

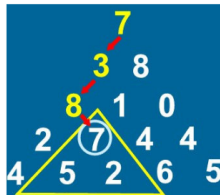
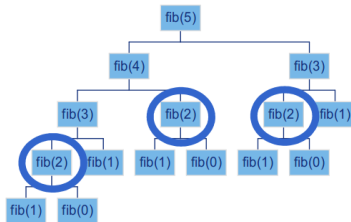
Programação Dinâmica

Pedro Ribeiro

DCC/FCUP

2016/2017

i \ j	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	1	2	3	3	4
2	2	2	2	2	3	4
3	3	3	3	3	3	4
4	4	3	4	4	4	3
5	5	4	4	5	5	4



Números de Fibonacci

Sequência de números muito famosa definida por Leonardo Fibonacci



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

Números de Fibonacci

$$F(0) = 0$$

$$F(1) = 1$$

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

Números de Fibonacci

- Como implementar?
- Implementação directa a partir da definição:

Fibonacci (a partir da definição)

fib(n):

Se $n = 0$ ou $n = 1$ então

retorna n

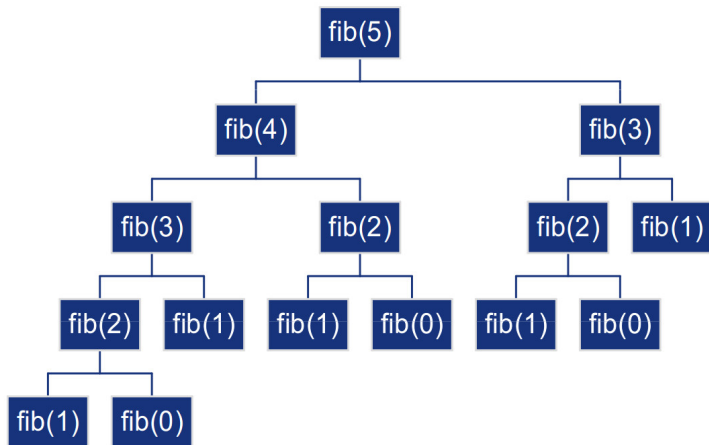
Senão

retorna $\text{fib}(n - 1) + \text{fib}(n - 2)$

- Pontos **negativos** desta implementação?

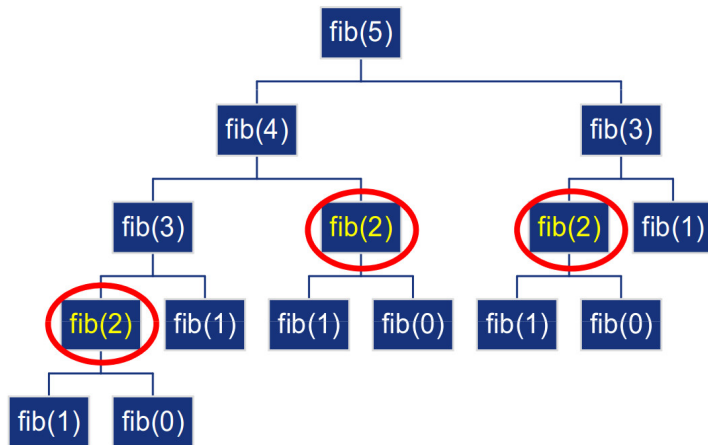
Números de Fibonacci

- Cálculo de **fib(5)**:



Números de Fibonacci

- Cálculo de **fib(5)**:



Por exemplo $\text{fib}(2)$ é chamado 3 vezes!

Números de Fibonacci

- Como **melhorar**?
 - ▶ Começar do zero e manter sempre em memória os dois últimos números da sequência

Fibonacci (versão iterativa mais eficiente)

fib(n):

Se $n = 0$ ou $n = 1$ então

retorna n

Senão

$f_1 \leftarrow 1$

$f_2 \leftarrow 0$

Para $i \leftarrow 2$ até n fazer

$f \leftarrow f_1 + f_2$

$f_2 \leftarrow f_1$

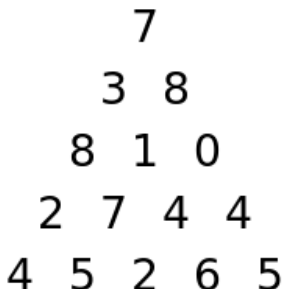
$f_1 \leftarrow f$

retorna f

- **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 em problemas mais "complicados"?**

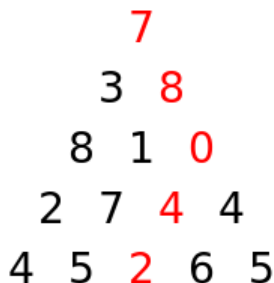
Pirâmide de Números

- Problema "clássico" das **Olimpíadas Internacionais de Informática de 1994**
- Calcular o **caminho**, 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.

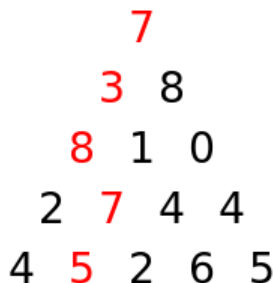


Pirâmide de Números

- Dois possíveis caminhos:



Soma = 21



Soma = 30

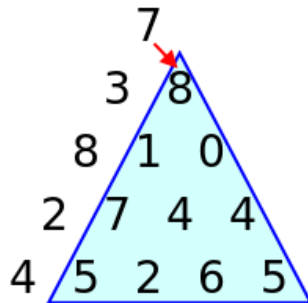
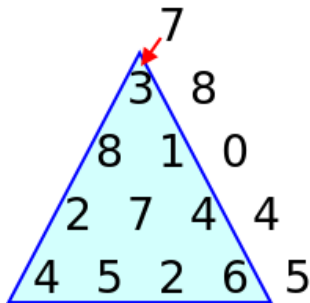
- Restrições:** todos os números são inteiros entre 0 e 99 e o número de linhas do triângulo é no máximo 100.

Pirâmide de Números

- Como resolver o problema?
- Ideia: **Algoritmo greedy**
 - ▶ Escolher sempre o maior número dos dois "descendentes"
Não funciona! (ex: na pirâmide dada daria 28 e o ótimo é 30)
- Ideia: **Pesquisa Exaustiva** (aka "Força Bruta")
 - ▶ Avaliar todos os caminhos possíveis e ver qual o melhor.
- Quanto tempo demoraria? Quantos caminhos existem?
- Análise da **complexidade temporal**:
 - ▶ Em cada linha podemos tomar **duas decisões**: esquerda ou direita
 - ▶ Seja n a altura da pirâmide. Um caminho são... $n - 1$ decisões!
 - ▶ Existem então 2^{n-1} caminhos diferentes
 - ▶ Um programa para calcular todos os caminhos tem portanto complexidade **$O(2^n)$** : exponencial!
 - ▶ $2^{99} \sim 6.34 \times 10^{29}$ (**633825300114114700748351602688**)

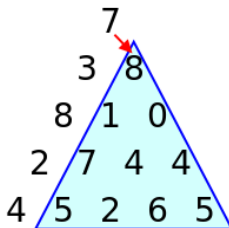
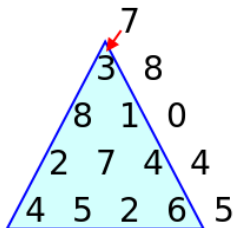
Pirâmide de Números

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



- Em cada um dos casos temos de ter em conta **todos os caminhos das respectivas subpirâmides**.

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 do melhor caminho 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 $\text{Max}(i, j)$ 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

Pirâmide de Números (a partir da definição recursiva)

Max(i, j):

Se $i = n$ então

retorna $P[i][j]$

Senão

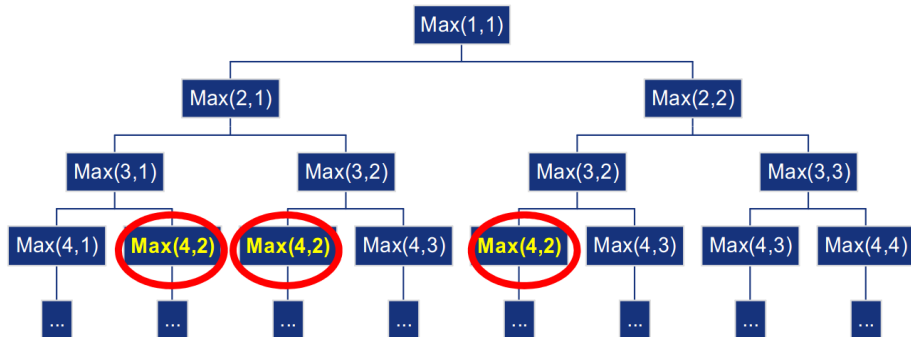
retorna $P[i][j] + \text{máximo} (\text{Max}(i + 1, j), \text{Max}(i + 1, j + 1))$

- Para resolver o problema basta chamar... **Max(1,1)**

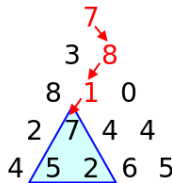
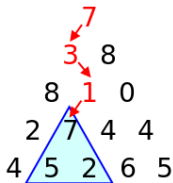
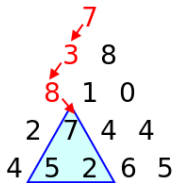
	1	2	3	4	5
1	7				
2	3	8			
3	8	1	0		
4	2	7	4	4	
5	4	5	2	6	5

Pirâmide de Números

- Continuamos com **crescimento exponencial**!



- Estamos a avaliar o mesmo subproblema várias vezes...

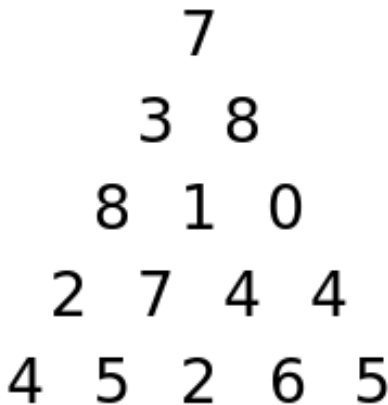


Pirâmide de Números

- 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?

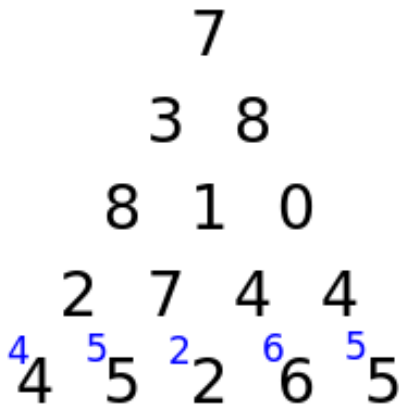
Pirâmide de Números

- Começar a partir do fim! (base da pirâmide)



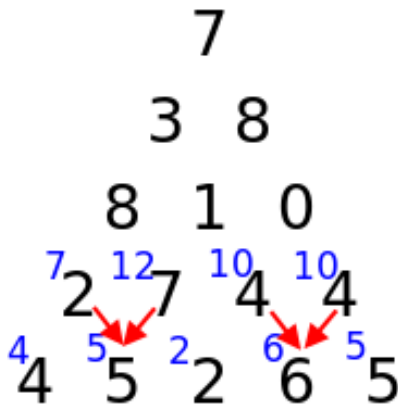
Pirâmide de Números

- Começar a partir do fim! (base da pirâmide)



Pirâmide de Números

- Começar a partir do fim! (base da pirâmide)



Pirâmide de Números

- Começar a partir do fim! (base da pirâmide)



Pirâmide de Números

- Tendo em conta a maneira como preenchemos a tabela, até podemos aproveitar $P[i][j]$:

Pirâmide de Números (solução polinomial - $\mathcal{O}(n^2)$)

Calcular():

Para $i \leftarrow n - 1$ até 1 fazer

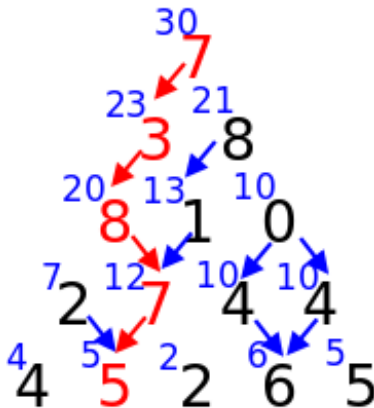
Para $j \leftarrow 1$ até i fazer

$P[i][j] \leftarrow P[i][j] + \text{máximo} (P[i + 1][j], P[i + 1][j + 1])$

- Com isto a solução fica em... $P[1][1]$
- Agora o tempo necessário para resolver o problema já só cresce polinomialmente ($\mathbf{O(n^2)}$) e já temos uma solução admissível para o problema ($99^2 = 9801$)

Pirâmide de Números

- E se fosse necessário saber a constituição do melhor caminho?
Basta usar a tabela já calculada!



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

Programação Dinâmica (PD)

Programação Dinâmica

Programação Dinâmica

Uma **técnica algorítmica**, normalmente usada em **problemas de otimização**, 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 Otimização:** encontrar a "melhor" solução entre todas as soluções possíveis, segundo um determinado critério (função objectivo). Geralmente descobrir um máximo ou mínimo.

Clássica troca de **espaço** por **tempo**

Quais são então as **características** que um problema deve apresentar para poder ser resolvido com PD?

- Subestrutura ótima
- Subproblemas coincidentes

Programação Dinâmica - Características

Subestrutura Ótima

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

Exemplo

No problema da pirâmides de números, a solução ótima contém nela própria os melhores percursos de subpirâmides, ou seja, soluções ótimas de subproblemas

- Se um problema apresenta esta característica, diz-se que respeita o **princípio da optimalidade**.

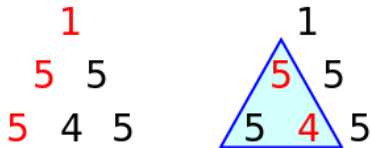
Programação Dinâmica - Características

Cuidado!

Nem sempre um problema tem subestrutura ótima!

Exemplo sem subestrutura ótima

Imagine que no problema da pirâmide de números o objectivo é encontrar o caminho que maximize o resto da divisão inteira entre 10 e a soma dos valores desse caminho



A solução ótima ($1 \rightarrow 5 \rightarrow 5$) não contém a solução ótima da subpirâmide assinalada ($5 \rightarrow 4$)

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 $n + (n - 1) + \dots + 1 < n^2$ subproblemas (crescem polinomialmente) pois, como já vimos, muitos subproblemas que aparecem são coincidentes

Programação Dinâmica - Características

Cuidado!

Também esta característica nem sempre acontece.

- Mesmo com subproblemas coincidentes são muitos subproblemas a resolver
ou
- Não existem subproblemas coincidentes.

Exemplo

No MergeSort, cada chamada recursiva é feita a um subproblema novo, diferente de todos os outros.

Programação Dinâmica - Metodologia

- Se um problema apresenta estas **duas características**, temos uma boa pista de que a PD se pode aplicar.
- Que **passos** seguir então para resolver um problema com PD?

Guia para resolver com PD

- 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 - apenas se for necessário)

1) Caracterizar a solução óptima do problema

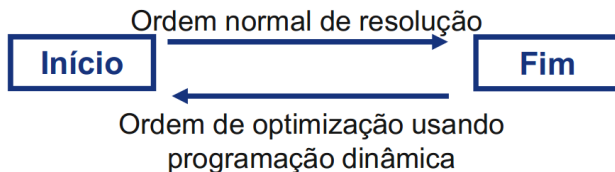
- **Compreender** bem o problema
- Verificar se um algoritmo que verifique todas as soluções à "**força bruta**" não é suficiente
- Tentar **generalizar o problema** (é preciso prática para perceber como generalizar da maneira correcta)
- Procurar **dividir o problema** em subproblemas do mesmo tipo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

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.

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, conhecida como "**memoization**", que permite resolver o problema pela ordem normal ("**top-down**").
- Usar a **função recursiva** obtida directamente a partir da 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 o resultado já calculado.

4) Reconstruir a solução ótima, 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

Subsequência Crescente

- Dada uma sequência de números:

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)

Subsequência Crescente

1) Caracterizar a solução óptima do problema

- Seja n o tamanho da sequência e $\text{num}[i]$ o i -ésimo número
- "Força bruta", quantas opções? **Exponencial!** (*binomial theorem*)
- Generalizar e resolver com subproblemas iguais:
 - ▶ Seja **best(i)** o valor da melhor subsequência a partir da i -ésima posição
 - ▶ **Caso base:** 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
 - ★ Para esses números, basta-nos saber o melhor a partir daí!
(**princípio da otimalidade**)
 - ★ O melhor a partir de uma posição é necessário para calcular todas as posições de índice inferior! (**subproblemas coincidentes**)

Subsequência Crescente

2) Definir recursivamente a solução ótima, em função de soluções ótimas de subproblemas

- **n** - tamanho da sequência
- **num[i]** - número na posição i
- **best(i)** - melhor subsequência a partir da posição i

Solução recursiva para Subsequência Crescente

$$\text{best}(n) = 1$$

$$\text{best}(i) = 1 + \text{máximo}\{\text{best}(j): i < j \leq n, \text{num}[j] > \text{num}[i]\}$$

para $1 \leq i < n$

Subsequência Crescente

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

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

Subsequência crescente (solução polinomial - $\mathcal{O}(n^2)$)

Calcular():

$best[n] \leftarrow 1$

Para $i \leftarrow n - 1$ até 1 fazer

$best[i] \leftarrow 1$

Para $j \leftarrow i + 1$ até n fazer

Se $num[j] > num[i]$ e $1 + best[j] > best[i]$ então

$best[i] \leftarrow 1 + best[j]$

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

Subsequência Crescente

4) Reconstruir a solução ótima

- Vamos exemplificar com uma tabela auxiliar que guarda as decisões
- Seja **next[i]** uma próxima posição para obter o melhor a partir da posição i ('-1' se é a última posição).

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	-1	5	7	7	8	-1	-1	-1

Problema do Troco

Vamos visitar um "velho conhecido" ...



O problema do troco (problema do cashier)

Input: Um conjunto de valores de moedas S e uma quantia K a criar com as moedas

Output: O menor número de moedas que fazem a quantia K (podemos repetir moedas)

- Já já vimos na semana anterior que um **algoritmo greedy não funciona** para todos os conjuntos de moedas
- **Pesquisa exaustiva** sobre todos os conjuntos de moedas também **não é exequível** em tempo útil (número exponencial de hipóteses)

Problema do Troco

1) Caracterizar a solução óptima do problema

- Se utilizar uma moeda de quantia Q então passo a querer saber o menor número de moedas para fazer $K - Q$ (**princípio da optimalidade**)
- Intuitivamente, vou precisar de saber muitas vezes o mínimo para uma dada quantia (**subproblemas coincidentes**)
ex: usando $3+5$ vou parar ao mesmo "sítio" que usando $1+1+6$ ou $4+4$)
- Seja $coins(i)$ o menor número de moedas para fazer a quantia i
Seja S_i a i -ésima moeda e N o número de moedas
 - ▶ $coins(i) = 1$ se existe moeda de quantia i
 - ▶ Para todas as outras quantias i basta-me ver as moedas S_i tal que $S_i < i$ e verificar $1 + coins(i - S_i)$
(usar a moeda mais o mínimo do restante)

Problema do Troco

2) Definir recursivamente a solução óptima

- K - quantia que queremos criar com as moedas
- S_i - i -ésima moeda
- N - número de moedas
- $\text{coins}[i]$ - menor número de moedas para fazer quantia i

Solução recursiva para problema do troco

$$\text{coins}(0) = 0$$

$$\text{coins}(i) = 1 + \text{mínimo}\{\text{coins}(i - S_j) : 1 \leq j \leq N, S_j < i\}$$

para $1 \leq i \leq K$

Problema do Troco

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

Problema do Troco (solução polinomial - $\mathcal{O}(N \times K)$)

Calcular():

$coins[0] \leftarrow 0$

Para $i \leftarrow 1$ até K fazer

$coins[i] \leftarrow \infty$

Para $j \leftarrow 1$ até N fazer

Se $(S[j] \leq i$ e $1 + coins[i - S[j]] < coins[i]$ então

$coins[i] \leftarrow 1 + coins[i - S[j]]$

Exemplo para $S = \{1, 5, 8, 11\}$, $K = 13$

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13
coins[i]	0	1	2	3	4	1	2	3	1	2	2	1	2	2

Problema do Troco

4) Reconstruir a solução ótima

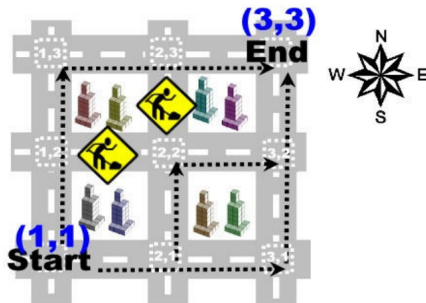
- Vamos exemplificar com uma tabela auxiliar que guarda as decisões
- Seja **use[i]** a moeda a 1ª moeda a usar para obter o mínimo número de moedas para fazer a quantia i
- No final bastava ir "percorrendo" o array de **use[i]**
 - ▶ $K = 13$, a primeira moeda é a de 5 ($use[13] = 5$)
 - ▶ $K = 13 - 5 = 8$, a moeda seguinte é a de 8 ($use[8] = 8$)
 - ▶ $K = 8 - 8 = 0$, mais nenhuma moeda é necessária

Exemplo para $S = \{1, 5, 8, 11\}$, $K = 13$

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13
coins[i]	0	1	2	3	4	1	2	3	1	2	2	1	2	2
use[i]	-	1	1	1	1	5	1	1	8	1	5	11	1	5

Obras na Estrada - PD com contagens

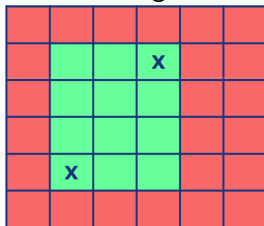
- Este problema saiu na **MIUP'2004**
(Maratona Inter-Universitária de Programação)
- Imagine um "quadriculado" de ruas em que:
 - ▶ Algumas estrada têm obras
 - ▶ Só se pode andar para norte e para este
 - ▶ Máximo tamanho do quadriculado: 30×30



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

1) Caracterizar a solução óptima do problema

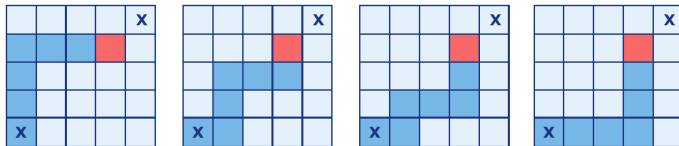
- "Força bruta", quantas opções?
 $N = 30$, existem $\sim 3 \times 10^{16}$ caminhos
- Para ir (x_1, y_1) para (x_2, y_2) pode ignorar-se tudo o que está "fora" desse rectângulo.



Obras na Estrada - PD com contagens

1) Caracterizar a solução óptima do problema

- **Número de maneiras** a partir de uma posição é igual a:
número de maneiras desde a posição a norte
+
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**!



Obras na Estrada - PD com contagens

2) Definir recursivamente a solução óptima

- **L** - número de linhas
- **C** - número de colunas
- **count(i,j)** - número de maneiras a partir da posição (i,j)
- **obra(i,j,D)** - valor booleano (V/F) indicando se existe obra a impedir deslocação de (i,j) na direcção D (NORTE ou ESTE)

Solução recursiva para Obras na Estrada

$$\text{count}(L, C) = 1$$

$$\text{count}(i, j) = \text{valor_norte}(i, j) + \text{valor_este}(i, j)$$

para $(i, j) \neq (L, C)$ onde:

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

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

Obras na Estrada - PD com contagens

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

Obras na Estrada (solução polinomial - $\mathcal{O}(L \times C)$)

Calcular():

Inicializar $count[][]$ com zeros

$count[L][C] \leftarrow 1$

Para $i \leftarrow L$ até 1 fazer

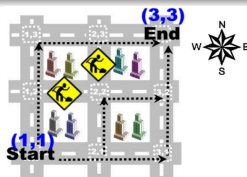
Para $j \leftarrow C$ até 1 fazer

Se $i < L$ e não($obra(i, j, NORTE)$) então
 $count[i][j] \leftarrow count[i][j] + count[i + 1][j]$

Se $j < C$ e não($obra(i, j, ESTE)$) então
 $count[i][j] \leftarrow count[i][j] + count[i][j + 1]$

count[][]

1	1	1
1	1	1
3	2	1



Jogo de Pedras



- Existem n pedras numa mesa
- Em cada jogada retira-se **1, 3 ou 8** pedras
(*generalizável para qualquer número de peças*)
- **Quem retirar as últimas pedras ganha o jogo!**

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 pedras na mesa: jogador A retira 1
3 pedras na mesa: jogador B retira 3
0 pedras na mesa: ganhou jogador B!

1) Caracterizar a solução óptima do problema

- "Força bruta", quantos jogos possíveis? $O(3^N)$!
- Como generalizar? Seja **win(i)** um valor booleano (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

2) Definir recursivamente a solução óptima

- **N** - número de pedras
- **win[i]** - valor booleano (V/F) indicando se posição i é ganhadora

Solução recursiva para Jogo de Pedras

$\text{win}(0) = \text{falso}$

$\text{win}(i) =$

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

para $1 \leq i \leq N$

Jogo de Pedras

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

Jogo de Pedras (solução polinomial)

Calcular():

Para $i \leftarrow 0$ até N fazer

Se $(i \geq 1 \text{ e } \text{win}[i - 1] = \text{falso})$ ou

$(i \geq 3 \text{ e } \text{win}[i - 3] = \text{falso})$ ou

$(i \geq 8 \text{ e } \text{win}[i - 8] = \text{falso})$ então

$\text{win}[i] \leftarrow \text{verdadeiro}$

Senão

$\text{win}[i] \leftarrow \text{falso}$

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13
win[i]	F	V	F	V	F	V	F	V	V	V	V	F	V	F

Distância de Edição

- Vamos agora ver um exemplo mais "complexo"

Problema - Distância de Edição

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? Chamamos a esta "medida" **distância de edição (de)**.

Exemplo

Para transformar "gotas" em "afoga" precisamos de 4 transformações:

(1) (3) (3) (2)
gotas \rightarrow gota_ \rightarrow fota \rightarrow foga \rightarrow afoga

1) Caracterizar a solução óptima do problema

- 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 (temos de fazer **inserções**)
 - ▶ $de(a, "")$, para qualquer palavra a ? É o tamanho da palavra a (temos de fazer **remoções**)
- E nos outros casos? Temos de tentar dividir o problema por etapas, onde decidimos de acordo com subproblemas.

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 a
 - ▶ Seja l_b a última letra de b e b' o resto da palavra b
- Se $l_a = l_b$, então só nos falta descobrir a distância de edição entre a' e b' (instância mais pequena do mesmo problema!)
- Caso contrário, temos três operações possíveis:
 - ▶ **Substituir** l_a por l_b . Gastamos 1 transformação e precisamos de saber a distância de edição entre a' e b' .
 - ▶ **Apagar** l_a . Gastamos 1 transformação e precisamos de saber a distância de edição entre a' e b .
 - ▶ **Inserir** l_b no final de a . Gastamos 1 transformação e precisamos de saber a distância de edição entre a e b' .

Distância de Edição

2) Definir recursivamente a solução óptima

- $|a|$ e $|b|$ - tamanho (comprimento) das palavras a e b
- $a[i]$ e $b[i]$ - letra na posição i das palavras 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

Solução recursiva para Distância de Edição

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

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

$de(i, j) = \text{mínimo}(de(i-1, j-1) + \{0 \text{ se } a[i] = b[j], 1 \text{ se } a[i] \neq b[j]\},$
 $de(i-1, j) + 1,$
 $de(i, j-1) + 1)$

para $1 \leq i \leq |a|$ e $1 \leq j \leq |b|$

Distância de Edição

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

Distância de Edição (solução polinomial)

Calcular():

Para $i \leftarrow 0$ até $|a|$ fazer $de[i][0] \leftarrow i$

Para $j \leftarrow 0$ até $|b|$ fazer $de[0][j] \leftarrow j$

Para $i \leftarrow 1$ até $|a|$ fazer

Para $j \leftarrow 1$ até $|j|$ fazer

Se $(a[i] = b[j])$ então $valor \leftarrow 0$

Senão $valor \leftarrow 1$

$de[i][j] = \text{mínimo}(de[i-1][j-1] + valor, \\ de[i-1][j] + 1, \\ de[i][j-1] + 1)$

Distância de Edição

- Vejam os a **tabela** para a distância de edição entre "gotas" e "afoga":

		j	0	1	2	3	4	5
i			<<>>	A	F	O	G	A
		<<>>	0	1	2	3	4	5
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	4	3
5	S	5	4	4	5	5	5	4

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

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

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

