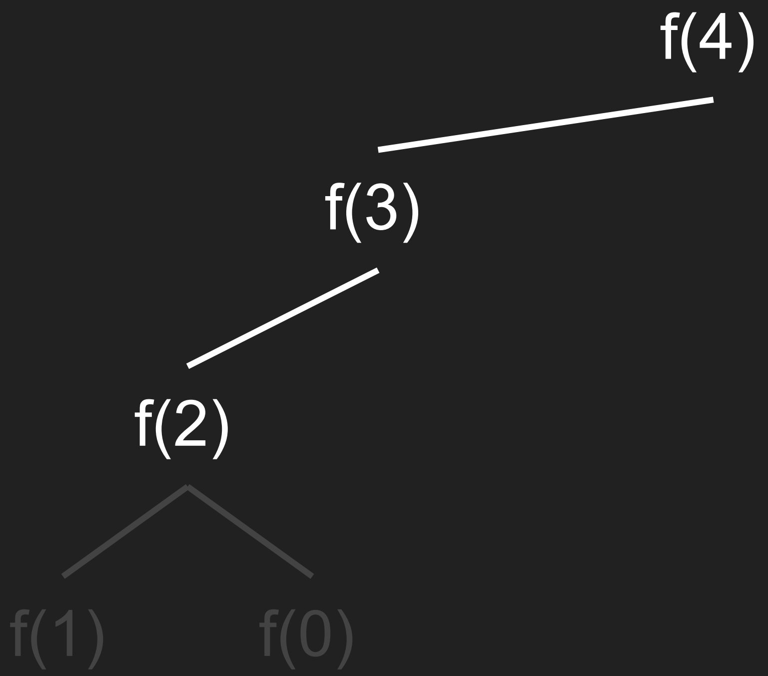
Um outro exemplo: sequência de Fibonacci



f(0)
f(2)
f(3)
f(4)

STACK

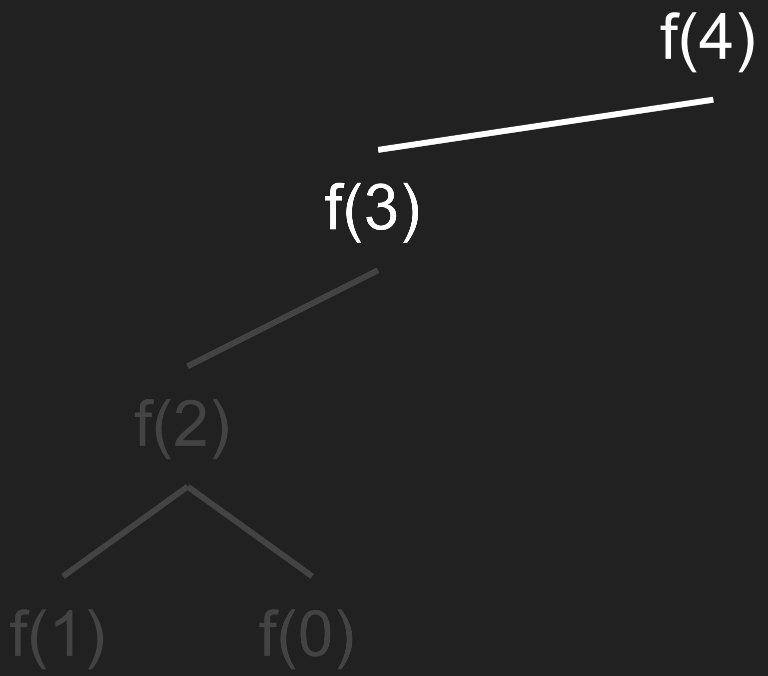
Um outro exemplo: sequência de Fibonacci



f(2)
f(3)
f(4)

STACK

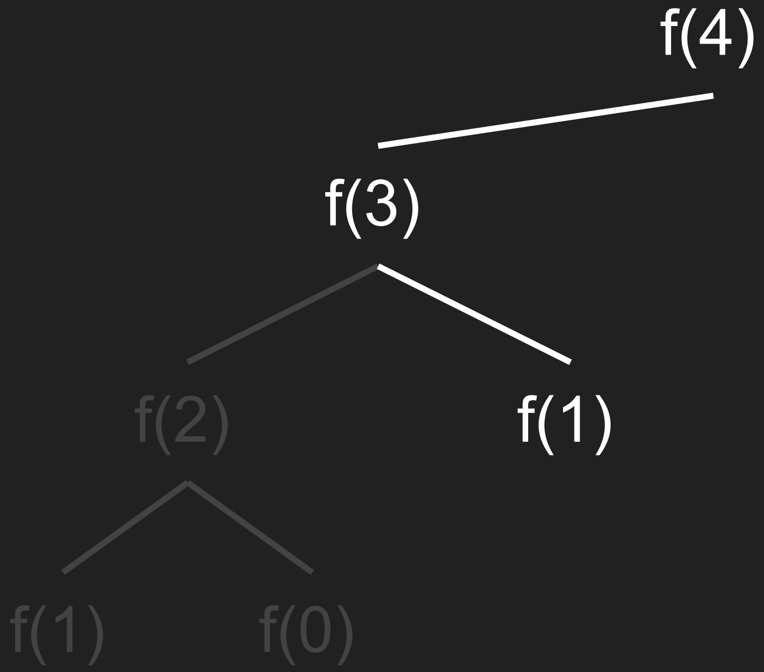
Um outro exemplo: sequência de Fibonacci



f(3)
f(4)

STACK

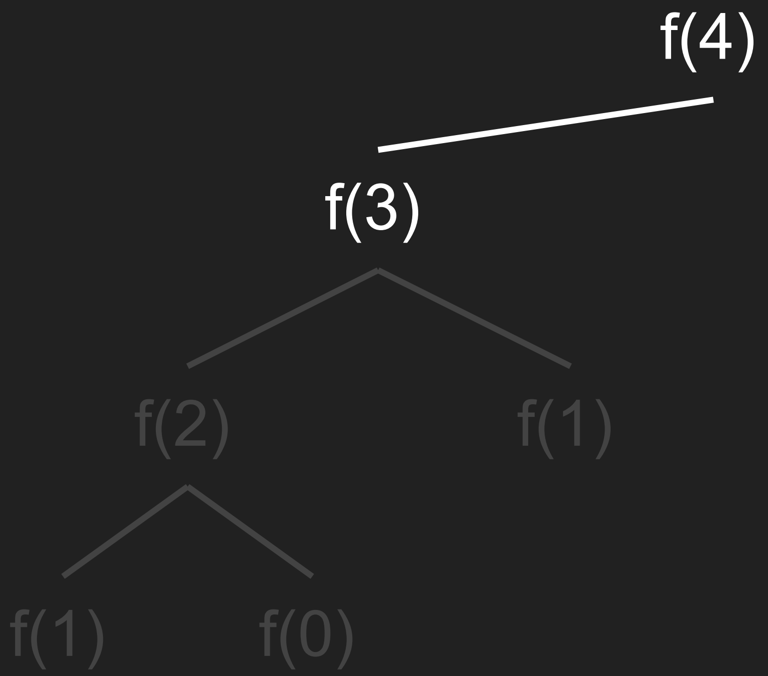
Um outro exemplo: sequência de Fibonacci



f(1)
f(3)
f(4)

STACK

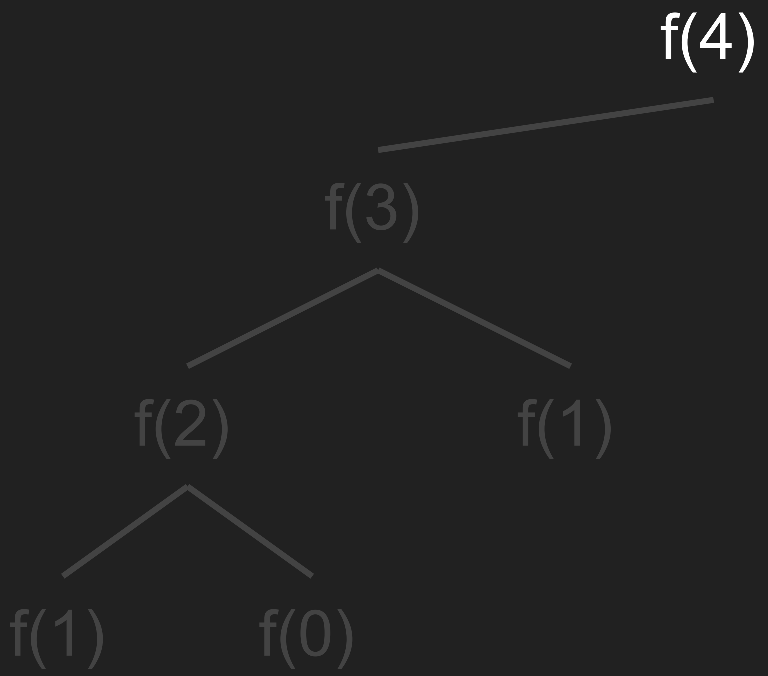
Um outro exemplo: sequência de Fibonacci



f(3)
f(4)

STACK

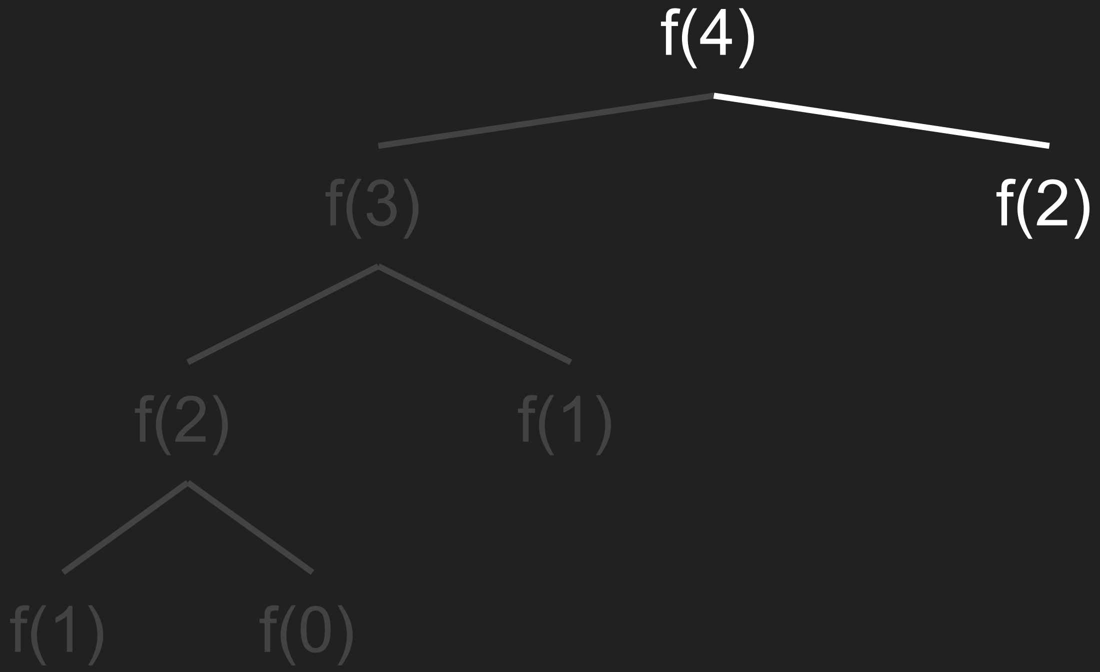
Um outro exemplo: sequência de Fibonacci



f(4)

STACK

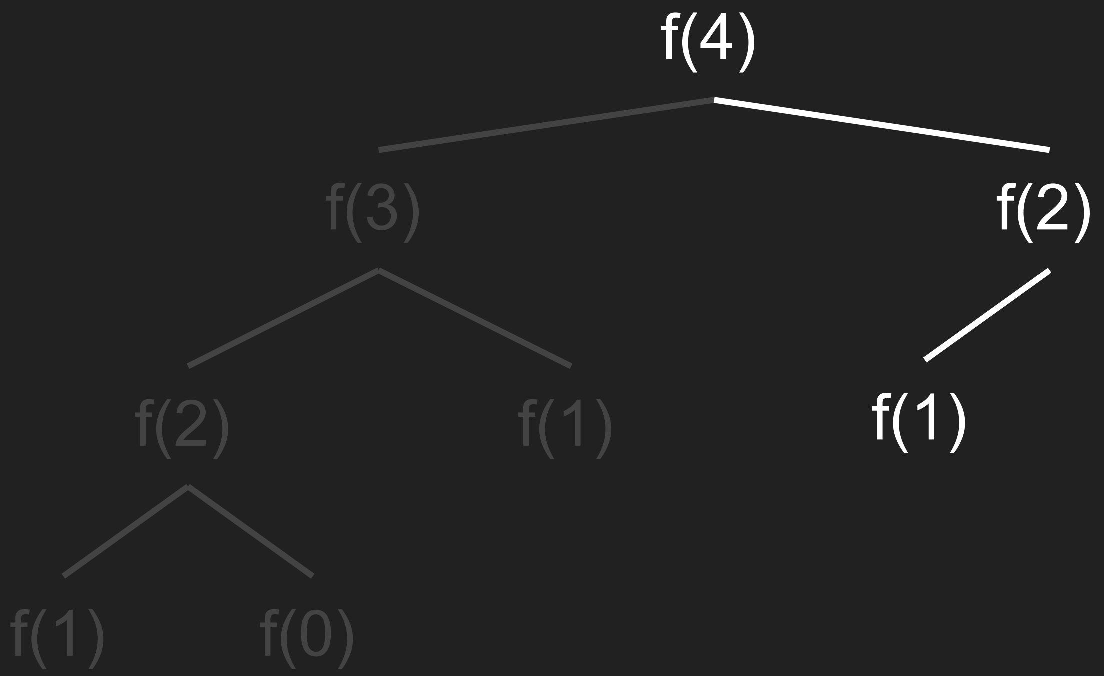
Um outro exemplo: sequência de Fibonacci



f(2)
f(4)

STACK

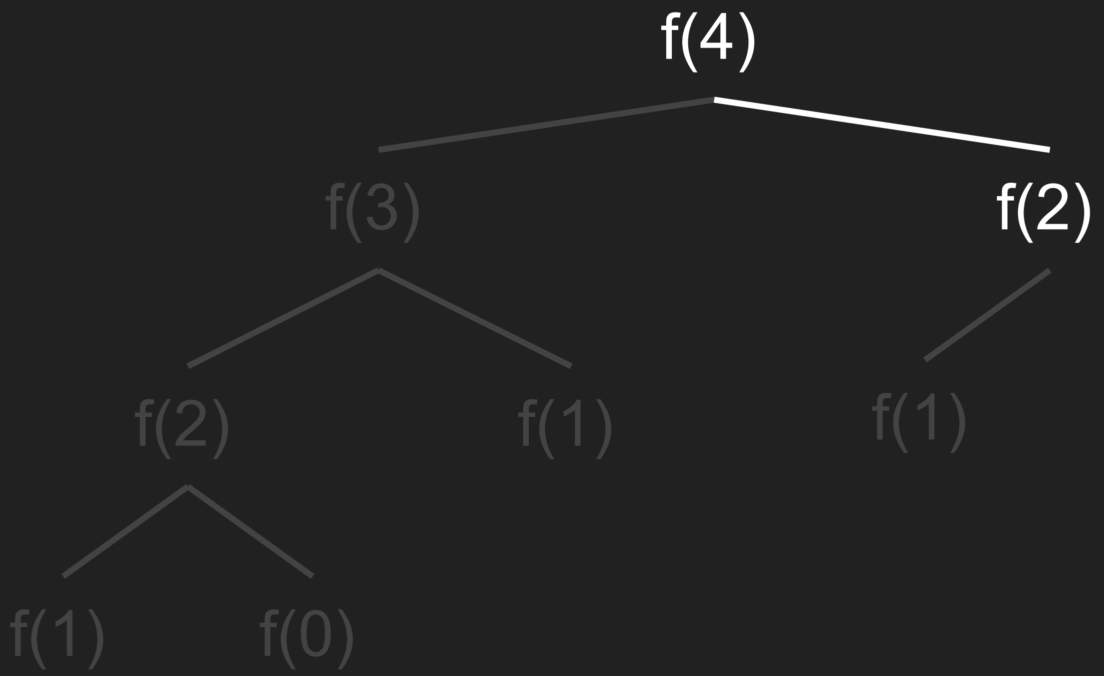
Um outro exemplo: sequência de Fibonacci



f(1)
f(2)
f(4)

STACK

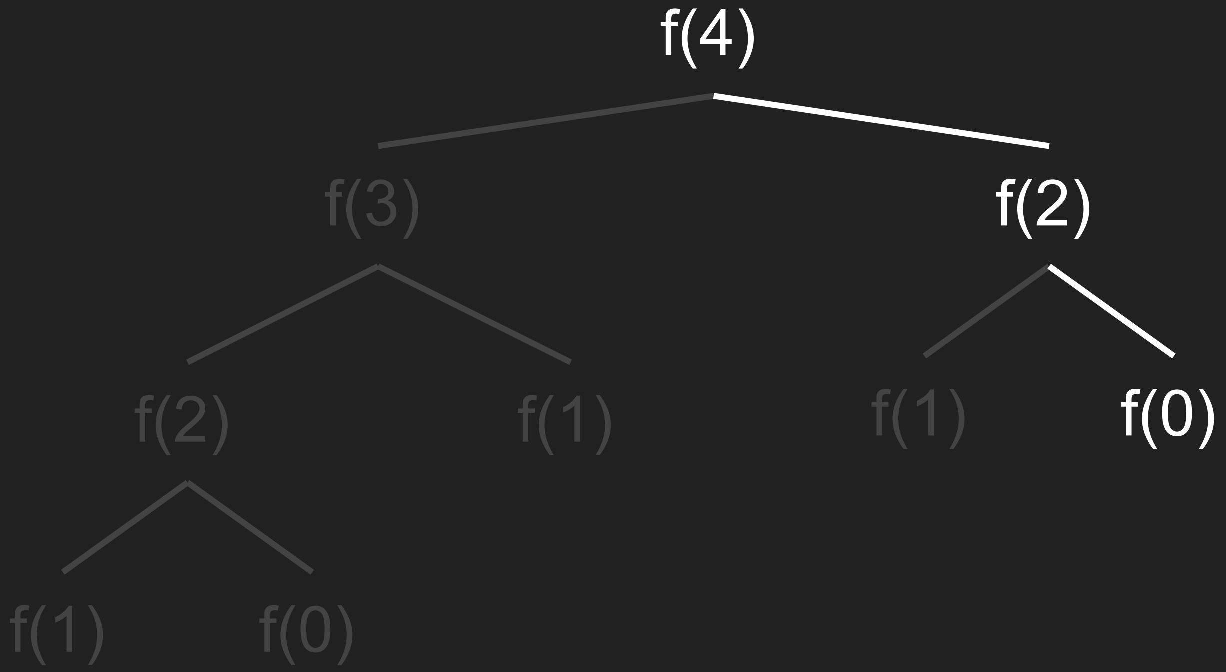
Um outro exemplo: sequência de Fibonacci



f(2)
f(4)

STACK

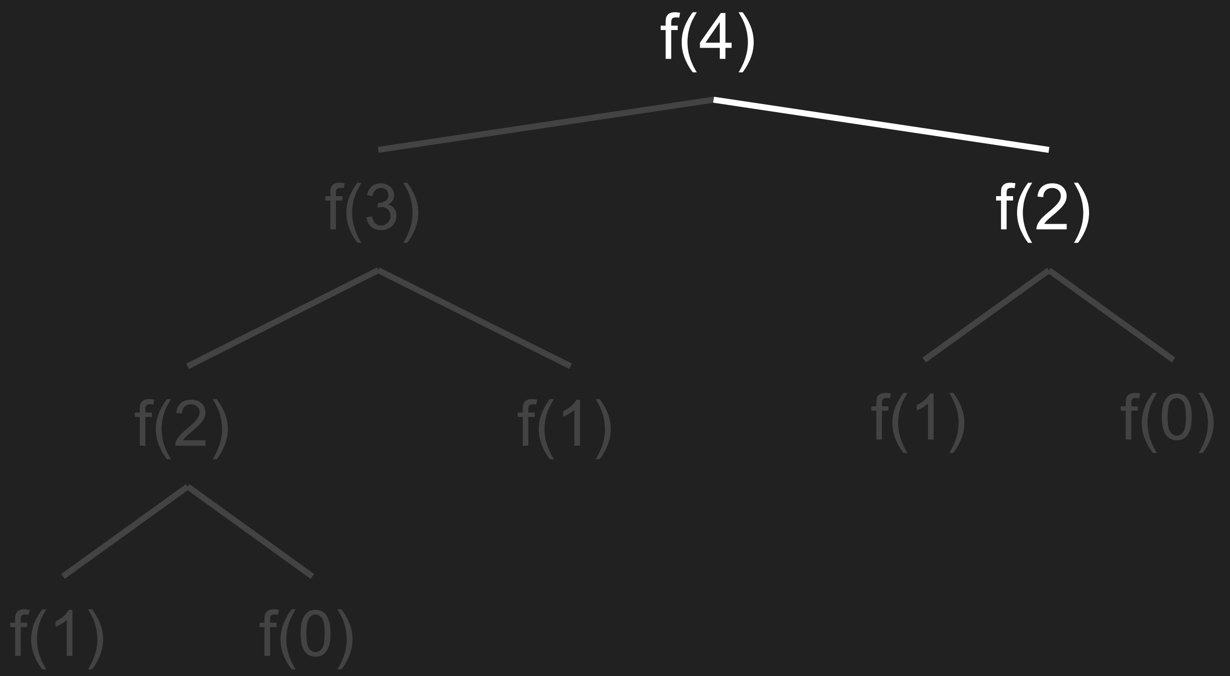
Um outro exemplo: sequência de Fibonacci



f(0)
f(2)
f(4)

STACK

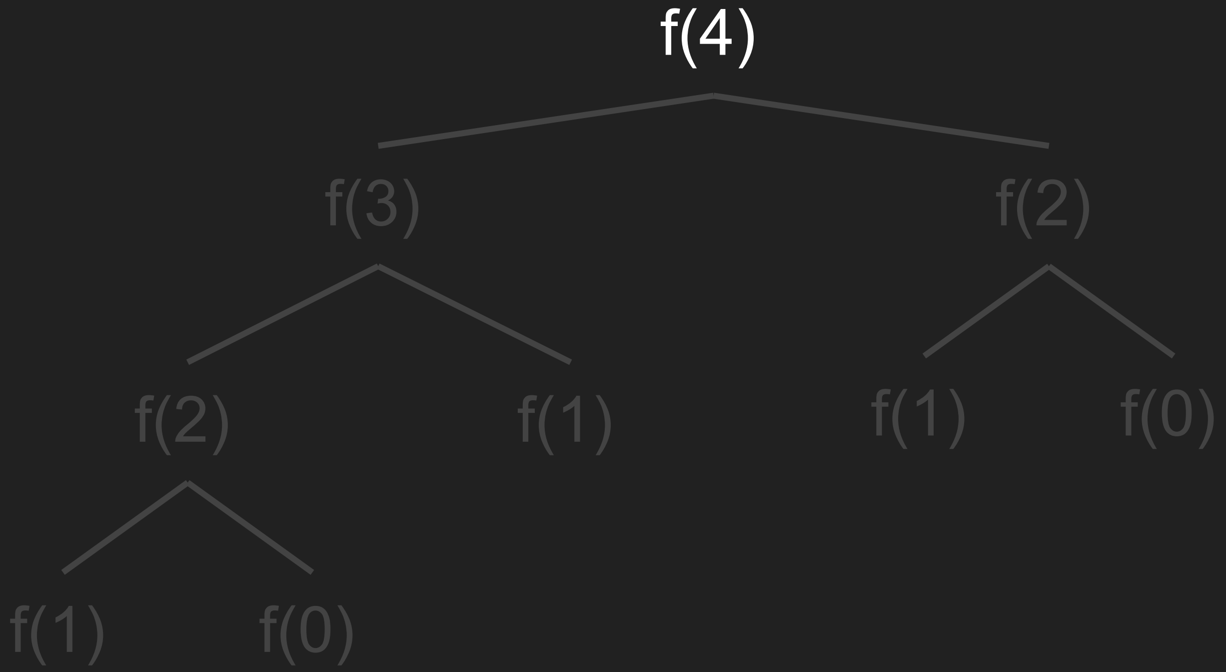
Um outro exemplo: sequência de Fibonacci



f(2)
f(4)

STACK

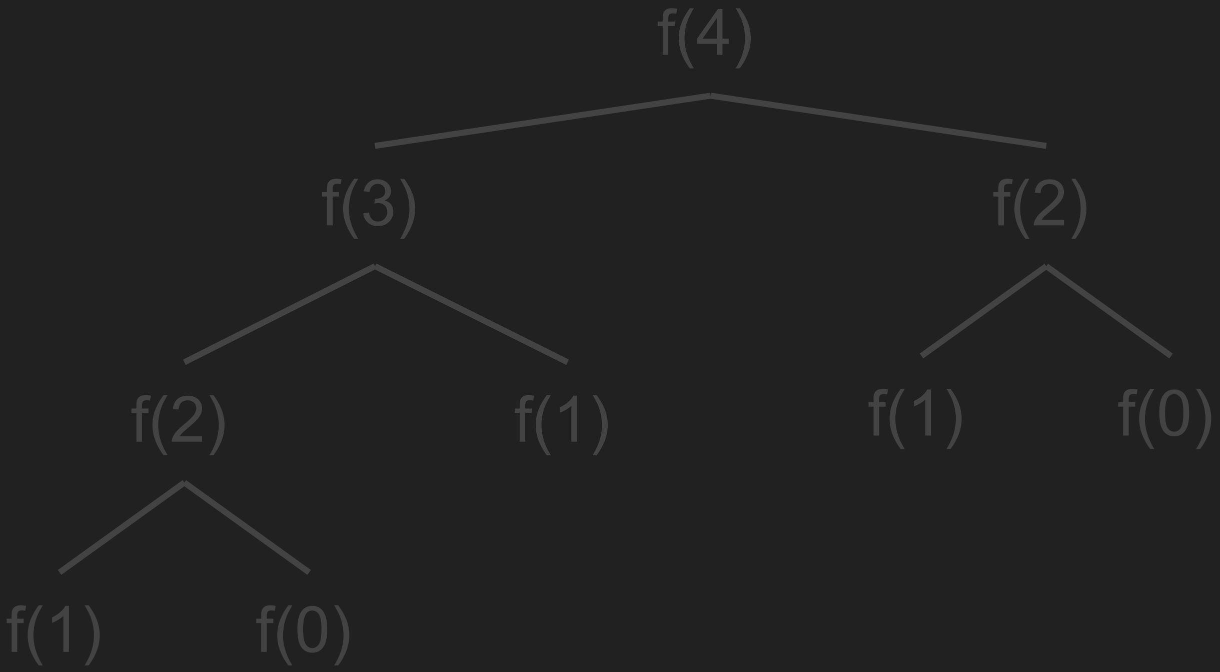
Um outro exemplo: sequência de Fibonacci



f(4)

STACK

Um outro exemplo: sequência de Fibonacci



STACK

Sequência de Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

Considerações:

- Versão recursiva é com certeza mais elegante, compacta e fácil de entender.
- Mas também parece ser menos eficiente.
- Quão pior ela é em relação à versão iterativa?
- Alguns testes práticos:
    - versão recursiva se torna bem mais lenta, conforme n aumenta.
    - Mas por quê?
    - E o quão mais lenta ela é?

Sequência de Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

Algoritmo iterativo:

- Laço: bloco executa (n - 1) vezes, com cada execução levando a ms.
- Demais operações: executadas uma vez só, levando todas elas b ms.

- $T(n) = a(n - 1) + b = an - a + b$
- $T(n) \sim a * n$
- $T(n) = \Theta(n)$

Sequência de Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

Algoritmo recursivo:

- $T(n) = T(n - 1) + T(n - 2) + c = ???$
- $T_{max}(n) = 2\ T_{max}(n - 1) + c = \Theta(2^n)$
- $T_{min}(n) = 2\ T_{min}(n - 2) + c = \Theta(sqrt(2)^n)$
- $T_{min}(n) <= T(n) <= T_{max}(n)$

Sequência de Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

Dá para “consertar” o desempenho ruim da versão recursiva?

- **Sim, usando programação dinâmica: as soluções dos subproblemas já resolvidos são armazenadas, e estes resultados são consultados quando necessário.**

Quão melhor ela é a versão que usa programação dinâmica?

Sequência de Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

Versão usando programação dinâmica:

- Simulação.

- #chamadas = (n + 1) + (n - 2) = 2n - 1
- $T(n) = c(2n - 1)$
- $T(n) \sim 2cn$
- $T(n) \sim \Theta(n)$ (mesma complexidade da versão iterativa)