Programação dinâmica

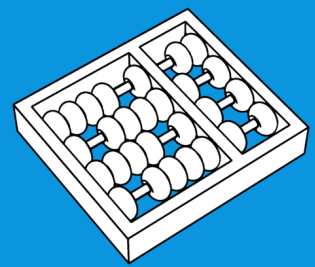
MO417 - Complexidade de Algoritmos I

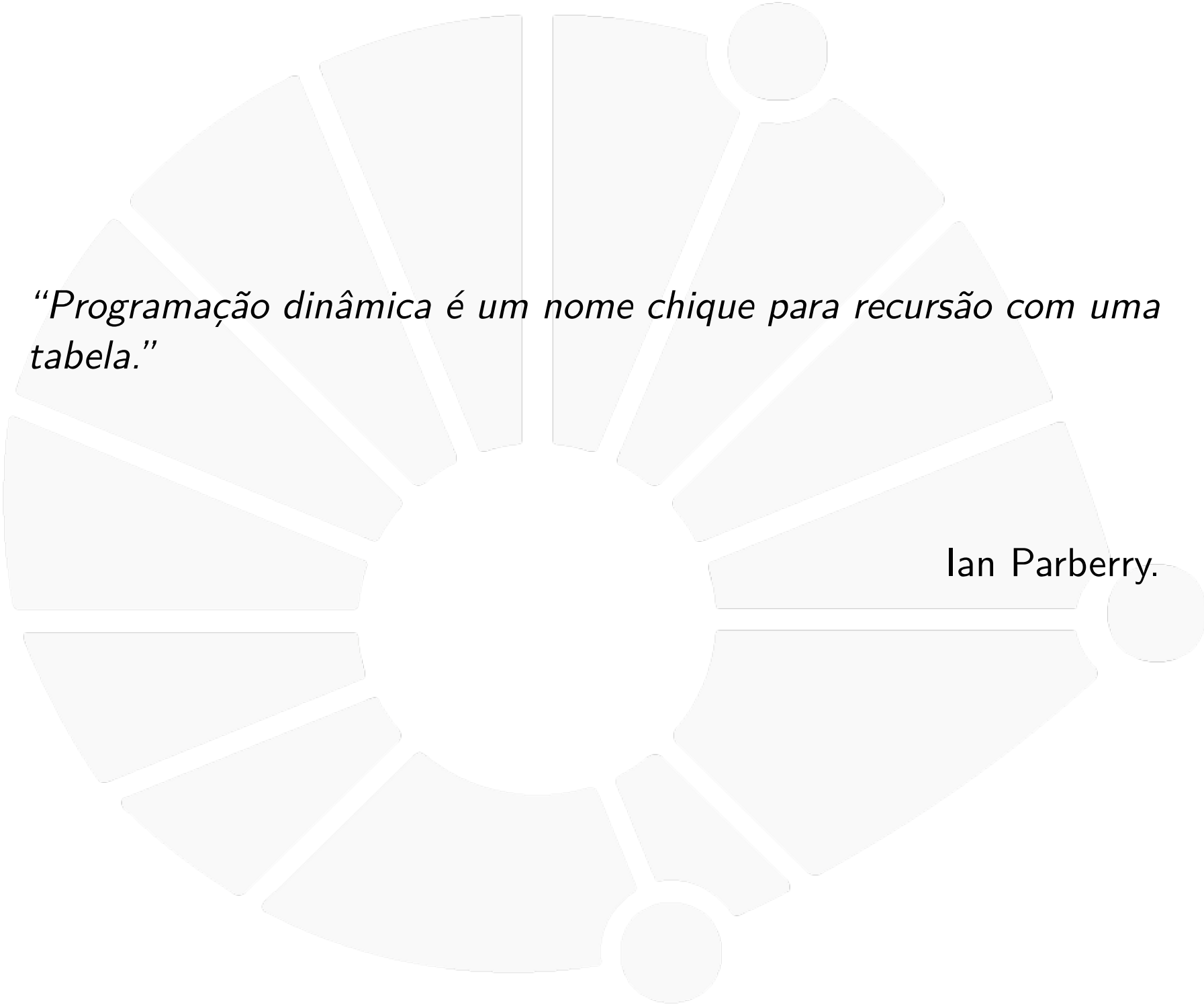
04/24

14

Santiago Valdés Ravelo
https://ic.unicamp.br/~santiago/
ravelo@unicamp.br

UNICAMP

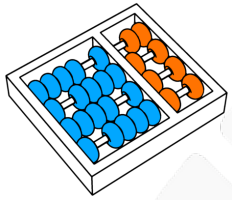




"Programação dinâmica é um nome chique para recursão com uma tabela."

Ian Parberry.

Mochila binária



Problema

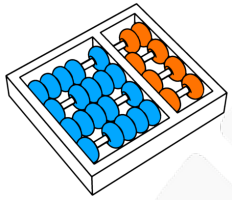
Indiana Jones descobriu um novo sítio arqueológico:

- O sítio contém diversos itens históricos.
- Ele irá levar alguns itens na mochila.
- Mas a mochila tem um limite de peso W.

Ele tem n itens disponíveis:

- Cada item i tem um peso w_i.
- Cada item i tem um valor c_i.

Pergunta: Quais itens escolher para **maximizar** o valor?



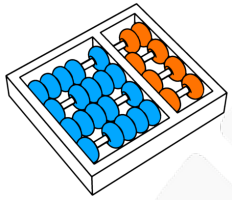
Problema da mochila binário

Problema

Entrada: *Um inteiro não negativo W representando a capacidade, um vetor de inteiros não negativos w representando pesos e um vetor de inteiros não negativos c representando valores.*
Solução: *Um subconjunto $I \subseteq \{1, 2, \ldots, n\}$ tal que $\sum_{i \in I} w_i \leq W$.*
Objetivo: ***MAXIMIZAR*** $\sum_{i \in I} c_i$.

Podemos fazer algumas suposições:

1. $\sum_{i=1}^{n} w_i > W$.
2. $0 < w_i \leq W$ para todo $i = 1, \ldots, n$.



Força bruta

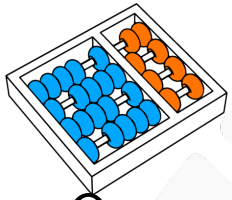
Algoritmo de força bruta:

- Há 2^n possíveis subconjuntos de itens.
- Testar cada um deles é impraticável!

Subproblema:

- Consideramos os itens em uma ordem definida $1, \ldots, n$.
- Considere números k e d tais que $0 \leq d \leq W$ e $1 \leq k \leq n$.
- Defina $z[k, d]$ como o **maior valor** de uma solução para uma instância com os k primeiros itens e uma mochila com capacidade d.

Vamos computar $z[k, d]$ utilizando indução em k.



Projeto por Indução

Queremos computar $z[k, d]$.

Base:

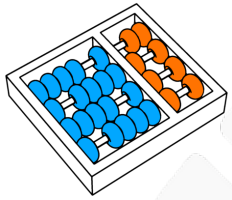
- Se $k = 0$, então $z[0, d] = 0$.

Hipótese de indução:

- Sabemos computar $z[k - 1, d']$ para todo d'.

Passo:

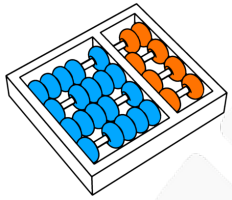
- Considere o último item k do subproblema.
  1. Se ele pertencer à solução:
     - $z[k, d] = z[k - 1, d - w_k] + c_k$
  2. Se ele **NÃO** pertencer:
     - $z[k, d] = z[k - 1, d]$
- Note que se $w_k > d$, então k não pertence à solução.
- Do contrário, não sabemos se k está na solução.
- Computamos $z[k, d]$ como o máximo entre os dois casos.



Projeto por Indução

Podemos computar $z[k, d]$ com a recorrência:

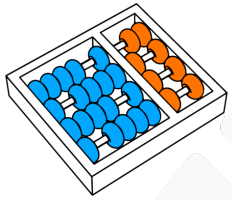
$$z[k,d] = \begin{cases} 0 & \text{se } k = 0 \\ z[k-1,d] & \text{se } w_k > d \\ \max\left\{z[k-1,d],\ z[k-1,d-w_k] + c_k\right\} & \text{se } w_k \leq d \end{cases}$$



Programação dinâmica

Observe que as capacidades são **inteiros**:

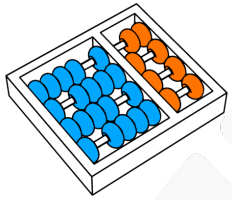
- Criamos uma tabela com $n+1$ linhas e $W+1$ colunas.
- O número de entradas da tabela é $(n+1)(W+1)$.
- Inicializamos a primeira linha com 0.
- O cálculo de uma entrada só depende da **linha anterior**!
- O valor do problema original será $z[n, W]$.



Algoritmo de programação dinâmica

Algoritmo: MOCHILA(W, w, c, n)

```
1 para d ← 0 até W
2     z[0, d] ← 0
3 para k ← 1 até n
4     para d ← 0 até W
5         z[k, d] ← z[k − 1, d]
6         se w_k ≤ d e z[k, d] < c_k + z[k − 1, d − w_k]
7             z[k, d] ← c_k + z[k − 1, d − w_k]
8 devolva z[n, W]
```



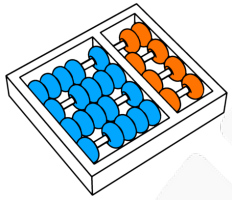
Análise

Tempo de execução:

- A complexidade de tempo é $O(nW)$.
- É um algoritmo **pseudo-polinomial**:
  - A complexidade é polinomial em n e W.
  - Mas W é um número da entrada.

Recuperando a solução:

- O algoritmo **NÃO** devolve uma solução ótima, somente o valor.
- Podemos obtê-la a partir da tabela z.

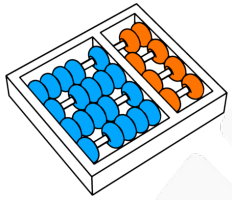


Recuperação da solução

Algoritmo: MOCHILA-SOLUÇÃO(W, z, n)

```
1 d ← W
2 para k ← n até 1
3     se z[k, d] = z[k − 1, d]
4         x[k] ← 0
5     senão
6         x[k] ← 1
7         d ← d − w_k
8 devolva x
```

- O vetor x que indica os itens de uma solução ótima.

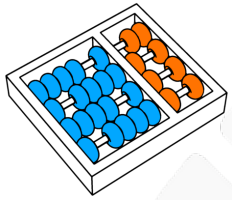


Complexidade de espaço

É possível economizar memória:

- A complexidade de espaço é $O(nW)$.
- Precisamos apenas da última linha para a recorrência.
- Assim, mantemos no máximo **duas linhas**.
- O algoritmo melhorado usa apenas $O(W)$ de espaço.
- Mas isso inviabiliza a recuperação da solução.

Árvore de busca ótima



Árvores binárias de busca

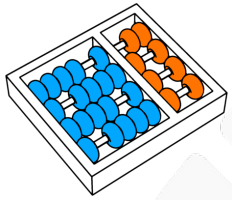
Considere um conjunto de chaves de busca:

- Elas podem ser ordenadas de forma que $e_1 < e_2 < \cdots < e_n$.
- f_i é a frequência de consulta para a chave e_i.

Vamos criar uma árvore binária de busca:

- Respeita a **propriedade de busca**.
- Ou seja, nós na subárvore esquerda de e_i têm chaves menores que e_i e nós na subárvore direita têm chaves maiores.
- A consulta a uma chave acessa os nós antecedentes.

Pergunta: Qual árvore minimiza o número de acessos a nós?



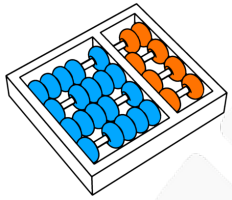
Problema da árvore de busca ótima

Problema

Entrada: *Uma sequência de chaves* $e_1 < e_2 < \cdots < e_n$ *e as frequências de consulta* f_i *para cada chave* e_i*.*

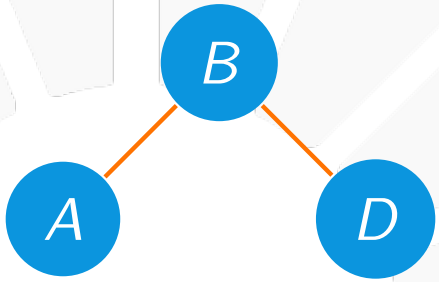
Solução: *Uma árvore binária de busca com nós* $e_1, e_2, \ldots, e_n$*.*

Objetivo: ***MINIMIZAR*** *número total de nós consultados.*



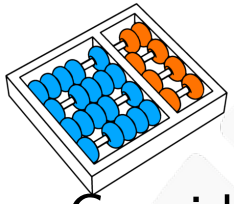
Propriedade da árvore de busca

Considere a árvore a seguir:



Seja T uma árvore de busca com $A < B < C < D$:

- **Pergunta:** A árvore acima pode ser subárvore de T?
- **Resposta: NÃO,** ela deveria conter o elemento C como raiz, mas aí não seria uma árvore de busca.



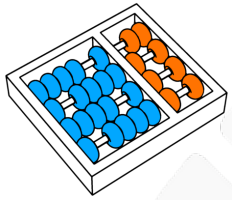
Propriedade da árvore de busca

Considere uma árvore de busca T:

- Seja T_k a subárvore enraizada em algum elemento e_k.
- Suponha que T_k contém elementos e_i e e_j para $i < j$.
- Então T_k contém **todos** os elementos e_ℓ de e_i até e_j.

Podemos mostrar isso da seguinte forma:

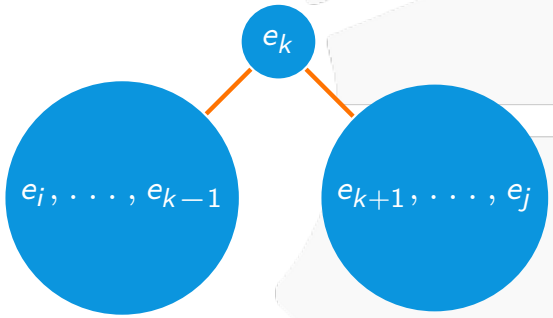
- Considere e_ℓ com $i < \ell < j$.
- Suponha que e_ℓ também não é descendente de e_k.
- Tampouco e_ℓ pode ser antecessor de e_k.
- Se e_a é o primeiro ancestral comum a e_k e a e_ℓ.
- Então $k < a < \ell$ ou $\ell < a < k$:
  1. No primeiro caso teríamos $j < a < \ell$.
  2. No segundo caso teríamos $\ell < a < i$.
- Isso é uma contradição, então e_ℓ é descendente de e_k.



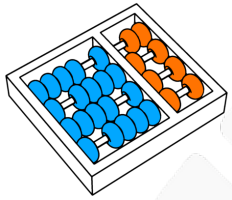
Subestrutura ótima

Considere uma árvore de busca T enraizada em e_k.

- Suponha que ela contém $\{e_i, \ldots, e_{k-1}, e_k, e_{k+1}, \ldots, e_j\}$:
  - No ramo esquerdo deve haver os elementos $e_i, \ldots, e_{k-1}$.
  - No ramo direito deve haver os elementos $e_{k+1}, \ldots, e_j$.



- Se T é uma árvore de busca ótima.
- Então as **subárvores** esquerda e direita também são.



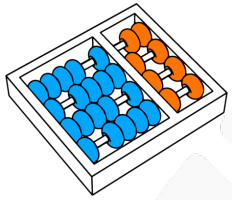
Subproblema

Definimos o seguinte subproblema:

- Considere um par de índices (i, j).
- Seja $a[i, j]$ o **menor** número de acessos de uma árvore de busca que contém $e_i, \ldots, e_j$.

Podemos computar $a[i, j]$ usando uma recorrência:

$$a[i,j] = \begin{cases} 0 & \text{se } i > j, \\ \displaystyle\sum_{t=i}^{j} f_t + \min_{i \le k \le j} \Big\{ a[i, k-1] + a[k+1, j] \Big\} & \text{se } i \le j \end{cases}$$



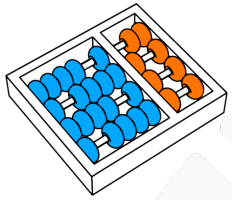
Correção da recorrência

Lema

O número de acessos a nós de uma árvore de busca que contém $e_i, \ldots, e_j$ *é pelo menos* $a[i, j]$.

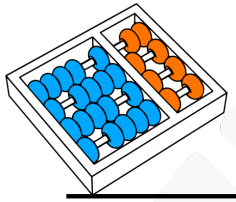
Iremos mostrar por indução em $n = j - i + 1$:

- Para $n = 0$, a árvore é vazia e o número de acessos é 0.
- Suponha que $n \geq 1$ e seja e_j a raiz de uma árvore ótima.
- Para cada $t = i, \ldots, j$ devemos acessar a raiz f_t vezes.
- Além disso, devemos somar os acessos às subárvores.
- Pela hipótese de indução, o número de acessos da árvore esquerda é pelo menos $a[i, k-1]$.
- O da subárvore direta é computado por $a[k+1, j]$.
- Como consideramos o mínimo entre todos k, o lema segue.



Implementando o algoritmo

- Podemos pré-computar $f_{ij} = \sum_{t=i}^{j} f_t$ em $\Theta(n^2)$.
- Considere elementos dummy e_0 e e_{n+1} com $f_0 = f_{n+1} = 0$.
- $a[i, j]$ é o número acessos para a subárvore com $e_i, \ldots, e_j$.
- $r[i, j]$ indica o índice raiz dessa subárvore.
- Preenchemos a tabela em ordem do tamanho $u = j - i + 1$.



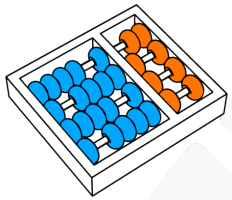
Algoritmo

Algoritmo: ÁRVORE-BUSCA(f)

```
1  para i ← 1 até n + 1
2      a[i, i − 1] ← 0        ▷ subárvore vazia
3  para u ← 1 até n
4      para i ← 1 até n − u + 1
5          j ← i + u − 1
6          a[i, j] ← ∞
7          para k ← i até j
8              aux ← f_ij + a[i, k − 1] + a[k + 1, j]
9              se a[i, j] > aux
10                 a[i, j] ← aux
11                 r[i, j] ← k
12 devolva a[1, n]
```

- A complexidade de tempo é $O(n^3)$

Subsequência comum máxima



Subsequência

Vamos trabalhar com sequências de símbolos:

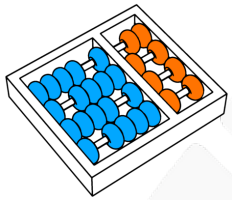
- Considere uma sequência $X = \langle x_1, x_2, \ldots, x_m \rangle$
- e uma outra sequência $Z = \langle z_1, z_2, \ldots, z_k \rangle$.

Dizemos que Z é uma **subsequência** de X se:

- Existe uma sequência crescente de índices $\langle i_1, i_2, \ldots, i_k \rangle$
- $x_{i_j} = z_j$ para cada $j = 1, 2, \ldots, k$.

Exemplo:

- Sequência $X = ABCDEFG$.
- Subsequência $Z = ADFG$.



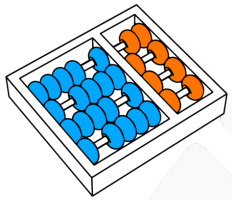
Subsequência comum máxima

Problema (Subsequência comum máxima)

Entrada: *Duas sequências sequência X e Y.*

Solução: *Uma subsequência* ***comum*** *Z de X e Y.*

Objetivo: ***MAXIMIZAR*** *o comprimento de Z.*



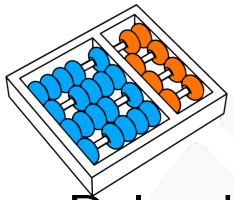
Subestrutura ótima

Vamos estudar uma solução ótima:

- Considere as sequências $X = \langle x_1 \dots x_m \rangle$ e $Y = \langle y_1 \dots y_n \rangle$:
- A subsequência comum máxima é $\mathbf{Z = \langle z_1 \dots z_k \rangle}$.

Seja S uma sequência de comprimento n:

- Denotamos por S_i o **prefixo** de S de comprimento i.
- Por exemplo, se $S = ABCDEFG$, então $S_2 = AB$.



Projeto por indução

Relembrando: $X = \langle x_1 \dots x_m \rangle$ e $Y = \langle y_1 \dots y_n \rangle$.

Base:

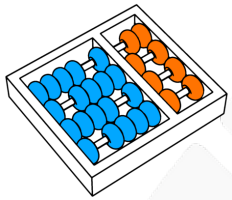
- Se $m = 0$ ou $n = 0$, então $Z = \emptyset$ com tamanho 0.

Hipótese de indução:

- Sabemos computar a solução se X' tem tamanho m' e Y' tem tamanho n', onde $0 \leq m' < m$ e $0 \leq n' < n$.

Passo:

1. Se $x_m = y_n$:
   - $\mathbf{z_k} = x_m = y_n$ é o último elemento da solução ótima.
   - $\mathbf{Z_{k-1}}$ é subsequência comum máxima de X_{m-1} e Y_{n-1}.
2. Se $x_m \neq y_n$:
   - Pelo menos x_m ou y_n não é parte da solução ótima.
   - Logo, $\mathbf{Z}$ é subsequência comum máxima de X_{m-1} e Y,
   - ou $\mathbf{Z}$ é subsequência comum máxima de X e Y_{n-1}.



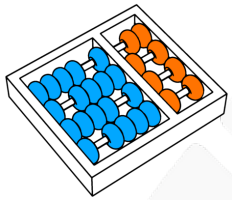
Subproblema

Definimos o seguinte **subproblema**:

- Considere $i = 0, 1, \ldots, m$ e $j = 0, 1, \ldots, n$.
- Seja $c[i, j]$ o comprimento da subsequência comum mais longa dos prefixos X_i e Y_j.

Podemos utilizar a seguinte recorrência:

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

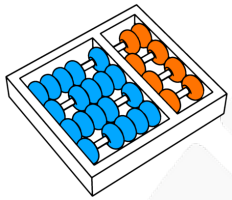


Algoritmo de programação dinâmica

Algoritmo: $\mathrm{SCM}(X, m, Y, n)$

```
1  para i ← 0 até m
2      c[i, 0] ← 0
3  para j ← 1 até n
4      c[0, j] ← 0
5  para i ← 1 até m
6      para j = 1 até n
7          se x_i = y_j
8              c[i, j] ← c[i − 1, j − 1] + 1
9              b[i, j] ←"xy"
10         senão
11             se c[i, j − 1] > c[i − 1, j]
12                 c[i, j] ← c[i, j − 1]
13                 b[i, j] ←"x"
14             senão
15                 c[i, j] ← c[i − 1, j]
16                 b[i, j] ←"y"
17 devolva c[m, n]
```

- A tabela b guarda quais subproblemas foram escolhidos.



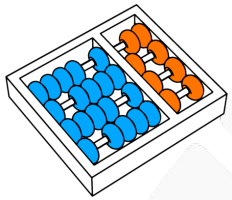
Análise

Complexidade de tempo:

- Cada entrada da tabela é preenchida em tempo constante.
- Assim, o algoritmo gasta tempo $O(mn)$.

Complexidade de espaço:

- O algoritmo gasta tabelas de tamanho total $O(mn)$.
- Podemos manter apenas duas linhas ou colunas.
- O algoritmo melhorado usa memória $O(\min\{m, n\})$.
- Mas manter a tabela b permite encontrar a solução.



Recuperando uma solução

Algoritmo: RECUPERA-SCM(b, X, i, j)

```
1  se i = 0 ou j = 0
2      retorne
3  se b[i, j] = "xy"
4      RECUPERA-SCM(b, X, i − 1, j − 1)
5      imprima x_i
6  senão
7      se b[i, j] = "x"
8          RECUPERA-SCM(b, X, i, j − 1)
9      senão
10         RECUPERA-SCM(b, X, i − 1, j)
```

- A chamada inicial é RECUPERA-SCM(b, X, m, n).

Programação dinâmica

MO417 - Complexidade de Algoritmos I

04/24

14

Santiago Valdés Ravelo
https://ic.unicamp.br/~santiago/
ravelo@unicamp.br

UNICAMP