Técnicas para Programação Competitiva

Paradigmas de Resolução de Problemas

Prof. Andrei Braga

UNIVERSIDADE FEDERAL DA FRONTEIRA SUL

Conteúdo

- Programação dinâmica
- Referências

Motivação

- Vamos estudar paradigmas de resolução de problemas comumente utilizados para tratar problemas de competições de programação
- Para ter sucesso em competições de programação, precisamos ter um bom domínio sobre estes paradigmas, sabendo usar a opção apropriada para o problema em questão

Problemas de Otimização

- Um **problema de otimização** é um problema tal que
  - cada **solução** para o problema tem um **valor** associado e
  - o objetivo é encontrar uma **solução de valor *ótimo*** – valor mínimo ou valor máximo – **ou apenas** este **valor ótimo**
- Também chamamos uma solução de valor ótimo de **solução ótima**
- Um problema de otimização pode ter mais de uma solução ótima – por isso, usamos o termo ***uma*** solução ótima, em vez de *a* solução ótima, para o problema

Problemas de Otimização

- Um **problema de otimização** é um problema tal que
  - cada **solução** para o problema tem um **valor** associado e
  - o objetivo é encontrar uma **solução de valor *ótimo*** – valor mínimo ou valor máximo – **ou apenas** este **valor ótimo**

- Exemplo:
  Problema da Mochila: Dado um conjunto de itens com seus valores e pesos, qual é o maior valor total de itens que um ladrão consegue carregar em sua mochila (sem ultrapassar a capacidade de peso da mochila)?

Problemas de Otimização

- Um algoritmo para resolver um problema de otimização geralmente realiza uma sequência de passos, a cada passo fazendo uma ou mais escolhas
- Algoritmos deste tipo podem ser classificados em categorias bastante conhecidas
- Duas destas categorias são **algoritmos gulosos** e **algoritmos de programação dinâmica**

- Vale ressaltar que algoritmos de programação dinâmica também comumente se aplicam a **problemas de contagem** do tipo “Conte de quantas maneiras é possível fazer …”

Problema

- **Problema:** Dados um valor *n* em centavos e moedas de 4, 3 e 1 centavo (suponha que estas moedas existem 😁), determine o menor número possível de moedas tal que a soma das moedas seja *n*. Não há restrições quanto a usar várias moedas de um mesmo valor.

- Anteriormente, para resolver uma versão deste problema onde os valores das moedas são 50, 10, 5 e 1, usamos o algoritmo guloso ao lado
- Este algoritmo resolve o problema acima?
  Não!
  Se $n = 6$, a solução (3, 3) com 2 moedas é ótima, mas o algoritmo retorna 3 (o valor da solução (4, 1, 1))

MinNumMoedas(*n*)

1. Se n == 0:
2.   Retorne 0
3. Escolha uma moeda de maior valor *c* tal que $c \leq n$
4. Retorne (1 + MinNumMoedas(*n* - *c*))

Solução – Busca completa (recursiva – backtracking)

- **Problema:** Dados um valor *n* em centavos e moedas de 4, 3 e 1 centavo (suponha que estas moedas existem 😁), determine o menor número possível de moedas tal que a soma das moedas seja *n*. Não há restrições quanto a usar várias moedas de um mesmo valor.

- Vamos escrever uma função recursiva *f*(*n*) que
  - considera todas as soluções possíveis e
  - retorna o valor ótimo

Solução – Busca completa (recursiva – backtracking)

- Função recursiva *f*(*n*) que
  - considera todas as soluções possíveis e
  - retorna o valor ótimo

Ideia:

f(6) $\xrightarrow{1}$ (1), *f*(5) $\xrightarrow{1}$ (1, 1), *f*(4) $\xrightarrow{1}$ (1, 1, 1), *f*(3) $\xrightarrow{1}$ (1, 1, 1, 1), *f*(2) $\xrightarrow{1}$ (1, 1, 1, 1, 1), *f*(1) $\xrightarrow{1}$ (1, 1, 1, 1, 1, 1), *f*(0)

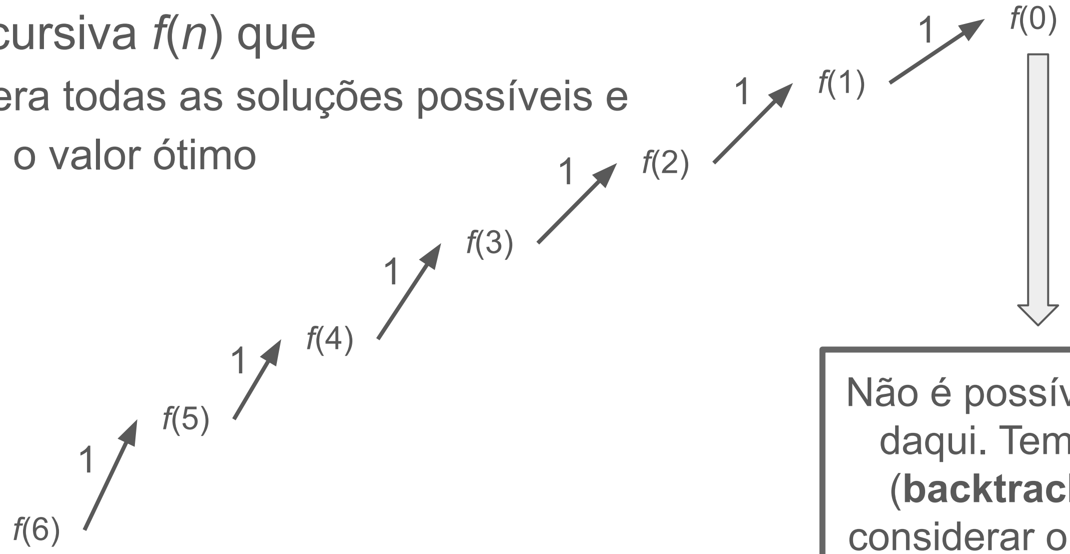
Não é possível ir adiante a partir daqui. Temos que retroceder (**backtrack**) para conseguir considerar outras possibilidades

Moedas: 4, 3 e 1
n: 6

Solução – Busca completa (recursiva – backtracking)

- Função recursiva *f*(*n*) que
  - considera todas as soluções possíveis e
  - retorna o valor ótimo

Ideia:



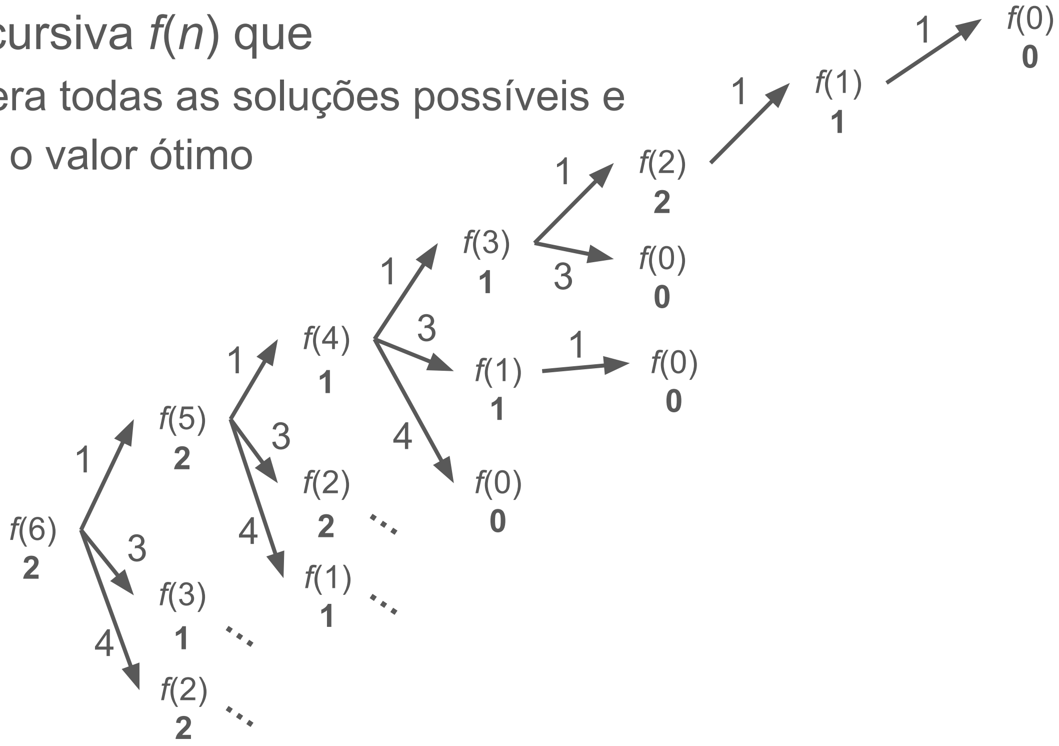
Não é possível ir adiante a partir daqui. Temos que retroceder (**backtrack**) para conseguir considerar outras possibilidades

Moedas: 4, 3 e 1
n: 6

Solução – Busca completa (recursiva – backtracking)

- Função recursiva *f*(*n*) que
  - considera todas as soluções possíveis e
  - retorna o valor ótimo

Ideia:



Moedas: 4, 3 e 1
n: 6

Solução – Busca completa (recursiva – backtracking)

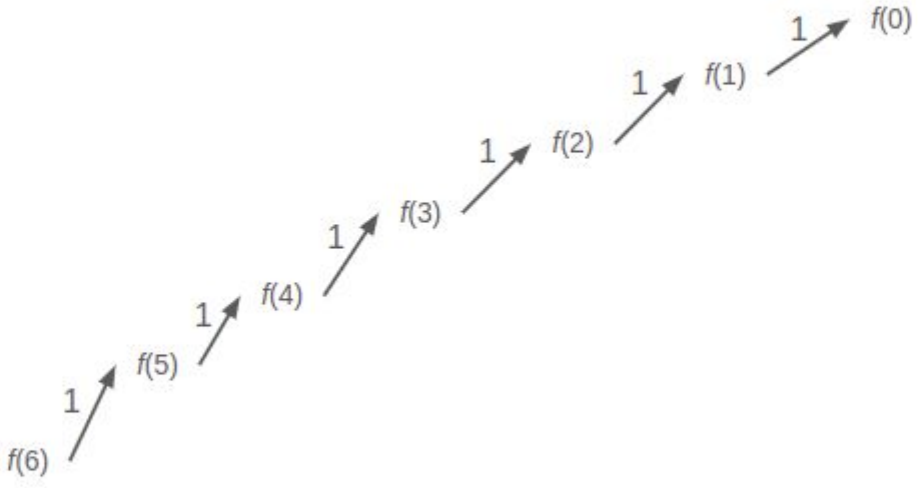
- Função recursiva *f*(*n*) que
  - considera todas as soluções possíveis e
  - retorna o valor ótimo

```
vector<int> moedas;
```

```
int f(int n) {


    for (auto m : moedas) {
        if (m <= n)
            f(n-m);
    }

}
```



Solução – Busca completa (recursiva – backtracking)

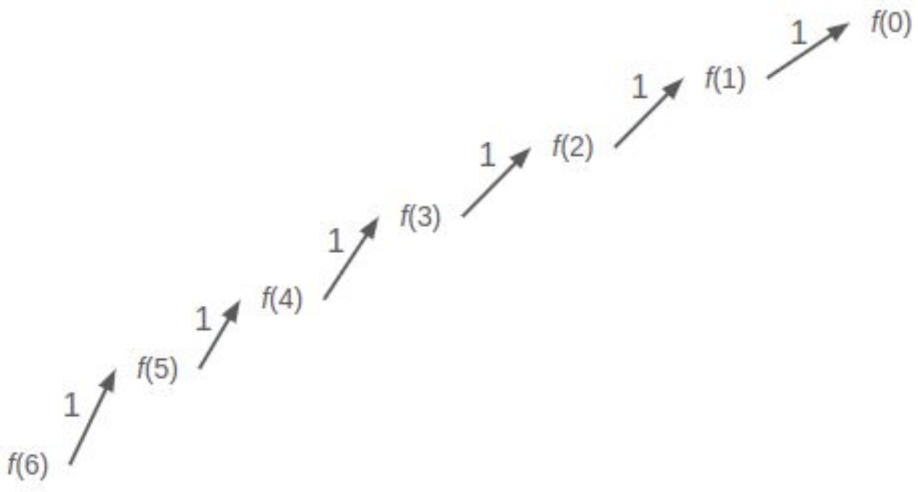
- Função recursiva *f*(*n*) que
  - considera todas as soluções possíveis e
  - retorna o valor ótimo

```
vector<int> moedas;
```

```
int f(int n) {
    if (n == 0) return 0;

    for (auto m : moedas) {
        if (m <= n)
            f(n-m);
    }

}
```



Solução – Busca completa (recursiva – backtracking)

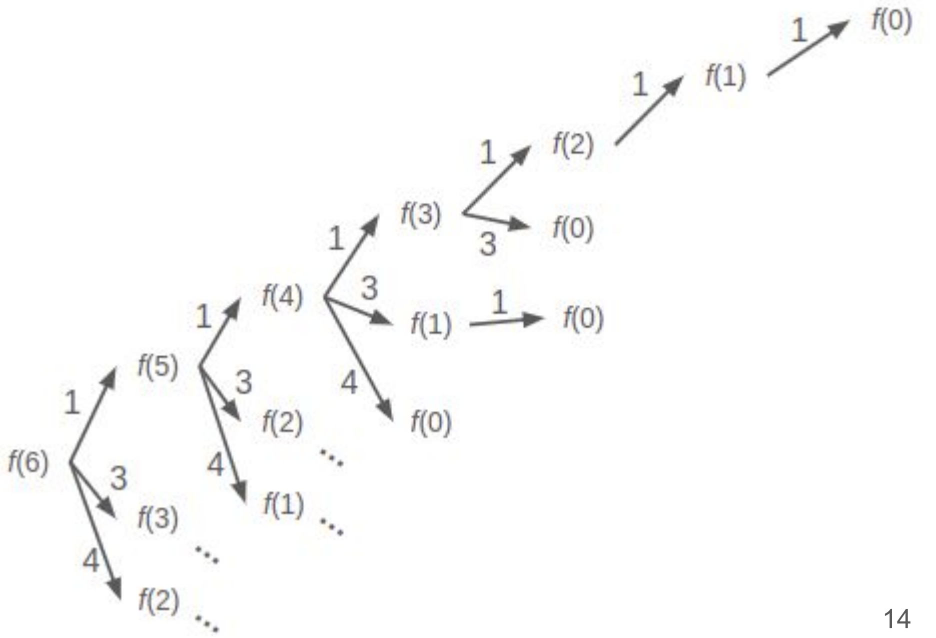
- Função recursiva *f*(*n*) que
    - considera todas as soluções possíveis e
    - retorna o valor ótimo

```
vector<int> moedas;
```

```
int f(int n) {
    if (n == 0) return 0;

    for (auto m : moedas) {
        if (m <= n)
            f(n-m);
    }

}
```

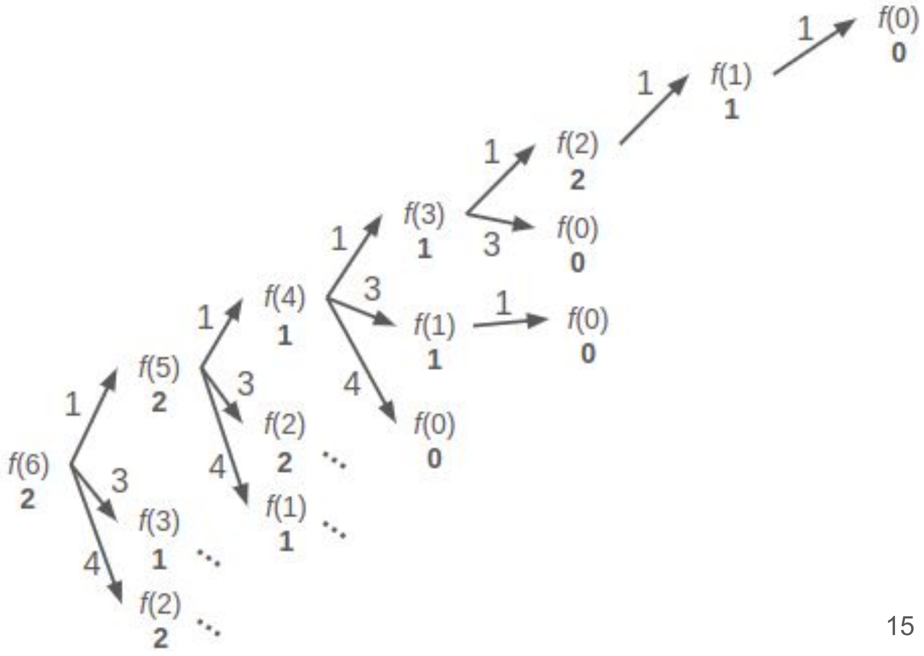


Solução – Busca completa (recursiva – backtracking)

- Função recursiva *f*(*n*) que
  - considera todas as soluções possíveis e
  - retorna o valor ótimo

```
vector<int> moedas;
```

```
int f(int n) {
    if (n == 0) return 0;
    int ret = INF;
    for (auto m : moedas) {
        if (m <= n)
            ret = min(ret, 1+f(n-m));
    }
    return ret;
}
```



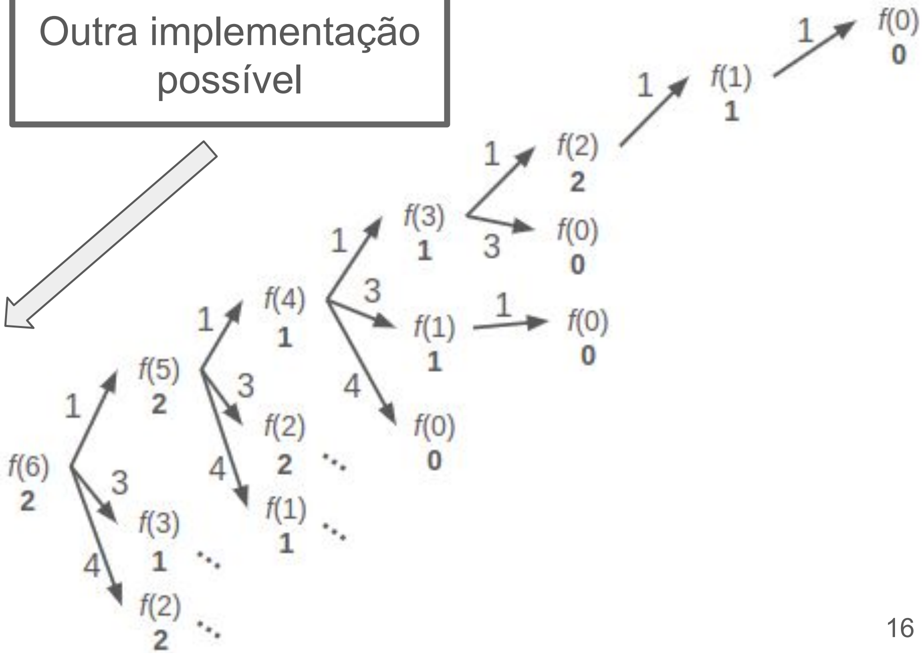
Solução – Busca completa (recursiva – backtracking)

- Função recursiva *f*(*n*) que
  - considera todas as soluções possíveis e
  - retorna o valor ótimo

```
vector<int> moedas;
```

```
int f(int n) {
    if (n < 0) return INF;
    if (n == 0) return 0;
    int ret = INF;
    for (auto m : moedas) {
        ret = min(ret, 1+f(n-m));
    }
    return ret;
}
```

Outra implementação possível



Solução – Busca completa (recursiva – backtracking)

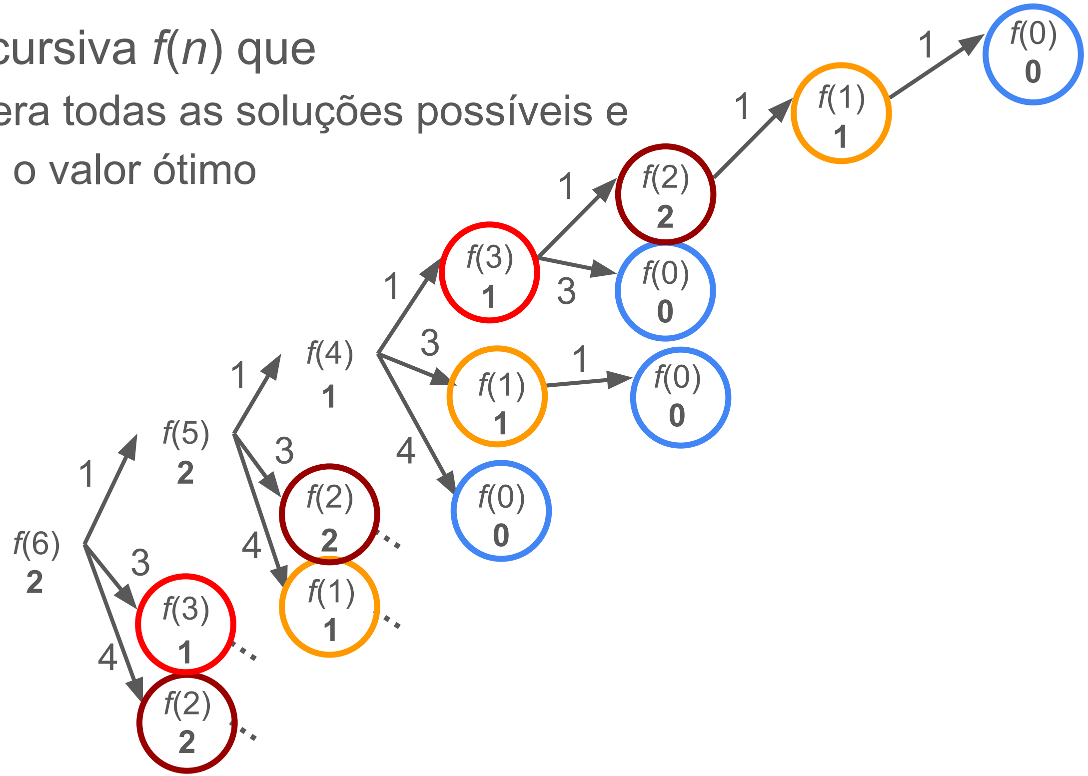
- Função recursiva *f*(*n*) que
  - considera todas as soluções possíveis e
  - retorna o valor ótimo

- Esta função determina o valor ótimo do problema, mas (como esperado) é **muito lenta**!
- Questão: É possível implementar a função de maneira mais eficiente?
  - Ponto a considerar: A função realiza algum trabalho repetido que poderia ser eliminado (ou feito de forma mais rápida)?

Solução – Busca completa (recursiva – backtracking)

- Função recursiva *f*(*n*) que
  - considera todas as soluções possíveis e
  - retorna o valor ótimo

Ideia:



Moedas: 4, 3 e 1
n: 6

Solução – Programação dinâmica

- A função anterior pode ser implementada de maneira mais eficiente com o uso da técnica de **programação dinâmica**
- No termo programação dinâmica, “programação” significa o ato de **tabular** os resultados e não o ato de escrever um código em uma linguagem de programação

Solução – Programação dinâmica

- Podemos reimplementar a função anterior de modo que, para cada possível valor de n, o valor de $f(n)$ seja computado explicitamente **apenas uma vez**
- Isto pode ser feito de uma forma simples!

Solução – Programação dinâmica

- Reimplementação da função recursiva *f*(*n*):
  - Vamos usar um vetor *memo* para armazenar os valores de *f*(*n*) que já foram computados (**tabulação** dos resultados)
    - O vetor *memo* terá *N*+1 posições, sendo *N* o valor máximo de *n*
    - Cada elemento *i* = 0, 1, …, *N* deste vetor representará o valor de *f*(*i*)
    - Vamos inicializar os elementos deste vetor com -1
  - No início de cada chamada à *f*(*n*), vamos verificar se *memo*[*n*] != -1, ou seja, se o valor de *f*(*n*) já foi computado
    - Se sim, simplesmente, vamos retornar *memo*[*n*] (não vamos computar novamente *f*(*n*))
    - Se não, vamos computar o valor de *f*(*n*) normalmente e armazenar este valor em *memo*[*n*] (em outras chamadas à função, o valor de *f*(*n*) não será computado novamente)

Solução – Programação dinâmica

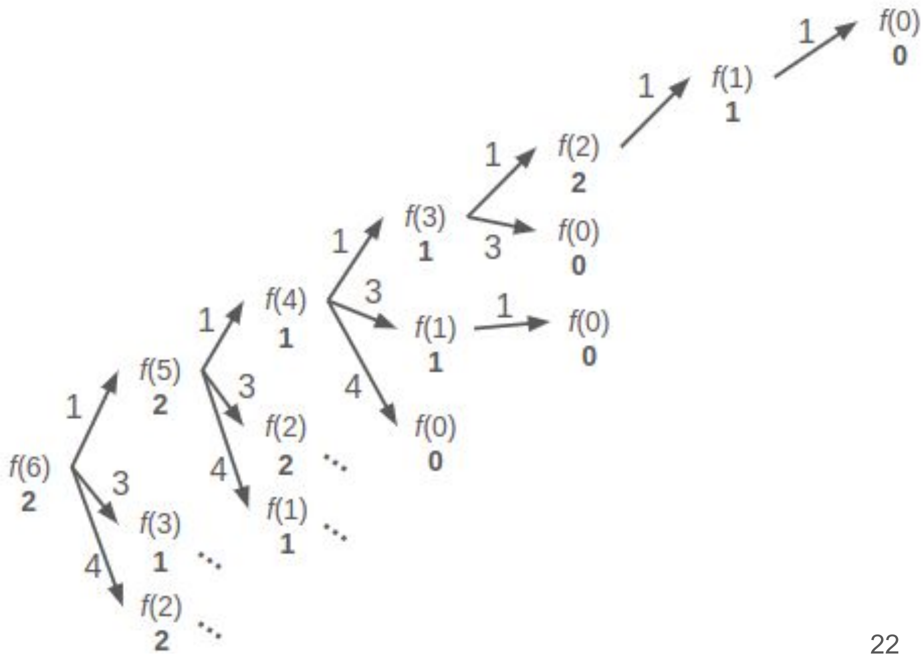
- Função recursiva *f*(*n*) que
  - considera todas as soluções possíveis e
  - retorna o valor ótimo

```
vector<int> moedas;
```

```
int f(int n) {
    if (n < 0) return INF;
    if (n == 0) return 0;

    int ret = INF;
    for (auto m : moedas) {
        ret = min(ret, 1+f(n-m));
    }

    return ret;
}
```



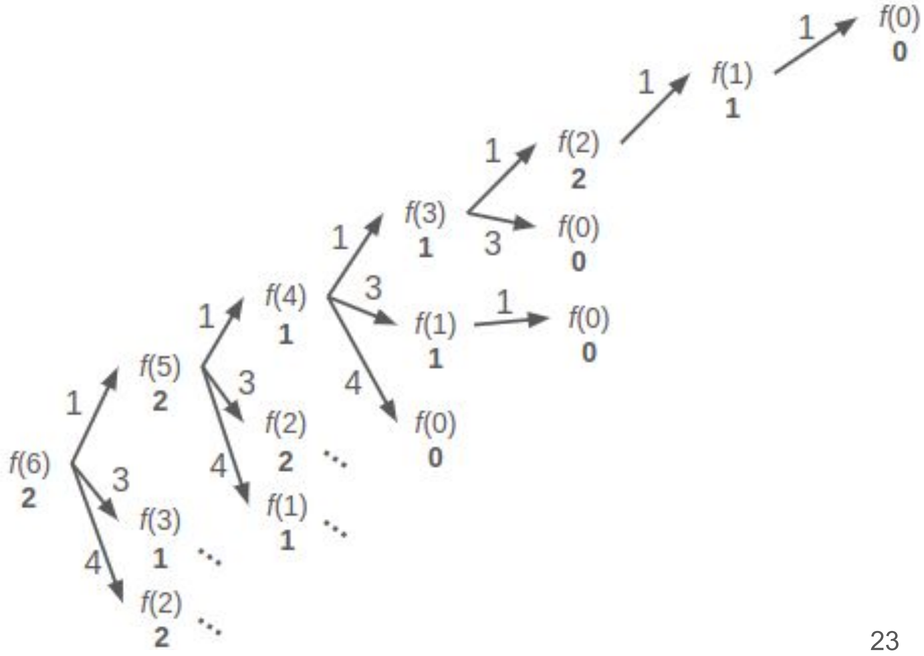
Solução – Programação dinâmica

memo :	0	1	2	1	1	2	2	-1	-1
	0	1	2	3	4	5	6	…	*N*

- Função recursiva *f*(*n*) que
  - considera todas as soluções possíveis **tabulando os resultados** e
  - retorna o valor ótimo

```
vector<int> moedas; vector<int> memo(N+1);
```

```
int f(int n) {
    if (n < 0) return INF;
    if (n == 0) return 0;
    if (memo[n] != -1) return memo[n];
    int ret = INF;
    for (auto m : moedas) {
        ret = min(ret, 1+f(n-m));
    }
    memo[n] = ret;
    return ret;
}
```



Solução – Programação dinâmica

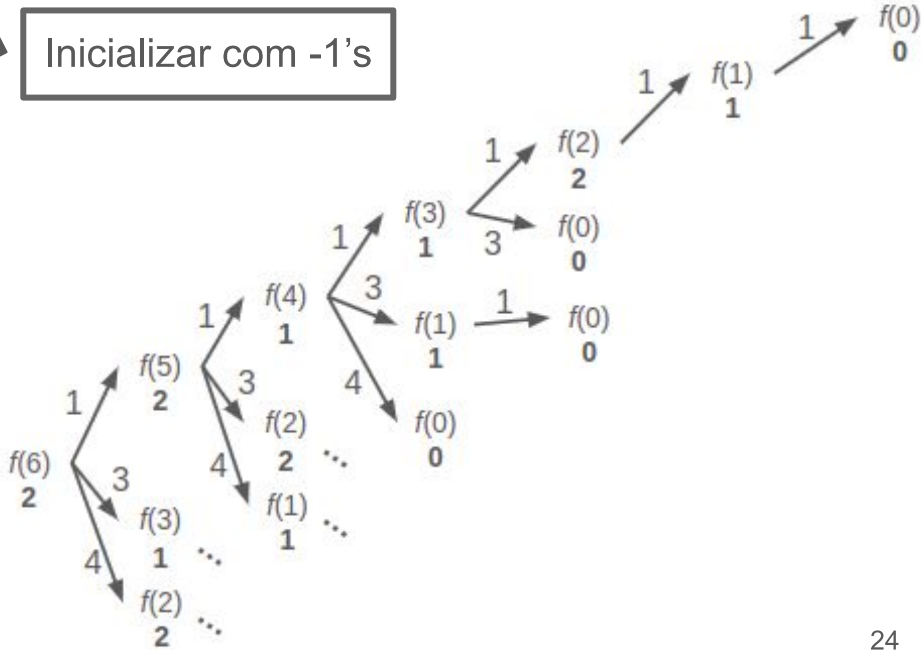
memo :	0	1	2	1	1	2	2	-1	-1
	0	1	2	3	4	5	6	…	*N*

- Função recursiva *f*(*n*) que
  - considera todas as soluções possíveis **tabulando os resultados** e
  - retorna o valor ótimo

Inicializar com -1's

```
vector<int> moedas; vector<int> memo(N+1);
```

```
int f(int n) {
    if (n < 0) return INF;
    if (n == 0) return 0;
    if (memo[n] != -1) return memo[n];
    int ret = INF;
    for (auto m : moedas) {
        ret = min(ret, 1+f(n-m));
    }
    memo[n] = ret;
    return ret;
}
```



Solução – Programação dinâmica

- A reimplementação que fizemos da função *f*(*n*) é um algoritmo de programação dinâmica do tipo **top-down**
- Também podemos fazer uma implementação iterativa que é conhecida como um algoritmo de programação dinâmica **bottom-up**

Solução – Programação dinâmica (bottom-up)

- Para escrever a versão **bottom-up**, devemos
    - nos concentrar no vetor *memo* (a tabela de resultados) e
    - determinar uma ordem de preenchimento deste vetor tal que,
        - se o valor de um elemento i depende dos valores dos elementos j_1, j_2, …, j_k,
        - então, quando formos preencher o elemento i, os valores dos elementos j_1, j_2, …, j_k já devem estar preenchidos

`memo:`

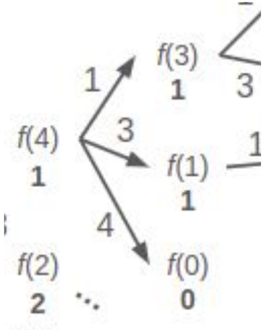
0	1	2	3	4	5	6	…	N
0	1	2	1	1	2	2	-1	-1

Solução – Programação dinâmica (bottom-up)

- Versão **bottom-up**, preenchimento do vetor *memo*:
  - se o valor de um elemento *i* depende dos valores dos elementos j_1, j_2, …, j_k,
  - então, quando formos preencher o elemento *i*, os valores dos elementos j_1, j_2, …, j_k já estejam preenchidos

`memo:`

0	1	2	1	1	2	2	-1	-1
0	1	2	3	4	5	6	…	*N*



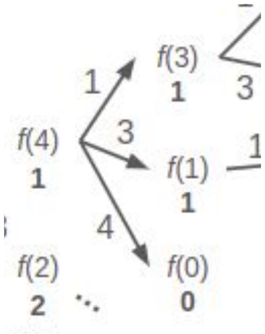
- Exemplo:
  - O valor do elemento *memo*[4] (ou seja, o valor de *f*(4)) depende dos valores dos elementos *memo*[3], *memo*[1] e *memo*[0] (ou seja, os valores de *f*(3), *f*(1) e *f*(0))
  - Então, precisamos preencher *memo*[0], *memo*[1] e *memo*[3] antes de preencher *memo*[4]

Solução – Programação dinâmica (bottom-up)

- Versão **bottom-up**, preenchimento do vetor *memo*:
  - se o valor de um elemento *i* depende dos valores dos elementos $j_1, j_2, \ldots, j_k$,
  - então, quando formos preencher o elemento *i*, os valores dos elementos $j_1, j_2, \ldots, j_k$ já estejam preenchidos

`memo:`

0	1	2	3	4	5	6	…	*N*
0	1	2	1	1	2	2	-1	-1



- Neste caso, existe uma regra geral simples:
  O valor de cada elemento *memo*[*i*] depende dos valores de elementos que garantidamente tem índice menor que *i*
- Então, podemos preencher os elementos do vetor em **ordem crescente** dos seus índices

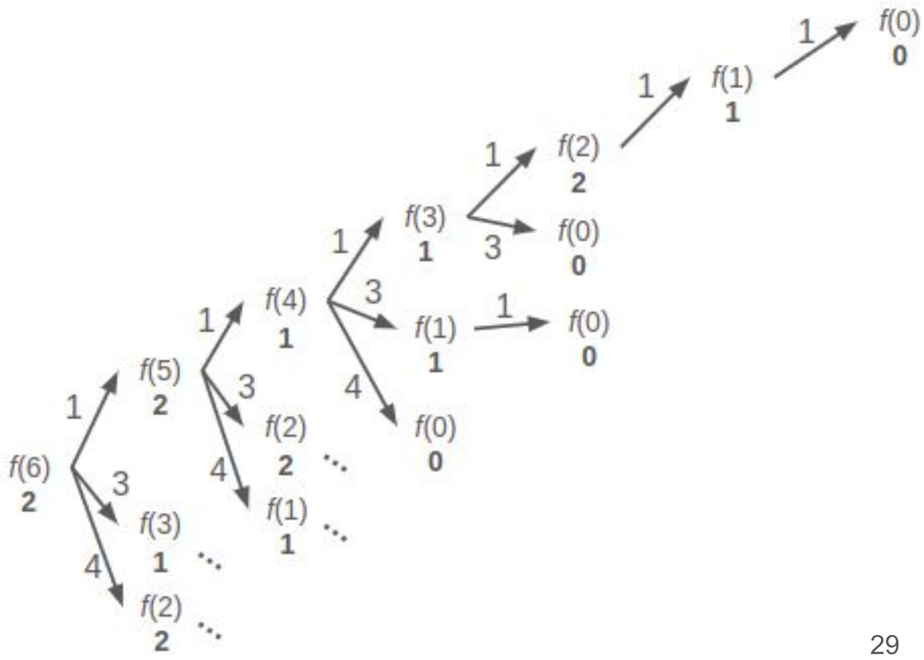
Solução – Programação dinâmica (bottom-up)

- Função iterativa *f*(*n*) que
  - considera todas as soluções possíveis **tabulando os resultados** e
  - retorna o valor ótimo

memo :	0	1	2	1	1	2	2	-1	-1
	0	1	2	3	4	5	6	…	*N*

```
vector<int> moedas; vector<int> memo(N+1);
```

```
int f(int n) {
    memo[0] = 0;
    for (int i = 1; i <= n; i++) {
        memo[i] = INF;
        for (auto m : moedas) {
            if (m <= i)
                memo[i] = min(memo[i],
                    1+memo(i-m));
        }
    }
    return memo[n];
}
```



Solução – Programação dinâmica

- As duas versões estudadas do algoritmo de programação dinâmica (top-down e bottom-up) são eficientes
- Em ambos os casos, a complexidade é O(*nk*), onde *k* é o número de moedas
- No entanto, em muitos casos, será **mais vantajoso** usar a versão **bottom-up** pois
  - as constantes presentes na complexidade desta versão são menores e
  - pode não ser viável usar a versão top-down caso a extensão da cadeia de recursões realizadas para uma dada entrada para o problema seja muito grande

Programação dinâmica

- Para que seja possível de resolver através de um algoritmo de programação dinâmica, um problema deve ter duas propriedades:
  - Propriedade da **subestrutura ótima**:
    - Uma solução ótima para o problema contém soluções ótimas para subproblemas
  - Propriedade da **sobreposição de subproblemas**:
    - A quantidade de subproblemas existentes não é muito grande – geralmente polinomial no tamanho da entrada – e isto ocorre porque existem muitos subproblemas repetidos

Programação dinâmica

- Em geral, na versão top-down de um algoritmo de programação dinâmica, temos uma função recursiva $f(n)$ onde vale o seguinte:
  - Usamos uma estrutura de dados para armazenar e acessar de maneira eficiente os valores de $f(n)$ que já foram computados (**tabulação** dos resultados)
  - No início de cada chamada à $f(n)$, vamos verificar se o valor de $f(n)$ já foi computado
    - Se sim, simplesmente, vamos retornar este valor (sem computar novamente $f(n)$)
    - Se não, vamos computar o valor de $f(n)$ normalmente e armazenar este valor na estrutura de dados (em outras chamadas à função, o valor de $f(n)$ não será computado novamente)

Referências

- Esta apresentação é baseada nos seguintes materiais:

1. Capítulo 14 do livro
   Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. Introduction to Algorithms. 4th. ed. MIT Press, 2022.
2. Capítulo 3 do livro
   HALIM, S.; HALIM, F.; EFFENDY, S. Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s, book 1, chs. 1-4. Lulu, 2018.
3. Capítulo 6 do livro
   LAAKSONEN, A. Guide to Competitive Programming: Learning and Improving Algorithms Through Contests, 2. ed. Springer, 2020.