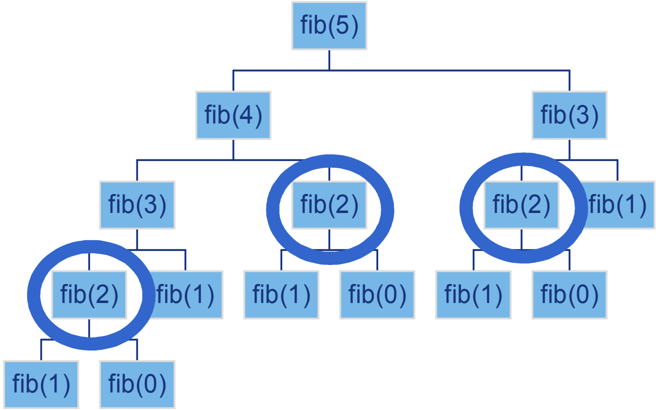
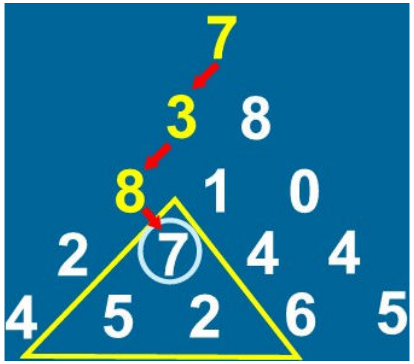
| i \ j | 0 | 1 A | 2 F | 3 O | 4 G | 5 A |
|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 3 | 4 | 5 |
| 1 G | 1 | 1 | 2 | 3 | 3 | 4 |
| 2 O | 2 | 2 | 2 | 2 | 3 | 4 |
| 3 T | 3 | 3 | 3 | 3 | 3 | 4 |
| 4 A | 4 | 3 | 4 | 4 | 4 | 3 |
| 5 S | 5 | 4 | 4 | 5 | 5 | 4 |




# Programação Dinâmica

## Uma metodologia de resolução de problemas


**Pedro Ribeiro**

Center for Research in Advanced Computing Systems
(CRACS & INESC-Porto LA)

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto

# Conteúdo

- **Motivação e conceitos base**
  - Exemplos iniciais e cálculos repetidos
- **Programação Dinâmica**
  - Definição
  - Características de um problema de PD
  - Passos para chegar a uma solução
- **Exemplos de aplicação**
  - PD clássica, de contagem, com jogos e mais complexa

# Números de Fibonacci

- **Sequência de números** muito famosa definida por Leonardo Fibonacci

0,1,1,2,3,5,8,13,21,34,...

- F(0) = 0
  F(1) = 1
  F(n) = F(n-1) + F(n-2)

# Números de Fibonacci

- Como implementar?

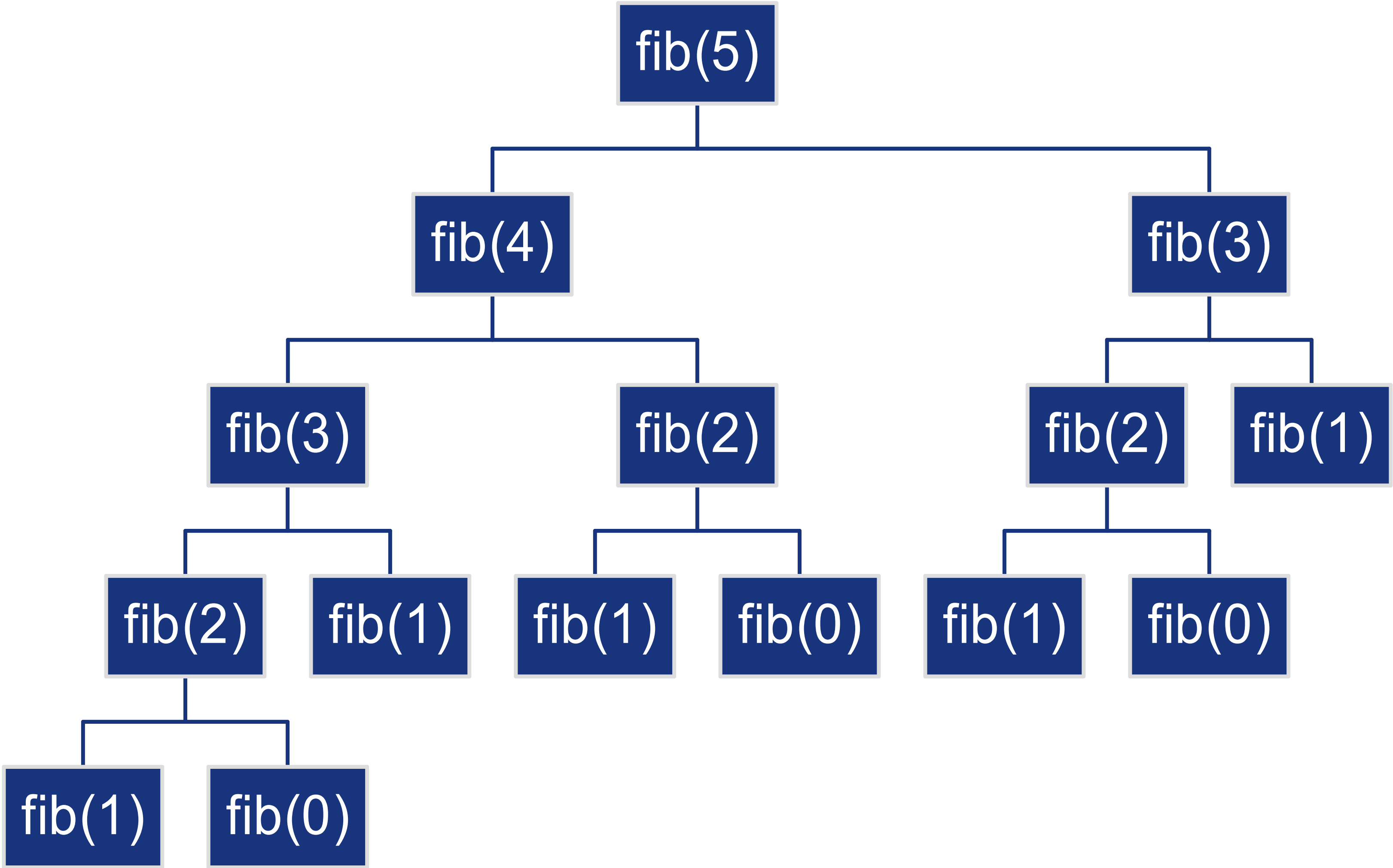
- Implementação directa a partir da definição:

```
fib(n):
    Se n=0 ou n=1 então
        retornar n
    Senão
        retornar fib(n-1) + fib(n-2)
```

- Pontos negativos desta implementação?

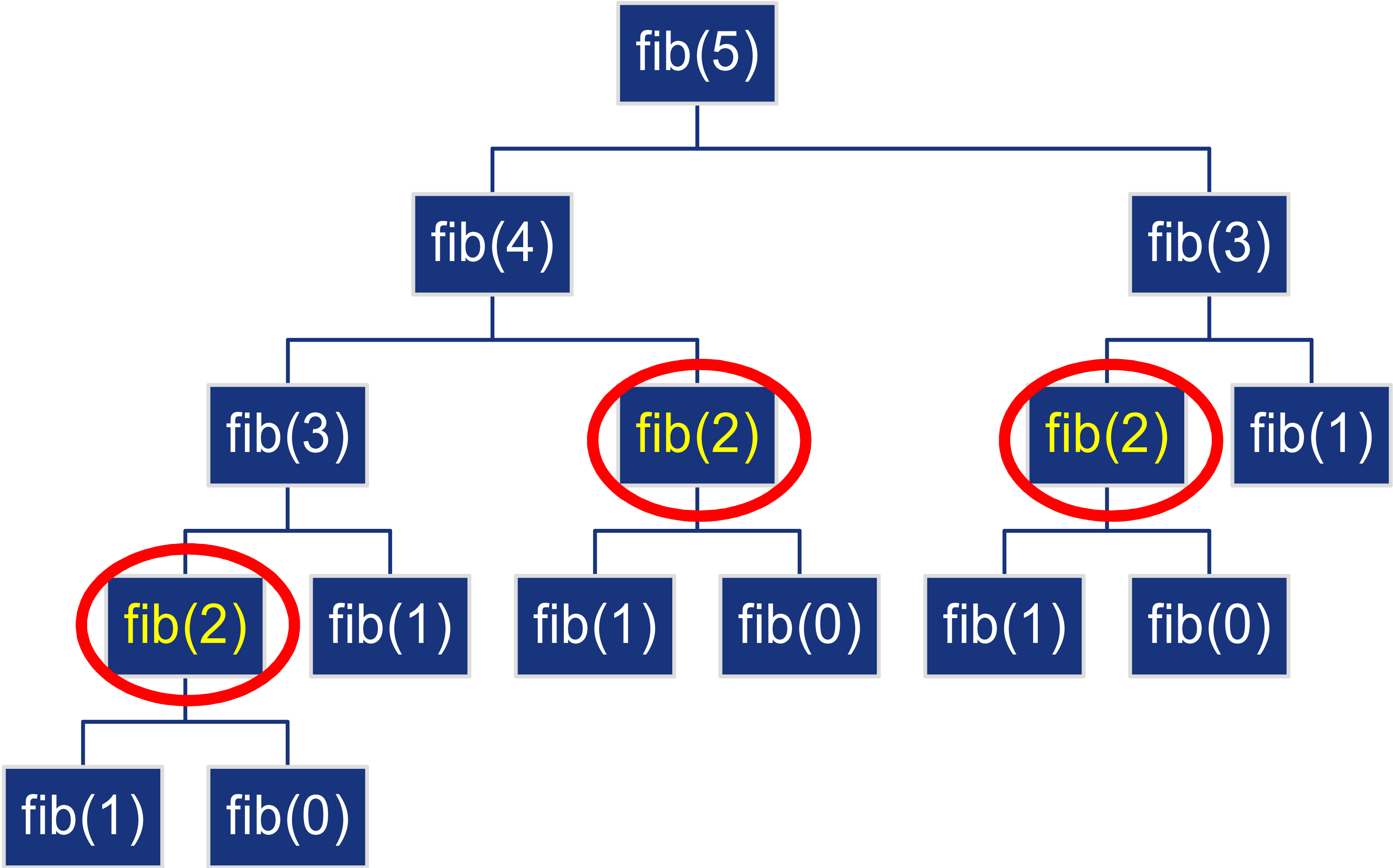
# Números de Fibonacci

- Cálculos de fib(5)



# Números de Fibonacci

- Cálculos de fib(5)




Por exemplo, fib(2) é chamado 3 vezes!

# Números de Fibonacci

- Como melhorar?
  - Por exemplo, começando do zero e mantendo sempre em memória os **dois últimos números** da sequência

```
fib(n):
    Se n=0 ou n=1 então
        retornar n
    Senão
        f1 = 1
        f2 = 0
        Para  i: 2 até n fazer
            f = f1 + f2
            f2 = f1
            f1 = f
        retornar  f
```

# Números de Fibonacci

- **Conceitos** a reter:
  - Divisão de um problema em **subproblemas do mesmo tipo**
  - Calcular o mesmo subproblema **apenas uma vez**

- Será que estas ideias podem também ser usadas noutros problemas mais complicados?

# Pirâmide de Números

- Problema clássico das **Olimpíadas Internacionais de Informática de 1994**

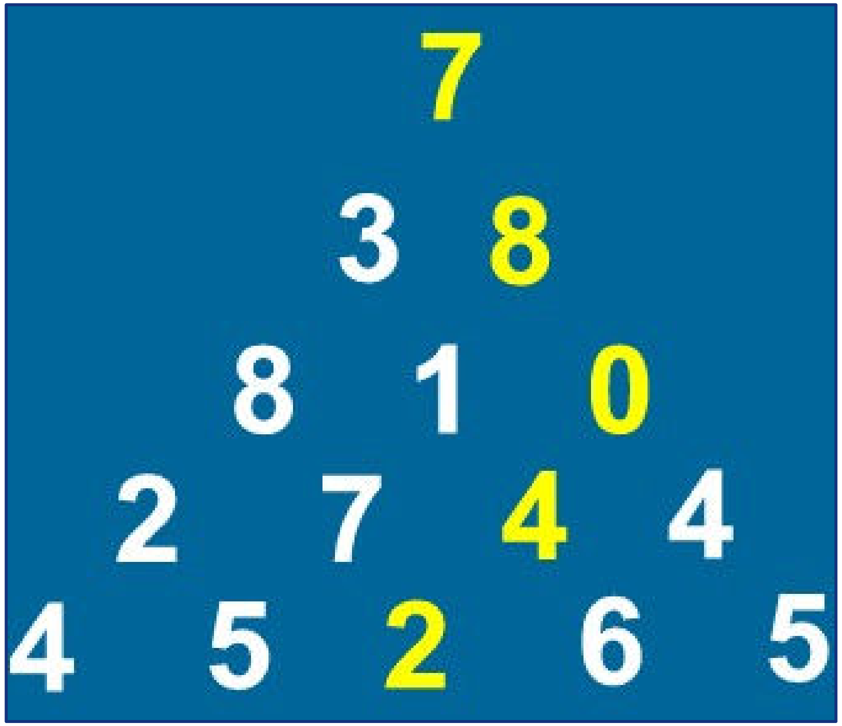
```
        7
      3   8
    8   1   0
  2   7   4   4
4   5   2   6   5
```

**Problema**

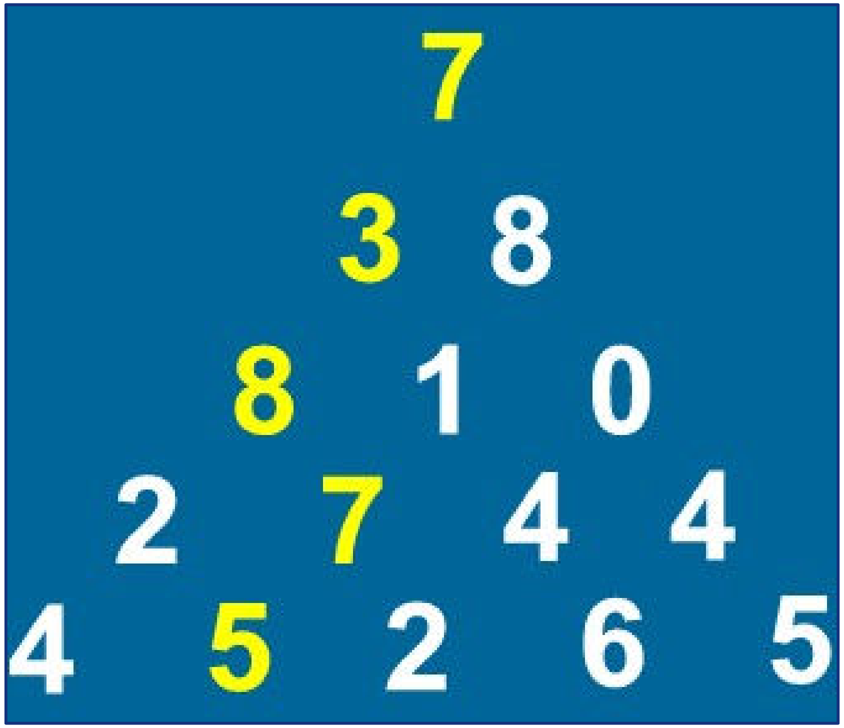
Calcular a rota, que começa no topo da pirâmide e acaba na base, com maior soma . Em cada passo podemos ir diagonalmente para baixo e para a esquerda ou para baixo e para a direita.

# Pirâmide de Números

- Duas possíveis rotas




**Soma = 21**




**Soma = 30**

- **Limites:** todos os números da pirâmide são inteiros entre 0 e 99 e o número de linhas do triângulo é no máximo 100.

# Pirâmide de Números

- Como resolver o problema?

- Ideia: **Força Bruta!**
  - avaliar todos os caminhos possíveis e ver qual o melhor.

- Mas quanto tempo demora isto?
  - Quantos caminhos existem?

# Pirâmide de Números

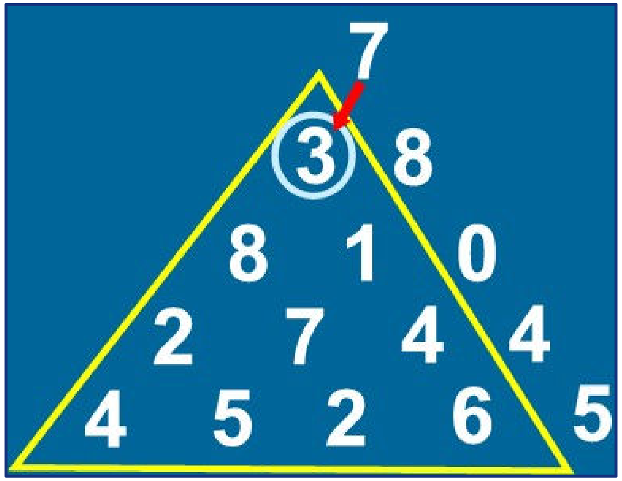
## Análise da complexidade

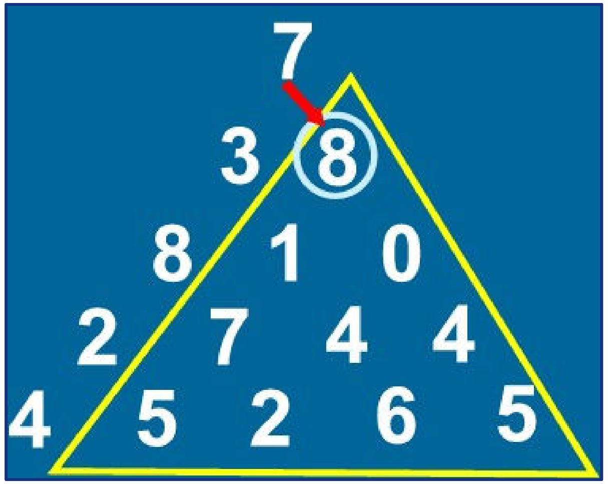
- Em cada linha podemos tomar **duas** decisões diferentes: esquerda ou direita
- Seja n a altura da pirâmide. Uma rota é constituída por **n–1** decisões diferentes.
- Existem $\mathbf{2^{n-1}}$ caminhos diferentes. Então, um programa que calculasse todas rotas teria complexidade temporal $\mathbf{O(2^n)}$: crescimento exponencial!
- Note-se que $\mathbf{2^{99} \approx 6{,}34 \times 10^{29}}$, que é um número demasiado grande!

**633825300114114700748351602688**

# Pirâmide de Números

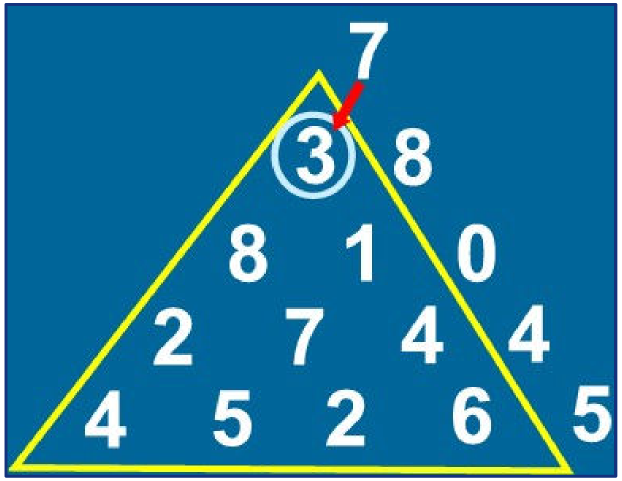
- Quando estamos no topo da pirâmide, temos duas decisões possíveis (esquerda ou direita):

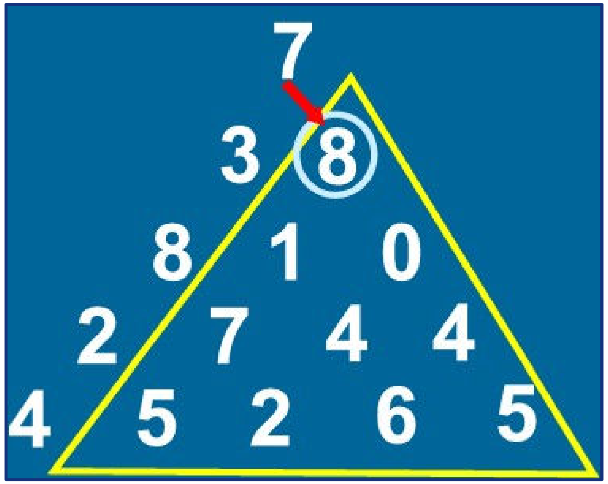



- Em cada um dos casos, temos de ter em conta todas as rotas das respectivas subpirâmides assinaladas a amarelo.

# Pirâmide de Números




- Mas o que nos interessa saber sobre estas subpirâmides?

**Apenas interessa o valor da sua melhor rota interna (que é um instância mais pequena do mesmo problema)!**

- Para o exemplo, a solução é 7 mais o máximo entre o valor da melhor rota de cada uma das subpirâmides

# Pirâmide de Números

- Então este problema pode ser resolvido recursivamente.
  - Seja **P[i][j]** o j-ésimo número da i-ésima linha
  - Seja **Max(i,j)** o melhor que conseguimos a partir da posição i,j

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 7 | | | | |
| 2 | 3 | 8 | | | |
| 3 | 8 | 1 | 0 | | |
| 4 | 2 | 7 | 4 | 4 | |
| 5 | 4 | 5 | 2 | 6 | 5 |

7
3 8
8 1 0
2 7 4 4
4 5 2 6 5

# Pirâmide de Números

- Então:

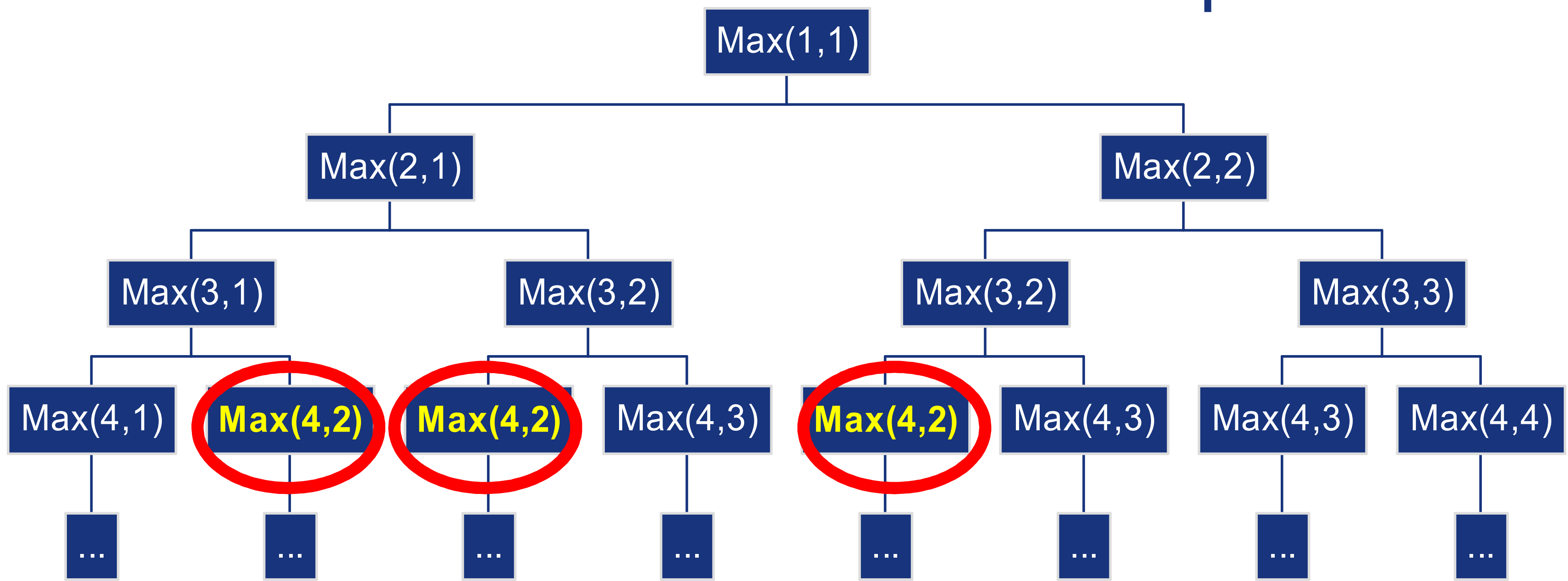
```
Max(i,j):
    Se  i = n  então
        Max(i,j) = P[i][j]
    Senão
        Max(i,j) = P[i][j] + máximo(Max(i+1,j), Max(i+1,j+1))
```

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 7 | | | | |
| 2 | 3 | 8 | | | |
| 3 | 8 | 1 | 0 | | |
| 4 | 2 | 7 | 4 | 4 | |
| 5 | 4 | 5 | 2 | 6 | 5 |

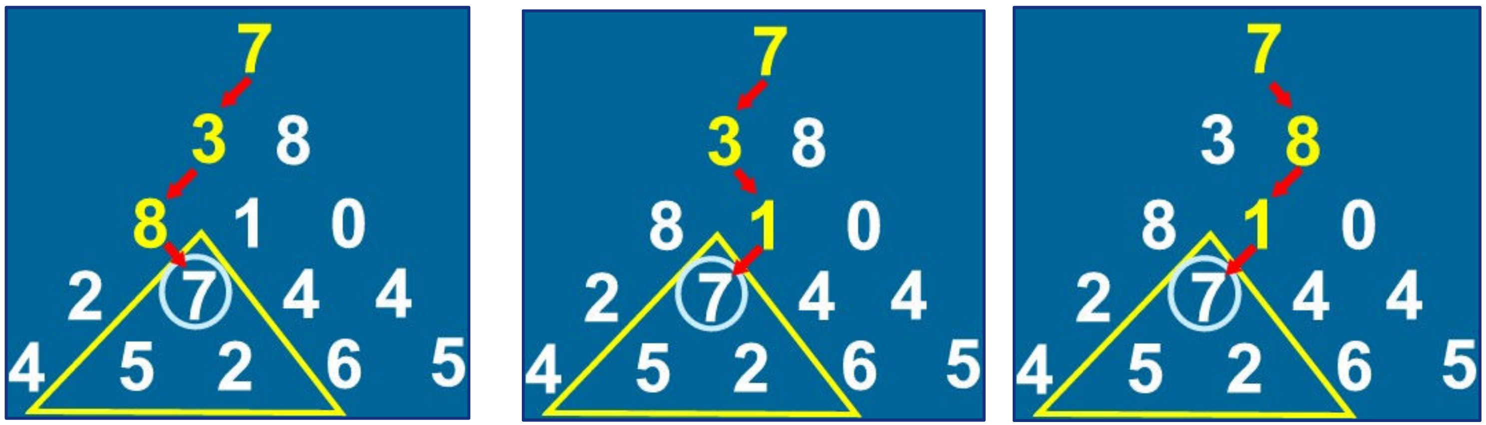
Para resolver o problema basta chamar Max(1,1)

# Pirâmide de Números

- Continuamos com crescimento exponencial!




- Estamos a avaliar o mesmo subproblema várias vezes!

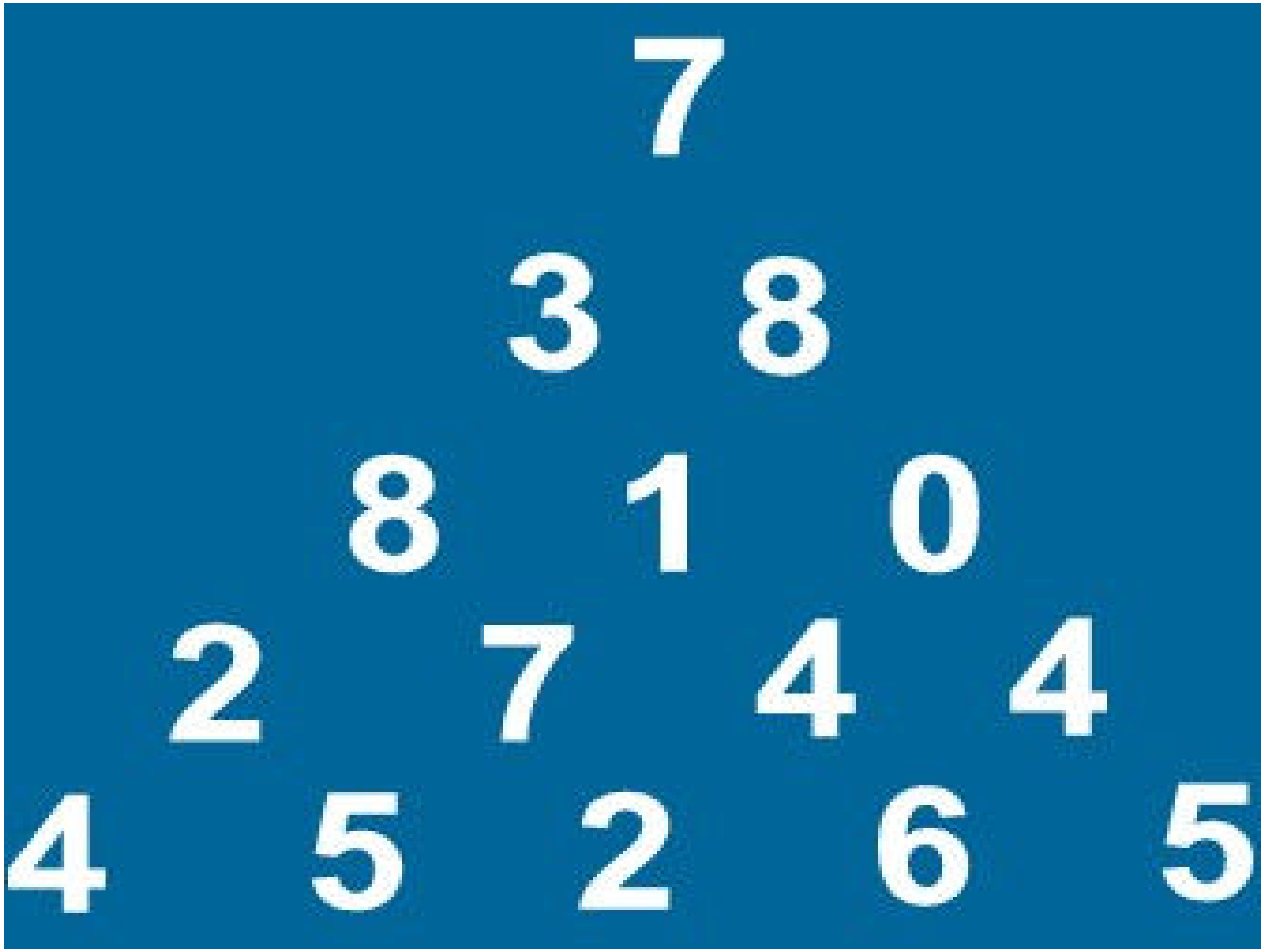


# Pirâmide de Números

- Temos de **reaproveitar** o que já calculamos
  - Só calcular uma vez o mesmo subproblema!
- Ideia: criar uma **tabela** com o valor obtido para cada subproblema!
  - Matriz M[i][j]
- Será que existe uma **ordem para preencher a tabela** de modo a que quando precisamos de um valor já o temos?

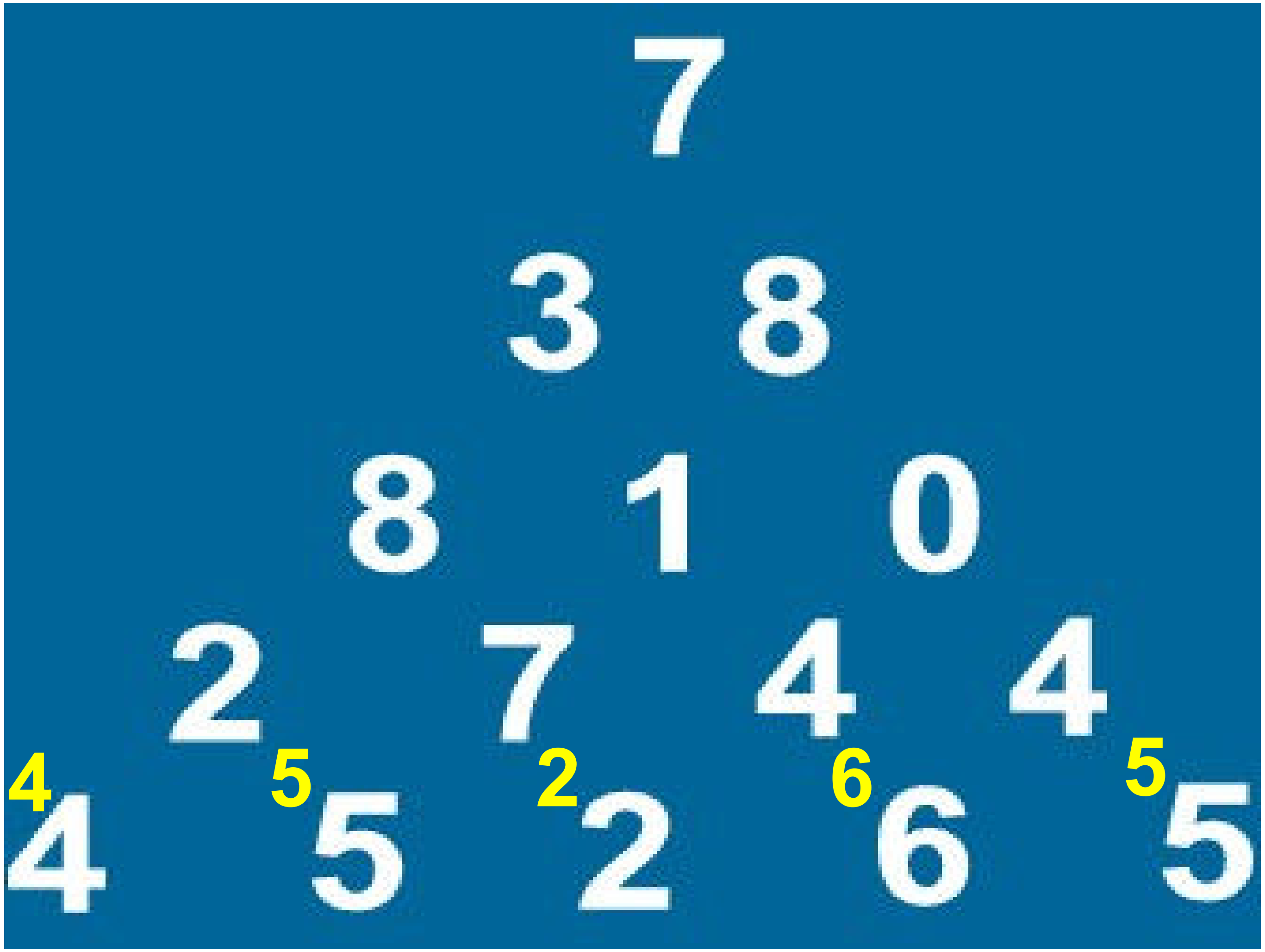
# Pirâmide de Números

- Começar a partir do fim!



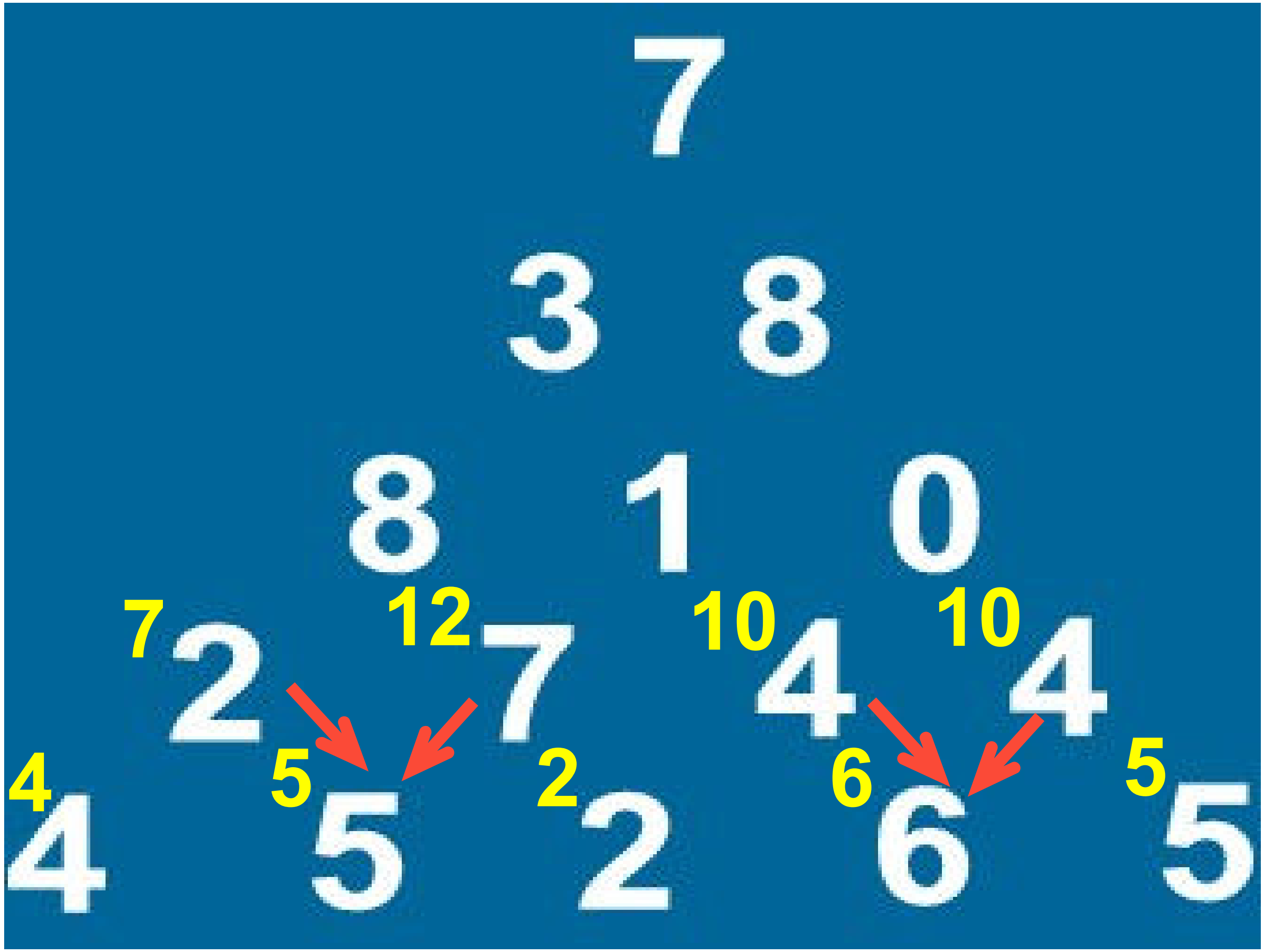
# Pirâmide de Números

- Começar a partir do fim!



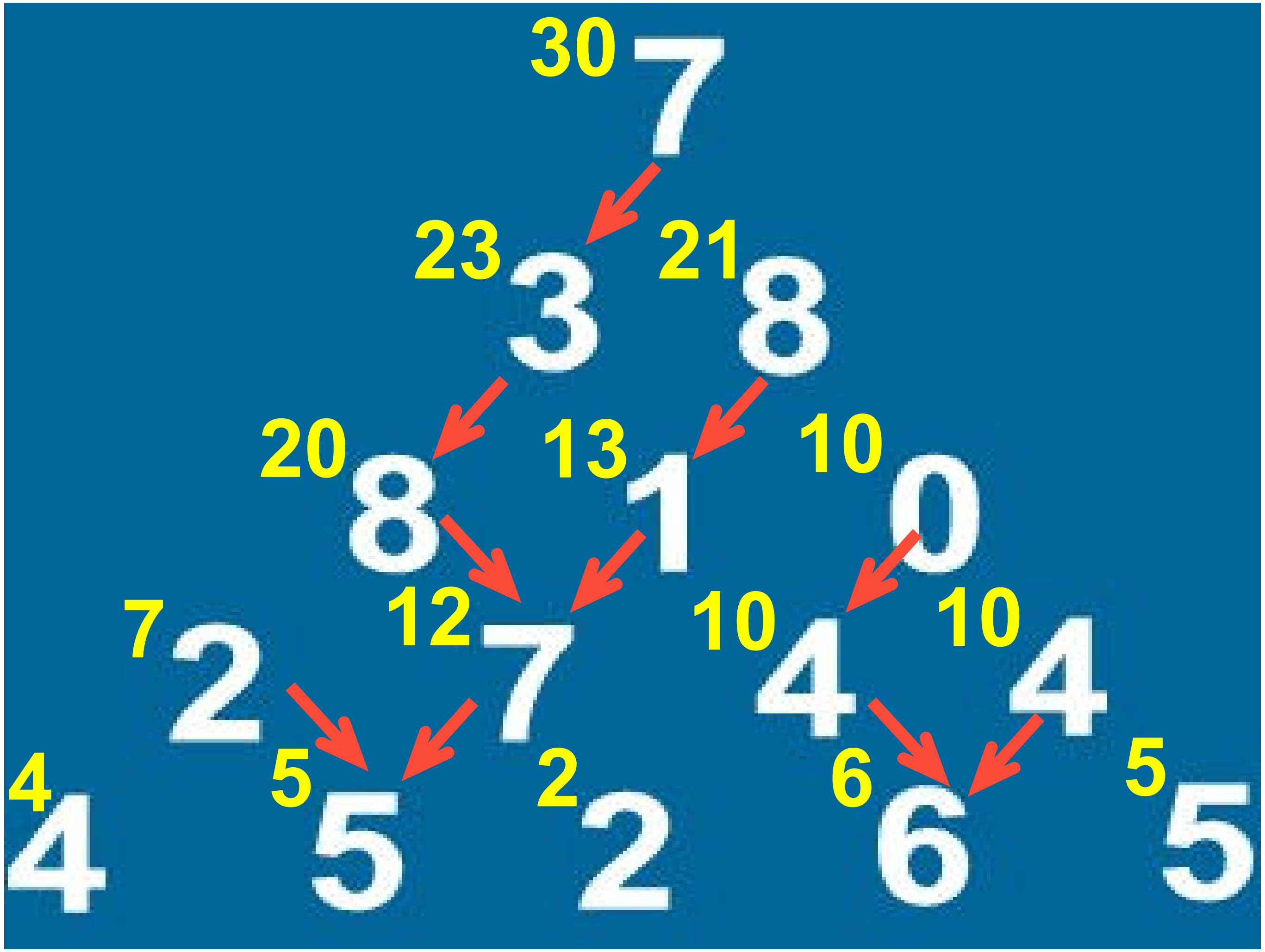
# Pirâmide de Números

- Começar a partir do fim!



# Pirâmide de Números

- Começar a partir do fim!



# Pirâmide de Números

- Tendo em conta a maneira como preenchemos a tabela, até podemos aproveitar **P[][]**!

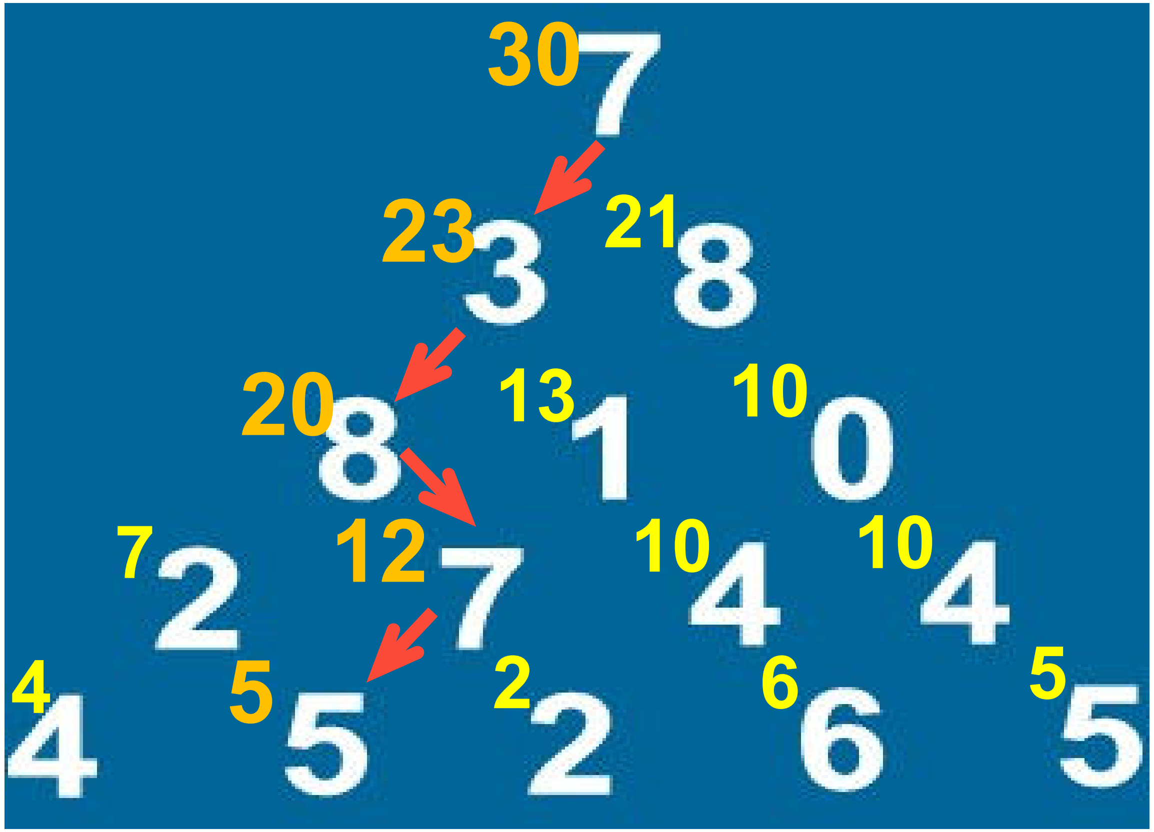
```
Calcular ():
    Para  i: n-1 até 1 fazer
        Para  j: 1 até i fazer
            P[i][j] = P[i][j] + máximo(P[i+1][j], P[i+1][j+1])
```

- Com isto solução fica em **P[1][1]**!

- Agora, o tempo necessário para resolver o problema já só cresce **polinomialmente** ( **O($n^2$)** ) e já temos uma solução admissível para o problema (**$99^2$=9801**)

# Pirâmide de Números

- Se fosse necessário saber constituição da melhor solução?
  - Basta usar a tabela já calculada!



# Pirâmide de Números

- Para resolver o problema da pirâmide de números usamos...

**Programação Dinâmica (PD)**

# Programação Dinâmica (PD)

- **Definição (adaptado de NIST-DADSP*):**
  - uma técnica algorítmica, normalmente usada em problemas de optimização, que é baseada em guardar os resultados de subproblemas, em vez de os recalcular.

  - **Técnica algorítmica**: método geral para resolver problemas que têm algumas características em comum
  - **Problema de optimização**: quando se pretende encontrar a "melhor" solução entre todas as soluções admissíveis, mediante um determinado critério (**função objectivo**).

**Clássica troca de espaço por tempo**

* **NIST** – National Institute of Standarts and Technology
**DADSP** – Dictionary of Algorithms, Data Structures, and Problems

# PD - Características

- Quais são então as características que um problema deve apresentar para poder ser resolvido com PD?
  - Subestrutura óptima
  - Subproblemas coincidentes

# PD - Características

- **Subestrutura óptima** **("optimal substructure")**:
  - Quando a solução óptima de um problema contém nela própria soluções óptimas para subproblemas do mesmo tipo.

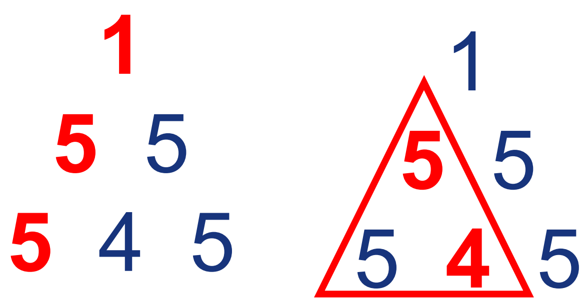
> **Exemplo:** *No problema das pirâmides de números, a solução óptima contém nela própria os melhores percursos de subpirâmides, ou seja, soluções óptimas de subproblemas.*

  - Quando um problema apresenta esta característica diz-se que ele respeita o **princípio de optimalidade** ("optimality principle").

# PD - Características

- **Subestrutura óptima** **("optimal substructure"):**
  - É preciso ter cuidado porque isto nem sempre acontece!

> **Exemplo:** *se, no problema das pirâmides, o objectivo fosse encontrar a rota que maximizasse o resto da divisão inteira entre 10 e o valor dessa rota.*




A solução óptima (1→5→5) não contém a solução óptima para a subpirâmide assinalada a amarelo (5→4)

# PD - Características

## Subproblemas coincidentes:

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

**Exemplo:** *no problema das pirâmides, para um determinada instância do problema, existem apenas* $n+(n-1)+...+1<n^2$ *subproblemas (crescem polinomialmente) pois, como já vimos, muitos subproblemas que aparecem são coincidentes.*

- Também esta característica nem sempre acontece, quer porque mesmo com subproblemas coincidentes são muitos subproblemas a resolver, quer porque não existem subproblemas coincidentes.

**Exemplo:** *no quicksort, cada chamada recursiva é feita a um subproblema novo, diferente de todos os outros.*

# PD - Metodologia

- Se um problema apresenta estas **duas características**, temos uma boa **pista** de que a PD se pode aplicar. No entanto, nem sempre isso acontece.

- Que **passos** devemos então tomar se quisermos resolver um problema usando PD?

# PD - Metodologia

(**nota:** estes passos representam apenas um guia de resolução)

**1)** Caracterizar a solução óptima do problema

**2)** Definir recursivamente a solução óptima, em função de soluções óptimas de subproblemas

**3)** Calcular as soluções de todos os subproblemas: "de trás para a frente" ou com "memoization"

**4)** Reconstruir a solução óptima, baseada nos cálculos efectuados (opcional)

# PD - Metodologia

## 1) Caracterizar a solução óptima de um problema

- Compreender bem o problema
- Verificar se um algoritmo que verifique todas as soluções à força bruta não é suficiente
- Tentar generalizar o problema (é preciso prática para perceber como generalizar da maneira correcta)
- Procurar dividir o problema em subproblemas do mesmo tipo
- Verificar se o problema obedece ao princípio de optimalidae
- Verificar se existem subproblemas coincidentes

# PD - Metodologia

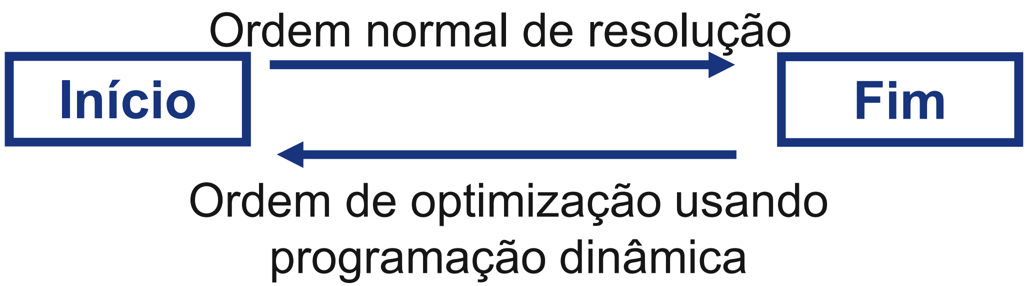
## 2) Definir recursivamente a solução óptima, em função de soluções óptimas de subproblemas.

- Definir recursivamente o valor da solução óptima, com rigor e exactidão, a partir de subproblemas mais pequenos do mesmo tipo
- Imaginar sempre que os valores das soluções óptimas já estão disponíveis quando precisamos deles
- Não é necessário codificar. Basta definir matematicamente a recursão.

# PD - Metodologia

## 3) Calcular as soluções de todos os subproblemas: "de trás para a frente"

- Descobrir a ordem em que os subproblemas são precisos, a partir dos subproblemas mais pequenos até chegar ao problema global (**"bottom-up"**) e codificar, usando uma tabela.
- Normalmente, esta ordem é inversa à ordem normal da função recursiva que resolve o problema



# 3) Calcular as soluções de todos os subproblemas: "memoization"

- Existe uma técnica, chamada "**memoization**", que permite resolver o problema pela ordem normal ("**top-down**")
- Usar a função recursiva obtida directamente a partir definição da solução e ir mantendo uma tabela com os resultados dos subproblemas.
- Quando queremos aceder a um valor pela primeira vez temos de calculá-lo e a partir daí basta ver qual é.

# 4) Reconstruir a solução óptima, baseada nos cálculos efectuados

- Pode ou não ser requisito do problema
- Duas alternativas:
  - **Directamente** a partir da tabela dos sub-problemas
  - **Nova tabela** que guarda as decisões em cada etapa
- Não necessitando de saber qual a melhor solução, podemos por vezes poupar espaço

# Exemplo – Subsequência crescente

- Dada uma sequência de números inteiros:

7, 6, 10, 3, 4, 1, 8, 9, 5, 2

Descobrir qual a **maior subsequência crescente** (não necessariamente contígua)

**7**, 6, **10**, 3, 4, 1, 8, 9, 5, 2 – Tamanho 2
7, **6**, 10, 3, 4, 1, **8**, **9**, 5, 2 – Tamanho 3
7, 6, 10, **3**, **4**, 1, **8**, **9**, 5, 2 – Tamanho 4

# Exemplo – Subsequência crescente

## 1. Caracterizar solução óptima

- Seja **N** o tamanho da sequência, e **num[i]** o i-ésimo número
- Força bruta, quantas opções? *(binomial theorem:* $2^{n-1}$*)*

- Generalizar e resolver com subproblemas iguais:
  - seja **best(i)** o valor da melhor subsequência a partir da *i*-ésima posição
  - **Caso fácil:** a melhor subsequência a começar da última posição tem tamanho 1!
  - **Caso geral**: para um dado i, podemos seguir para todos os números entre i+1 e N, desde que sejam maiores

**7, 6, 10, 3, 4, 1, 8, 9, 5, 2**

# Exemplo – Subsequência crescente

## 1) Caracterizar solução óptima

- **Caso geral**: para um dado i, podemos seguir para todos os números entre i+1 e N, desde que sejam maiores
  - Para esses números, basta-nos saber o melhor a partir daí! **(princípio da optimalidade)**
  - O melhor a partir de uma posição é necessário para calcular todas as posições de índice inferior! (**subproblemas coincidentes**)

**7, 6, 10, 3, 4, 1, 8, 9, 5, 2**

# Exemplo – Subsequência crescente

## 2) Definir solução recursiva em função de soluções óptimas de subproblemas

**N** – tamanho da sequência

**num[i]** – número na posição i

**best(i)** – melhor subsequência a partir da posição i

**best(N)** = 1

**best(i)** = 1 + máximo{best(j): i<j≤N, num[j]>num[i]}
para 1≤i<N

# Exemplo – Subsequência crescente

## 3) Calcular solução óptima: trás para a frente

- Seja best[] tabela para guardar valores de best()

```
Calcular ():
    best[N] = 1
    Para  i: n-1 até 1 fazer
        best[i] = 1
        Para  j: i+1 até N fazer
            Se num[j]>num[i] e 1+best[j]>best[i] então
                best[i] = 1+best[j]
```

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| **num[i]** | **7** | **6** | **10** | **3** | **4** | **1** | **8** | **9** | **5** | **2** |
| **best[i]** | **3** | **3** | **1** | **4** | **3** | **3** | **2** | **1** | **1** | **1** |

# Exemplo – Subsequência crescente

## 4) Reconstruir solução

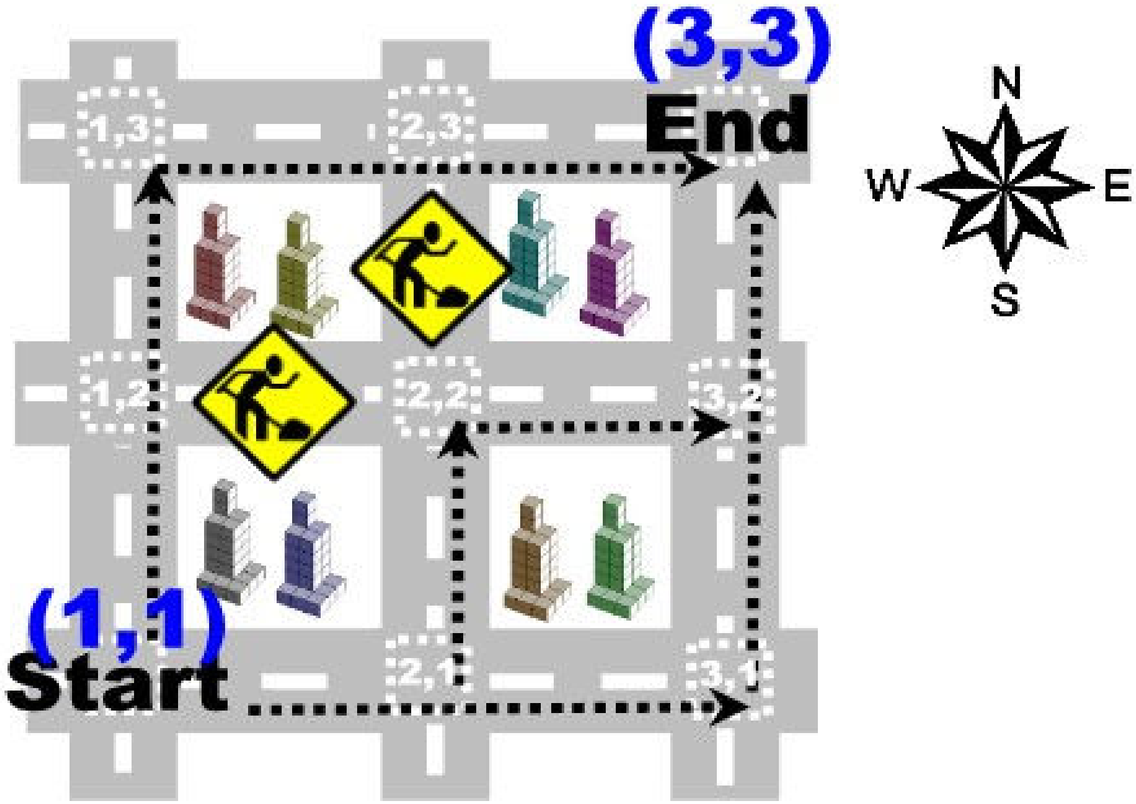
- Exemplo de tabela auxiliar com decições

Seja **next[i]** a próxima posição para obter o melhor a partir da posição i (‘X’ se é a última da sequência)

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| **num[i]** | **7** | **6** | **10** | **3** | **4** | **1** | **8** | **9** | **5** | **2** |
| **best[i]** | **3** | **3** | **1** | **4** | **3** | **3** | **2** | **1** | **1** | **1** |
| **next[i]** | **7** | **7** | **X** | **5** | **7** | **7** | **8** | **X** | **X** | **X** |

# Exemplo – Obras na estrada

- Cidade com "quadriculado" de ruas (MIUP'2004)
  - Algumas estradas têm obras
  - Só se pode andar para norte e para este
  - N máximo: 30

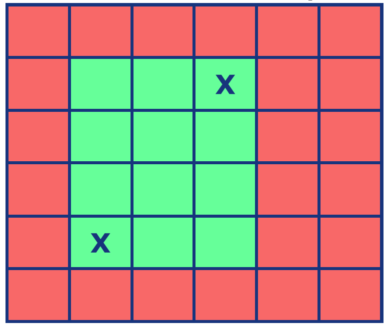



**De quantas maneiras diferentes se pode ir de $(x_1,y_1)$ para $(x_2,y_2)$ ?**

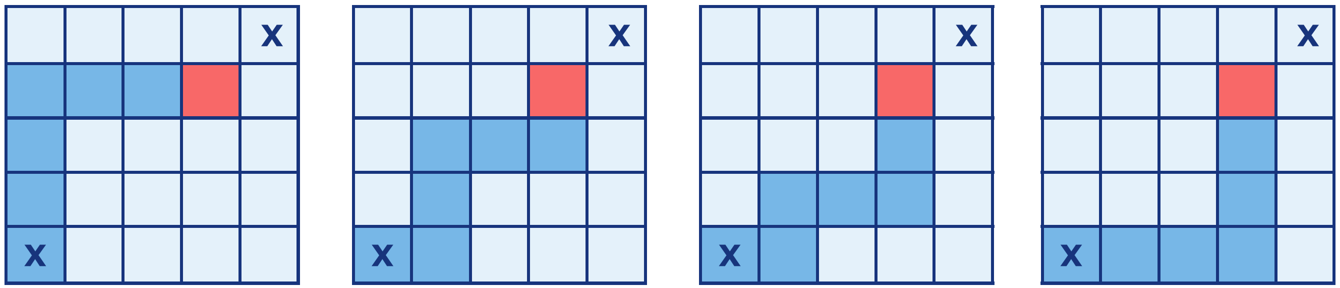
# Exemplo – Obras na estrada

## 1) Caracterizar o problema

- Força bruta? (com N=30, existem ~1,2x10$^{17}$ caminhos)




- Ir de $(x_1,y_1)$ para $(x_2,y_2)$: pode ignorar-se tudo o que está "fora "desse rectângulo
- Número de maneiras a partir de uma posição é igual ao número de maneiras desde a posição a norte mais o número de maneiras desde a posição a este!
  - **Subproblema igual** com solução não dependente do problema "maior" (equivalente a **princípio da optimalidade**)
  - Existem muitos **subproblemas coincidentes**!



# Exemplo – Obras na estrada

## 2) Definir solução recursiva

**L** – número de linhas

**C** – número de colunas

**count(i,j)** – número de maneiras a partir de posição (i,j)

**obra(i,j,D)** – valor V/F indicando se existe obra a impedir deslocação de (i,j) na direcção D (NORTE ou ESTE)

**count(L,C)** = 1

**count(i,j)** = valor_norte(i,j) + valor_este(i,j)

para (i,j)≠(L,C)

onde:

valor_norte(i,j): $\begin{cases} 0 & \text{se (j=L ou obra(i,j,NORTE))} \\ \text{count(i+1,j)} & \text{caso contrário} \end{cases}$

valor_este(i,j): $\begin{cases} 0 & \text{se (j=L ou obra(i,j,ESTE))} \\ \text{count(i,j+1)} & \text{caso contrário} \end{cases}$

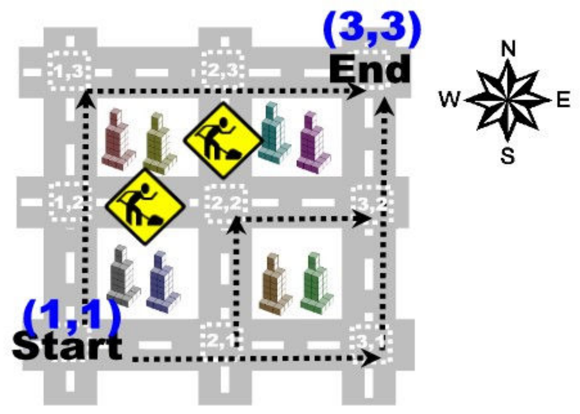
# Exemplo – Obras na estrada

## 3) Calcular solução óptima: trás para a frente

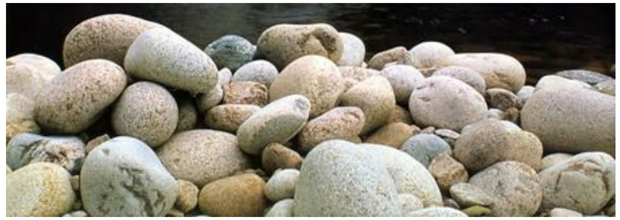
```
Calcular ():
    Inicializar count[][] com zeros
    count[L][C] = 1
    Para  i: L até 1 fazer
        Para  j: C até 1 fazer
            Se i<L e não(obra(i,j,NORTE)) então
                count[i][j] = count[i][j] + count[i+1][j]
            Se j<C e não(obra(i,j,ESTE)) então
                count[i][j] = count[i][j] + count[i][j+1]
```

Count[][]

| | | |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 1 | 1 |
| 3 | 2 | 1 |



# Exemplo – Jogo de Pedras



- Existem N pedras numa mesa
  - Dois jogadores
  - Em cada jogada retira-se 1, 3 ou 8 pedras (generalizável para qualquer número de peças)
  - Quem retirar as últimas pedras ganha o jogo!
  - **Pergunta: dado o número de pedras, o jogador que começa a jogar pode garantidamente ganhar?**

**Exemplo:**

15 pedras na mesa: jogador A retira 8
7 pedras na mesa: jogador B retira 3
4 pedra na mesa: jogador A retira 1
3 pedras na mesa: jogador B retira 3
0 pedras na mesa: ganhou jogador B!

# Exemplo – Jogo de Pedras

## 1) Caracterizar solução óptima

- Solução bruta: avaliar todos jogos possíveis! O(3^N)

- Como generalizar? Seja **win(i)** um valor V/F representando se com *i* pedras conseguimos ganhar **(posição ganhadora)**
  - Claramente win(1), win(3) e win(8) são verdadeiras
  - E para os outros casos?
    - Se a nossa jogada for dar a uma posição ganhadora, então o adversário pode forçar a nossa derrota
    - Então, a nossa posição é ganhadora se conseguirmos chegar a uma posição que não o seja!
    - Caso todas as jogadas forem dar a posições ganhadoras, a nossa posição é perdedora

# Exemplo – Jogo de Pedras

## 2) Definir solução recursiva em função de soluções óptimas de subproblemas

**N** – número de pedras

**win[i]** – valor V/F indicando se estar com i pedras é estar numa posição ganhadora

**win(0)** = falso

$$\textbf{win(i)} = \begin{cases} \text{verdadeiro} & \text{se } \sim win[i-1] \text{ ou } \sim win[i-3] \text{ ou } \sim win[i-8] \\ \text{falso} & \text{caso contrário} \end{cases}$$

para $1 \leq i \leq N$

# Exemplo – Jogo de Pedras

## 3) Calcular solução óptima: trás para a frente

```
Calcular ():
    Para  i: 0 até N fazer
        Se (i≥1 e não(win[i-1])) ou
           (i≥3 e não(win[i-3])) ou
           (i≥8 e não(win[i-8])) então
              win[i] = true
        Senão
           win[i] = false
```

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| win[i] | F | V | F | V | F | V | F | V | V | V | V | F | V |

# Exemplo – Distância de Edição

- Vamos a um último exemplo um pouco mais complexo

- **Problema:** consideremos duas palavras $pal_1$ e $pal_2$. O nosso objectivo é **transformar $pal_1$ em $pal_2$** usando apenas três tipos de transformações:
  **1)** Apagar uma letra
  **2)** Introduzir uma nova letra
  **3)** Substituir uma letra por outra
  Qual o mínimo número de transformações que temos de fazer para transformar uma palavra na outra?
  Chamemos a esta "medida" **distância de edição** (**de**).

Exemplo: para transformar "gotas" em "afoga" precisamos de quatro transformações:

$$\text{gotas} \xrightarrow{(1)} \text{gota\_} \xrightarrow{(3)} \textbf{f}\text{ota} \xrightarrow{(3)} \text{fo}\textbf{g}\text{a} \xrightarrow{(2)} \textbf{a}\text{foga}$$

# Exemplo – Distância de Edição

## 1) Caracterizar solução óptima

- Seja **de(a,b)** a distância de edição entre as palavras a e b.
- Seja **«»** uma palavra vazia

- Existem alguns casos simples?
  - Claramente **de(«», «»)** é zero.
  - **de(«»,b),** para qualquer palavra b? É o tamanho da palavra b (porque temos de fazer **inserções**)
  - **de(a, «»)** para qualquer palavra a? É o tamanho da palavra a pois temos de **apagar** todas as letras de a.

- E nos outros casos? Temos de tentar dividir o problema por etapas, onde decidimos de acordo com subproblemas

# Exemplo – Distância de Edição

- Nenhuma das palavras é vazia
- Como podemos igualar o final das duas palavras?
  - Seja $l_a$ a última letra de **a** e **a'** o resto da palavra (do mesmo modo definimos $l_b$ e **b'**).
  - Se $l_a = l_b$ então só nos falta descobrir a distância de edição entre **a'** e **b'** (que é uma instância mais pequena do mesmo problema!).
  - Caso contrário podemos fazer três operações:
    - <u>Substituir</u> $l_a$ por $l_b$. Gastamos uma operação e precisamos de saber a distância de edição entre **a'** e **b'**.
    - <u>Apagar</u> $l_a$. Gastamos uma operação e precisamos de saber a distância de edição entre **a'** e **b**.
    - <u>Inserir</u> $l_b$ no final de **a**. Gastamos uma operação e precisamos de saber a distância de edição entre **a** e **b'**.

# Exemplo – Distância de Edição

## 2) Definir solução recursiva

**|a| e |b|** – comprimentos das palavras *a* e *b*

**a[i] e b[i]** – letra na posição *i* a palavra *a* e *b*

**de(i,j)** – distância de edição entre as palavras formadas pelas primeiras ***i*** letras de **a** e as primeiras ***j*** letras de ***b***

**de(i,0)** = i, para $0 \leq i \leq |a|$
**de(0,j)** = j, para $0 \leq j \leq |b|$

**de(i,j)** = min(de(i-1,j-1) + {0 se a[i]=b[i], 1 se a[i]≠b[i]},
de(i-1,j)+1,
de(i,j-1)+1),
para $1 \leq i \leq |a|$ e $1 \leq j \leq |b|$

# Exemplo – Distância de Edição

- Calcular soluções de trás para a frente

```
Calcular ():
    Para  i: 0 até |a| fazer  de[i][0] = i
    Para  j: 0 até |b| fazer  de[0][j] = j

    Para  i: 1 até |a| fazer
        Para  j: 1 até |b| fazer
            Se a[i]=b[j] então valor = 0
            Senão valor = 1
            de[i][j] = min( de[i-1][j-1]+value,
                            de[i-1][j],
                            de[i-1][j] )
```

# Exemplo – Distância de Edição

- Vejamos a tabela a distância de edição entre “gotas” e”afoga”:

| i \ j | | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|---|
| | | «» | A | F | O | G | A |
| 0 | «» | 0 | 1 | 2 | 3 | 4 | 5 |
| 1 | G | 1 | 1 | 2 | 3 | 3 | 4 |
| 2 | O | 2 | 2 | 2 | 2 | 3 | 4 |
| 3 | T | 3 | 3 | 3 | 3 | 3 | 4 |
| 4 | A | 4 | 3 | 4 | 4 | 4 | 3 |
| 5 | S | 5 | 4 | 4 | 5 | 5 | 4 |

**de(i,0)** = i, para $0 \leq i \leq |a|$
**de(0,j)** = j, para $0 \leq j \leq |b|$

**de(i,j)** = min(de(i-1,j-1) + {0 se a[i]=b[i], 1 se a[i]≠b[i]},
de(i-1,j)+1,
de(i,j-1)+1),
para $1 \leq i \leq |a|$ e $0 \leq j \leq |b|$

# Exemplo – Distância de Edição

- Se fosse preciso reconstruir a solução?

| | j | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|---|
| i | | «» | A | F | O | G | A |
| 0 | «» | 0 | 1 | 2 | 3 | 4 | 5 |
| 1 | G | 1 | 1 | 2 | 3 | 3 | 4 |
| 2 | O | 2 | 2 | 2 | 2 | 3 | 4 |
| 3 | T | 3 | 3 | 3 | 3 | 3 | 4 |
| 4 | A | 4 | 3 | 4 | 4 | 4 | 3 |
| 5 | S | 5 | 4 | 4 | 5 | 5 | 4 |

← Inserir letra

↑ Apagar letra

↖ Substituir letra

⇠ Manter letra

gotas → gota_ → goga → foga → afoga

goga → gfoga → afoga

# Conclusão

- Nem sempre a PD representa a **melhor solução** para um problema, mas no entanto apresenta normalmente ganhos muito significativos sobre algoritmos exponenciais de força-bruta;

- A PD é uma **técnica** activamente usada na vida real, tanto no meio empresarial, como no meio académico;

- A **ideia base** da PD é muito simples, mas, como foi visto neste último problema, nem sempre é fácil chegar à sua solução. Quanto a mim, a parte mais difícil é generalizar o problema da maneira correcta, de modo a podermos escrever a solução em função de soluções óptimas de subproblemas;

- Só existe uma maneira de dominar a PD: **treinar, treinar, treinar**!

# Fim

**http://www.dcc.fc.up.pt/~pribeiro/**

**pribeiro@dcc.fc.up.pt**