Programação dinâmica

SCC0601 - Introdução à Ciência de Computação II

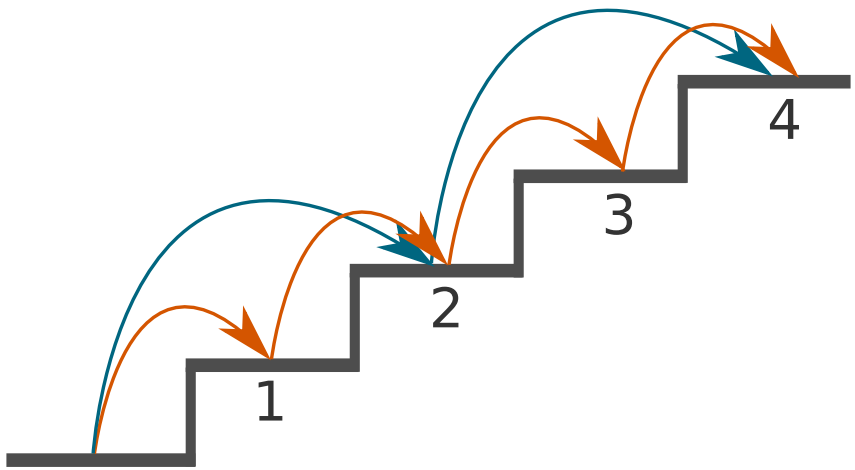
Paola Tatiana Llerena Valdivia

Instituto de Ciências Matemáticas e de Computação
USP - São Carlos

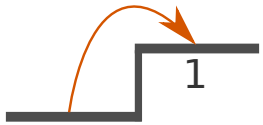
Roteiro

Problema da escadaria

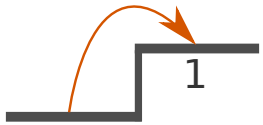
Temos uma escadaria com n degraus. Suponha que só podemos subir esta escadaria dando passos de um degrau ou saltando por cima de um degrau.
De quantas formas possíveis podemos subir a escadaria?

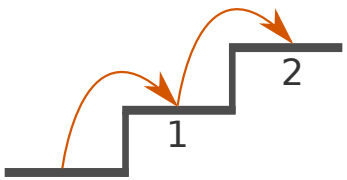


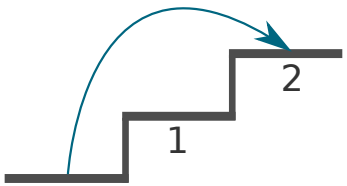
Problema da escadaria - 1 2 e 3 degraus



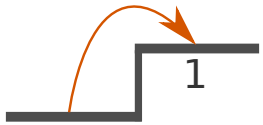
Problema da escadaria - 1 2 e 3 degraus

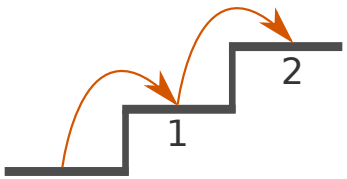


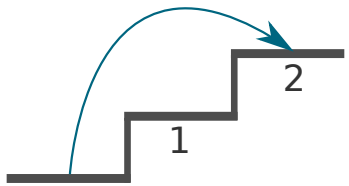


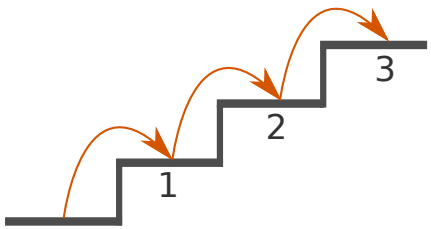


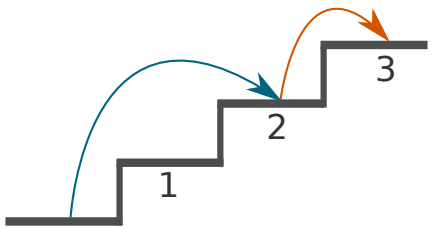
Problema da escadaria - 1 2 e 3 degraus

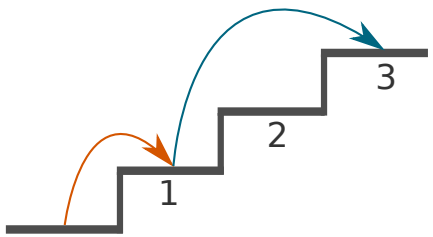




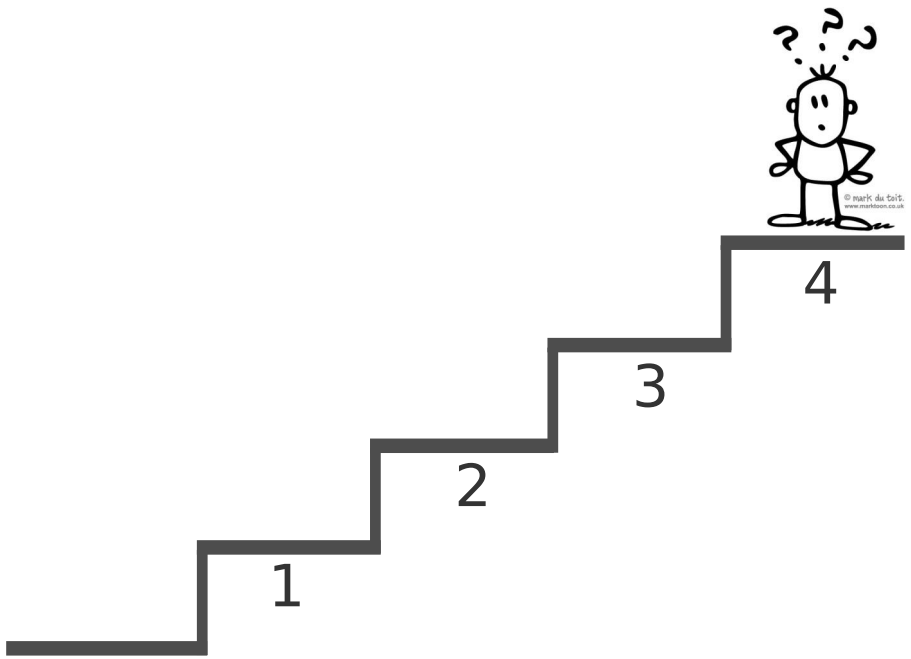




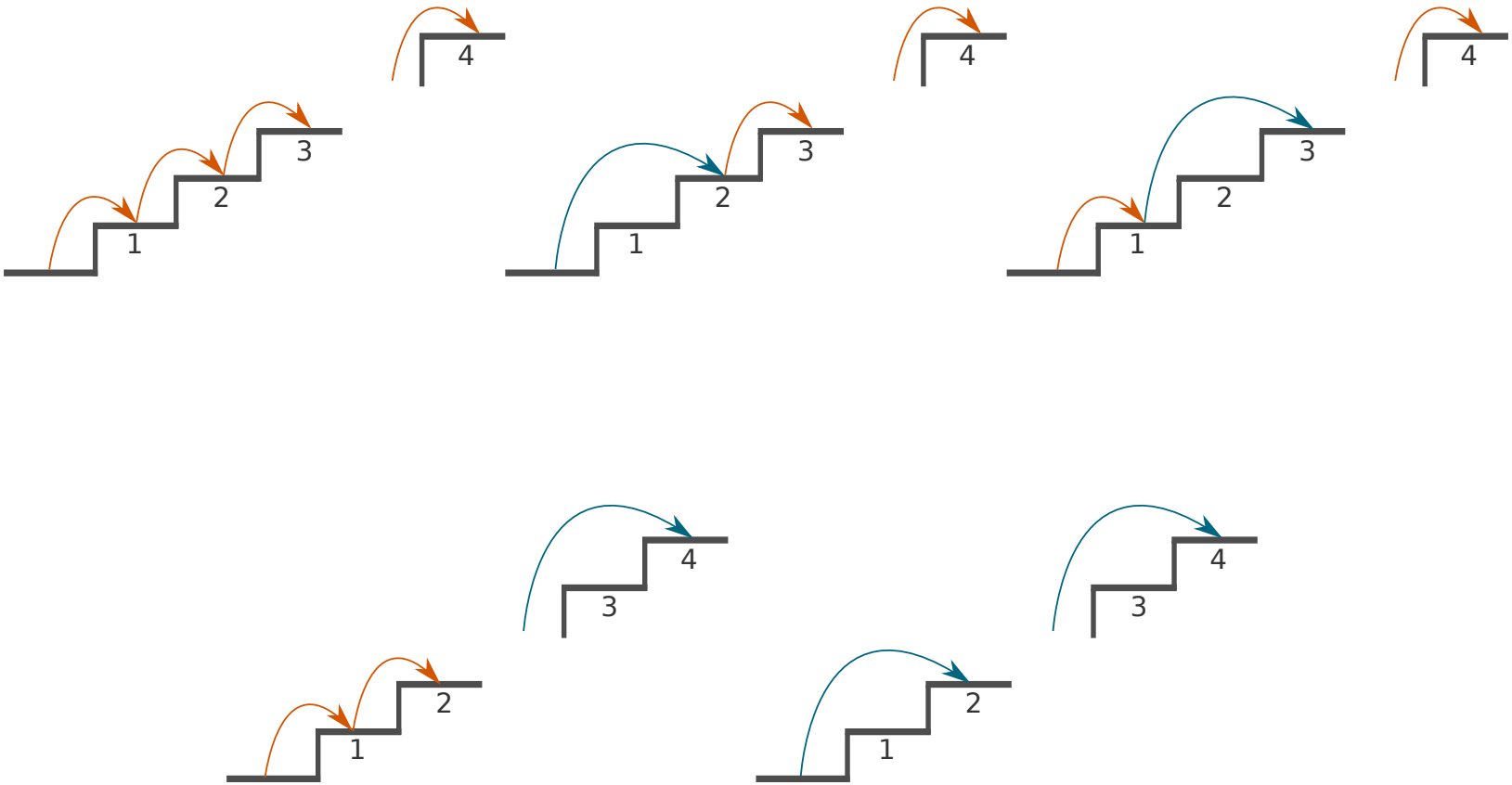




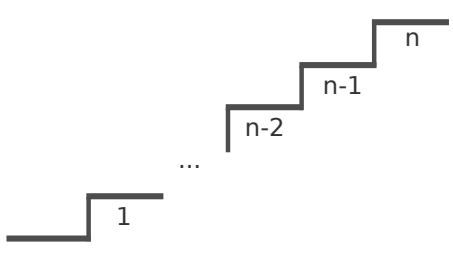
Problema da escadaria - 4 degraus



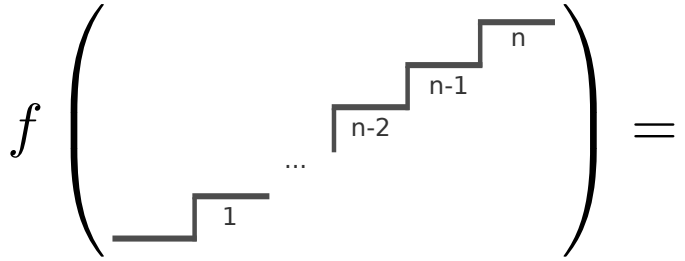
Problema da escadaria - 4 degraus

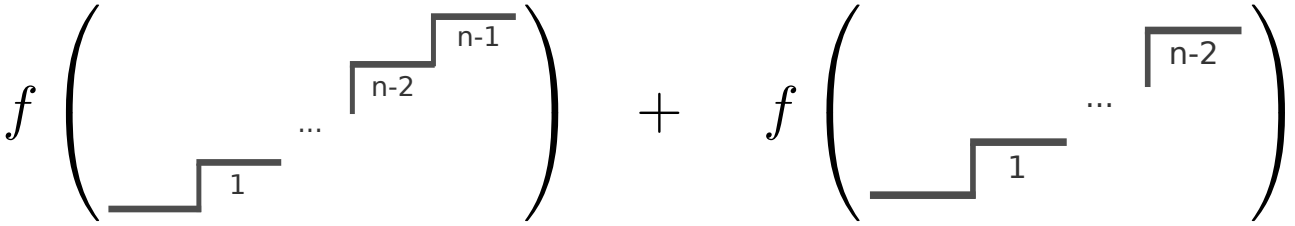


Problema da escadaria - n degraus

$$f\left(\text{  }\right) =$$

Problema da escadaria - n degraus





Problema da escadaria - n degraus

$$f\left(\text{1} \cdots \text{n-2} \text{n-1} \text{n}\right) =$$

$$f\left(\text{1} \cdots \text{n-2} \text{n-1}\right) + f\left(\text{1} \cdots \text{n-2}\right)$$

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

Problema da escadaria - Recursão

```
int fib(int n) {
    int resultado;
    if (n<2) resultado=1;
    else resultado = fib(n-1)+fib(n-2);
    return(resultado);
}
```

Problema da escadaria - Recursão

```
int fib(int n) {
    int resultado;
    if (n<2) resultado=1;
    else resultado = fib(n-1)+fib(n-2);
    return(resultado);
}
```

Complexidade

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

Problema da escadaria - Recursão

```
int fib(int n) {
    int resultado;
    if (n<2) resultado=1;
    else resultado = fib(n-1)+fib(n-2);
    return(resultado);
}
```

Complexidade

$$T(n) = T(n-1) + T(n-2) + 1$$
$$T(n) \geq 2\,T(n-2)$$
$$= 2^{n/2} \quad \text{Exponencial!}$$

Problema da escadaria - Recursão

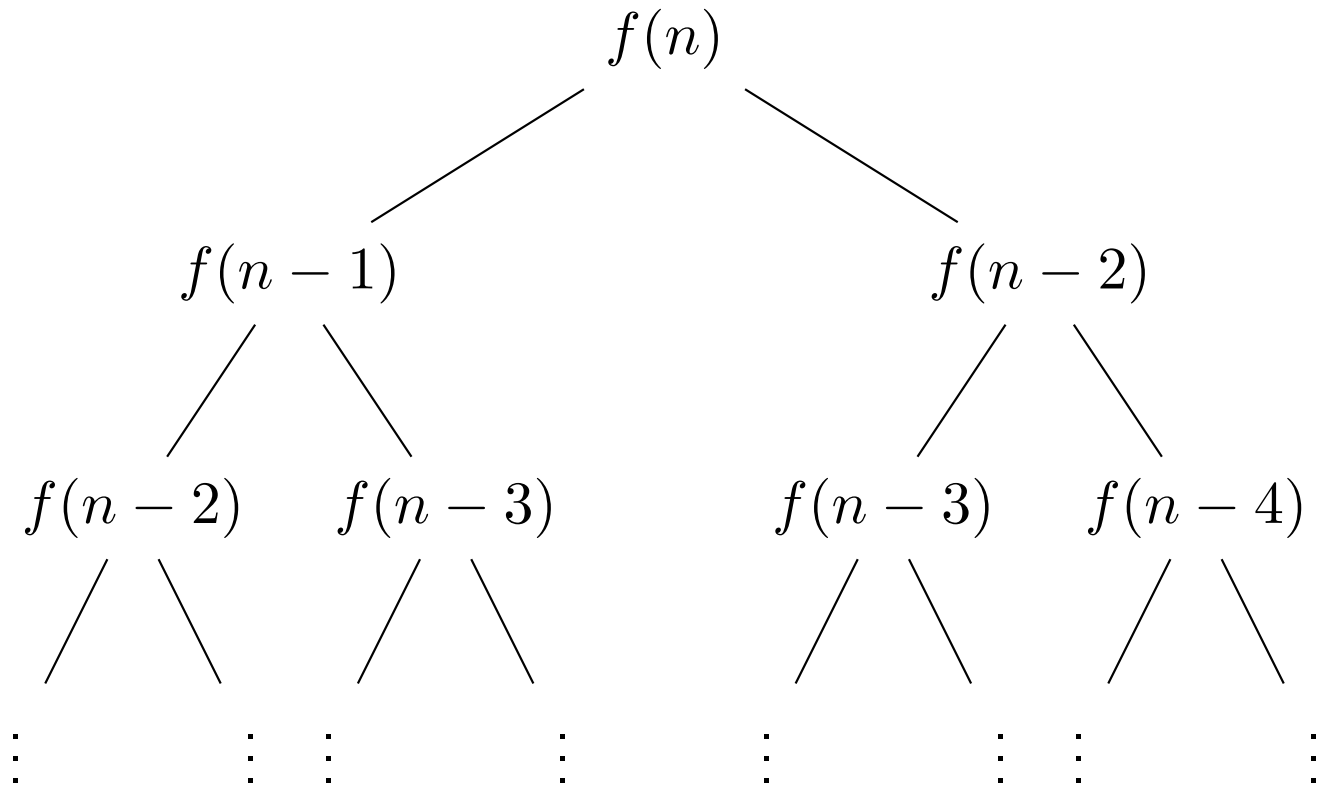
```
int fib(int n) {
    int resultado;
    if (n<2) resultado=1;
    else resultado = fib(n-1)+fib(n-2);
    return(resultado);
}
```

Complexidade

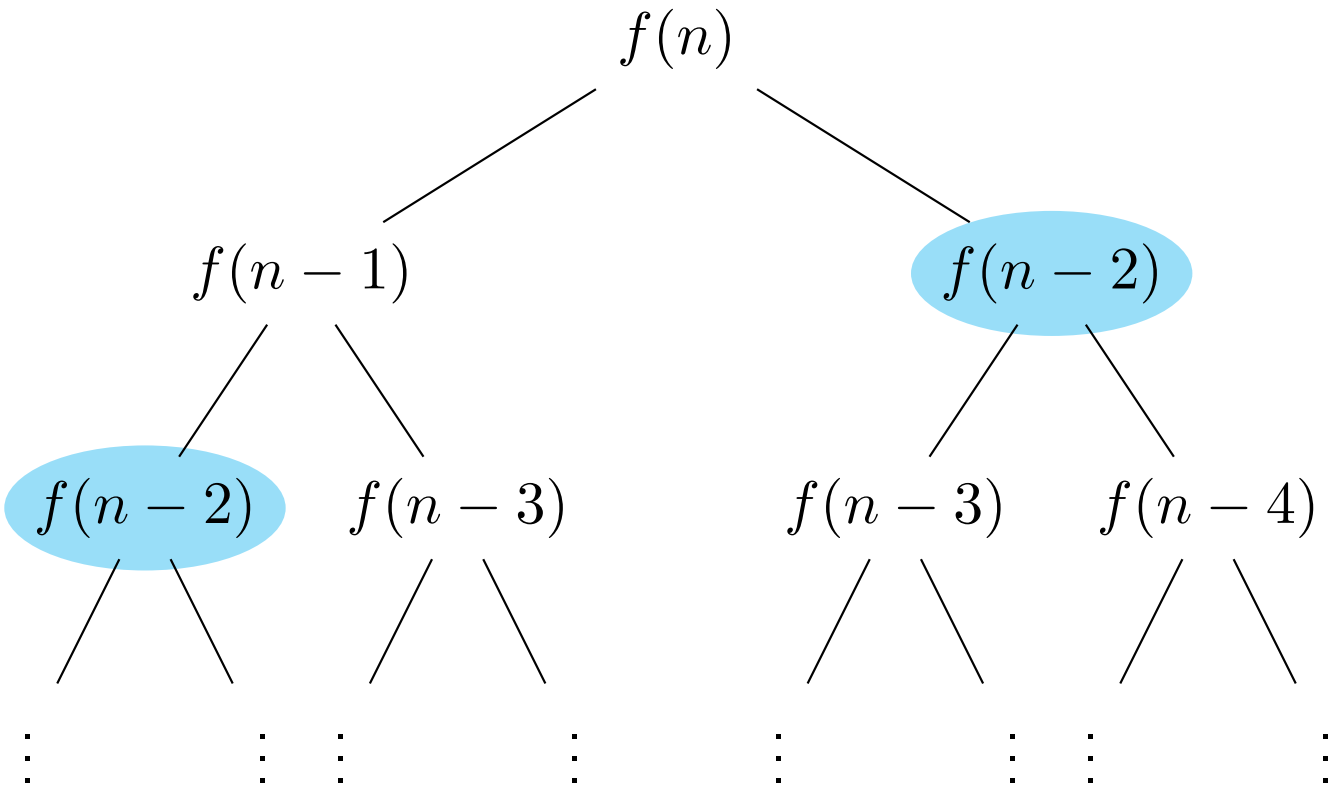
$$T(n) = 2^{n/2}$$ Exponencial!

Podemos melhorar esta solução?

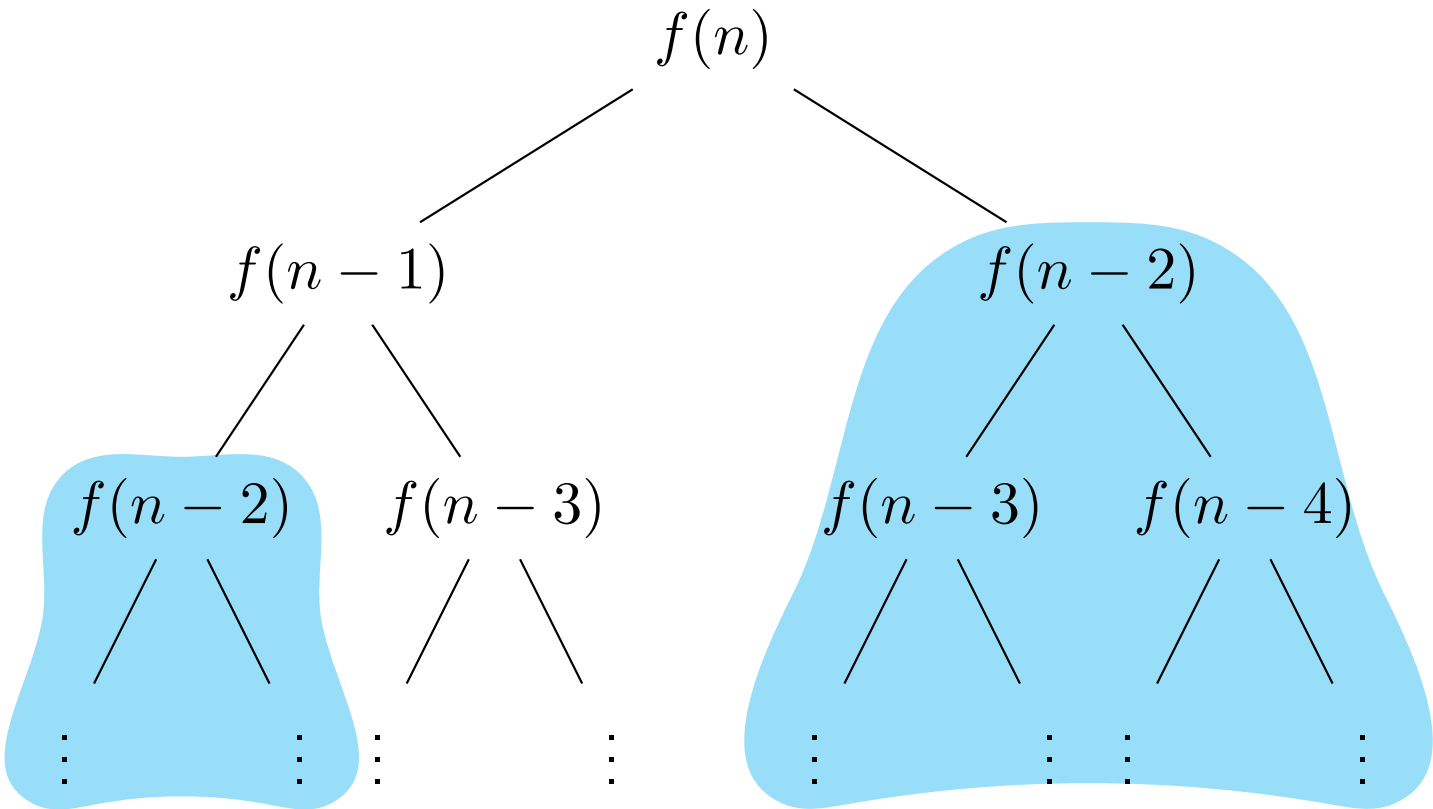
Problema da escadaria - Árvore de recursão



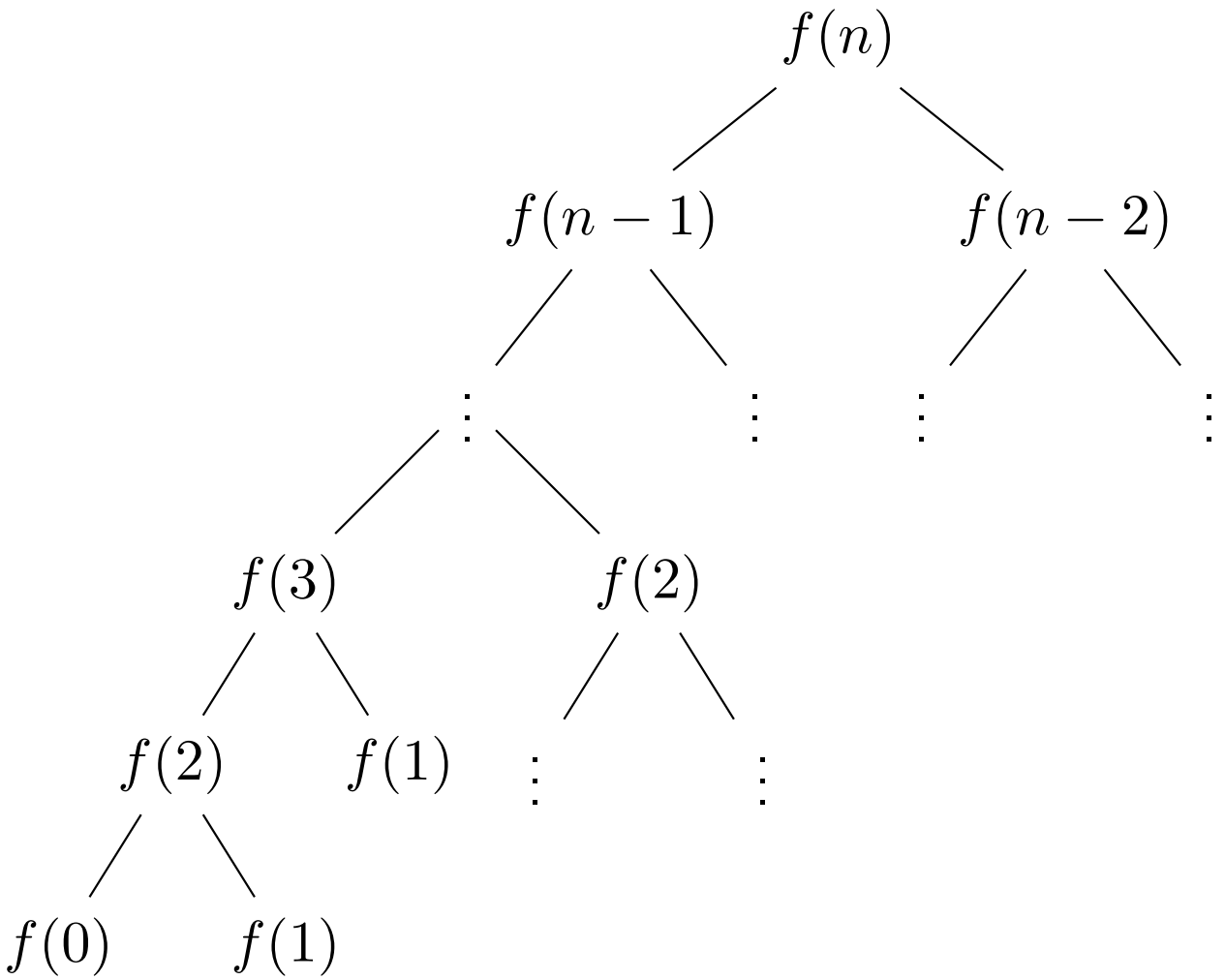
Problema da escadaria - Árvore de recursão



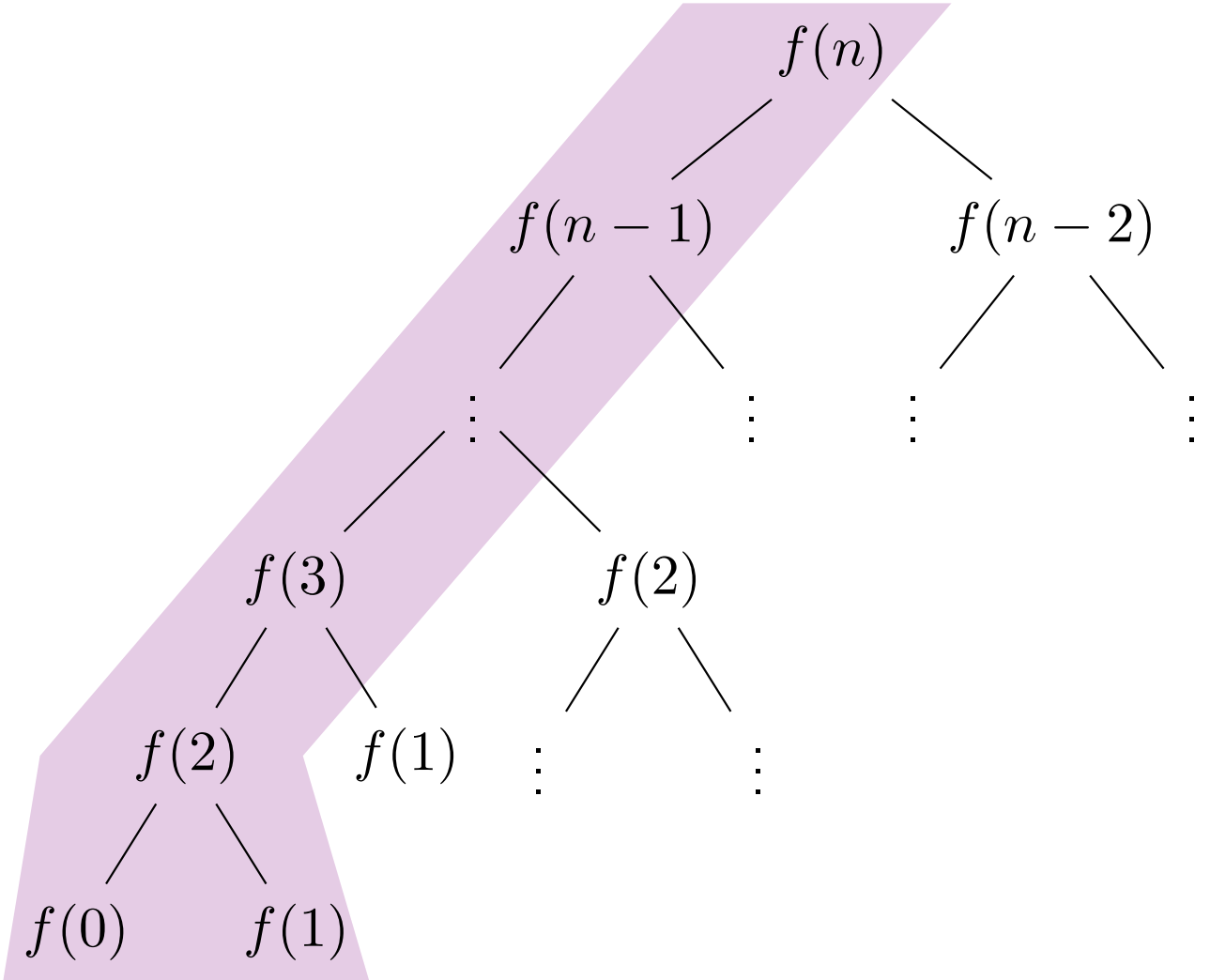
Problema da escadaria - Árvore de recursão



Problema da escadaria - Árvore de recursão



Problema da escadaria - Árvore de recursão



Problema da escadaria - Memoização

```
int memo[50]; // inicializado com zeros
int memo_fib(int n) {
    if (memo[n]!=0) return memo[n];
    int resultado;
    if (n < 2) resultado = 1;
    else resultado = fib(n-1)+fib(n-2);
    memo[n] = resultado;
    return(resultado);
}
```

Problema da escadaria - Memoização

```
int memo[50]; // inicializado com zeros
int memo_fib(int n) {
    if (memo[n]!=0) return memo[n];
    int resultado;
    if (n < 2) resultado = 1;
    else resultado = fib(n-1)+fib(n-2);
    memo[n] = resultado;
    return(resultado);
}
```

Complexidade

Problema da escadaria - Memoização

```
int memo[50]; // inicializado com zeros
int memo_fib(int n) {
    if (memo[n]!=0) return memo[n];
    int resultado;
    if (n < 2) resultado = 1;
    else resultado = fib(n-1)+fib(n-2);
    memo[n] = resultado;
    return(resultado);
}
```

Complexidade

- chamadas que foram memoizadas: $\Theta(1)$
- n chamadas que não foram memoizadas
- trabalho não recursivo: $\Theta(1)$

$$T(n) = n$$

Roteiro

Programação dinâmica

Ideia principal:

- Dividir o problema em subproblemas.
- Calcular cada subproblema somente uma vez.

Programação dinâmica

Ideia principal:

- Dividir o problema em subproblemas. (Recursão)
- Calcular cada subproblema somente uma vez. (Memoização)

Programação dinâmica

Ideia principal:

- Dividir o problema em subproblemas. (Recursão)
- Calcular cada subproblema somente uma vez. (Memoização)

PD = Recursão + Memoização

(Abordagem top-down)

Programação dinâmica

Ideia principal:

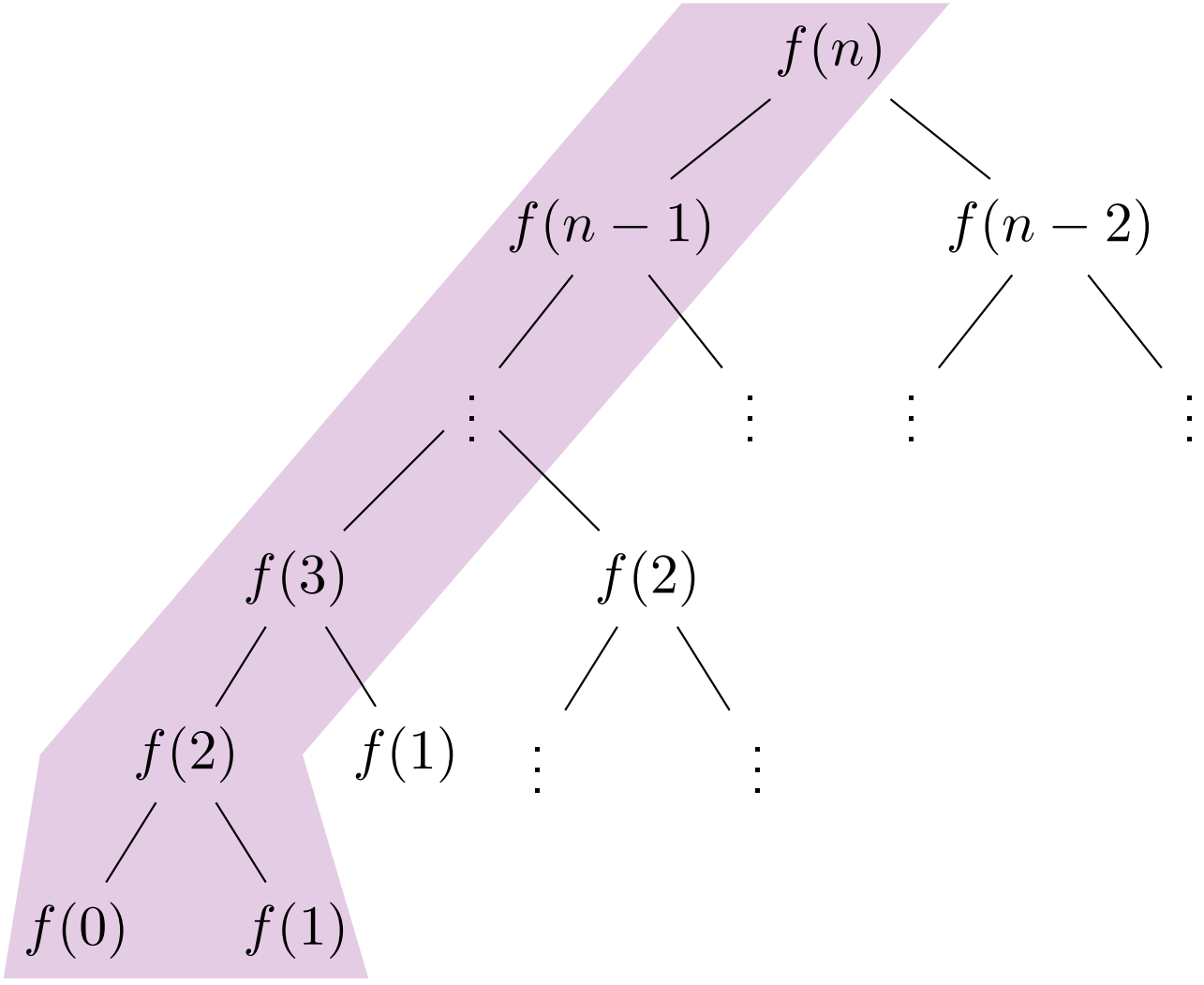
- Dividir o problema em subproblemas. (Recursão)
- Calcular cada subproblema somente uma vez. (Memoização)

PD = Recursão + Memoização

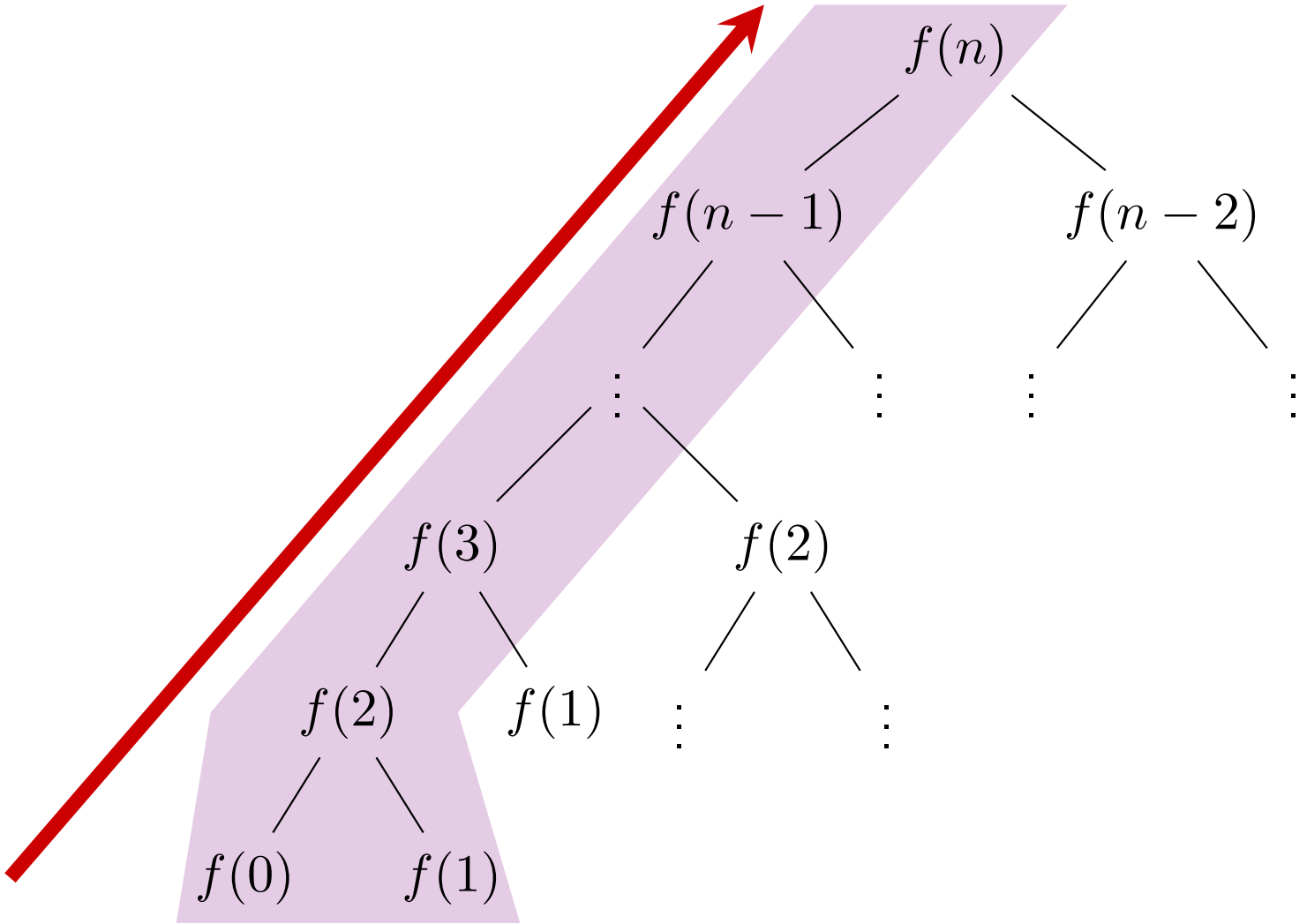
(Abordagem top-down)

Alternativa: Abordagem bottom-up.

Problema da escadaria - Bottom-up



Problema da escadaria - Bottom-up



Problema da escadaria - Bottom-up

```
int bottom_up_fib(int n) {
    int k, r, f[50];
    for (k=0; k<=n; ++k) {
        if(k<2) r = 1;
        else r = f[k-1] + f[k-2];
        f[k] = r;
    }
    return(f[n]);
}
```

Problema da escadaria - Bottom-up

```
int bottom_up_fib(int n) {
    int k, r, f[50];
    for (k=0; k<=n; ++k) {
        if(k<2) r = 1;
        else r = f[k-1] + f[k-2];
        f[k] = r;
    }
    return(f[n]);
}
```

Complexidade

Problema da escadaria - Bottom-up

```
int bottom_up_fib(int n) {
    int k, r, f[50];
    for (k=0; k<=n; ++k) {
        if(k<2) r = 1;
        else r = f[k-1] + f[k-2];
        f[k] = r;
    }
    return(f[n]);
}
```

Complexidade

$$T(n) = n$$

Programação dinâmica

Definição

Técnica algorítmica

Definição

Técnica algorítmica

*** Técnica algorítmica:** método geral para resolver problemas que têm algumas características em comum.

Programação dinâmica

Definição

Técnica algorítmica que é baseada em dividir um problema em uma serie de subproblemas *coincidentes*, guardar os resultados e construir soluções para problemas maiores.

*** Técnica algorítmica:** método geral para resolver problemas que têm algumas características em comum.

Programação dinâmica - Características

Características de um problema para poder ser resolvido com PD:

- Subestrutura ótima
- Subproblemas coincidentes

Programação dinâmica - Subestrutura ótima

Subestrutura ótima

Quando a solução ótima de um problema contém nela própria soluções ótimas para subproblemas do mesmo tipo.

Exemplo:

Programação dinâmica - Subproblemas coincidentes

Subproblemas coincidentes

Quando um espaço de subproblemas é pequeno, isto é, não são muitos os subproblemas a resolver, pois muitos deles são exatamente iguais uns aos outros.

Programação dinâmica - Subproblemas coincidentes

Subproblemas coincidentes

Quando um espaço de subproblemas é pequeno, isto é, não são muitos os subproblemas a resolver, pois muitos deles são exatamente iguais uns aos outros.

Exemplo:

No problema da escadaria (sequência de Fibonnaci), para n escadas, existem apenas n subproblemas, pois os outros subproblemas são coincidentes.

Programação dinâmica - Etapas

1. Definir a estrutura de uma solução ótima
2. Definir a solução ótima de forma recursiva
3. Usar PD (recursão + memoização) para solucionar o problema
   3.1 Alternativa: algoritmo iterativo bottom-up
4. Construir uma solução ótima a partir do valor ótimo calculado (opcional)

Roteiro

Cortar barras de ferro

Uma empresa compra barras de ferro, as corta e revende as partes. Assume-se que:

- o corte é sempre em unidades de medida inteiras: $1, 2, 3, \ldots$;
- os cortes são de graça e
- sabemos o preço para os possíveis tamanhos das barras.

Deseja-se saber qual é a forma de cortar as barras que gera maiores ingressos (melhor forma de cortar as barras).

Cortar barras de ferro

Uma empresa compra barras de ferro, as corta e revende as partes. Assume-se que:

- o corte é sempre em unidades de medida inteiras: $1, 2, 3, \ldots$;
- os cortes são de graça e
- sabemos o preço para os possíveis tamanhos das barras.

Deseja-se saber qual é a forma de cortar as barras que gera maiores ingressos (melhor forma de cortar as barras).

tamanho i	1	2	3	4
preço p_i	1	5	8	9

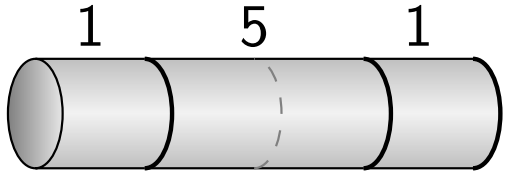
Cortar barras de ferro

Uma empresa compra barras de ferro, as corta e revende as partes. Assume-se que:

- o corte é sempre em unidades de medida inteiras: $1, 2, 3, \ldots$;
- os cortes são de graça e
- sabemos o preço para os possíveis tamanhos das barras.

Deseja-se saber qual é a forma de cortar as barras que gera maiores ingressos (melhor forma de cortar as barras).

tamanho i	1	2	3	4
preço p_i	1	5	8	9



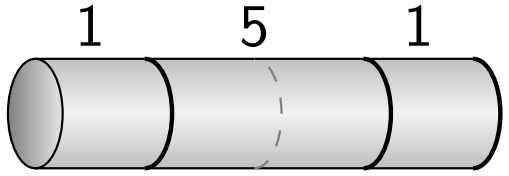
Cortar barras de ferro

Uma empresa compra barras de ferro, as corta e revende as partes. Assume-se que:

- o corte é sempre em unidades de medida inteiras: $1, 2, 3, \ldots$;
- os cortes são de graça e
- sabemos o preço para os possíveis tamanhos das barras.

Deseja-se saber qual é a forma de cortar as barras que gera maiores ingressos (melhor forma de cortar as barras).

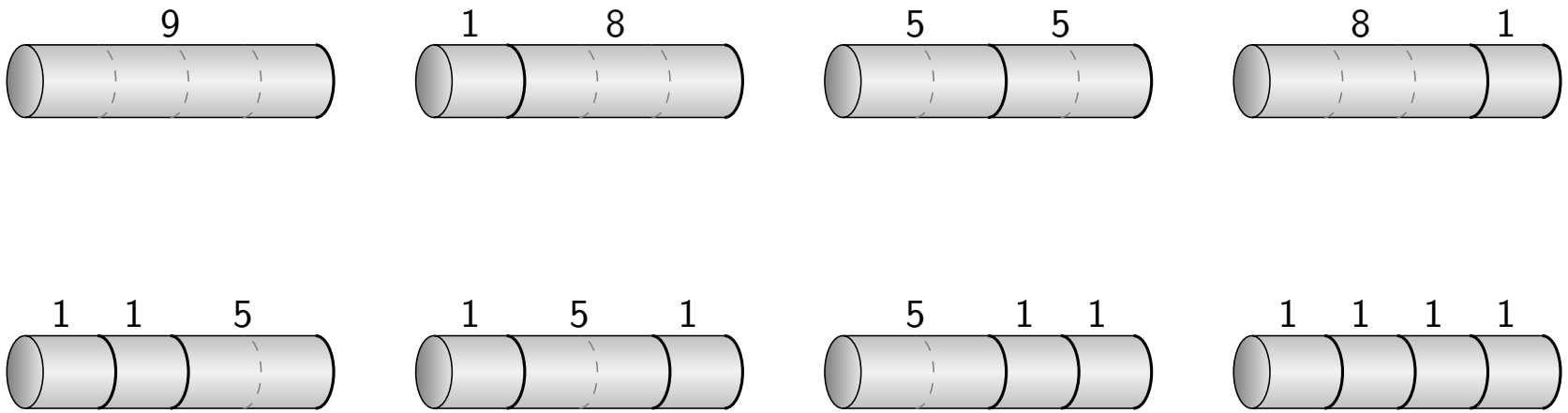
tamanho i	1	2	3	4
preço p_i	1	5	8	9



$$r = 1 + 5 + 1 = 7$$

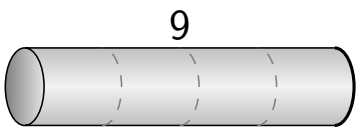
Cortar barras de ferro - $n = 4$

tamanho i	1	2	3	4
preço p_i	1	5	8	9

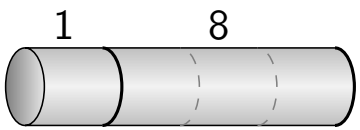


Cortar barras de ferro - $n = 4$

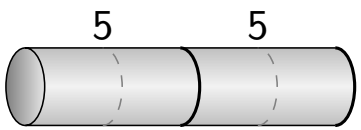
tamanho i	1	2	3	4
preço p_i	1	5	8	9



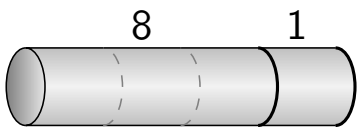
$r = 9$



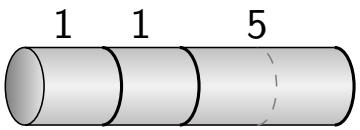
$r = 9$



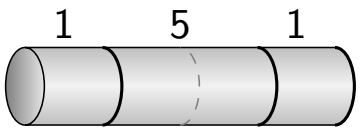
$r = 10$



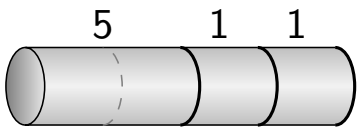
$r = 9$



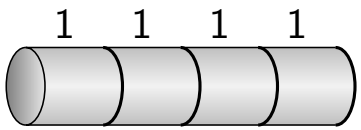
$r = 7$



$r = 7$



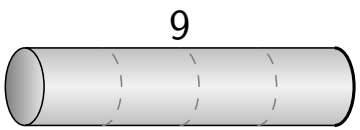
$r = 7$



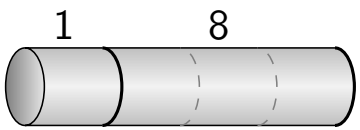
$r = 4$

Cortar barras de ferro - $n = 4$

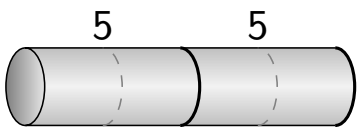
tamanho i	1	2	3	4
preço p_i	1	5	8	9



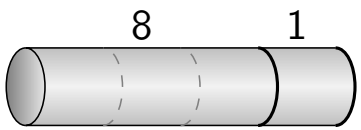
$r = 9$



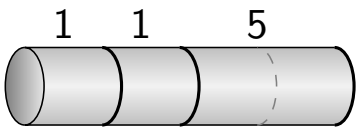
$r = 9$



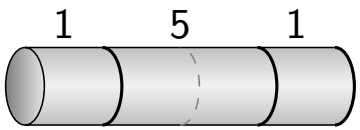
$r = 10$



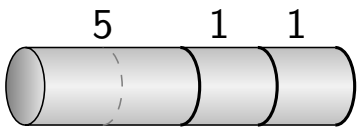
$r = 9$



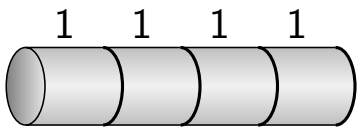
$r = 7$



$r = 7$



$r = 7$



$r = 4$

$$r(4) = 10$$

Cortar barras de ferro - Solução ótima

Etapa 1: Definir a estrutura de uma solução ótima

Cortar barras de ferro - Solução ótima

- Subestrutura ótima?
- Subproblemas coincidentes?

Etapa 1: Definir a estrutura de uma solução ótima

Cortar barras de ferro - Solução recursiva

$$r(4) = \max(p_1 + r(3),\ \ p_2 + r(2),\ \ p_3 + r(1),\ \ p_4)$$

Cortar barras de ferro - Solução recursiva

$$r(4) = \max(p_1 + r(3),\ \ p_2 + r(2),\ \ p_3 + r(1),\ \ p_4)$$

$$r(n) = \max\left(p_1 + r(n-1), p_2 + r(n-1), \ldots, p_{n-1} + r(1), p_n + r(0)\right)$$

$$r(0) = 0$$

Cortar barras de ferro - Solução recursiva

Solução ótima de forma recursiva

$$r(n) = \max_{1 \leq i \leq n} (p_i + r(n-i))$$

$$r(0) = 0$$

Etapa 2: Definir a solução ótima de forma recursiva

Cortar barras de ferro - Solução recursiva

```
int cut_rod(int *p, int n) {
    if(n==0) return 0;
    int i, q=-1;
    for(i=1; i<=n; ++i)
        q = max(q, p[i]+cut_rod(p,n-i));
    return q;
}
```

Cortar barras de ferro - Solução recursiva

```
int cut_rod(int *p, int n) {
    if(n==0) return 0;
    int i, q=-1;
    for(i=1; i<=n; ++i)
        q = max(q, p[i]+cut_rod(p,n-i));
    return q;
}
```

Complexidade

Cortar barras de ferro - Solução recursiva

```
int cut_rod(int *p, int n) {
    if(n==0) return 0;
    int i, q=-1;
    for(i=1; i<=n; ++i)
        q = max(q, p[i]+cut_rod(p,n-i));
    return q;
}
```

Complexidade

$$T(n) = 1 + \sum_{j=1}^{n-1} T(j)$$

$$= 2^n$$

Exponencial!

Cortar barras de ferro - PD: recursão + memoização

```
int memo[50]; // inicializado com -1
int memo_cut_rod(int *p, int n) {









}
```

Cortar barras de ferro - PD: recursão + memoização

```
int memo[50]; // inicializado com -1
int memo_cut_rod(int *p, int n) {
    if (memo[n]>=0) return memo[n];
    int q, i;
    if (n == 0) resultado = 0;
    else{
        q = -1;
        for(i=1; i<=n; ++i)
            q = max(q, p[i]+memo_cut_rod(p,n-i));
    }
    memo[n] = q;
    return(q);
}
```

Cortar barras de ferro - PD: recursão + memoização

```
int memo[50]; // inicializado com -1
int memo_cut_rod(int *p, int n) {
    if (memo[n]>=0) return memo[n];
    int q, i;
    if (n == 0) resultado = 0;
    else{
        q = -1;
        for(i=1; i<=n; ++i)
            q = max(q, p[i]+memo_cut_rod(p,n-i));
    }
    memo[n] = q;
    return(q);
}
```

Complexidade

Cortar barras de ferro - PD: recursão + memoização

```
int memo[50]; // inicializado com -1
int memo_cut_rod(int *p, int n) {
    if (memo[n]>=0) return memo[n];
    int q, i;
    if (n == 0) resultado = 0;
    else{
        q = -1;
        for(i=1; i<=n; ++i)
            q = max(q, p[i]+memo_cut_rod(p,n-i));
    }
    memo[n] = q;
    return(q);
}
```

Complexidade

$$T(n) = n^2$$

Cortar barras de ferro - PD: bottom-up

```
int bottom_up_cut_rod(int *p, int n) {









}
```

Cortar barras de ferro - PD: bottom-up

```
int bottom_up_cut_rod(int *p, int n) {
    int j, i, r[50];
    r[0] = 0;
    for (j=1; j<=n; ++j) {
        q = -1;
        for (i=1; i<=j; ++i)
            q = max(q, p[i]+r[j-i]);
            r[j] = q;
        }
    return(r[n]);
}
```

Cortar barras de ferro - PD: bottom-up

```
int bottom_up_cut_rod(int *p, int n) {
    int j, i, r[50];
    r[0] = 0;
    for (j=1; j<=n; ++j) {
        q = -1;
        for (i=1; i<=j; ++i)
            q = max(q, p[i]+r[j-i]);
            r[j] = q;
        }
    return(r[n]);
}
```

Complexidade

Cortar barras de ferro - PD: bottom-up

```
int bottom_up_cut_rod(int *p, int n) {
    int j, i, r[50];
    r[0] = 0;
    for (j=1; j<=n; ++j) {
        q = -1;
        for (i=1; i<=j; ++i)
            q = max(q, p[i]+r[j-i]);
            r[j] = q;
        }
    return(r[n]);
}
```

Complexidade

$$T(n) = n^2$$

Cortar barras de ferro - Reconstruir uma solução

```
void extended_bottom_up_cut_rod(int *p, int n,
                                int *r, int *s) {
   int j, i;
   r[0] = 0;
   for (j=1; j<=n; ++j) {
      q = -1;
      for (i=1; i<=j; ++i)
         if (q< p[i]+r[j-i]){
            q = p[i]+r[j-i];
            s[j] = i;
         }
         r[j] = q;
      }
}
```

Roteiro

Subsequência comum mais longa

Em biologia algumas aplicações exigem comparação de DNA de organismos.

- Cadeia de DNA: cadeia de bases ($\{$A,C,G,T$\}$)

Exemplo

$S_1 =$ ACCGGTCGAGCAA
$S_2 =$ GTCGAGTGCAA

S_1 e S_2 são semelhantes?

Subsequência comum mais longa

Em biologia algumas aplicações exigem comparação de DNA de organismos.

- Cadeia de DNA: cadeia de bases ($\{A,C,G,T\}$)

Exemplo

$S_1 =$ ACCGGTCGAGCAA
$S_2 =$ GTCGAGTGCAA

S_1 e S_2 são semelhantes?

Critério 1. uma é subcadeia da outra

Subsequência comum mais longa

Em biologia algumas aplicações exigem comparação de DNA de organismos.

- Cadeia de DNA: cadeia de bases ($\{A,C,G,T\}$)

Exemplo

$S_1 = ACCGGTCGAGCAA$
$S_2 = GTCGAGTGCAA$

S_1 e S_2 são semelhantes?

Critério 1. uma é subcadeia da outra

Critério 2. número de alterações para transformar S_1 em S_2

Subsequência comum mais longa

Em biologia algumas aplicações exigem comparação de DNA de organismos.

- Cadeia de DNA: cadeia de bases ($\{A,C,G,T\}$)

Exemplo

$S_1 = ACCGGTCGAGCAA$
$S_2 = GTCGAGTGCAA$

S_1 e S_2 são semelhantes?

Critério 1. uma é subcadeia da outra

Critério 2. número de alterações para transformar S_1 em S_2

Critério 3. encontrar S_3: cadeia mais longa de bases que aparecem em S_1 e S_2 na mesma ordem, mas não necessariamente consecutivamente.

Subsequência comum mais longa

Em biologia algumas aplicações exigem comparação de DNA de organismos.

- Cadeia de DNA: cadeia de bases ($\{A,C,G,T\}$)

Exemplo

$S_1 = ACCGGTCGAGCAA$
$S_2 = GTCGAGTGCAA$

S_1 e S_2 são semelhantes?

Critério 1. uma é subcadeia da outra

Critério 2. número de alterações para transformar S_1 em S_2

Critério 3. encontrar S_3: cadeia mais longa de bases que aparecem em S_1 e S_2 na mesma ordem, mas não necessariamente consecutivamente.

Subsequência comum mais longa

Em biologia algumas aplicações exigem comparação de DNA de organismos.

- Cadeia de DNA: cadeia de bases ($\{A,C,G,T\}$)

Exemplo

$S_1 =$ ACCGGTCGAGCAA
$S_2 =$ GTCGAGTGCAA

S_1 e S_2 são semelhantes?

Critério 1. uma é subcadeia da outra

Critério 2. número de alterações para transformar S_1 em S_2

Critério 3. encontrar S_3: cadeia mais longa de bases que aparecem em S_1 e S_2 na mesma ordem, mas não necessariamente consecutivamente.
$S_3 =$ GTCGAGCAA

Subsequência comum mais longa

Uma **subsequência** de uma sequência dada é a mesma com zero ou mais elementos omitidos.

Subsequência comum mais longa

Uma **subsequência** de uma sequência dada é a mesma com zero ou mais elementos omitidos.

Formalmente, dadas as sequências $X = \langle x_1, x_2, \ldots, x_m \rangle$ e $Z = \langle z_1, z_2, \ldots, z_k \rangle$, Z é uma **subsequência** de X se existe uma sequência crescente de índices de X, $\langle i_1, i_2, \ldots, i_k \rangle$ tais que para todo $j = 1, 2, \ldots, k$, temos que $x_{i_j} = z_j$.

Subsequência comum mais longa

Uma **subsequência** de uma sequência dada é a mesma com zero ou mais elementos omitidos.

Formalmente, dadas as sequências $X = \langle x_1, x_2, \ldots, x_m \rangle$ e $Z = \langle z_1, z_2, \ldots, z_k \rangle$, Z é uma **subsequência** de X se existe uma sequência crescente de índices de X, $\langle i_1, i_2, \ldots, i_k \rangle$ tais que para todo $j = 1, 2, \ldots, k$, temos que $x_{i_j} = z_j$.

Z é uma **subsequência comum** de X e Y se é uma subsequência de ambas.

Subsequência comum mais longa

Uma **subsequência** de uma sequência dada é a mesma com zero ou mais elementos omitidos.

Formalmente, dadas as sequências $X = \langle x_1, x_2, \ldots, x_m \rangle$ e $Z = \langle z_1, z_2, \ldots, z_k \rangle$, Z é uma **subsequência** de X se existe uma sequência crescente de índices de X, $\langle i_1, i_2, \ldots, i_k \rangle$ tais que para todo $j = 1, 2, \ldots, k$, temos que $x_{i_j} = z_j$.

Z é uma **subsequência comum** de X e Y se é uma subsequência de ambas.

Queremos encontrar a **subsequência comum mais longa** (LCS).

Subsequência comum mais longa - Solução ótima

Etapa 1: Definir a estrutura de uma solução ótima

Subsequência comum mais longa - Solução ótima

- Subestrutura ótima?
- Subproblemas coincidentes?

Etapa 1: Definir a estrutura de uma solução ótima

Subsequência comum mais longa - Solução ótima

Substrutura ótima de uma LCS

Sejam as sequências $X = \langle x_1, x_2, \ldots, x_m \rangle$ e $Y = \langle y_1, y_2, \ldots, y_n \rangle$ e seja $Z = \langle z_1, z_2, \ldots, z_k \rangle$ uma LCS de X e Y, então:

1. Se $x_m = y_n$, então $z_k = x_m = y_n$ e Z_{k-1} é uma LCS de X_{m-1} e Y_{n-1}.
2. Se $x_m \neq y_n$, então $z_k \neq x_m$ implica que Z é uma LCS de X_{m-1} e Y.
3. Se $x_m \neq y_n$, então $z_k \neq y_n$ implica que Z é uma LCS de X_m e Y_{n-1}.

Notação: X_i é definido por $X_i = \langle x_1, x_2, \ldots, x_i \rangle$, para $i \leq m$.

Subsequência comum mais longa - Solução ótima

- Se $x_m = y_n$:

Subsequência comum mais longa - Solução ótima

- Se $x_m = y_n$: deve ser encontrada uma LCS de X_{m-1} e Y_{n-1}.

Subsequência comum mais longa - Solução ótima

- Se $x_m = y_n$: deve ser encontrada uma LCS de X_{m-1} e Y_{n-1}.

  Concatenando x_m nessa LCS obtém-se a LCS de X e Y.

Subsequência comum mais longa - Solução ótima

- Se $x_m = y_n$: deve ser encontrada uma LCS de X_{m-1} e Y_{n-1}. Concatenando x_m nessa LCS obtém-se a LCS de X e Y.

- Se $x_m \neq y_n$:

Subsequência comum mais longa - Solução ótima

- Se $x_m = y_n$: deve ser encontrada uma LCS de X_{m-1} e Y_{n-1}.

  Concatenando x_m nessa LCS obtém-se a LCS de X e Y.

- Se $x_m \neq y_n$: há dois subproblemas a resolver
  1. Encontrar LCS de X_{m-1} e Y.
  2. Encontrar LCS de X_m e Y_{n-1}.

Subsequência comum mais longa - Solução ótima

- Se $x_m = y_n$: deve ser encontrada uma LCS de X_{m-1} e Y_{n-1}.

  Concatenando x_m nessa LCS obtém-se a LCS de X e Y.

- Se $x_m \neq y_n$: há dois subproblemas a resolver
  1. Encontrar LCS de X_{m-1} e Y.
  2. Encontrar LCS de X_m e Y_{n-1}.

  A LCS de X e Y é a maior dessas LCSs.

Subsequência comum mais longa - Solução recursiva

Seja $c[i, j]$ o comprimento de uma LCS de X_i e Y_j:

$$c[i,j] = \begin{cases} 0 & \text{se } i = 0 \text{ ou } j = 0, \\ c[i-1, j-1] + 1 & \text{se } i, j > 0 \text{ e } x_i = y_j, \\ \max(c[i, j-1], c[i-1, j]) & \text{se } i, j > 0 \text{ e } x_i \neq y_j \end{cases}$$

Subsequência comum mais longa - Solução recursiva

Seja $c[i, j]$ o comprimento de uma LCS de X_i e Y_j:

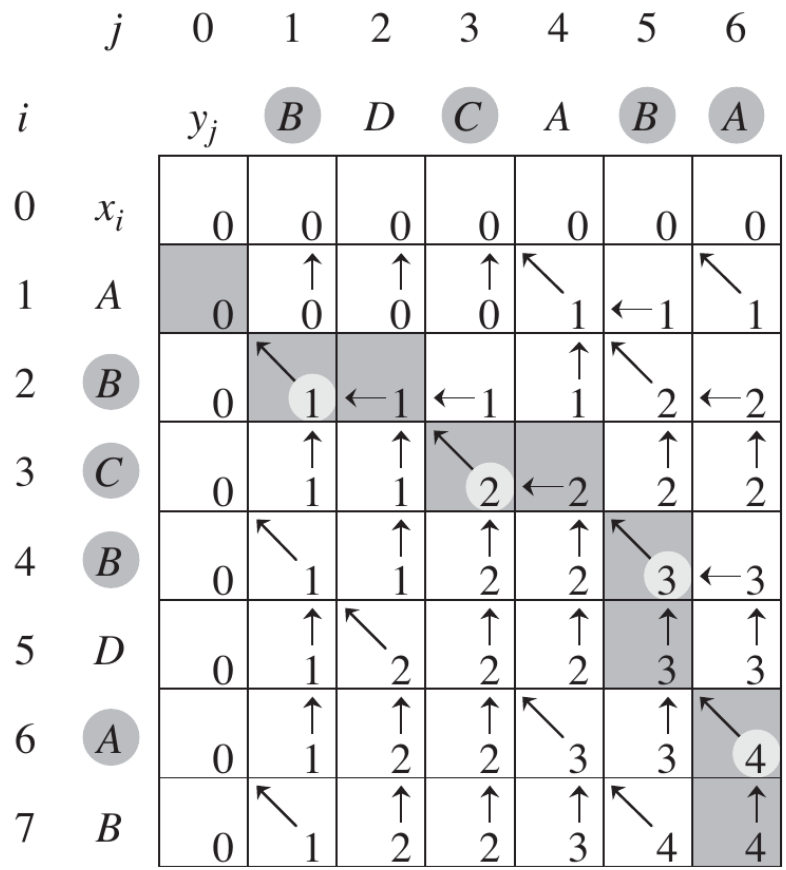
$$c[i,j] = \begin{cases} 0 & \text{se } i = 0 \text{ ou } j = 0, \\ c[i-1, j-1] + 1 & \text{se } i, j > 0 \text{ e } x_i = y_j, \\ \max(c[i, j-1], c[i-1, j]) & \text{se } i, j > 0 \text{ e } x_i \neq y_j \end{cases}$$

Etapa 2: Definir a solução ótima de forma recursiva

Subsequência comum mais longa - PD: bottom-up

```
LCS-LENGTH(X, Y)
 1  m = X.length
 2  n = Y.length
 3  for i = 1 to m
 4     c[i, 0] = 0
 5  for j = 1 to n
 6     c[0, j] = 0
 7  for i = 1 to m
 8     for j = 1 to n
 9        if x_i == y_j
10           then c[i, j] = c[i − 1, j − 1] + 1
11                b[i, j] = "↖"
12           else if c[i − 1, j] ≥ c[i, j − 1]
13                   then c[i, j] = c[i − 1, j]
14                        b[i, j] = "↑"
15                   else c[i, j] = c[i, j − 1]
16                        b[i, j] = "←"
17  return c and b
```

Subsequência comum mais longa - Exemplo



i		j	0	1	2	3	4	5	6
		y_j		B	D	C	A	B	A
0	x_i		0	0	0	0	0	0	0
1	A		0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B		0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C		0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B		0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D		0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A		0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B		0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Tabelas c e b para as sequências $X = \langle A, B, C, B, D, A, B \rangle$ e $Y = \langle B, D, C, A, B, A \rangle$

Referencias

Cormen, T.H.; Leiserson, C.E.; Rivest, R.L. e Stein, C.
Introduction To Algorithms
MIT Press, 2001

Demaine, Erik, e Srinivas Devadas.
6.006 Introduction to Algorithms, Fall 2011.
MIT OpenCourseWare

Obrigada!

paolalv@icmc.usp.br