Programação dinâmica

→ Constrói a solução para um problema gerando subproblemas menores, resolvendo-os e combinando as soluções (div. e conq.)
↳ Porém, só resolve um subproblema se ele já não foi resolvido antes.

→ Usamos uma "tabela" (memória extra) para armazenar o que já foi resolvido.
↳ Então temos que analisar quantos subproblemas diferentes há

Abordagens

→ TOP-DOWN: forma recursiva natural, que só faz recursão caso o subproblema não tenha sido resolvido e salva o que for resolvido na tabela. Tem 2 funções: recursiva e inicialização

→ BOTTOM-UP: forma iterativa, que resolve os subproblemas do menor para o maior, salvando na tabela. Ao resolver um subproblema, temos certeza que os menores já foram resolvidos.

→ Em geral têm o mesmo tempo assintótico:
tempo para resolver um subproblema × qtd. de subproblemas

A sequência de Fibonacci é 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ... e F_n é dado por

$$F_n = \begin{cases} 1 & \text{se } n = 1 \text{ ou } n = 2 \\ F_{n-1} + F_{n-2} & \text{se } n > 2 \end{cases}$$

Problema: *Número de Fibonacci*

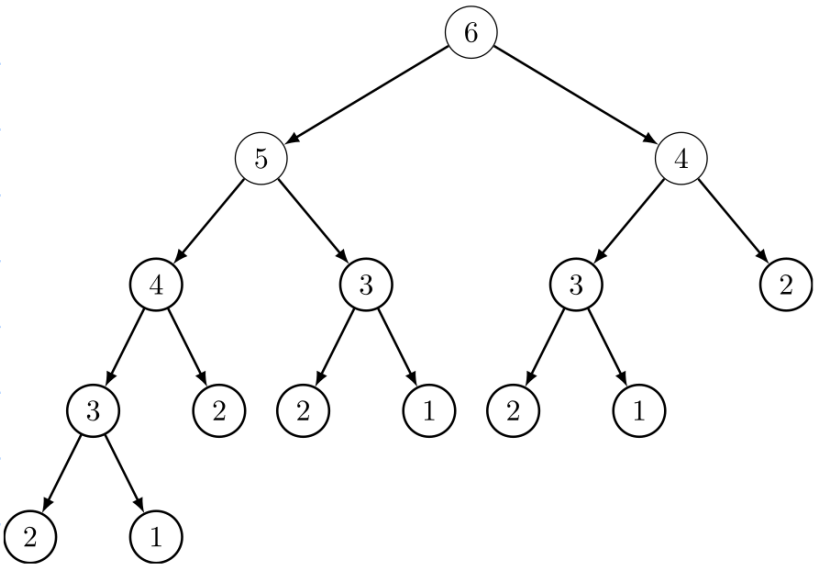
Entrada: $\langle n \rangle$, onde $n \geq 1$ é um inteiro.
Saída: F_n.

O seguinte algoritmo recursivo resolve* o problema:

```
FibonacciRecursivo(n)
1 se n ≤ 2 então
2     devolve 1
3 devolve FibonacciRecursivo(n − 1) + FibonacciRecursivo(n − 2)
```

* Resolve, pois podemos provar a frase $P(n)$ = "FibonacciRecursivo(n) devolve F_n" por indução em n.

Porém veja na árvore de recursão para FibonacciRecursivo(6) como há repetição de subproblemas:



Tempo de execução de FibonacciRecursivo(n)

De fato, se $T(n)$ é o tempo de FibonacciRecursivo(n), então

$$T(n-1)+T(n-2)+1 \leq T(n) \leq T(n-1)+T(n-2)+n$$

Pelo método da substituição, podemos provar que $T(n) \geq c\left(\frac{1+\sqrt{5}}{2}\right)^n$

BASE: $n=1$. $T(1)=1$ e $c\cdot\left(\frac{1+\sqrt{5}}{2}\right)^1 = c\cdot\left(\frac{1+\sqrt{5}}{2}\right)$.

Então o caso base vale se $c \leq \frac{2}{1+\sqrt{5}}$.

HIP: $T(k) \geq c\left(\frac{1+\sqrt{5}}{2}\right)^k$ para $1 \leq k < n$.

PASSO: Temos $T(n) \geq T(n-1)+T(n-2)+1$, com $n-1<n$ e $n-2<n$.

Então por HI vale:

$$T(n) \geq T(n-1)+T(n-2)+1 \geq c\left(\frac{1+\sqrt{5}}{2}\right)^{n-1} + c\left(\frac{1+\sqrt{5}}{2}\right)^{n-2} + 1$$

$$= c\frac{\left(\frac{1+\sqrt{5}}{2}\right)^n}{\left(\frac{1+\sqrt{5}}{2}\right)} + c\frac{\left(\frac{1+\sqrt{5}}{2}\right)^n}{\left(\frac{1+\sqrt{5}}{2}\right)^2} + 1 = c\left(\frac{1+\sqrt{5}}{2}\right)^n\left(\left(\frac{1+\sqrt{5}}{2}\right)+\left(\frac{1+\sqrt{5}}{2}\right)^2\right) + 1$$

$$\geq c\left(\frac{1+\sqrt{5}}{2}\right)^n + 1 \geq c\left(\frac{1+\sqrt{5}}{2}\right)^n$$

pois $\left(\frac{1+\sqrt{5}}{2}\right)+\left(\frac{1+\sqrt{5}}{2}\right)^2 \geq 1$, já que $\sqrt{5} > \sqrt{4} = 2$.

Então $T(n)$ é $\Omega\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$ (ou seja, é pelo menos exponencial).

Problema Número de Fibonacci

→ Note como só existem n subproblemas a serem resolvidos:
$F_n, F_{n-1}, F_{n-2}, \ldots, F_2, F_1$

→ Cada subproblema é totalmente descrito por um valor i.
↳ Então um vetor de tamanho n consegue armazenar esses resultados → diretamente na posição i, guarda F_i.

Fibonacci com programação dinâmica - Abordagem Top Down

```
FIBONACCIRECURSIVO-TOPDOWN(n)
1 se F[n] == −1 então
2 |  F[n] = FIBONACCIRECURSIVO-TOPDOWN(n − 1) +
  |    FIBONACCIRECURSIVO-TOPDOWN(n − 2)
3 devolve F[n]
```

praticamente igual à versão recursiva

```
FIBONACCI-TOPDOWN(n)
1 Cria vetor F[1..n] global
2 F[1] = 1
3 F[2] = 1
4 para i = 3 até n, incrementando faça
5 |  F[i] = −1
6 devolve FIBONACCIRECURSIVO-TOPDOWN(n)
```

Fibonacci com programação dinâmica - Abordagem Bottom Up

```
FIBONACCI-BOTTOMUP(n)
1 se n ≤ 2 então
2 |  devolve 1
3 Seja F[1..n] um vetor de tamanho n
4 F[1] = 1
5 F[2] = 1
6 para i = 3 até n, incrementando faça
7 |  F[i] = F[i − 1] + F[i − 2]
8 devolve F[n]
```

Problema *Corte de Barras de Ferro*

Entrada: $\langle n, p \rangle$, onde n é um inteiro positivo indicando o tamanho da barra, sendo que cada tamanho i de barra, com $1 \leq i \leq p$, tem um preço p_i de venda.
Saída: Maior lucro obtido com a venda de pedaços inteiros de uma barra de tamanho n.

Exemplo: $n = 6$

	1	2	3	4	5	6
p	1	3	11	16	19	10
	1	1,5	3,67	4	3,8	1,67

(entrada)

Soluções viáveis:

6 pedaços de tam. 1 → $6 \cdot p_1 = 6$
3 tam. 2 → $3 \cdot p_2 = 9$
1 tam. 6 → $p_6 = 10$
1 tam. 1 + 1 tam. 2 + 1 tam. 3 → $p_1 + p_2 + p_3 = 15$
1 tam. 5 + 1 tam. 1 → $p_5 + p_1 = 20$
1 tam. 4 + 1 tam. 2 → $p_4 + p_2 = 19$

Sol. ótima:

2 tam. 3 → $2 \cdot p_3 = 22$

Como resolver esse problema?

→ Existe um número finito de soluções viáveis
↳ Descreva uma solução $S = (s_1, s_2, \ldots, s_{n-2}, s_{n-1})$ onde $s_i = 1$ se houve corte à distância i do início da barra e $s_i = 0$ c.c.

Barra de tamanho 6:

$S = (0, 1, 0, 0, 1, 0)$

↳ Conclusão: existem $\leq 2^{n-1}$ soluções

→ Então podemos fazer um algoritmo de força bruta:

```
lucro_max = -1
para cada solução S
    se S é viável e p(S) > lucro_max
        lucro_max = p(S)
devolve lucro_max
```

Como resolver esse problema?

→ Podemos criar um algoritmo guloso: precisamos escolher tamanhos de pedaços da barra.
  ↳ Escolha por melhor valor
  ↳ Escolha por melhor razão $\frac{\text{valor}}{\text{tam.}}$

→ São polinomiais e viáveis.

→ São ótimos? Não! Por quê?

→ mas note que quando cortamos um pedaço de tamanho i, temos uma barra de tamanho $n-i$ para cortar.
  ↳ Temos um subproblema!

→ Alguns cuidados:
  ↳ caso base: qual o menor tamanho de barra que conseguimos resolver diretamente?
  ↳ Se $i=n$, $n-i$ não reduz o tamanho!

Abordagem Recursiva

```
CORTA (n, p)
    se n == 0
        devolve 0
    escolha um valor i entre 1 e n-1
    S = CORTA (n-i, p)
    se S + p[i] > p[n]
        devolve S + p[i]
    devolve p[n]
```

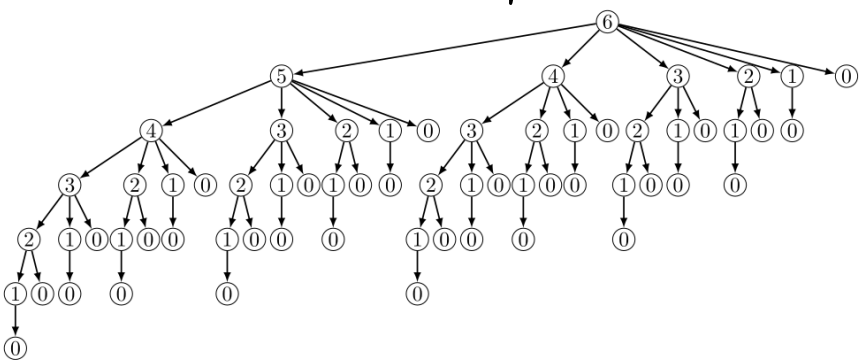
→ Qual valor de i escolher?
  ↳ De maior p_i?
  ↳ De maior p_i/i?

→ São só $n-1$ valores possíveis!

Abordagem recursiva ótima

```
CorteBarras(n)
1 se n == 0 então
2     devolve 0
3 lucro = −1
4 para i = 1 até n, incrementando faça
5     valor = p_i + CorteBarras(n − i)
6     se valor > lucro então
7         lucro = valor
8 devolve lucro
```

→ Árvore de recursão para $n = 6$.



Abordagem recursiva

① Esse algoritmo é ótimo?

Se L_t é o lucro ótimo de uma barra de tamanho t, então

$$L_t = \max_{1 \le i \le t} \{ p_i + L_{t-i} \}$$

porque a solução ótima para a barra t contém soluções ótimas para barras menores, como provado a seguir.

Seja $S = (c_1, c_2, \ldots, c_k)$ uma sequência de cortes ótima para t. Seja c_j um corte qualquer em S. Note que $S - c_j$ deve ser ótima para a barra $t - c_j$. Se não for, seja S' ótima para $t - c_j$. Veja que $S' + c_j$ é viável para t. Porém $S' + c_j$ tem lucro maior do que S (já que S' tem lucro maior que $S - c_j$), uma contradição.

② Qual seu tempo de execução?

```
CorteBarras(n)
1 se n == 0 então
2     devolve 0
3 lucro = −1
4 para i = 1 até n, incrementando faça
5     valor = p_i + CorteBarras(n − i)
6     se valor > lucro então
7         lucro = valor
8 devolve lucro
```

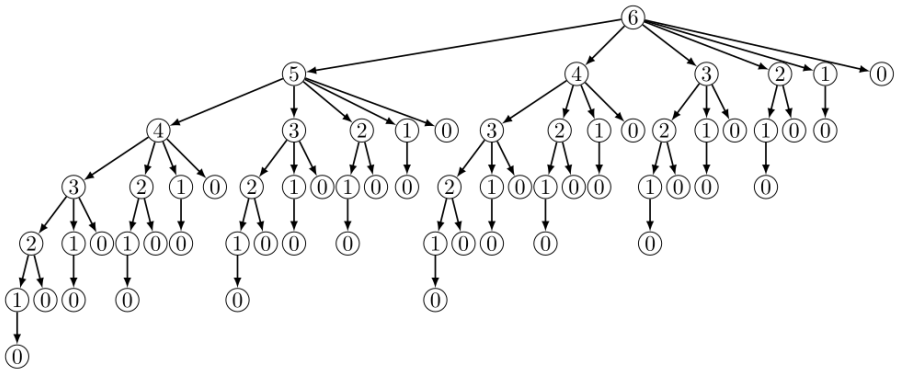
$$T(n) = \sum_{i=1}^{n} T(n-i) + n$$

Pelo método da substituição, $T(n) \geq 2^n$:

$$T(n) = n + \sum_{i=1}^{n} T(n-i) \geq n + \sum_{i=1}^{n} 2^{n-i}$$
$$= n + 2^{n-1} + 2^{n-2} + \ldots + 2^0 = n + 2^n - 1$$
$$\geq 2^n \quad (\text{se } n \geq 1).$$

③ Dá para melhorar?

Note que existem subproblemas repetidos sendo calculados várias vezes:



Mas só existem $n+1$ subproblemas ao todo!

- ↳ Um para cada tamanho de barra, de 0 a n.
- ↳ Cada um pode ser descrito por um valor i, $0 \leq i \leq n$
- ↳ Um vetor $B[0..n]$ consegue armazená-los.
  - ↳ $B[i]$ terá o lucro máximo obtido ao cortar uma barra de tamanho i
  - ↳ Queremos calcular $B[n]$

PD para Corte de Barras (abordagem top-down)

```
CorteBarrasRecursivo-TopDown(k)
1 se B[k] == −1 então
2     lucro = −1
3     para i = 1 até k, incrementando faça
4         valor = p_i + CorteBarrasRecursivo-TopDown(k − i)
5         se valor > lucro então
6             lucro = valor
7             S[k] = i
8     B[k] = lucro
9 devolve B[k]
```

```
CorteBarras-TopDown(n)
1 Cria vetores B[0..n] e S[0..n] globais
2 B[0] = 0
3 para i = 1 até n, incrementando faça
4     B[i] = −1
5 devolve CorteBarrasRecursivo-TopDown(n)
```

PD para Corte de Barras (abordagem bottom-up)

```
CorteBarras-BottomUp(n)
1  Cria vetores B[0..n] e S[0..n]
2  B[0] = 0
3  para k = 1 até n, incrementando faça
4      lucro = −1
5      para i = 1 até k, incrementando faça
6          valor = p_i + B[k − i]
7          se valor > lucro então
8              lucro = valor
9              S[k] = i
10     B[k] = lucro
11 devolve B[n]
```

Construindo uma solução

→ Para cortar a barra n e obter lucro ótimo $B[n]$, houve um corte de tamanho $S[n]$.

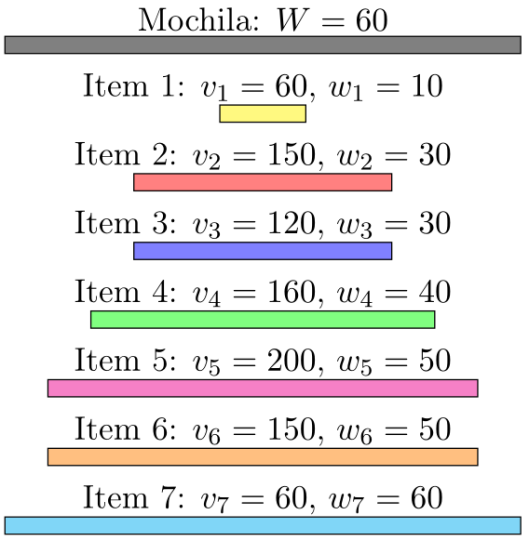
→ Nos resta uma barra $n - S[n]$, portanto.
Para o lucro ótimo dela, $B[n - S[n]]$, houve um corte de tamanho $S[n - S[n]]$, e assim por diante.
Então o seguinte algoritmo mostra os cortes que foram feitos:

```
1 enquanto n > 0 faça
2     Imprime S[n]
3     n = n − S[n]
```

Problema *Mochila Inteira*

Entrada: $\langle I, n, w, v, W\rangle$, onde $I = \{1, 2, \ldots, n\}$ é um conjunto de n itens sendo que cada $i \in I$ tem um peso w_i e um valor v_i associados, e W é a capacidade total da mochila.

Saída: Subconjunto $S \subseteq I$ de itens tal que $\sum_{i \in S} w_i \leq W$ e $\sum_{i \in S} v_i$ é máximo.



Soluções viáveis:

$S_1 = \{7\}$ valor = 60

$S_2 = \{2,1\}$ valor = 210

$S_3 = \{4,1\}$ valor = 220

$S_4 = \{5,1\}$ valor = 260

$S_5 = \{2,3\}$ valor = 270

Como resolver esse problema?

→ Existe um número finito de soluções
↳ Descreva uma solução $S \subseteq I$ como uma sequência $(s_1, \ldots, s_n)$ onde $s_i = 1$ sse $i \in S$.
↳ Existem 2^n soluções ∴ força bruta é exponencial

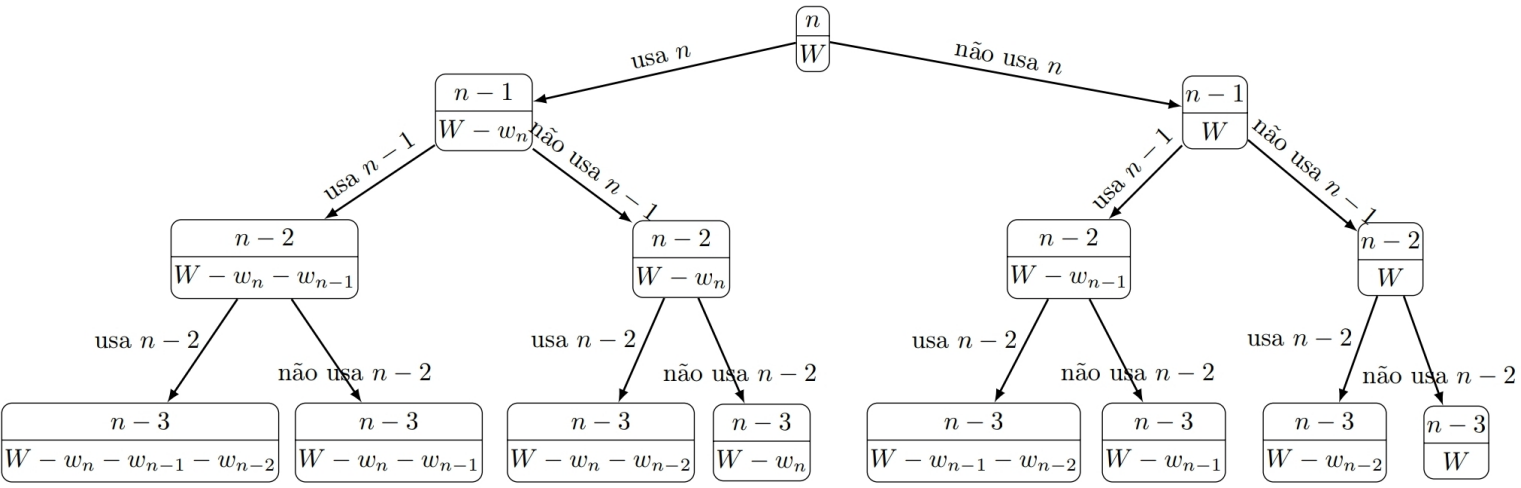
→ Guloso (escolhe item a item enquanto cabe):
↳ Escolha por melhor valor
↳ Escolha por melhor razão valor/peso.

→ Divisão e conquista:
↳ Para um certo item i, podemos escolher usá-lo ou não.

```
MochilaInteira(n, W)
1 se n == 0 ou W == 0 então
2 |  devolve 0
3 se w_n > W então
4 |  devolve MochilaInteira(n − 1, W)
5 senão
6 |  usa = v_n + MochilaInteira(n − 1, W − w_n)
7 |  naousa = MochilaInteira(n − 1, W)
8 |  devolve max{usa, naousa}
```

→ Podia ser qualquer item, mas o n-ésimo é mais conveniente para implementação

→ Problema: continua exponencial e repete subproblemas



Abordagem recursiva

① Esse algoritmo é ótimo?

Se $V_{j,x}$ é o valor ótimo para j itens e capacidade X, então

$$V_{j,X} = \begin{cases} V_{j-1,X} & \text{se } w_j > X \\ \max\{V_{j-1,X},\ V_{j-1,X-w_j} + v_j\} & \text{se } w_j \leq X \end{cases}$$

pois uma solução ótima para (j, X) contém soluções ótimas para subproblemas, como provado a seguir.

Seja $S \subseteq I$ ótima para (j, X) com $|I| = j$. Se $j \in S$, então note que $S \setminus \{j\}$ é ótima para $(j-1, X-w_j)$. Se não fosse, haveria S' ótima para $(j-1, X-w_j)$ tal que $S' \cup \{j\}$ teria valor melhor para (j, X), uma contrad. Se $j \notin S$, então note que S é ótima para $(j-1, X)$. Se não fosse, haveria S' ótima para $(j-1, X)$ tal que S' teria valor melhor do que S para (j, X), uma contradição.

② Qual o tempo de execução?

```
MochilaInteira(n, W)
1 se n == 0 ou W == 0 então
2     devolve 0
3 se w_n > W então
4     devolve MochilaInteira(n − 1, W)
5 senão
6     usa = v_n + MochilaInteira(n − 1, W − w_n)
7     naousa = MochilaInteira(n − 1, W)
8     devolve max{usa, naousa}
```

$$T(n, W) \leq T(n-1, W) + T(n-1, W-w_n) + \Theta(1)$$

$$\equiv$$

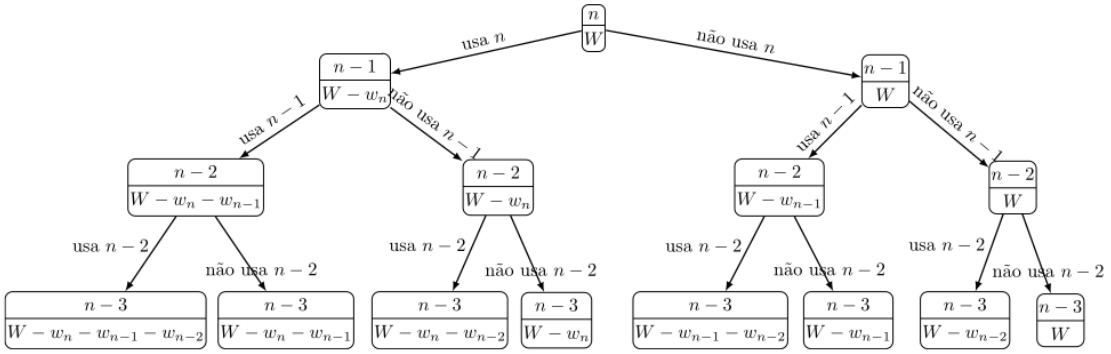
$$S(n) = 2S(n-1) + 1$$

$$\hookrightarrow \Theta(2^n)$$

$$\Rightarrow O(2^n)$$

③ Dá para melhorar?

Note que existem subproblemas repetidos sendo calculados várias vezes.



É no entanto existem $(n+1) \times (W+1)$ subproblemas diferentes!

↳ Cada um pode ser descrito por um par (j, X), com $0 \leq j \leq n$ e $0 \leq X \leq W$.

↳ Uma matriz $M[0..n][0..W]$ consegue armazená-los.

↳ $M[j][X]$ terá o maior valor ao escolher itens no subpr. (j, X), isto é, tendo j itens disp. e capac. X restante.

↳ Queremos calcular $M[n][W]$.

PD para Mochila Inteira (abordagem Top-Down)

```
MochilaInteiraRecursivo-TopDown(j, X)
1 se M[j][X] == −1 então
2     se w_j > X então
3         M[j][X] = MochilaInteiraRecursivo-TopDown(j − 1, X)
4     senão
5         usa = v_j + MochilaInteiraRecursivo-TopDown(j − 1, X − w_j)
6         naousa = MochilaInteiraRecursivo-TopDown(j − 1, X)
7         M[j][X] = max{usa, naousa}
8 devolve M[j][X]
```

```
MochilaInteira-TopDown(n, W)
1 Seja M[0..n][0..W] uma matriz global
2 para X = 0 até W, incrementando faça
3     M[0][X] = 0
4     para j = 1 até n, incrementando faça
5         M[j][X] = −1
6         M[j][0] = 0
7 devolve MochilaInteiraRecursivo-TopDown(n, W)
```

PD para Mochila Inteira (Abordagem Bottom-Up)

```
MochilaInteira-BottomUp(n, W)
1  Seja M[0..n][0..W] uma matriz
2  para X = 0 até W, incrementando faça
3      M[0][X] = 0
4  para j = 0 até n, incrementando faça
5      M[j][0] = 0
6  para j = 1 até n, incrementando faça
7      para X = 0 até W, incrementando faça
8          se w_j > X então
9              M[j][X] = M[j − 1][X]
10         senão
11             usa = v_j + M[j − 1][X − w_j]
12             naousa = M[j − 1][X]
13             M[j][X] = max{usa, naousa}
14 devolve M[n][W]
```

Construindo uma solução

→ Para obter uma solução ótima, de custo $M[n][W]$, o item n pode ter sido usado ou não.
↳ Se foi, então $M[n][W] = M[n-1][W-w_n] + v_n$
↳ Se não foi, então $M[n][W] = M[n-1][W]$

```
ConstroiMochila(n, W, M)
1  S = ∅
2  X = W
3  j = n
4  enquanto j ≥ 1 faça
5      se M[j][X] == M[j − 1][X − w_j] + v_j então
6          S = S ∪ {j}
7          X = X − w_j
8      j = j − 1
9  devolve S
```

Problema *Alinhamento de Sequências*

Entrada: $\langle X, n, Y, m, \alpha \rangle$, onde X e Y são duas sequências sobre um mesmo alfabeto Σ, em que $X = x_1 x_2 \cdots x_m$, $Y = y_1 y_2 \cdots y_n$ e $x_i, y_j \in \Sigma$, e α é uma função de pontuação.

Saída: Alinhamento entre X e Y de pontuação máxima.

$X = AGGGCT$ $m=6$ $\quad$ $Y = AGGCA$ $m=5$

```
AGGGCT-----
------AGGCA
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 = -11
```

```
AGGGCT
AGGCA-
2 2 2 -4 -4 -1 = -3
```

```
AGGGCT
AGG-CA
2 2 2 -1 2 -4 = 3
```

$\alpha(a,b) = -4$
$\alpha(a,a) = 2$
$\alpha(gap) = -1$

```
----AGGGCT
AGGCA-----
-1 -1 -1 -1 2 -1 -1 -1 -1 -1 = -7
```

```
AGGGCT-
AGG-C-A
2 2 2 -1 2 -1 -1 = 5
```

Como resolver esse problema?

→ Existe um número finito de soluções.
↳ Dados $X = x_1 \ldots x_m$ e $Y = y_1 \ldots y_n$, um alinhamento qualquer terá tamanho entre $\max\{m, n\}$ e $m + n$.

→ Podemos fazer força bruta:
↳ tente alinhar x_m com gap ou qualquer outro de Y.

→ mas na posição final do alinhamento, só há três possibilidades:
↳ x_m está alinhado com y_n
↳ x_m " " " gap
↳ y_n " " " gap

Abordagem recursiva

① Esse algoritmo é ótimo?

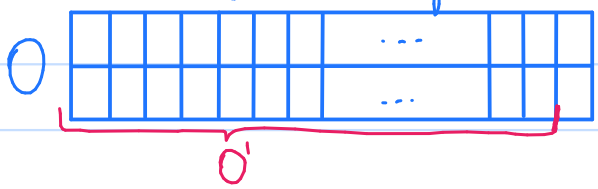
Se $P_{i,j}$ é a pontuação ótima para alinhar $x_1 \dots x_i$ com $y_1 \dots y_j$, então

$$P_{i,j} = \max \begin{cases} \alpha(x_i, x_j) + P_{i-1,j-1} \\ \alpha(gap) + P_{i-1,j} \\ \alpha(gap) + P_{i,j-1} \end{cases} \quad \text{se } i \neq 0 \text{ e } j \neq 0$$

e $P_{i,0} = i \cdot \alpha(gap)$, $P_{0,j} = j \cdot \alpha(gap)$.

Isso vale pois o alinhamento ótimo para $x_1 \dots x_i$ com $y_1 \dots y_j$ contém alinhamentos para sequências menores.

Seja O um alinhamento ótimo para $x_1 \dots x_i$ com $y_1 \dots y_j$:



Se na última posição de O tivermos x_i alinhado com y_j, então $O' = O$ sem essa posição deve ser ótimo para $x_1 \dots x_{i-1}$ com $y_1 \dots y_{j-1}$.

Se na última posição de O tivermos x_i alinhado com gap, então $O' = O$ sem essa posição deve ser ótimo para $x_1 \dots x_{i-1}$ com $y_1 \dots y_j$.

Se na última posição de O tivermos y_j alinhado com gap, então $O' = O$ sem essa posição deve ser ótimo para $x_1 \dots x_i$ com $y_1 \dots y_{j-1}$.

Se O' não fosse ótimo para os subproblemas, encontraríamos uma solução melhor que poderia ser combinada com a última posição de O e que melhoraria O.

② Qual o tempo de execução?

$$T(m,n) = T(m-1,n) + T(m,n-1) + T(m-1,n-1) + \Theta(1)$$
$$\geq 3T(m-1,n-1) + \Theta(1)$$

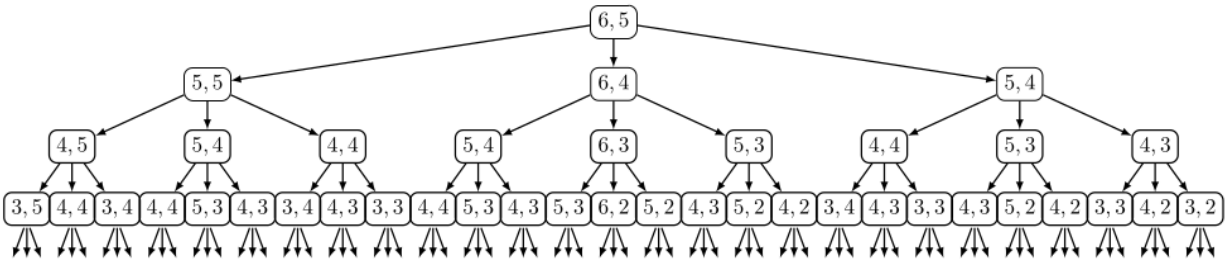
$$R(x) = 3R(x-1) + 1$$
$$= 9R(x-2) + 2$$
$$= 3^i R(x-i) + i$$
$$= 3^x + x \geq 3^x$$

$x = \min\{m,n\}$

$\Rightarrow T(m,n) = \Omega(3^{\min\{m,n\}})$

③ Dá para melhorar?

Note que existem subproblemas repetidos sendo calculados várias vezes.



Mas só existem $(m+1) \times (n+1)$ subproblemas!

↳ Cada um pode ser descrito por um par (i,j), com $0 \leq i \leq m$ e $0 \leq j \leq n$.

↳ Uma matriz $M[0..m][0..n]$ consegue armazená-los.

↳ $M[i][j]$ terá a pontuação ótima do subproblema (i,j), que é alinhar $x_1 \dots x_i$ com $y_1 \dots y_j$.

↳ Queremos calcular $M[m][n]$.

PD para Alinhamento de Sequências (Bottom-Up)

ALINHAMENTO-BOTTOMUP(X, m, Y, n, α)

```
1 Seja M[0..m][0..n] uma matriz
2 para i = 0 até m, incrementando faça
3     M[i][0] = i × α(gap)
4 para j = 0 até n, incrementando faça
5     M[0][j] = j × α(gap)
6 para i = 1 até m, incrementando faça
7     para j = 1 até n, incrementando faça
8         M[i][j] =
            max{M[i−1][j−1]+α(x_i, y_j), M[i−1][j]+α(gap), M[i][j−1]+α(gap)}
9 devolve M[m][n]
```