Estruturas de Dados e Algoritmos II

Vasco Pedro

Departamento de Informática
Universidade de Évora

2021/2022

Pseudo-código

Exemplo

PESQUISA-LINEAR(V, k)

```
1 n <- |V|                                  // inicialização
2 i <- 1
3 while i <= n and V[i] != k do             // pesquisa
4     i <- i + 1
5 if i <= n then                            // resultado:
6     return i                              //  - sucesso
7 return INEXISTENTE                        //  - insucesso
```

`|V|`	nº de elementos de um vector — $O(1)$
`V[1..|V|]`	elementos do vector
`and` e `or`	só é avaliado o segundo operando se necessário
`variável.campo`	acesso a um campo de um "objecto"

(`INEXISTENTE` é uma constante, com valor -1, por exemplo)

Análise da complexidade (1)

Exemplo

Análise da complexidade temporal, no pior caso, da função PESQUISA-LINEAR, por linha de código

1. Obtenção da dimensão de um vector, afectação: operações com complexidade (temporal) constante

$$O(1) + O(1) = O(1)$$

2. Afectação: $O(1)$
3. Acessos a `i`, `n`, `V[i]` e `k`, comparações e saltos condicionais com complexidade constante

$$4\,O(1) + 2\,O(1) + 2\,O(1) = O(1)$$

Executada, no pior caso, $|\mathtt{V}|+1$ vezes

$$(|\mathtt{V}| + 1) \times O(1) = O(|\mathtt{V}|)$$

Análise da complexidade (2)

Exemplo

4. Acesso a `i`, soma e afectação: $O(1) + O(1) + O(1) = O(1)$
   Executada, no pior caso, `|V|` vezes

$$|\texttt{V}| \times O(1) = O(|\texttt{V}|)$$

5. Acesso a `i` e `n`, comparação e salto condicional com complexidade constante

$$2\,O(1) + O(1) + O(1) = O(1)$$

6. Saída de função com complexidade constante: $O(1)$
7. Saída de função com complexidade constante: $O(1)$

Análise da complexidade (3)

Exemplo

Juntando tudo

$$O(1) + O(1) + O(|\mathtt{V}|) + O(|\mathtt{V}|) + O(1) + \max\{O(1), O(1)\} =$$
$$= 4\,O(1) + 2\,O(|\mathtt{V}|) =$$
$$= O(|\mathtt{V}|)$$

No pior caso, a função PESQUISA-LINEAR tem complexidade temporal linear na dimensão do vector `V`

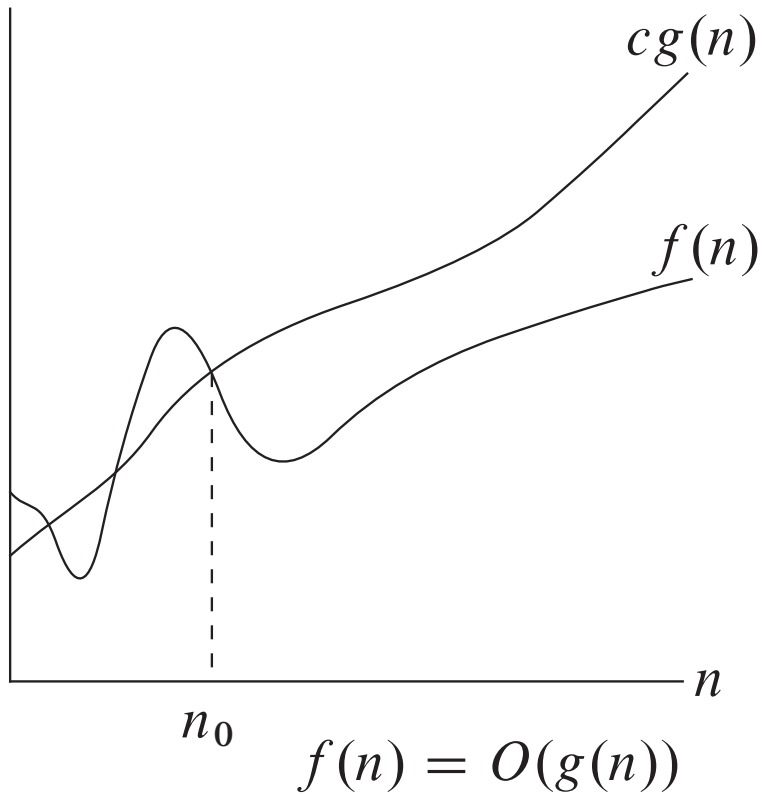
Se n representar a dimensão do vector `V`, o tempo $T(n)$ que a função demora a executar tem complexidade linear em n

$$T(n) = O(n)$$

Isto significa que o tempo que a função demora a executar varia linearmente com a dimensão do *input*

A notação O (1)

$$O(g(n)) = \{f(n) : \exists_{c,n_0>0} \text{ tais que } \forall_{n \geq n_0} \; 0 \leq f(n) \leq c\, g(n)\}$$



A notação O (2)

$$O(g(n)) = \{f(n) : \exists_{c,n_0>0} \text{ tais que } \forall_{n \geq n_0} \; 0 \leq f(n) \leq c\, g(n)\}$$

- $O(n) = \{f(n) : \exists_{c,n_0>0} \text{ tais que } \forall_{n \geq n_0} \; 0 \leq f(n) \leq c\, n\}$

  $n = O(n)$ $\quad$ $2n + 5 = O(n)$ $\quad$ $\log n = O(n)$ $\quad$ $n^2 \neq O(n)$

- $O(n^2) = \{f(n) : \exists_{c,n_0>0} \text{ tais que } \forall_{n \geq n_0} \; 0 \leq f(n) \leq c\, n^2\}$

  $n^2 = O(n^2)$ $\quad$ $4n^2 + n = O(n^2)$ $\quad$ $n = O(n^2)$ $\quad$ $n^3 \neq O(n^2)$

- $O(\log n) = \{f(n) : \exists_{c,n_0>0} \text{ tais que } \forall_{n \geq n_0} \; 0 \leq f(n) \leq c \log n\}$

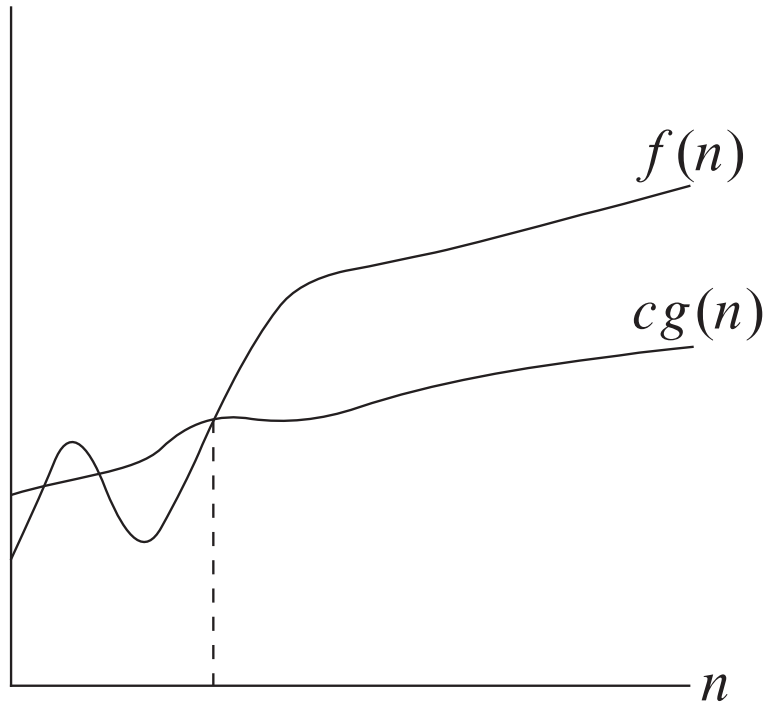
  $1 + \log n = O(\log n)$ $\quad$ $\log n^2 = O(\log n)$ $\quad$ $n \neq O(\log n)$

$f(n) = O(g(n))$ significa $f(n) \in O(g(n))$

Lê-se $f(n)$ é O de $g(n)$

A notação O (3)

$$\Omega(g(n)) = \{f(n) : \exists_{c,n_0>0} \text{ tais que } \forall_{n \geq n_0} \; 0 \leq c\, g(n) \leq f(n)\}$$

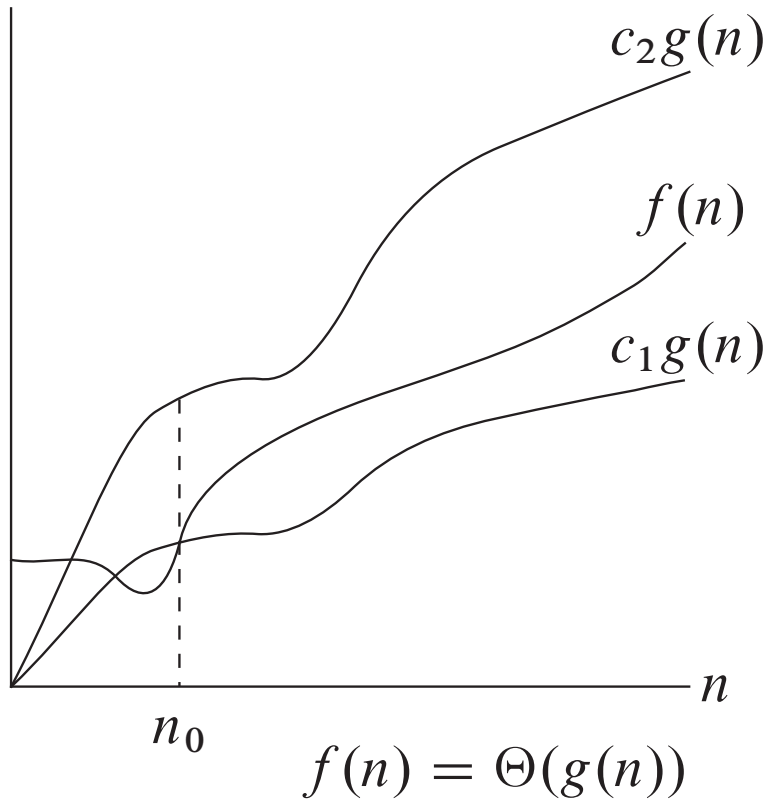


$f(n) = \Omega(g(n))$

$$n = \Omega(n) \qquad n^2 = \Omega(n) \qquad \log n \neq \Omega(n^2)$$

A notação O (4)

$$\Theta(g(n)) = \{f(n) : \exists_{c_1,c_2,n_0>0} \text{ t.q. } \forall_{n \geq n_0} \; 0 \leq c_1\, g(n) \leq f(n) \leq c_2\, g(n)\}$$



$f(n) = \Theta(g(n))$

$3n^2 + n = \Theta(n^2)$ $\quad$ $n \neq \Theta(n^2)$ $\quad$ $n^2 \neq \Theta(n)$

A notação O (5)

$$o(g(n)) = \{f(n) : \forall_{c>0}\ \exists_{n_0>0} \text{ tal que } \forall_{n \geq n_0}\ 0 \leq f(n) < c\, g(n)\}$$

$$n = o(n^2) \qquad n^2 \neq o(n^2) \qquad \log n = o(n)$$

$$\forall_{k>0}\ n^k = o(2^n)$$

$$\forall_{k>0}\ \forall_{b>1}\ n^k = o(b^n)$$

$$\omega(g(n)) = \{f(n) : \forall_{c>0}\ \exists_{n_0>0} \text{ tal que } \forall_{n \geq n_0}\ 0 \leq c\, g(n) < f(n)\}$$

$$n = \omega(\log n) \qquad n^2 = \omega(\log n) \qquad \log n \neq \omega(\log n)$$

$$\forall_{k>0}\ 2^n = \omega(n^k)$$

$$\forall_{b>1}\ \forall_{k>0}\ b^n = \omega(n^k)$$

A notação O (6)

Traduzindo...

$f(n) = O(g(n))$	$f(n)$ não cresce mais depressa que $g(n)$
$f(n) = o(g(n))$	$f(n)$ cresce mais devagar que $g(n)$
$f(n) = \Omega(g(n))$	$f(n)$ não cresce mais devagar que $g(n)$
$f(n) = \omega(g(n))$	$f(n)$ cresce mais depressa que $g(n)$
$f(n) = \Theta(g(n))$	$f(n)$ e $g(n)$ crescem ao mesmo ritmo

Ainda a pesquisa linear

De um valor num vector ordenado

PESQUISA-LINEAR-ORD(V, k)

```
1 n <- |V|                                  // inicialização
2 i <- 1
3 while i <= n and V[i] < k do              // pesquisa
4     i <- i + 1
5 if i <= n and V[i] = k then               // resultado:
6     return i                              //  - sucesso
7 return INEXISTENTE                        //  - insucesso
```

Pesquisa dicotómica ou binária

De um valor num vector ordenado

PESQUISA-DICOTÓMICA(V, k)

```
1 n <- |V|
2 return PESQUISA-DICOTÓMICA-REC(V, k, 1, n)
```

PESQUISA-DICOTÓMICA-REC(V, k, i, f)

```
1 if i > f then
2     return INEXISTENTE          // intervalo vazio
3 m <- (i + f) / 2
4 if k < V[m] then
5     return PESQUISA-DICOTÓMICA-REC(V, k, i, m - 1)
6 if k > V[m] then
7     return PESQUISA-DICOTÓMICA-REC(V, k, m + 1, f)
8 return m                        // V[m] = k
```

Complexidade das pesquisas linear e dicotómica

Pior caso e caso esperado para a complexidade temporal das pesquisas num vector de dimensão n

Pesquisa linear

$$T(n) = O(n)$$

Pesquisa linear num vector ordenado

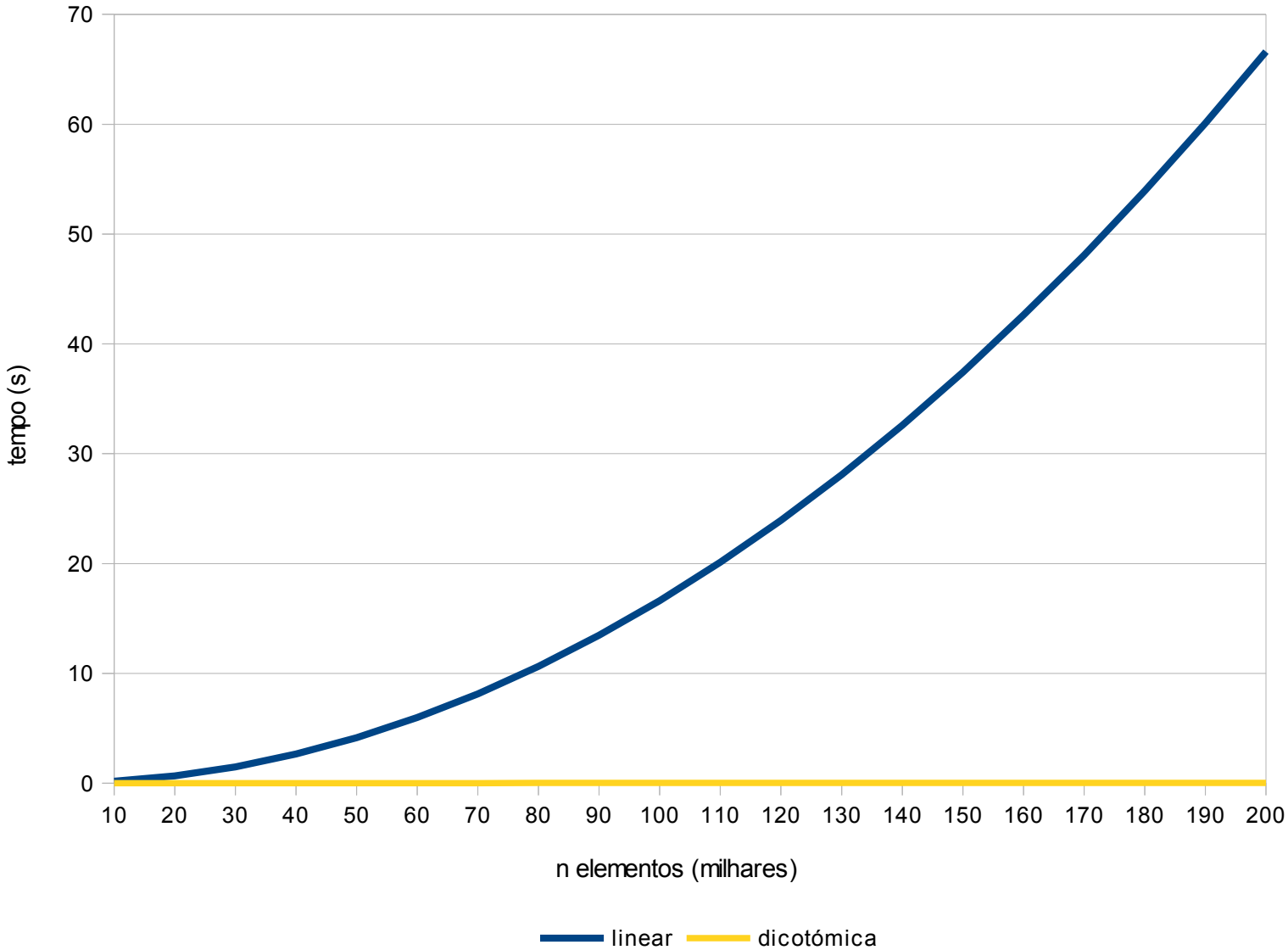
$$T(n) = O(n)$$

Pesquisa dicotómica

$$T(n) = O(\log n)$$

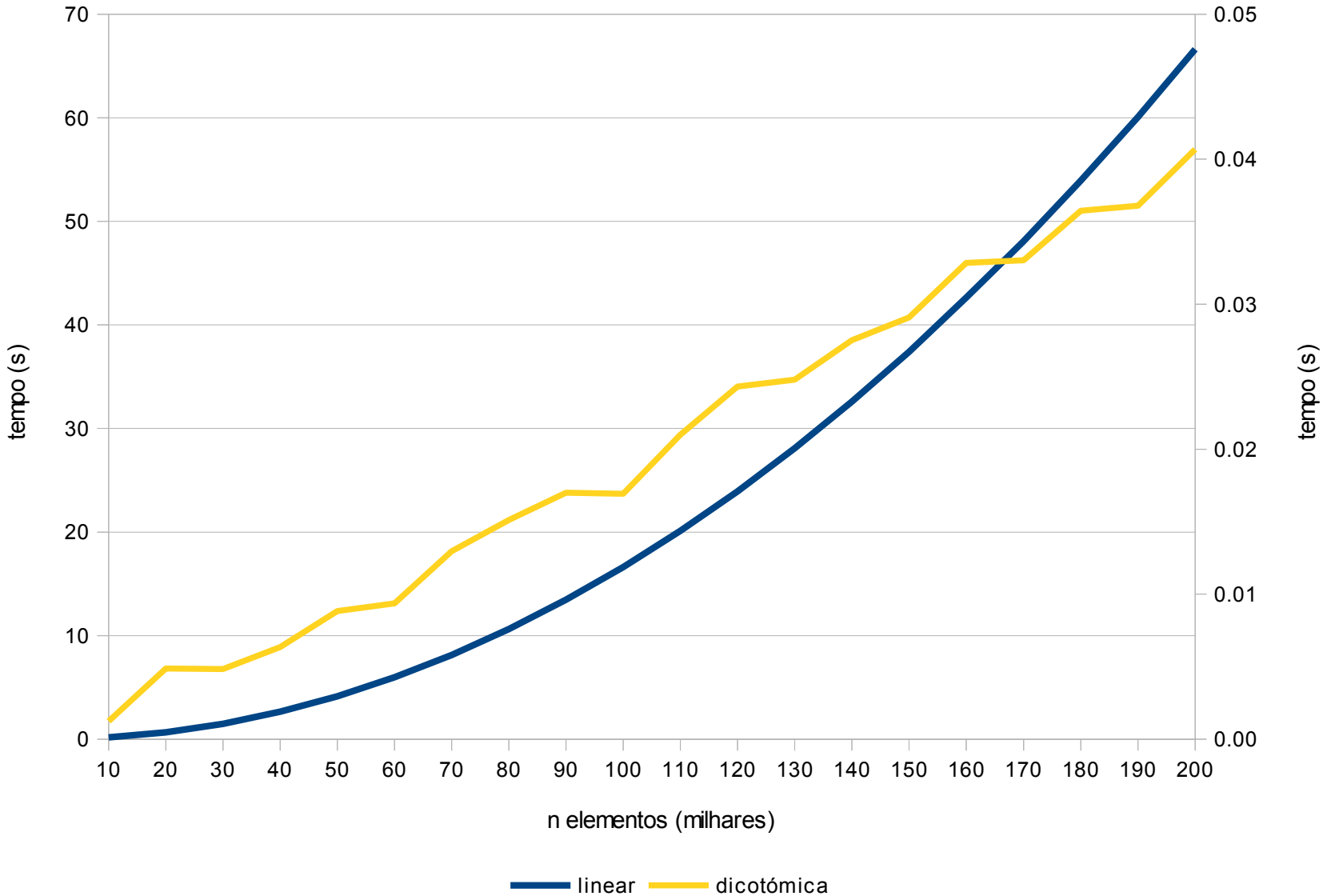
Pesquisas linear e dicotómica

Dos *n* elementos de um vector



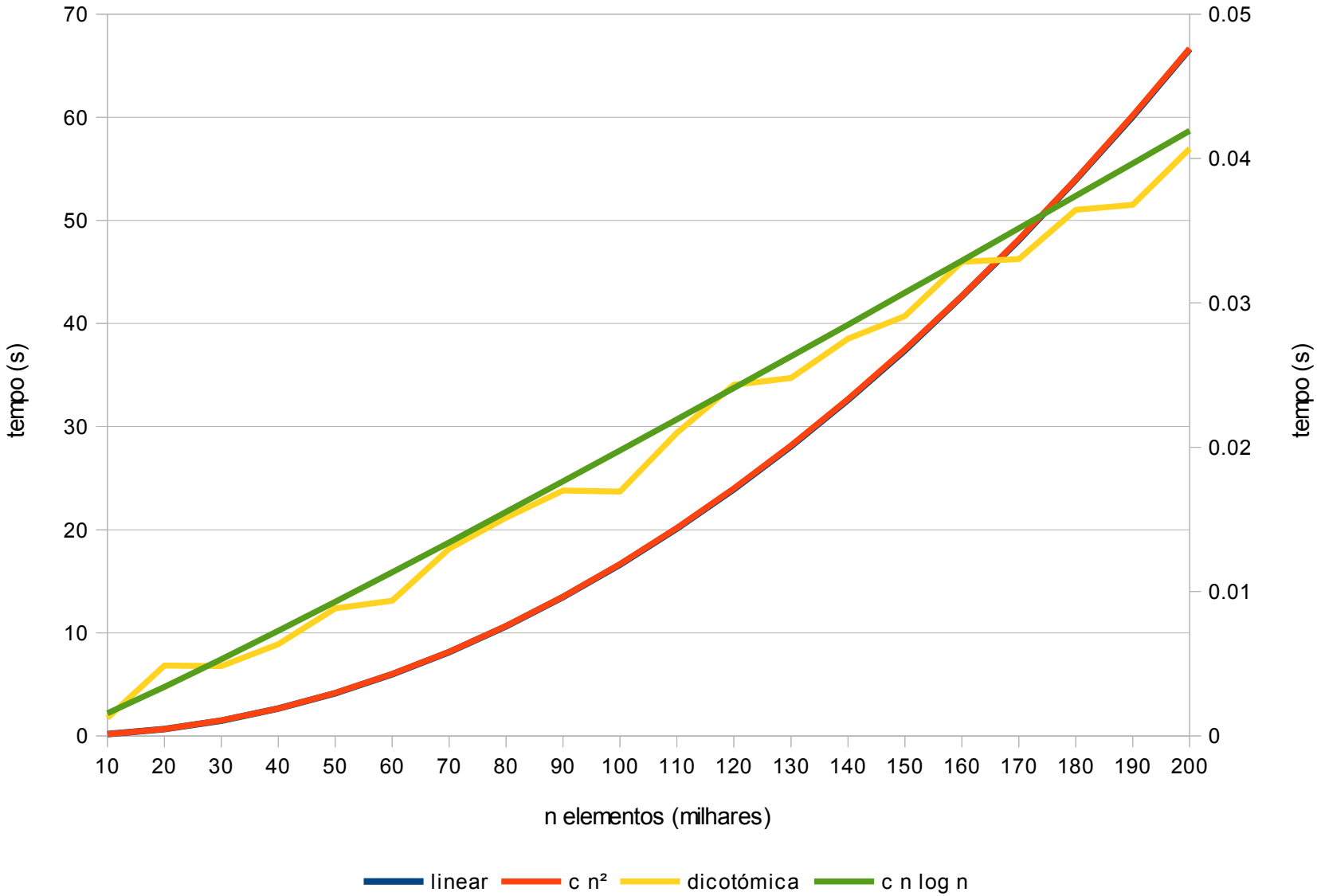
Pesquisas linear e dicotómica (com escalas diferentes)

Dos *n* elementos de um vector



Pesquisas linear e dicotómica (com escalas diferentes)

Dos *n* elementos de um vector



Números de Fibonacci

Versão recursiva

```
public static int fibonacci(int n)
{
  if (n == 0)
    return 0;

  if (n == 1)
    return 1;

  return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Números de Fibonacci

Versão iterativa com tabelação

```
public static int fibonacci(int n)
{
  int[] tabela = new int[n + 1];

  // casos base
  tabela[0] = 0;
  tabela[1] = 1;

  for (int i = 2; i <= n; ++i)
    tabela[i] = tabela[i - 1] + tabela[i - 2];

  return tabela[n];
}
```

Números de Fibonacci

Versão iterativa

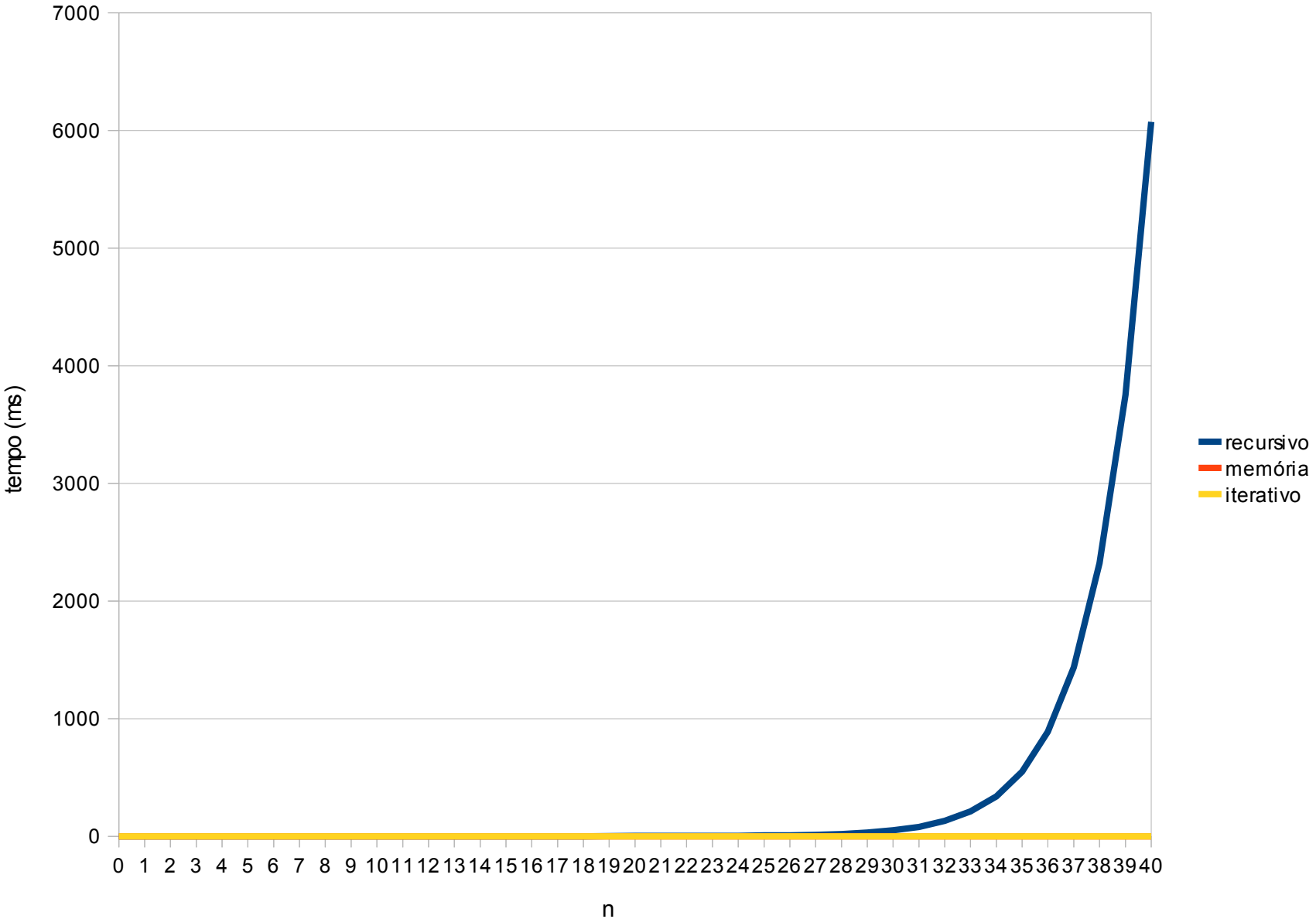
```
public static int fibonacci(int n)
{
  int i = 0;

  int corrente = 0;      // fibonacci(i)
  int anterior = 1;      // fibonacci(i - 1)

  while (i < n)
    {
      // fibonacci(i + 1)
      int proximo = corrente + anterior;

      anterior = corrente;
      corrente = proximo;

      i++;
    }

  return corrente;
}
```

Números de Fibonacci

Versão recursiva com memória

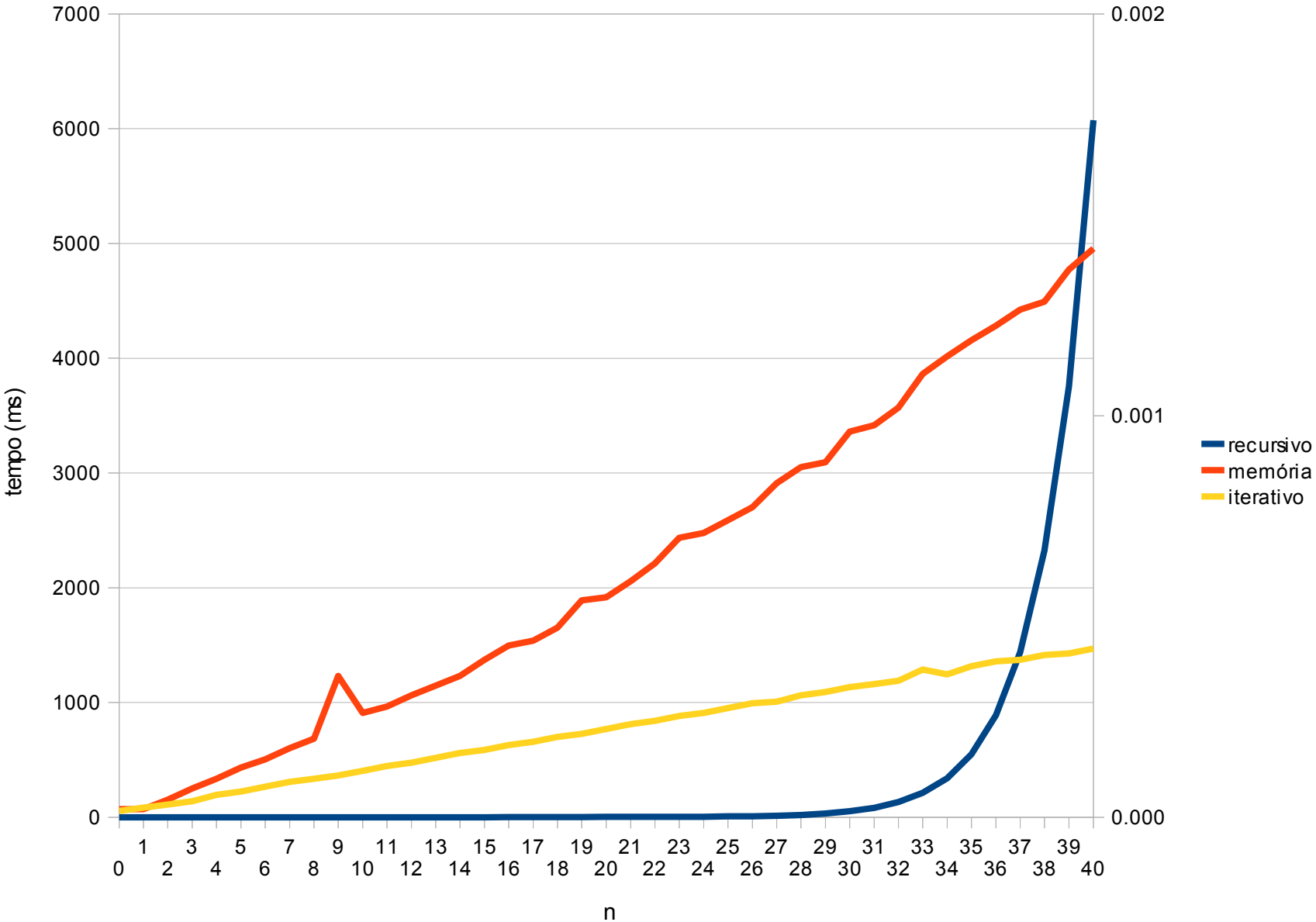
```
private static int CARDINAL_DOMINIO = ...;
private static int[] memoria;
static {
  memoria = new int[CARDINAL_DOMINIO];
  memoria[1] = 1;
}

public static int fibonacci(int n)
{
  if (n > 1 && memoria[n] == 0)
    memoria[n] = fibonacci(n - 1) + fibonacci(n - 2);

  return memoria[n];
}
```

Números de Fibonacci



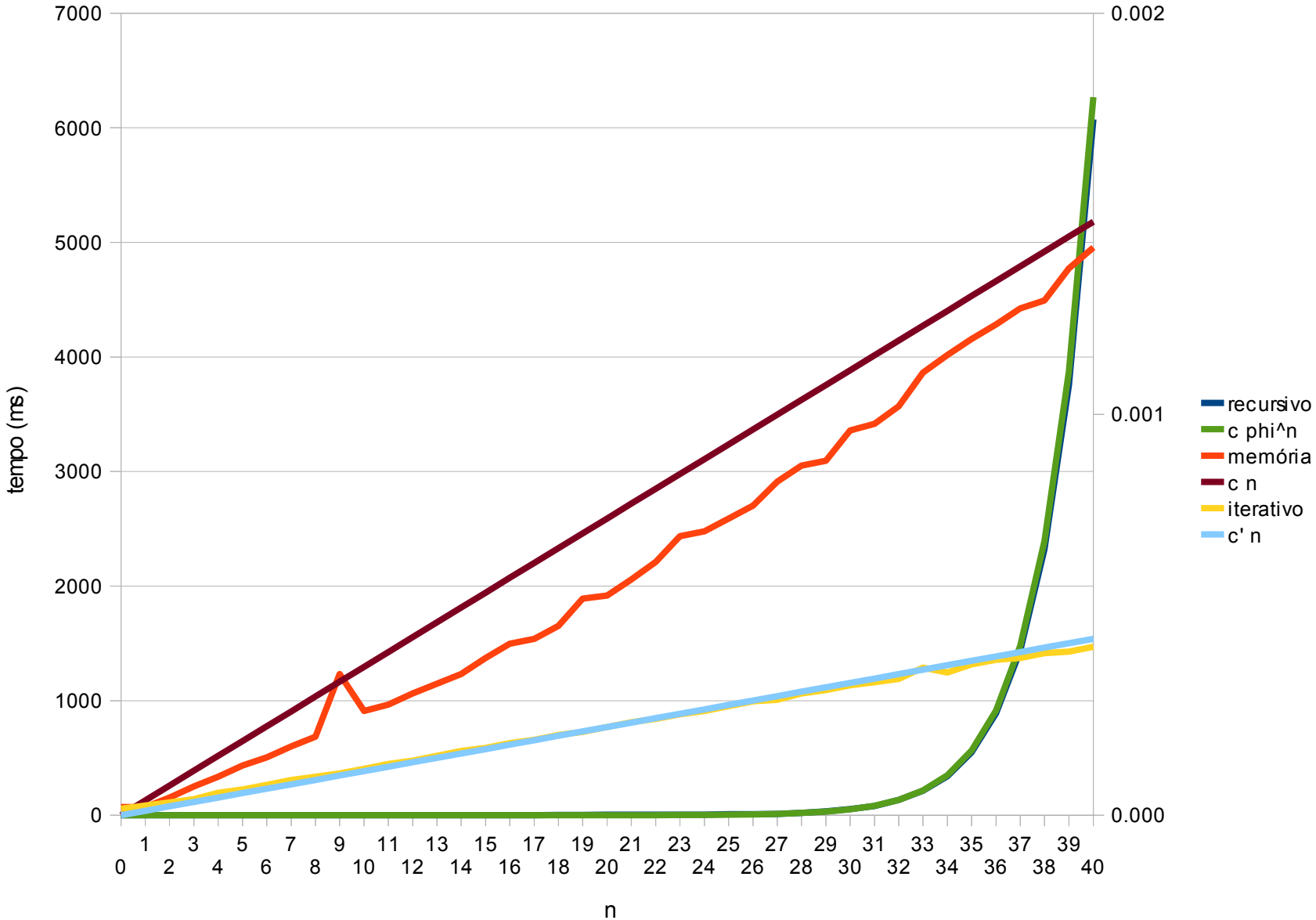
Números de Fibonacci

Escalas diferentes



Números de Fibonacci

Escalas diferentes



Técnicas de construção de algoritmos

- Força bruta
- Divisão e conquista
- Abordagem *greedy*
- Programação dinâmica

Força bruta

Também conhecida como abordagem ingénua

A solução é calculada da maneira mais directa possível, sem recorrer a qualquer técnica para diminuir o número de operações feitas

Inclui os algoritmos de geração e teste

Pode ser útil:

- Quando a dimensão dos problemas a tratar é pequena
- Para chegar a uma primeira implementação
- Para ajudar a ter confiança noutros algoritmos, cuja correcção é mais difícil de estabelecer

Divisão e conquista

Abordagem

1. Dividir o problema em subproblemas (mais pequenos)
2. Conquistar, resolvendo os subproblemas
3. Combinar as soluções dos subproblemas para obter a solução do problema original

Exemplos

- *Merge sort*
- *Quicksort*

Abordagem *greedy*

Aplica-se, em geral, para resolver problemas de optimização

- Nos problemas de optimização procura-se uma solução que é melhor, de acordo com algum critério
  - O maior lucro
  - O menor custo
  - A que requer menos operações
  - ...

A solução é construída fazendo, em cada momento, a escolha que parece ser a melhor

Nem todos os problemas de optimização podem ser resolvidos através de um algoritmo *greedy*

Também conhecidos como algoritmos gananciosos, ansiosos, gulosos, ...

Programação dinâmica

Método usado na construção de soluções iterativas para problemas cuja solução recursiva tem uma complexidade elevada (exponencial, em geral)

Aplica-se, normalmente, a problemas de optimização

- Um problema de optimização é um problema em que se procura minimizar ou maximizar algum valor associado às suas soluções
- Uma solução com essa característica diz-se óptima
- Pode haver várias soluções óptimas

Venda de varas a retalho

Uma empresa compra varas de aço, corta-as e vende-as aos pedaços

O preço de venda de cada pedaço depende do seu comprimento

Problema

Como cortar uma vara de comprimento n de forma a maximizar o seu valor de venda?

Comprimento i	1	2	3	4	5	6	7	8	9	10
Preço p_i	1	5	7	11	11	17	20	20	24	27

Corte de varas

Caracterização de uma solução óptima (1)

Soluções possíveis, para uma vara de comprimento 10

- Um corte de comprimento 1, mais as soluções para uma vara de comprimento 9
- Um corte de comprimento 2, mais as soluções para uma vara de comprimento 8
- Um corte de comprimento 3, mais as soluções para uma vara de comprimento 7

...

- Um corte de comprimento 9, mais as soluções para uma vara de comprimento 1
- Um corte de comprimento 10, mais as soluções para uma uma vara de comprimento 0

Qual a melhor?

Corte de varas

Caracterização de uma solução óptima (2)

Sejam os tamanhos dos cortes possíveis

$$1, 2, \ldots, n$$

com preços

$$p_1, p_2, \ldots, p_n$$

O valor máximo de venda de uma vara de comprimento n é o máximo que se obtém

- fazendo um corte inicial de comprimento $1 \leq i \leq n$, de valor p_i, somado com
- o valor máximo de venda de uma vara de comprimento $n - i$

Corte de varas

Função recursiva

Corte de uma vara de comprimento n

Tamanho dos cortes: $i = 1, \ldots, n$
Preços: $P = (p_1\, p_2\, \ldots\, p_n)$

$v_P(0..n)$: função t.q. $v_P(l)$ é o valor máximo de venda de uma vara de comprimento l, dados os preços P

$$v_P(l) = \begin{cases} 0 & \text{se} \quad l = 0 \\ \max\limits_{1 \le i \le l} \{p_i + v_P(l-i)\} & \text{se} \quad l > 0 \end{cases}$$

Chamada inicial da função

Valor máximo de venda de uma vara completa: $v_P(n)$

Corte de varas

Implementação recursiva

CUT-ROD(p, l)

```
1 if l = 0 then
2   return 0
3 q <- −∞
4 for i <- 1 to l do
5   q <- max(q, p[i] + CUT-ROD(p, l - i))
6 return q
```

Argumentos

- p Preços das varas de comprimentos $\{1, 2, \ldots, n\}$
- l Comprimento da vara a cortar

Chamada inicial da função: CUT-ROD(p, n)

Corte de varas

Alguns números

Número de cortes possíveis

$$2^{n-1}$$

Exemplo ($n = 4$)

$4 \quad 1+3 \quad 2+2 \quad 3+1$

$1+1+2 \quad 1+2+1 \quad 2+1+1 \quad 1+1+1+1$

Número de cortes distintos possíveis

$$O\left(\frac{e^{\pi\sqrt{\frac{2n}{3}}}}{4n\sqrt{3}}\right)$$

Exemplo ($n = 4$)

$4 \quad 1+3 \quad 2+2 \quad 1+1+2 \quad 1+1+1+1$

Corte de varas

Implementação recursiva com *memoização*

MEMOIZED-CUT-ROD(p, n)

```
1 let v[0..n] be a new array
2 for l <- 0 to n do
3       v[l] <- −∞
4 return MEMOIZED-CUT-ROD-2(p, n, v)
```

MEMOIZED-CUT-ROD-2(p, l, v)

```
1 if v[l] = −∞ then
2   if l = 0 then
3     q <- 0
4   else
5     q <- −∞
6     for i <- 1 to l do
7       q <- max(q, p[i] + MEMOIZED-CUT-ROD-2(p, l - i, v))
8   v[l] <- q
9 return v[l]
```

Corte de varas

Cálculo iterativo de $v[n]$ (1)

p_i	1	5	7	11	11	17	20	20	24	27

Preenchimento do vector v

	0	1	2	3	4	5	6	7	8	9	10
$v[l]$	0	1	5	7	11	12	17	20	22	25	28

1. Caso base: $v[0] \leftarrow 0$
2. $v[1] \leftarrow \max\{\, p_1 + v[0] \,\} = \max\{\, 1 + 0 \,\}$
3. $v[2] \leftarrow \max\{\, p_1 + v[1], p_2 + v[0] \,\} = \max\{\, 1 + 1, 5 + 0 \,\}$
4. $v[3] \leftarrow \max\{\, p_1 + v[2], p_2 + v[1], p_3 + v[0] \,\} =$
$= \max\{\, 1 + 5, 5 + 1, 7 + 0 \,\}$

...

11. $v[10] \leftarrow \max\{\, p_1 + v[9], p_2 + v[8], \ldots, p_4 + v[6], \ldots, p_{10} + v[0] \,\}$

Corte de varas

Cálculo iterativo de $v[n]$ (2)

BOTTOM-UP-CUT-ROD(p, n)

```
1 let v[0..n] be a new array
2 v[0] <- 0
3 for l <- 1 to n do
4   q <- −∞
5   for i <- 1 to l do
6     q <- max(q, p[i] + v[l - i])
7   v[l] <- q
8 return v[n]
```

Corte de varas

Complexidade

Complexidade de BOTTOM-UP-CUT-ROD($p_1\ p_2\ \ldots\ p_n$)

Ciclo 3–7 é executado n vezes

Ciclo 5–6 é executado l vezes, $l = 1, \ldots, n$

$$1 + 2 + \ldots + n = \sum_{l=1}^{n} l = \frac{n(n+1)}{2} = \Theta(n^2)$$

Todas as operações têm custo constante

Complexidade temporal $\Theta(n^2)$

Complexidade espacial $\Theta(n)$

Corte de varas

Construção da solução (1)

O valor máximo de venda de uma vara é calculado pela função BOTTOM-UP-CUT-ROD

Quais os cortes a fazer para obter esse valor?

Para o preenchimento da posição `l` do vector `v[]`, é escolhido o valor máximo de `p[i] + v[l - i]`

- A inclusão da parcela `p[i]` significa a inclusão de um pedaço de vara de comprimento `i`

Logo, o valor máximo de venda de uma vara de comprimento `l` (vector `c[]`) será obtido:

- Com um pedaço de comprimento `i` e
- Os pedaços que levam ao valor máximo de venda de uma vara de comprimento `l - i`

Corte de varas

Construção da solução (2)

p_i	1	5	7	11	11	17	20	20	24	27

	0	1	2	3	4	5	6	7	8	9	10
$v[l]$	0	1	5	7	11	12	17	20	22	25	28
$c[l]$		1	2	3	4	1	6	7	2	2	4

1. Caso base: $v[0] \leftarrow 0$
2. $v[1] \leftarrow \max\{\, p_1 + v[0] \,\} = \max\{\, 1 + 0 \,\}$, $c[1] \leftarrow 1$
3. $v[2] \leftarrow \max\{\, p_1 + v[1], p_2 + v[0] \,\} = \max\{\, 1+1, 5+0 \,\}$, $c[2] \leftarrow 2$
4. $v[3] \leftarrow \max\{\, p_1 + v[2], p_2 + v[1], p_3 + v[0] \,\} =$
   $= \max\{\, 1 + 5, 5 + 1, 7 + 0 \,\}$, $c[3] \leftarrow 3$

   ...

11. $v[10] \leftarrow \max\{\, p_1 + v[9], p_2 + v[8], \ldots, p_4 + v[6], \ldots, p_{10} + v[0] \,\}$,
    $c[10] \leftarrow 4$

Corte de varas

Construção da solução (3)

$c[1..n]$: $c[l]$ é o primeiro corte a fazer numa vara de comprimento l

EXTENDED-BOTTOM-UP-CUT-ROD(p, n)

```
 1 let v[0..n] and c[1..n] be new arrays
 2 v[0] <- 0
 3 for l <- 1 to n do
 4   q <- -∞
 5   for i <- 1 to l do
 6     if q < p[i] + v[l - i] then
 7       q <- p[i] + v[l - i]
 8       c[l] <- i                  // corte de tamanho i
 9   v[l] <- q
10 return v and c
```

Corte de varas

Resolução completa

PRINT-CUT-ROD-SOLUTION(p, n)

```
1 (v, c) <- EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
2 print "The best price is ", v[n]
3 print "Cuts:"
4 while n > 0 do
5   print c[n]
6   n <- n - c[n]
```

Resultado, para a vara de comprimento 10

```
The best price is 28
Cuts:
4
6
```

Programação dinâmica

Condições de aplicabilidade

A programação dinâmica aplica-se a problemas que apresentam as características seguintes:

Subestrutura óptima (*Optimal substructure*)

- Um problema tem subestrutura óptima se uma sua solução óptima é construída com recurso a soluções óptimas de subproblemas

Subproblemas repetidos (*Overlapping subproblems*)

- Existem subproblemas repetidos quando os subproblemas de um problema têm subproblemas em comum

Programação dinâmica

Aplicação

1. Caracterização de uma solução óptima
2. Formulação recursiva do cálculo do valor de uma solução óptima
3. Cálculo iterativo do valor de uma solução óptima, por tabelamento
4. Construção de uma solução óptima

Produto de matrizes

Cálculo do produto de uma sequência de matrizes (*Matrix-chain multiplication*)

Problema

Dada uma sequência de matrizes a multiplicar

$$A_1 \, A_2 \, \ldots \, A_n, \quad n > 0$$

com dimensões

$$p_0 \times p_1 \quad p_1 \times p_2 \quad \ldots \quad p_{n-1} \times p_n$$

por que ordem efectuar os produtos de modo a minimizar o número de multiplicações entre elementos das matrizes?

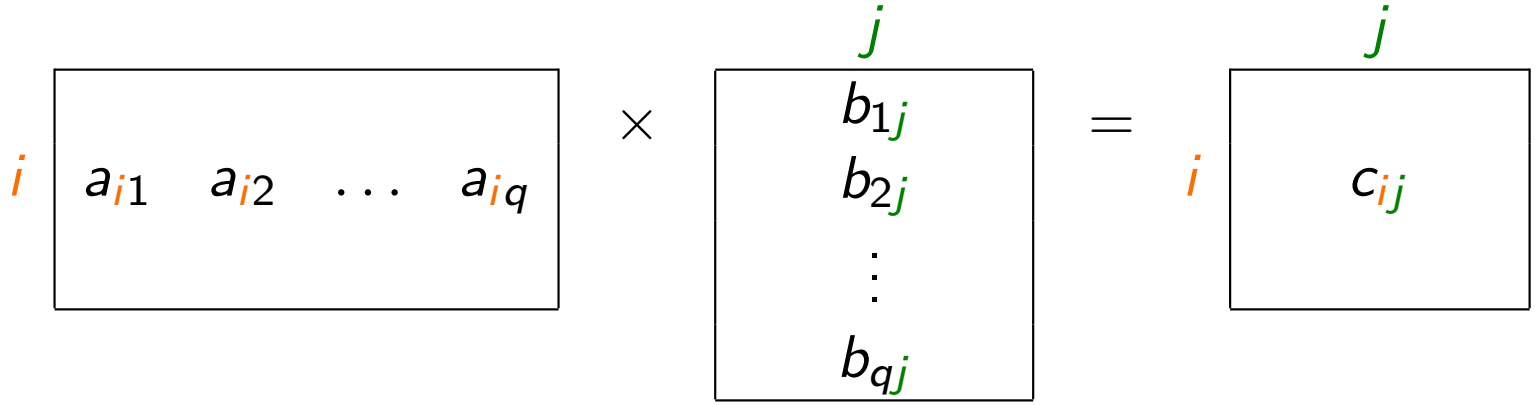
(NOTA 1: A matriz A_i tem dimensão $p_{i-1} \times p_i$)

(NOTA 2: O produto de matrizes é uma operação associativa)

Produto de matrizes

Cálculo do produto de duas matrizes (1)

$$A\ (p \times q) \quad \times \quad B\ (q \times r) \quad = \quad C\ (p \times r)$$



$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \ldots + a_{iq}b_{qj} = \sum_{k=1}^{q} a_{ik}b_{kj}$$

No cálculo de cada elemento de C, são efectuadas q multiplicações (escalares)

Produto de matrizes

Cálculo do produto de duas matrizes (2)

MATRIX-MULTIPLY(A[1..p, 1..q], B[1..q, 1..r])

```
1 let C[1..p,1..r] be a new matrix
2 for i <- 1 to p do
3   for j <- 1 to r do
4     C[i,j] <- 0
5     for k <- 1 to q do
6       C[i,j] <- C[i,j] + A[i,k] * B[k,j]
7 return C
```

Número de multiplicações

Se A e B são matrizes com dimensões $p \times q$ e $q \times r$, respectivamente, no cálculo de $C = AB$, o número de multiplicações efectuadas entre elementos das matrizes é

$$p \times q \times r$$

(C tem $p \times r$ elementos e são efectuadas q multiplicações para o cálculo de cada um)

Produto de uma sequência de matrizes

Exemplo

Sejam A_1, A_2 e A_3 matrizes com dimensões

$$10 \times 100,\ 100 \times 5 \text{ e } 5 \times 50$$

Ordens de avaliação possíveis para o produto $A_1 A_2 A_3$

$$(A_1 A_2) A_3$$
$$A_1 (A_2 A_3)$$

Número de multiplicações

$(A_1 A_2) A_3$

$$10 \times 100 \times 5 + 10 \times 5 \times 50 = 5000 + 2500 = 7500$$

$A_1 (A_2 A_3)$

$$100 \times 5 \times 50 + 10 \times 100 \times 50 = 25000 + 50000 = 75000$$

Produto de uma sequência de matrizes

Colocação de parêntesis

Formulação alternativa

Como colocar parêntesis no produto $A_1 A_2 \ldots A_n$ de modo a realizar o menor número de multiplicações possível?

Número de colocações de parêntesis distintas

$$\Omega\left(\frac{4^n}{n^{\frac{3}{2}}}\right)$$

Produto de uma sequência de matrizes

Caracterização de uma solução óptima (1)

O produto $A_1 A_2 \ldots A_n$ será calculado de uma das formas

$$
\begin{array}{l}
A_1 \, (A_2 \ldots A_n) \\
(A_1 A_2) \, (A_3 \ldots A_n) \\
(A_1 \ldots A_3) \, (A_4 \ldots A_n) \\
\qquad \vdots \\
(A_1 \ldots A_{n-2}) \, (A_{n-1} A_n) \\
(A_1 \ldots A_{n-1}) \, A_n
\end{array}
$$

O número m de multiplicações a efectuar para o cálculo de

$$(A_1 \ldots A_k) \, (A_{k+1} \ldots A_n)$$

para qualquer $1 \leq k < n$, será

$$\mathrm{m}(A_1 \ldots A_k) + \mathrm{m}(A_{k+1} \ldots A_n) + p_0 \, p_k \, p_n$$

Produto de uma sequência de matrizes

Caracterização de uma solução óptima (2)

Procura-se o valor mínimo de

$$\mathrm{m}(A_1 \ldots A_n)$$

que depende do valor mínimo de

$$\mathrm{m}(A_1 \ldots A_k) \quad \text{e de} \quad \mathrm{m}(A_{k+1} \ldots A_n)$$

para algum valor de k

O número mínimo m de multiplicações a efectuar será obtido para o valor de k que minimiza

$$m(A_1 \ldots A_k) + m(A_{k+1} \ldots A_n) + p_0\, p_k\, p_n$$

Produto de uma sequência de matrizes

Função recursiva

Sequência de matrizes a multiplicar

$$A_1\, A_2\, \ldots\, A_n, \quad n > 0$$

Dimensões das matrizes: $P = (p_0\, p_1 \ldots p_n)$

$m_P(1..n, 1..n)$: $m_P(i,j)$ é o menor número de multiplicações a fazer para calcular o produto $A_i \ldots A_j$

$$m_P(i,j) = \begin{cases} 0 & \text{se} \quad i = j \\ \min\limits_{i \le k < j} \{m_P(i,k) + m_P(k+1,j) + p_{i-1} p_k p_j\} & \text{se} \quad i < j \end{cases}$$

Chamada inicial da função

Nº mínimo de multiplicações para a sequência completa: $m_P(1,n)$

Produto de uma sequência de matrizes

Cálculo de $m[i, j]$

m

	1	2	3	4	5
1	0	m_{12}	m_{13}	m_{14}	m_{15}
2		0	m_{23}	m_{24}	m_{25}
3			0	m_{34}	m_{35}
4				0	m_{45}
5					0

Ordem de cálculo

1. Sequências de comprimento 1: m_{11}, m_{22}, m_{33}, m_{44}, m_{55} (Caso base)
2. Sequências de comprimento 2: m_{12}, m_{23}, m_{34}, m_{45}
3. Sequências de comprimento 3: m_{13}, m_{24}, m_{35}
4. Sequências de comprimento 4: m_{14}, m_{25}
5. Sequências de comprimento 5: m_{15}

Produto de uma sequência de matrizes

Cálculo iterativo de $m[1, n]$

Cálculo por comprimento crescente de sequência

MATRIX-CHAIN-ORDER(P)

```
 1 n <- |P| - 1                                  // p[0..n]
 2 let m[1..n,1..n] be a new array
 3 for i <- 1 to n do                            // caso base
 4      m[i, i] <- 0
 5 for ℓ <- 2 to n do                            // ℓ é o comprimento
 6     for i <- 1 to n - ℓ + 1 do
 7         j <- i + ℓ - 1                        // |Ai..Aj| = ℓ
 8         m[i, j] <- +∞
 9         for k <- i to j - 1 do
10             q <- m[i, k] + m[k + 1, j] +
                    p[i - 1] * p[k] * p[j]
11             if q < m[i, j] then
12                 m[i, j] <- q
13 return m[1, n]
```

Produto de uma sequência de matrizes

Complexidade de MATRIX-CHAIN-ORDER($p_0\ p_1\ \ldots\ p_n$)

Todas as operações executadas têm custo constante

Ciclo 3–4 é executado n vezes

Ciclo 5–12 é executado $n-1$ vezes (variável ℓ)

Ciclo 6–12 é executado $n-\ell+1$ vezes (variável i)

Ciclo 9–12 é executado $\ell-1$ vezes (variável k)

$$\sum_{\ell=2}^{n}\sum_{i=1}^{n-\ell+1}\sum_{k=i}^{i+\ell-2} 1 = \sum_{\ell=2}^{n}\sum_{i=1}^{n-\ell+1} \ell-1 = \sum_{\ell=2}^{n}(n-(\ell-1))(\ell-1) = \sum_{\ell=1}^{n-1}(n-\ell)\ell =$$

$$n\sum_{\ell=1}^{n-1}\ell - \sum_{\ell=1}^{n-1}\ell^2 = n\frac{(n-1)n}{2} - \frac{(n-1)n(2n-1)}{6} = \frac{n^3-n}{6} = \Theta(n^3)$$

Complexidade temporal $\Theta(n^3)$

Complexidade espacial $\Theta(n^2)$

Produto de uma sequência de matrizes

Construção da solução

MATRIX-CHAIN-ORDER(P)

```
 1 n <- |P| - 1                                  // p[0..n]
 2 let m[1..n,1..n] and s[1..n-1,2..n] be new arrays
 3 for i <- 1 to n do                            // caso base
 4      m[i, i] <- 0
 5 for ℓ <- 2 to n do                            // ℓ é o comprimento
 6     for i <- 1 to n - ℓ + 1 do
 7         j <- i + ℓ - 1                        // |Ai..Aj| = ℓ
 8         m[i, j] <- +∞
 9         for k <- i to j - 1 do
10             q <- m[i, k] + m[k + 1, j] +
                    p[i - 1] * p[k] * p[j]
11             if q < m[i, j] then
12                 m[i, j] <- q
13                 s[i, j] <- k                 // parte na matriz k
14 return m and s
```

Produto de uma sequência de matrizes

Exemplo

$$P = (10 \quad 100 \quad 5 \quad 50 \quad 3) \quad n = 4$$

Matriz `m` (multiplicações)

	1	**2**	**3**	**4**
1	0	5000	7500	5250
2		0	25000	2250
3			0	750
4				0

Matriz `s` (separação)

	2	**3**	**4**
1	1	2	1
2		2	2
3			3

Número mínimo de multiplicações para calcular ...

$$\begin{aligned}
A_1A_2 &= 5000 \\
A_2A_3 &= 25000 \\
A_1A_2A_3 &= 7500 \\
A_2A_3A_4 &= 2250 \\
A_1A_2A_3A_4 &= 5250
\end{aligned}$$

Separação dos produtos

$$\begin{aligned}
A_1 \ldots A_2 &= (A_1)(A_2) \\
A_1 \ldots A_3 &= (A_1A_2)(A_3) \\
A_2 \ldots A_4 &= (A_2)(A_3A_4) \\
A_1 \ldots A_4 &= (A_1)(A_2 \ldots A_4) \\
&= (A_1)(A_2(A_3A_4))
\end{aligned}$$

Produto de uma sequência de matrizes

Melhor colocação de parêntesis

$s[1..n-1, 2..n]$: $s[i, j]$ é a posição onde a sequência $A_i \ldots A_j$ é dividida: $(A_i \ldots A_{s[i,j]})(A_{s[i,j]+1} \ldots A_j)$

PRINT-OPTIMAL-PARENS(s, i, j)

```
1 if i = j then
2   print "A"i
3 else
4   print "("
5   PRINT-OPTIMAL-PARENS(s, i, s[i, j])
6   PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
7   print ")"
```

Produto de uma sequência de matrizes

Cálculo iterativo de $m[1, n]$ — Variante 1

Cálculo por linhas

MATRIX-CHAIN-ORDER(P)

```
 1 n <- |P| - 1                              // p[0..n]
 2 let m[1..n,1..n] be a new array
 3 for i <- 1 to n do                        // caso base
 4      m[i, i] <- 0
 5 for i <- n - 1 downto 1 do
 6     for j <- i + 1 to n do
 7         m[i, j] <- +∞
 8         for k <- i to j - 1 do
 9             q <- m[i, k] + m[k + 1, j] +
                    p[i - 1] * p[k] * p[j]
10             if q < m[i, j] then
11                 m[i, j] <- q
12 return m[1, n]
```

Produto de uma sequência de matrizes

Cálculo iterativo de $m[1, n]$ — Variante 2

Cálculo por colunas

MATRIX-CHAIN-ORDER(P)

```
 1 n <- |P| - 1                              // p[0..n]
 2 let m[1..n,1..n] be a new array
 3 for i <- 1 to n do                        // caso base
 4      m[i, i] <- 0
 5 for j <- 2 to n do
 6     for i <- j - 1 downto 1 do
 7         m[i, j] <- +∞
 8         for k <- i to j - 1 do
 9             q <- m[i, k] + m[k + 1, j] +
                  p[i - 1] * p[k] * p[j]
10             if q < m[i, j] then
11                 m[i, j] <- q
12 return m[1, n]
```

Sequências e subsequências

Seja x a sequência

$$x_1\ x_2\ \ldots\ x_m, \quad m \geq 0$$

A sequência $z = z_1\ z_2 \ldots z_k$ é uma subsequência de x se

$$z_j = x_{i_j}, \quad j = 1, \ldots, k \quad \text{e} \quad i_j < i_{j+1}$$

Exemplo

$$x = \text{A}\ \ \text{B}\ \ \text{C}\ \ \text{B}\ \ \text{D}\ \ \text{A}\ \ \text{B}$$

São (algumas) subsequências de x:

A A B D B B B C B A B C B A

A B C B D A B (*toda a sequência*) λ (*a sequência vazia*)

Não são subsequências de x:

A A A D C E

Subsequências comuns

Sejam x e y as sequências

$$x_1\ x_2\ \ldots\ x_m \quad \text{e} \quad y_1\ y_2\ \ldots\ y_n, \quad m, n \geq 0$$

A sequência z é uma subsequência comum a x e y se

- z é uma subsequência de x e
- z é uma subsequência de y

Exemplo

$$x = \text{A B C B D A B}$$
$$y = \text{B D C A B A}$$

Subsequências comuns a x e a y

A A B C B A λ ...

Maiores subsequências comuns a x e a y

B C A B B C B A B D A B

Maior subsequência comum

Longest common subsequence

Problema

Dadas duas sequências x e y

$$x_1\ x_2\ \ldots\ x_m \quad \text{e} \quad y_1\ y_2\ \ldots\ y_n, \quad m, n \geq 0$$

determinar uma maior subsequência comum a x e a y

Número de subsequências de uma sequência de comprimento m

$$2^m$$

Maior subsequência comum

Caracterização de uma solução óptima (para sequências não vazias)

$$x = x_1\ x_2\ \ldots\ x_m \quad \text{e} \quad y = y_1\ y_2\ \ldots\ y_n, \quad m, n > 0$$

Se o último símbolo é o mesmo ($x_m = y_n$)

Uma maior subsequência comum a x e y será uma maior subsequência comum a

$$x_1\ x_2\ \ldots\ x_{m-1} \quad \text{e} \quad y_1\ y_2\ \ldots\ y_{n-1}$$

acrescida de x_m

Se terminam com símbolos diferentes ($x_m \neq y_n$)

Uma maior subsequência comum a x e y será uma maior de entre as maiores subsequências comuns a

$$x_1\ x_2\ \ldots\ x_{m-1} \quad \text{e} \quad y_1\ y_2\ \ldots\ y_n$$

e as maiores subsequências comuns a

$$x_1\ x_2\ \ldots\ x_m \quad \text{e} \quad y_1\ y_2\ \ldots\ y_{n-1}$$

Maior subsequência comum

Função recursiva

Comprimento de uma maior subsequência comum às sequências

$$x = x_1\, x_2\, \ldots\, x_m \quad \text{e} \quad y = y_1\, y_2\, \ldots\, y_n, \quad m, n \geq 0$$

$c_{xy}(0..m, 0..n)$: $c_{xy}(i, j)$ é o comprimento das maiores subsequências comuns a $x_1 \ldots x_i$ e $y_1 \ldots y_j$

$$c_{xy}(i,j) = \begin{cases} 0 & \text{se } i = 0 \vee j = 0 \\ 1 + c_{xy}(i-1, j-1) & \text{se } i, j > 0 \wedge x_i = y_j \\ \max\{c_{xy}(i-1, j), c_{xy}(i, j-1)\} & \text{se } i, j > 0 \wedge x_i \neq y_j \end{cases}$$

Chamada inicial da função

Comprimento de uma maior subsequência comum a x e y: $c_{xy}(m, n)$

Maior subsequência comum

Tabela para $c_{xy}(i, j)$

A função c_{xy} tem dois argumentos, logo, os valores da função serão guardados numa matriz

Valores possíveis para i

- O valor inicial é $m \geq 0$
- Nas chamadas recursivas, o valor do primeiro argumento mantém-se ou diminui em 1 unidade
- O caso base é atingido quando i é 0

Valores possíveis para j

- O valor inicial é $n \geq 0$
- Nas chamadas recursivas, o valor do segundo argumento mantém-se ou diminui em 1 unidade
- O caso base é atingido quando j é 0

A tabela terá índices $(0..m) \times (0..n)$

Maior subsequência comum

Tabulação de $c_{xy}(i, j)$

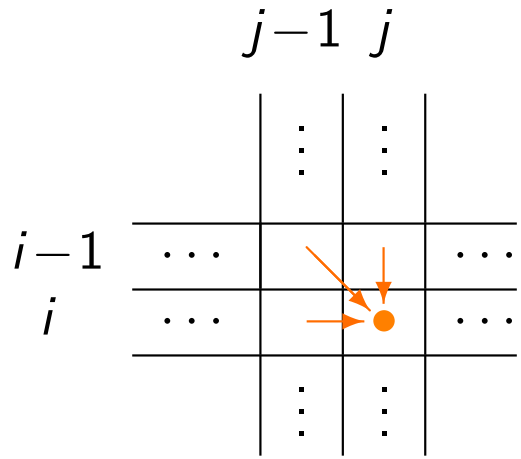
Caso base

Se $i = 0$ ou $j = 0$, então $c(i, j) = 0$

Casos recursivos

O valor de $c(i, j)$ depende:

- Do valor de $c(i - 1, j - 1)$ ou
- Dos valores de $c(i - 1, j)$ e de $c(i, j - 1)$



Para garantir que os valores necessários já estão calculados:

- As linhas são calculadas da linha 1 à linha m
- Dentro de cada linha, os valores são calculados da coluna 1 à coluna n

Argumentos da função iterativa

Os parâmetros do problema, que são as sequências x e y

Maior subsequência comum

Cálculo iterativo de $c[m, n]$

LONGEST-COMMON-SUBSEQUENCE-LENGTH(x, y)

```
 1 m <- |x|
 2 n <- |y|
 3 let c[0..m, 0..n] be a new array
 4 for i <- 1 to m do                    // caso base (j = 0)
 5   c[i, 0] <- 0
 6 for j <- 0 to n do                    // caso base (i = 0)
 7   c[0, j] <- 0
 8 for i <- 1 to m do                    // linhas 1 a m
 9   for j <- 1 to n do                  // colunas 1 a n
10     if x[i] = y[j] then
11       c[i, j] = 1 + c[i - 1, j - 1]
12     else if c[i - 1, j] >= c[i, j - 1] then
13       c[i, j] = c[i - 1, j]
14     else
15       c[i, j] = c[i, j - 1]
16 return c[m, n]
```

Maior subsequência comum

Análise da complexidade

LONGEST-COMMON-SUBSEQUENCE-LENGTH($x_1 \ldots x_m$, $y_1 \ldots y_n$)

Todas as operações executadas têm custo constante

Ciclo 4–5 é executado m vezes

Ciclo 6–7 é executado $n + 1$ vezes

Ciclo 8–15 é executado m vezes

Ciclo 9–15 é executado n vezes em cada iteração do ciclo 8–15

$$\Theta(m) + \Theta(n + 1) + \Theta(mn) = \Theta(mn)$$

Complexidade temporal $\Theta(mn)$

Complexidade espacial $\Theta(mn)$

Maior subsequência comum

Construção da sequência

Para identificar a maior subsequência comum, basta saber se a posição a partir da qual o valor de $c[i, j]$ foi calculado foi

$$\underset{\text{(NW)}}{c[i-1, j-1]} \quad \text{ou} \quad \underset{\text{(N)}}{c[i-1, j]} \quad \text{ou} \quad \underset{\text{(W)}}{c[i, j-1]}$$

Se foi a partir de $c[i-1, j-1]$, o símbolo $x_i = y_j$ é um elemento da subsequência

É possível reconstruir a maior subsequência comum calculada, seguindo as direcções NW, N e W, partindo da posição $[m, n]$ e até chegar à linha ou à coluna 0

Maior subsequência comum

Construção da solução

LONGEST-COMMON-SUBSEQUENCE(x, y)

```
 1 m <- |x|
 2 n <- |y|
 3 let c[0..m, 0..n] and d[1..m, 1..n] be new arrays
 4 for i <- 1 to m do
 5   c[i, 0] <- 0
 6 for j <- 0 to n do
 7   c[0, j] <- 0
 8 for i <- 1 to m do
 9   for j <- 1 to n do
10     if x[i] = y[j] then
11       c[i, j] = 1 + c[i - 1, j - 1]
12       d[i, j] = NW
13     else if c[i - 1, j] >= c[i, j - 1] then
14       c[i, j] = c[i - 1, j]
15       d[i, j] = N
16     else
17       c[i, j] = c[i, j - 1]
18       d[i, j] = W
19 return c and d
```

Maior subsequência comum

Resultado da aplicação a x = ABCBDAB e y = BDCABA

			B	D	C	A	B	A	y
		0	1	2	3	4	5	6	j
	0	0	0	0	0	0	0	0	
A	1	**0**	N 0	N 0	N 0	NW 1	W 1	NW 1	
B	2	0	**NW** **1**	**W** **1**	W 1	N 1	NW 2	W 2	
C	3	0	N 1	N 1	**NW** **2**	**W** **2**	N 2	N 2	
B	4	0	NW 1	N 1	N 2	N 2	**NW** **3**	W 3	
D	5	0	N 1	NW 2	N 2	N 2	**N** **3**	N 3	
A	6	0	N 1	N 2	N 2	NW 3	N 3	**NW** **4**	
B	7	0	NW 1	N 2	N 2	N 3	NW 4	**N** **4**	
x	i								

Maior subsequência comum a x e y calculada: B C B A

Maior subsequência comum

Reconstrução da subsequência

$d[1..m, 1..n]$: $d[i, j]$ é

- NW se $x_i = y_j$
- N se a subsequência calculada é comum a $x_1 \ldots x_{i-1}$ e $y_1 \ldots y_j$
- W se a subsequência calculada é comum a $x_1 \ldots x_i$ e $y_1 \ldots y_{j-1}$

PRINT-LCS(d, x, i, j)

```
1 if i = 0 or j = 0 then
2   return
3 if d[i, j] = NW then
4   PRINT-LCS(d, x, i - 1, j - 1)
5   print x[i]
6 else if d[i, j] = N then
7   PRINT-LCS(d, x, i - 1, j)
8 else // d[i, j] = W
9   PRINT-LCS(d, x, i, j - 1)
```

Dilema do ladrão — Versão 1

Problema

Produtos	Quantidade	Valor
A	10 kg	600
B	20 kg	1000
C	30 kg	1200
D	40 kg	1400

Capacidade do saco (Máximo a transportar)
50 kg

O que levar, para maximizar o produto do roubo?

$$10\,\text{kg} \times \text{A} + 20\,\text{kg} \times \text{B} + 20\,\text{kg} \times \text{C}$$

$$\text{Peso total} = 10 + 20 + 20 = 50$$

$$\text{Valor total} = 10 \times \frac{600}{10} + 20 \times \frac{1000}{20} + 20 \times \frac{1200}{30} = 2400$$

Dilema do ladrão — Versão 1

Solução

Algoritmo

1. Calcular valor por quilo
2. Ordenar produtos por ordem decrescente do valor por quilo
3. Enquanto ainda há algum produto disponível e espaço no saco

   Colocar no saco a maior quantidade possível do próximo produto disponível

É um algoritmo *greedy*

Dilema do ladrão — Versão 2

Artigos indivisíveis — *Knapsack* 0-1

Artigos	Peso (kg)	Valor
A	10	600
B	20	1000
C	30	1200
D	40	1400

Capacidade do saco (Máximo a transportar)
50 kg

O que levar, para maximizar o produto do roubo?

Artigos	Peso total	Valor
A + B	30	1600
A + C	40	1800
A + D	50	2000
B + C	50	2200

Dilema do ladrão — Versão 2

Como escolher?

Escolher artigos mais valiosos

$$D + A$$

Escolher artigos com maior valor/kg

$$A + B$$

Escolher... como, para chegar a?

$$B + C$$

Não existe uma estratégia *greedy* que funcione

Estratégia (exaustiva) possível

Examinar todas as alternativas e escolher uma a que corresponda o maior valor

Resulta? Quantas são? $O(2^{\text{Número de artigos}})$

Dilema do ladrão — Versão 2

Caracterização de uma solução óptima (1)

O maior valor possível de obter com os artigos $\{A, B, C, D\}$ e capacidade 50 kg é o máximo entre

- A soma entre o valor de D e o maior valor possível de obter com os artigos $\{A, B, C\}$ e capacidade $50 - 40 = 10$ kg e
- O maior valor possível de obter com os artigos $\{A, B, C\}$ e capacidade 50 kg

O maior valor possível de obter com os artigos $\{A, B, C\}$ e capacidade j kg é o máximo entre

- A soma entre o valor de C, v_C, e o maior valor possível de obter com os artigos $\{A, B\}$ e capacidade $j - w_C$ kg (se w_C, o peso de C, não for superior a j) e
- O maior valor possível de obter com os artigos $\{A, B\}$ e capacidade j kg

Dilema do ladrão — Versão 2

Caracterização de uma solução óptima (2)

Sejam v_i e w_i, respectivamente, o valor e o peso do artigo i

Em geral, o maior valor possível de obter com os artigos

$$1, \ldots, i$$

e capacidade j será o máximo entre:

- A soma de v_i e o maior valor possível de obter com os artigos

$$1, \ldots, i-1$$

e capacidade $j - w_i$ (só se $w_i \leq j$) e

- O maior valor possível de obter com os artigos

$$1, \ldots, i-1$$

e capacidade j

Dilema do ladrão — Versão 2

Função recursiva

n artigos, capacidade C

Valor dos artigos: $V = (v_1\ v_2\ \ldots\ v_n)$
Peso dos artigos: $W = (w_1\ w_2\ \ldots\ w_n)$

$m_{vw}(0..n, 0..C)$: $m_{vw}(i,j)$ é o maior valor possível com os artigos $1..i$ e capacidade j, dados os valores V e os pesos W

$$m_{vw}(i,j) = \begin{cases} 0 & \text{se } i = 0 \vee j = 0 \\ m_{vw}(i-1,j) & \text{se } i,j > 0 \wedge w_i > j \\ \max \begin{Bmatrix} v_i + m_{vw}(i-1, j-w_i), \\ m_{vw}(i-1,j) \end{Bmatrix} & \text{se } i,j > 0 \wedge w_i \leq j \end{cases}$$

Chamada inicial da função

Valor máximo total do conteúdo do saco: $m_{vw}(n, C)$

Dilema do ladrão — Versão 2

Cálculo iterativo de $m[n, C]$

BOTTOM-UP-MAX-LOOT-VALUE(V, W, c)

```
 1 n <- |v|
 2 let m[0..n, 0..c] be a new array

 3 for i <- 1 to n do                          // caso base
 4   m[i, 0] <- 0
 5 for j <- 0 to c do
 6   m[0, j] <- 0

 7 for i <- 1 to n do                          // casos recursivos
 8   for j <- 1 to c do
 9     if w[i] > j then
10       m[i, j] <- m[i - 1, j]
11     else if v[i] + m[i - 1, j - w[i]] >= m[i - 1, j] then
12       m[i, j] <- v[i] + m[i - 1, j - w[i]]
13     else
14       m[i, j] <- m[i - 1, j]

15 return m[n, c]
```

Dilema do ladrão — Versão 2

Análise da complexidade (1)

BOTTOM-UP-MAX-LOOT-VALUE($v_1 \ldots v_n$, $w_1 \ldots w_n$, c)

Ciclo 3–4 é executado n vezes

Ciclo 5–6 é executado $c + 1$ vezes

Ciclo 7–14 é executado n vezes

Ciclo 8–14 é executado c vezes em cada iteração do ciclo 7–14

Complexidade temporal $\Theta(nc)$

Complexidade espacial $\Theta(nc)$

Dilema do ladrão — Versão 2

Análise da complexidade (2)

A complexidade é expressa em função da dimensão das entradas

Entrada	Dimensão	
$v_1 \ldots v_n$	n	
$w_1 \ldots w_n$	n	
c	$\log c = b$	(nº de algarismos de c)

A complexidade temporal de BOTTOM-UP-MAX-LOOT-VALUE é, então

$$\Theta(nc) = \Theta(n\,2^b)$$

que é exponencial na dimensão de c

Trata-se de um algoritmo pseudo-polinomial

Dilema do ladrão — Versão 2

Construção da solução

BOTTOM-UP-MAX-LOOT(V, W, c)

```
 1 n <- |v|
 2 let m[0..n, 0..c] and a[0..n, 1..c] be new arrays
 3 for i <- 1 to n do                          // caso base
 4   m[i, 0] <- 0
 5 for j <- 0 to c do
 6   m[0, j] <- 0
 7   a[0, j] <- 0
 8 for i <- 1 to n do                          // casos recursivos
 9   for j <- 1 to c do
10     if w[i] > j then
11       m[i, j] <- m[i - 1, j]
12       a[i, j] <- a[i - 1, j]
13     else if v[i] + m[i - 1, j - w[i]] >= m[i - 1, j] then
14       m[i, j] <- v[i] + m[i - 1, j - w[i]]
15       a[i, j] <- i
16     else
17       m[i, j] <- m[i - 1, j]
18       a[i, j] <- a[i - 1, j]
19 return m and a
```

Dilema do ladrão — Versão 2

Artigos a escolher

$a[0..n, 1..c]$: $a[i, j]$ é o último dos artigos $1..i$ a escolher para obter o maior valor para a capacidade j

PRINT-LOOT(a, w, c)

```
1 n <- |w|
2 while c > 0 and a[n, c] != 0 do
3   k <- a[n, c]
4   print "article: " k
5   c <- c - w[k]
6   n <- k - 1
```

Grafos

Grafos

Orientados ou **não orientados**

Pesados (ou **etiquetados**) ou **não pesados** (**não etiquetados**)

Grafo $G = (V, E)$

V – conjunto dos **nós** (ou **vértices**)

$E \subseteq V^2$ – conjunto dos **arcos** (ou **arestas**)

$w : E \to \mathbb{R}$ – **peso** (ou **etiqueta**) de um arco

Vértices e arcos (1)

Se $G = (V, E)$ e $(u, v) \in E$

- O nó v diz-se adjacente ao nó u
- Os nós u e v são vizinhos

Se G é não orientado:

- Os nós u e v são as extremidades do arco (u, v)
- Os arcos (u, v) e (v, u) são o mesmo arco
- Logo, o nó u também é adjacente ao nó v
- O arco (u, v) liga os nós u e v
- O arco (u, v) é incidente no nó u e no nó v

Vértices e arcos (2)

Se G é orientado:

- O nó u é a origem do arco (u, v)
- O nó v é o destino do arco (u, v)
- O nó u é um predecessor (ou antecessor) do nó v
- O nó v é um sucessor do nó u
- O arco (u, v) sai, ou parte, do nó u
- O arco (u, v) chega ao nó v
- O arco (u, v) é incidente no nó v

O grau do nó u é o número de arcos $(u, v) \in E$

Subgrafos e grafos isomorfos

Subgrafo

Um subgrafo do grafo $G = (V, E)$ é um grafo $G' = (V', E')$ tal que $V' \subseteq V$ e $E' \subseteq E$

Grafos isomorfos

Os grafos $G_1 = (V_1, E_1)$ e $G_2 = (V_2, E_2)$ são isomorfos se existe uma função bijectiva $f : V_1 \to V_2$ tal que

$$(f(u), f(v)) \in E_2 \quad \text{sse} \quad (u, v) \in E_1$$

Caminhos

Um caminho num grafo $G = (V, E)$ qualquer é uma sequência não vazia de vértices $v_i \in V$

$$v_0\, v_1\, \ldots\, v_k \quad (k \geq 0)$$

tal que $(v_i, v_{i+1}) \in E$, para $i < k$

O comprimento do caminho $v_0 v_1 \ldots v_k$ é k, o número de arestas que contém

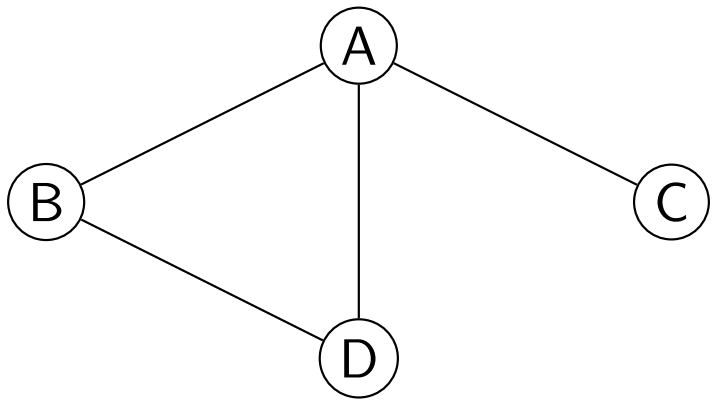
O caminho v_0 é o caminho de comprimento 0, de v_0 para v_0

Um caminho é simples se $v_i \neq v_j$ quando $i \neq j$

Exemplos de grafos

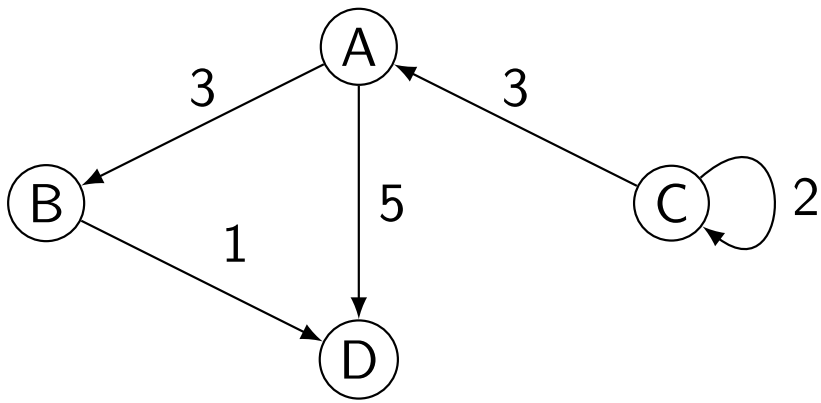
Grafo não orientado e não pesado

$$G = (\{A, B, C, D\}, \{(A, B), (B, D), (A, D), (C, A)\})$$



Grafo orientado pesado

$$G = (\{A, B, C, D\}, \{(A, B, 3), (B, D, 1), (A, D, 5), (C, A, 3), (C, C, 2)\})$$



Ciclos

Um ciclo, num grafo orientado, é um caminho em que

$$v_0 = v_k \quad \text{e} \quad k > 0$$

Num grafo não orientado, um caminho forma um ciclo se

$$v_0 = v_k \quad \text{e} \quad k \geq 3$$

Um ciclo é simples se $v_1, v_2, \ldots, v_k$ são distintos

Um grafo é acíclico se não contém qualquer ciclo simples

Representação / Implementação

Listas de adjacências

- Grafos esparsos $(|E| \ll |V|^2)$
- Permite descobrir rapidamente os vértices adjacentes a um vértice
- Complexidade espacial $\Theta(V + E)$

Matriz de adjacências

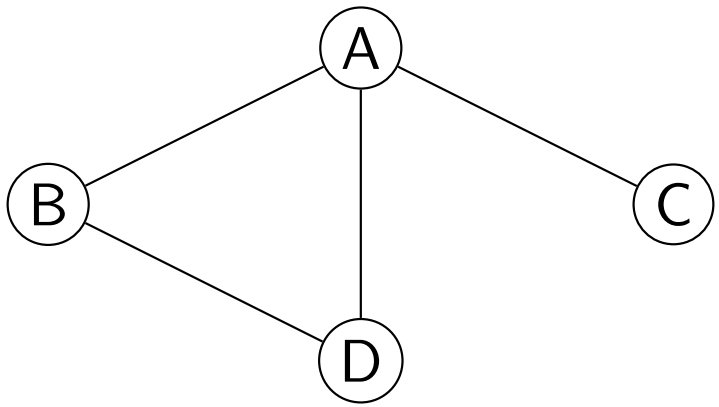
- Grafos densos $(|E| = O(V^2))$
- Permite verificar rapidamente se $(u, v) \in E$
- Complexidade espacial $\Theta(V^2)$

Na notação O, V e E significam, respectivamente, $|V|$ e $|E|$

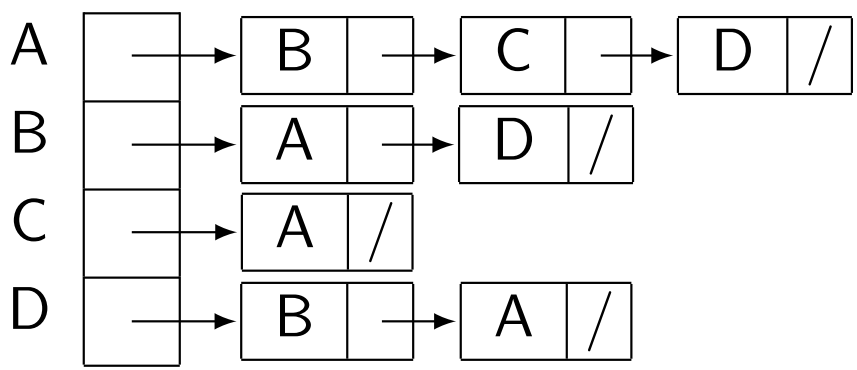
Representação / Implementação

Grafo não orientado e não pesado

Grafo $G = (\{A, B, C, D\}, \{(A, B), (B, D), (A, D), (C, A)\})$



Listas de adjacências



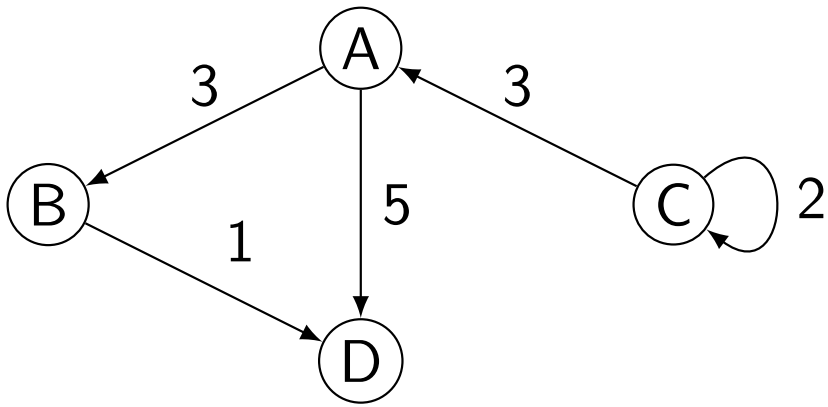
Matriz de adjacências

	A	B	C	D
A	0	1	1	1
B	1	0	0	1
C	1	0	0	0
D	1	1	0	0

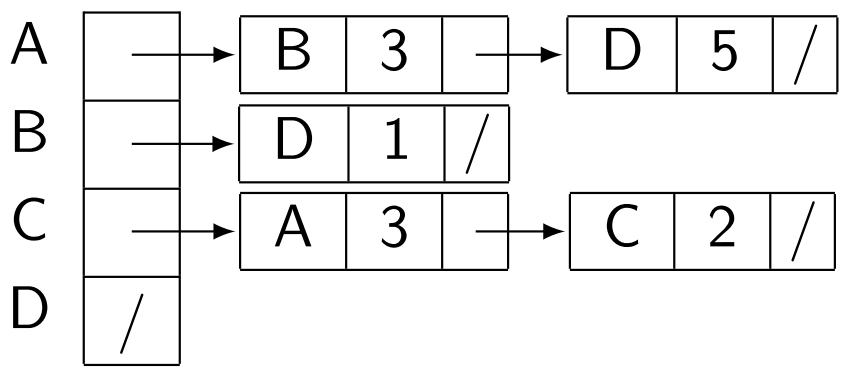
Representação / Implementação

Grafo orientado pesado

Grafo $G = (\{A, B, C, D\}, \{(A, B, 3), (B, D, 1), (A, D, 5), (C, A, 3), (C, C, 2)\})$



Listas de adjacências



Matriz de adjacências

	A	B	C	D
A	0	3	0	5
B	0	0	0	1
C	3	0	2	0
D	0	0	0	0

Implementação em Java

Grafo orientado não pesado

```
class Graph {
  int nodes; // number of nodes
  List<Integer>[] adjacents; // adjacency lists

  @SuppressWarnings("unchecked")
  public Graph(int nodes)
  {
    this.nodes = nodes;

    adjacents = new List[nodes];

    for (int i = 0; i < nodes; ++i)
      adjacents[i] = new LinkedList<>();
  }

  /* Adds the (directed) edge (U,V) to the graph. */
  public void addEdge(int u, int v)
  {
    adjacents[u].add(v);
  }

  ...
}
```

Percursos básicos em grafos

Percurso em largura

Nós são visitados por ordem crescente de distância ao nó em que o percurso se inicia

Percurso em profundidade

Nós são visitados pela ordem por que são encontrados

Percurso em largura (a partir do vértice s)

BFS(G, s)

```
 1  for each vertex u in G.V - {s} do
 2      u.color <- WHITE
 3      u.d <- INFINITY
 4      u.p <- NIL
 5  s.color <- GREY
 6  s.d <- 0
 7  s.p <- NIL
 8  Q <- EMPTY                          // fila (FIFO)
 9  ENQUEUE(Q, s)
10  while Q != EMPTY do
11      u <- DEQUEUE(Q)                 // próximo nó a explorar
12      for each vertex v in G.adj[u] do
13          if v.color = WHITE then
14              v.color <- GREY
15              v.d <- u.d + 1
16              v.p <- u
17              ENQUEUE(Q, v)
18      u.color <- BLACK                // u foi explorado
```

Percurso em largura

Breadth-first search

Descobre um caminho mais curto de um vértice s a qualquer outro vértice

Calcula o seu comprimento (linhas 3, 6 e 15)

Constrói a árvore da pesquisa em largura (linhas 4, 7 e 16), que permite reconstruir o caminho identificado

Atributos dos vértices

`color`	`WHITE`	não descoberto
	`GREY`	descoberto, mas não processado
	`BLACK`	processado
`d`		distância a s
`p`		antecessor do nó no caminho a partir de s

Análise da complexidade temporal de BFS (1)

Grafo implementado através de listas de adjacências

BFS(G, s)

```
1  for each vertex u in G.V - {s} do
2      u.color <- WHITE
3      u.d <- INFINITY
4      u.p <- NIL
```

- Ciclo das linhas 1–4 é executado $|V| - 1$ vezes

```
5  s.color <- GREY
6  s.d <- 0
7  s.p <- NIL
8  Q <- EMPTY                          // fila (FIFO)
9  ENQUEUE(Q, s)
```

- Linhas 5–9 com custo constante

Análise da complexidade temporal de BFS (2)

- Ciclo das linhas 10–18 é executado $|V|$ vezes, no pior caso

```
10  while Q != EMPTY do
11      u <- DEQUEUE(Q)                 // próximo nó a explorar
12      for each vertex v in G.adj[u] do
13          if v.color = WHITE then
14              v.color <- GREY
15              v.d <- u.d + 1
16              v.p <- u
17              ENQUEUE(Q, v)
18      u.color <- BLACK                // u foi explorado
```

- Mas o ciclo das linhas 12–17 é executado, no pior caso

$$\sum_{u \in V} |\,\mathtt{G.adj}[u]\,| = |E| \text{ (orientado) ou } 2 \times |E| \text{ (não orientado) vezes}$$

porque cada vértice só pode entrar na fila uma vez

Análise da complexidade temporal de BFS (3)

Considerando que todas as operações, incluindo a criação de uma fila vazia, ENQUEUE e DEQUEUE, têm custo $\Theta(1)$

- O ciclo das linhas 1–4 tem custo $\Theta(V)$
- Conjuntamente, os ciclos das linhas 10–18 e 12–17 têm custo $O(E)$ (pior caso)

Logo, a complexidade temporal de BFS é $O(V + E)$

Análise da complexidade temporal de BFS (4)

Grafo implementado através da matriz de adjacências

Na linha 12, é necessário percorrer uma linha da matriz, com $|V|$ elementos

```
12'     for each vertex v in G.V do
13'         if G.adjm[u,v] and v.color = WHITE then
```

Como o ciclo das linhas 10–18 é executado $|V|$ vezes, no pior caso, o custo combinado dos dois ciclos é $O(V^2)$

- Corresponde a aceder a todas as posições de uma matriz $|V| \times |V|$

Neste caso, a complexidade temporal de BFS será $O(V^2)$

Complexidade espacial de BFS

Memória usada pelo algoritmo

- 2 variáveis para vértices (`u` e `v`)

$$O(1)$$

- 3 valores escalares (`color`, `d` e `p`) por cada vértice

$$\Theta(V)$$

- Uma fila, que poderá ter, no pior caso, $|V| - 1$ vértices

$$O(V)$$

A complexidade espacial de BFS é $\Theta(V)$

BFS em Java (1)

```
public static final int INFINITY = Integer.MAX_VALUE;
public static final int NONE = -1;

private static enum Colour { WHITE, GREY, BLACK };

public int[] bfs(int s)
{
  Colour[] colour = new Colour[nodes];
  int[] d = new int[nodes];          // distância para S
  int[] p = new int[nodes];          // predecessor no caminho de S

  for (int u = 0; u < nodes; ++u)
    {
      colour[u] = Colour.WHITE;
      d[u] = INFINITY;
      p[u] = NONE;
    }

  colour[s] = Colour.GREY;
  d[s] = 0;

  Queue<Integer> Q = new LinkedList<>();

  Q.add(s);

  ...
```

BFS em Java (2)

```
...
while (!Q.isEmpty())
  {
    int u = Q.remove();                    // visita nó U
    for (Integer v : adjacents[u])
      if (colour[v] == Colour.WHITE)
        {
          colour[v] = Colour.GREY;         // V é um novo nó
          d[v] = d[u] + 1;
          p[v] = u;
          Q.add(v);
        }
    colour[u] = Colour.BLACK;              // nó U está tratado
  }
  return d ou p ou ...
}
```

Percurso em profundidade

DFS(G)

```
 1 for each vertex u in G.V do
 2     u.color <- WHITE
 3     u.p <- NIL
 4 time <- 0                          // variável global
 5 for each vertex u in G.V do        // explora todos os nós
 6     if u.color = WHITE then
 7         DFS-VISIT(G, u)
```

DFS-VISIT(G, u)

```
 1 time <- time + 1                 // instante da descoberta do
 2 u.d <- time                      // vértice u
 3 u.color <- GREY
 4 for each vertex v in G.adj[u] do // explora arco (u, v)
 5     if v.color = WHITE then
 6         v.p <- u
 7         DFS-VISIT(G, v)
 8 u.color <- BLACK                 // u foi explorado
 9 time <- time + 1                 // instante em que termina
10 u.f <- time                      // a exploração de u
```

Percurso em profundidade

Depth-first search

Constrói a floresta da pesquisa em profundidade (linhas 3 [DFS] e 6 [DFS-VISIT])

Atributos dos vértices

`color`	`WHITE`	não descoberto
	`GREY`	descoberto e em processamento
	`BLACK`	processado
`d`		instante em que foi descoberto
`f`		instante em que terminou de ser processado
`p`		antecessor do nó no caminho que levou à sua descoberta

Análise da complexidade temporal de DFS

O ciclo das linhas 1–3 [DFS] é executado $|V|$ vezes

DFS-VISIT é chamada para cada um dos $|V|$ vértices

Para cada vértice u (e considerando a implementação através de listas de adjacências), o ciclo das linhas 4–7 [DFS-VISIT] é executado

$$|\,\texttt{G.adj}[u]\,| \quad \text{vezes}$$

Tendo todas as operações custo constante, considerando todas as chamadas a DFS-VISIT, DFS corre em tempo

$$\Theta(V + \sum_{u \in V} |\,\texttt{G.adj}[u]\,|) = \Theta(V + E)$$

Complexidade espacial de DFS

Memória usada pelo algoritmo

- 4 valores escalares (`color`, `d`, `f` e `p`) por cada vértice

$$\Theta(V)$$

- Uma pilha, que poderá ter, no pior caso, $|V|$ vértices

$$O(V)$$

A pilha será implícita, quando algoritmo for implementado recursivamente, ou explícita, quando for implementado iterativamente

A complexidade espacial de DFS é $\Theta(V)$

Complexidades dos percursos

Resumo

$G = (V, E)$

	Complexidade		
	Temporal		Espacial
Percurso em largura	$O(V + E)$	$O(V^2)$	$\Theta(V)$
Percurso em profundidade	$\Theta(V + E)$	$\Theta(V^2)$	$\Theta(V)$
	Listas de adjacências	Matriz de adjacências	

Se, no percurso em largura, for percorrido todo o grafo (como é feito no percurso em profundidade), as complexidades temporais também serão $\Theta(V + E)$ e $\Theta(V^2)$, respectivamente

Se o percurso em profundidade for feito a partir de um único nó (como no percurso em largura), as complexidades temporais também serão $O(V + E)$ e $O(V^2)$, respectivamente

Ordenação topológica

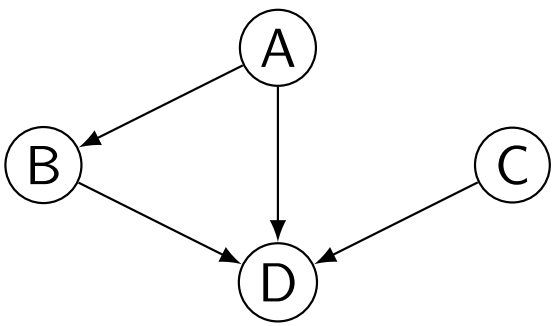
Seja $G = (V, E)$ um grafo orientado acíclico (DAG, de *directed acyclic graph*)

Ordem topológica

Se existe um arco de u para v, u está antes de v na ordem dos vértices

$$(u, v) \in E \Rightarrow u < v$$

Exemplo



Ordem

$A < B \quad A < D \quad B < D \quad C < D$

Ordenações possíveis

- A B C D
- A C B D
- C A B D

Ordenação topológica

Um algoritmo baseado no percurso em profundidade

Aplicação do percurso em profundidade

- Se não há caminho de u para nenhum outro vértice, u poderá ser o último vértice da ordenação topológica
- Se há um caminho de u para outro vértice v, então u estará antes de v, em qualquer ordenação topológica

TOPOLOGICAL-SORT(G)

1. Aplicar DFS(G)
2. Durante o percurso, quando termina o processamento de um vértice, inseri-lo à cabeça de uma lista
3. Devolver a lista, que contém os vértices por (alguma) ordem topológica

Ordenação topológica

Adaptação de DFS

$G = (V, E)$ – grafo orientado acíclico (DAG)

TOPOLOGICAL-SORT(G)

```
1 for each vertex u in G.V do
2     u.color <- WHITE
3 L <- EMPTY                          // lista, global
4 for each vertex u in G.V do
5     if u.color = WHITE then
6         DFS-VISIT'(G, u)
7 return L
```

DFS-VISIT'(G, u)

```
1 u.color <- GREY
2 for each vertex v in G.adj[u] do
3     if v.color = WHITE then
4         DFS-VISIT'(G, v)
5 u.color <- BLACK
6 LIST-INSERT-HEAD(L, u)
```

Ordenação topológica

Outro algoritmo (1)

- Se u não é o destino de nenhum arco, u pode ser o primeiro nó da ordenação topológica

- Uma vez ordenados todos os vértices u tais que $(u, v) \in E$, o vértice v pode ser colocado a seguir

Ordenação topológica

Outro algoritmo (2)

TOPOLOGICAL-SORT'(G)

```
 1 for each vertex u in G.V do
 2     u.i <- 0
 3 for each edge (u,v) in G.E do
 4     v.i <- v.i + 1                  // arcos incidentes em v
 5 L <- EMPTY                          // lista
 6 S <- EMPTY                          // conjunto
 7 for each vertex u in G.V do
 8     if u.i = 0 then
 9         SET-INSERT(S, u)
10 while S != EMPTY do
11     u <- SET-DELETE(S)              // retira um nó de S
12     for each vertex v in G.adj[u] do
13         v.i <- v.i - 1
14         if v.i = 0 then
15             SET-INSERT(S, v)
16     LIST-INSERT-TAIL(L, u)
17 return L
```

Complexidade dos algoritmos

$G = (V, E)$

	Compl. Temporal
Percurso em largura	$O(V + E)$
Percurso em profundidade	$\Theta(V + E)$
Ordenação topológica (ambos os algoritmos)	$\Theta(V + E)$

Pressupostos

Grafo representado através de listas de adjacências

Se, no percurso em largura, for percorrido todo o grafo (como é feito no percurso em profundidade), a complexidade temporal também será $\Theta(V + E)$

Conectividade (1)

Seja $G = (V, E)$ um grafo não orientado

G é conexo se existe algum caminho entre quaisquer dois nós:

$u, v \in V \Rightarrow$ existe (pelo menos) um caminho $v_0 v_1 \ldots v_k, k \geq 0$, com $v_0 = u$ e $v_k = v$

$V' \subseteq V$ é uma componente conexa de G se

- existe algum caminho entre quaisquer dois nós de V' e
- não existe qualquer caminho entre algum nó de V' e algum nó de $V \setminus V'$

Conectividade (2)

Seja $G = (V, E)$ um grafo orientado

G é fortemente conexo se existe algum caminho de qualquer nó para qualquer outro nó

$V' \subseteq V$ é uma componente fortemente conexa de G se

- existe algum caminho de qualquer nó de V' para qualquer outro nó de V' e
- se, qualquer que seja o nó $u \in V \setminus V'$
  - não existe qualquer caminho de algum nó de V' para u ou
  - não existe qualquer caminho de u para algum nó de V'

G é simplesmente conexo se, substituindo todos os arcos por arcos não orientados, se obtém um grafo conexo

Grafo transposto

O grafo transposto do grafo orientado $G = (V, E)$ é o grafo

$$G^T = (V, E^T)$$

tal que

$$E^T = \{(v, u) \mid (u, v) \in E\}$$

Componentes fortemente conexas

Strongly connected components

G – grafo orientado

SCC(G)

1. Executar DFS(G) para calcular o instante `u.f` em que termina o processamento de cada vértice `u`
2. Calcular G^T
3. Executar DFS(G^T), processando os vértices por ordem decrescente de `u.f` (calculado em 1), no ciclo principal de DFS (linha 5)
4. Devolver os vértices de cada árvore da floresta da pesquisa em profundidade (construída em 3) como uma componente fortemente conexa distinta

Árvore de cobertura mínima (1)

Minimum(-weight) spanning tree (MST)

Seja $G = (V, E)$ um grafo pesado não orientado conexo

Uma árvore é um grafo não orientado conexo acíclico

(Retirando qualquer arco de uma árvore, obtém-se um grafo não conexo)

Uma árvore de cobertura de G é um subgrafo $G' = (V', E')$ de G tal que

- $V' = V$
- $E' \subseteq E$
- G' é uma árvore

Árvore de cobertura mínima (2)

Minimum(-weight) spanning tree (MST)

Seja o peso de um grafo $w(G)$ a soma dos pesos dos arcos de G

$$w(G) = \sum_{e \in E} w(e)$$

Uma árvore de cobertura mínima de G é uma árvore de cobertura G' de peso mínimo:

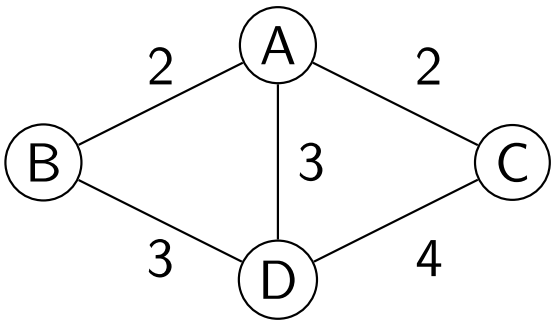
Para qualquer árvore de cobertura G'' de G tem-se

$$w(G') \leq w(G'')$$

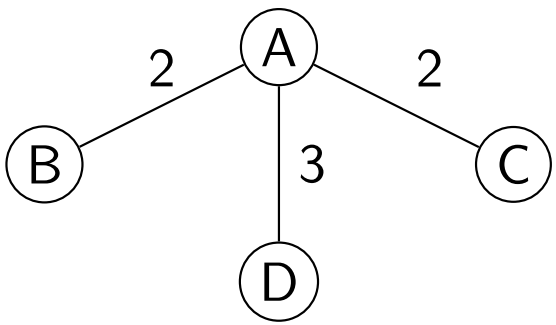
Árvore de cobertura mínima (3)

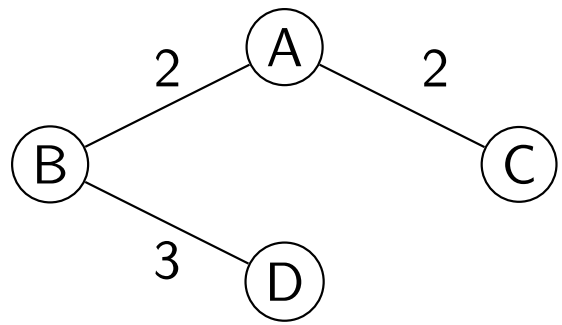
Minimum(-weight) spanning tree (MST)

Exemplo



Árvores de cobertura mínimas





O peso das árvores de cobertura mínimas deste grafo é $2+2+3 = 7$

Árvore de cobertura mínima

Algoritmo de Prim

$G = (V, E)$ – grafo pesado não orientado conexo

MST-PRIM(G, w, s)

```
 1 for each vertex u in G.V do
 2     u.key <- INFINITY          // custo de juntar u à MST
 3     u.p <- NIL                 // nó a que u é ligado
 4 s.key <- 0
 5 Q <- G.V                       // fila com prioridade, por
                                  // mínimos, chave é u.key
 6 while Q != EMPTY do
 7     u <- EXTRACT-MIN(Q)
 8     for each vertex v in G.adj[u] do
 9         if v in Q and w(u,v) < v.key then
10             v.p <- u           // arco (u,v) é candidato
11             v.key <- w(u,v) // pode alterar Q!
```

Filas com prioridade (1)

Uma fila com prioridade é uma fila em que a cada elemento está associado um valor (`key`), que determina a sua prioridade

Numa fila organizada por mínimos, a prioridade é maior quando o valor é menor

Numa fila organizada por máximos, a prioridade é maior quando o valor é maior

A operação DEQUEUE retira da fila um elemento com maior prioridade

Para tornar explícita a disciplina da fila, no pseudo-código, a operação DEQUEUE é denotada por EXTRACT-MIN, para filas organizadas por mínimos, e por EXTRACT-MAX, para filas organizadas por máximos

Filas com prioridade (2)

Implementação, para um conjunto limitado de elementos

Vector ou lista (duplamente) ligada

- Inserção de elemento com complexidade temporal constante
- Remoção de elemento com complexidade temporal linear no número de elementos na fila
- Ou vice-versa, se a fila for mantida ordenada por prioridade

Heap binário

- Ambas as operações com complexidade temporal logarítmica no número de elementos na fila
- Criação, a partir de um conjunto de elementos, com complexidade temporal linear no número de elementos

Filas com prioridade com actualização da prioridade (1)

Por vezes, como acontece no algoritmo de Prim, é necessário aumentar a prioridade de um elemento presente na fila

Operação denotada por DECREASE-KEY, numa fila organizada por mínimos, e por INCREASE-KEY, numa fila organizada por máximos

Nas implementações em vector ou em lista, essa operação tem complexidade temporal constante (ou linear, se a fila for mantida ordenada)

Heap binário

Na implementação normal com um *heap* binário, a operação tem complexidade temporal linear no número de elementos na fila, devido a ser preciso localizar o elemento

Se, além do *heap*, a implementação mantiver um mapa, com a posição de cada elemento, a operação pode ser realizada com complexidade temporal logarítmica no número de elementos na fila

Filas com prioridade com actualização da prioridade (2)

Poor man's approach

Uma alternativa à actualização da prioridade de um elemento presente na fila é uma nova inserção do elemento, com o novo valor associado

- Podem existir várias cópias do mesmo elemento na fila, com diferentes prioridades
- É necessário associar uma *flag* a cada elemento, que diga se ele já saiu da fila, e foi processado, ou não

A complexidade temporal das várias operações mantém-se

Aumenta o número de elementos que a fila pode conter

Análise da complexidade do algoritmo de Prim (1)

Operação da fila com prioridade

Fila implementada através de um *heap* binário

- Construção de uma fila com n elementos: $\Theta(n)$
- Ver se está vazia: $\Theta(1)$
- Remoção do elemento mínimo: $O(\log n)$
- Determinar se contém um dado elemento: $O(n)$

  Associando uma *flag* a cada vértice, pode-se reduzir a complexidade temporal desta operação para $\Theta(1)$

- Alterar o valor associado a um elemento: $O(n + \log n)$

  Mantendo, para cada vértice, o índice da posição em que se encontra, pode-se reduzir a complexidade temporal desta operação para $O(\log n)$

A seguir, considera-se uma implementação em que cada operação é realizada da maneira mais eficiente

Análise da complexidade do algoritmo de Prim (2)

Grafo representado através de listas de adjacências

Linhas

1–3 Ciclo executado $|V|$ vezes

5 Construção da fila com prioridade (*heap*): $\Theta(V)$

6–11 Ciclo executado $|V|$ vezes

7 Remoção do menor elemento da fila: $O(\log V)$

8–11 Ciclo executado $2|E|$ vezes **no total**

11 Alteração da prioridade de um elemento na fila: $O(\log V)$

Operação executada, no pior caso, $|E|$ vezes

Complexidade temporal do algoritmo

$$O(V + V + V \log V + 2E + E \log V) = O(E \log V)$$

Restantes operações com complexidade temporal constante

Árvore de cobertura mínima

Algoritmo de Kruskal

$G = (V, E)$ – grafo pesado não orientado conexo

MST-KRUSKAL(G, w)

```
 1 n <- |G.V|
 2 A <- EMPTY                        // conjunto dos arcos da MST
 3 P <- MAKE-SETS(G.V)               // partição de G.V, floresta
 4 Q <- G.E                          // fila com prioridade, por
                                     // mínimos, chave é w(u,v)
 5 e <- 0                            // número de arcos na MST
 6 while e < n - 1 do
 7     (u,v) <- EXTRACT-MIN(Q)
 8     if FIND-SET(P, u) != FIND-SET(P, v) then
 9         A <- A + {(u,v)}          // novo arco da MST
10         UNION(P, u, v)
11         e <- e + 1
12 return A
```

Análise da complexidade do algoritmo de Kruskal (1)

Linhas

3 Construção da partição

$$\text{MAKE-SETS}$$

4 Construção da fila com prioridade (*heap*)

$$\Theta(E)$$

6–11 Ciclo executado entre $|V| - 1$ e $|E|$ vezes

7 Remoção do menor elemento da fila (*heap*)

$$O(\log E) = O(\log V)$$

$$(\, |E| < |V|^2 \rightarrow \log |E| < log|V|^2 = 2 \log |V| = O(\log V)\,)$$

8 $2 \times$ FIND-SET

10 Executada $|V| - 1$ vezes

$$\text{UNION}$$

Restantes operações com complexidade temporal constante

Análise da complexidade do algoritmo de Kruskal (2)

Juntando tudo, obtém-se

$$\text{MAKE-SETS} + \Theta(E) + |E| \times O(\log V) +$$
$$|E| \times 2 \times \text{FIND-SET} + (|V| - 1) \times \text{UNION}$$

ou

$$\Theta(E) + |E| \times O(\log V) + f(V, E)$$

com

$$f(V, E) = \text{MAKE-SETS} + 2 \times |E| \times \text{FIND-SET} + (|V| - 1) \times \text{UNION}$$

Partição

Conjuntos disjuntos (*Disjoint sets*)

Abstracção da implementação de conjuntos disjuntos com os elementos do conjunto $\{1, 2, \ldots, n\}$

Operações suportadas

MAKE-SETS(n)
: Cria conjuntos singulares com os elementos $\{1, 2, \ldots, n\}$

FIND-SET(i)
: Devolve o representante do conjunto que contém o elemento i

UNION(i, j)
: Reúne os conjuntos a que pertencem os elementos i e j

Também é conhecido como *Union-Find*

Partição

Implementação em vector

MAKE-SETS(n)

```
1 let P[1..n] be a new array
2 for i <- 1 to n do
3   P[i] <- -1      // i is the representative for set {i}
4 return P
```

FIND-SET(P, i)

```
1 while P[i] > 0 do
2       i <- P[i]
3 return i
```

UNION(P, i, j)

```
1 P[FIND-SET(P,j)] <- FIND-SET(P, i)
```

Partição

Implementação em vector

Reunião por tamanho

Se P[i] $= -k$, o conjunto de que i é o representante contém k elementos

UNION-BY-SIZE(P, i, j)

```
1 ri <- FIND-SET(P, i)          // get i's set representative
2 rj <- FIND-SET(P, j)          // get j's set representative
3 if (P[rj] < P[ri])            // j's set is larger than i's
4     P[rj] <- P[rj] + P[ri]
5     P[ri] <- rj
6 else                 // i's is larger or both have the same size
7     P[ri] <- P[ri] + P[rj]
8     P[rj] <- ri
```

Reunião por altura

Se P[i] $= -h$, a árvore do conjunto de que i é o representante tem altura h ou inferior

UNION-BY-RANK(P, i, j)

```
1 ri <- FIND-SET(P, i)
2 rj <- FIND-SET(P, j)
3 if (P[rj] < P[ri])          // j's set tree taller than i's
4     P[ri] <- rj
5 else
6     if (P[rj] == P[ri])     // both heights are the same
7         P[ri] <- P[ri] - 1
8     P[rj] <- ri
```

Partição

Implementação em vector

Compressão de caminho

FIND-SET-WITH-PATH-COMPRESSION(P, i)

```
1 if P[i] < 0 then
2     return i
3 P[i] <- FIND-SET-WITH-PATH-COMPRESSION(P, P[i])
4 return P[i]
```

Análise da complexidade do algoritmo de Kruskal (3)

$$\Theta(E) + |E| \times O(\log V) + f(V, E)$$

$$f(V, E) = \text{MAKE-SETS} + 2 \times |E| \times \text{FIND-SET} + (|V| - 1) \times \text{UNION}$$

Implementação da Partição	Básica	União por tam./altura	+ Compressão de caminho
MAKE-SETS	$O(V)$	$O(V)$	$O((V + E)\,\alpha(V))$ [Tarjan 1975]
$2 \times \|E\| \times$ FIND-SET	$O(EV)$	$O(E \log V)$	
$(\|V\| - 1) \times$ UNION	$O(V^2)$	$O(V \log V)$	
$f(V, E)$	$O(EV)$	$O(E \log V)$	$O(E\,\alpha(V))$
Algoritmo de Kruskal	$O(EV)$	$O(E \log V)$	$O(E \log V)$

$\alpha(n) \leq 4$ para $n < 10^{80}$

Análise da complexidade do algoritmo de Kruskal (4)

$\alpha(n) = \min\{k \mid A_k(1) \geq n\}$

onde

$$A_k(j) = \begin{cases} j+1 & \text{se } k = 0 \\ A_{k-1}^{(j+1)}(j) & \text{se } k \geq 1 \end{cases}$$

$$A_0(1) = 2$$
$$A_1(1) = A_0(A_0(1)) = 3$$
$$A_2(1) = A_1(A_1(1)) = 7$$
$$A_3(1) = 2047$$
$$A_4(1) \gg 2^{2048} \gg 10^{80}$$

Iteração de uma função

$A_{k-1}^{(0)}(j) = j$ e $A_{k-1}^{(i)}(j) = A_{k-1}(A_{k-1}^{(i-1)}(j))$, para $i \geq 1$

Análise amortizada da complexidade

Estudo da complexidade com base no comportamento temporal de uma sequência de operações sobre uma estrutura de dados, no pior caso

Análise amortizada da complexidade

Técnicas

Análise agregada

Se uma sequência de n operações demora tempo $T(n)$, cada operação demora

$$\frac{T(n)}{n}$$

Método da contabilidade

A cada operação é associado um custo, cuja diferença para o custo real da operação pode ser usada como crédito para pagar operações posteriores ou ser abatida ao crédito existente

Método do potencial

...

Método do potencial (1)

O potencial $\Phi(D_i)$ do estado D_i de uma estrutura de dados representa energia que pode ser usada por operações futuras

- D_0 é o estado inicial da estrutura de dados
- D_i é o estado depois da i-ésima operação
- Φ é a função potencial
- $\Phi(D_0)$ é o potencial inicial, em geral 0
- $\Phi(D_i) - \Phi(D_j)$, $i > j$, é a diferença de potencial entre os estados D_j e D_i
- c_i é o custo real da i-ésima operação
- O custo amortizado da i-ésima operação, relativo a Φ, é

$$\widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

Método do potencial (2)

Pretende-se obter um majorante do custo da sequência de operações

$$\begin{aligned}\sum_{i=1}^{n} \widehat{c_i} &= \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0) \\ &\geq \sum_{i=1}^{n} c_i\end{aligned}$$

Logo, a função Φ tem de ser tal que, para qualquer $0 < i \leq n$,

$$\Phi(D_i) \geq \Phi(D_0)$$

Pilha com operação MULTIPOP (1)

Exemplo

Operações PUSH(S,x), POP(S) e STACK-EMPTY(S) com complexidade temporal $O(1)$

MULTIPOP(S, k)

```
1 while not STACK-EMPTY(S) and k > 0 do
2     POP(S)
3     k <- k - 1
```

Custo (real) das operações

PUSH	1
POP	1
MULTIPOP	$\min(\|S\|, k)$

Pilha com operação MULTIPOP (2)

Exemplo

Função potencial

$\Phi(D_i) = s_i$, onde s_i é o número de elementos na pilha

$$\Phi(D_i) = s_i \geq 0 = \Phi(D_0) \quad \text{(pilha inicialmente vazia)}$$

Custo amortizado das operações

$$\begin{aligned} \widehat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= c_i + s_i - s_{i-1} \end{aligned}$$

Pilha com operação MULTIPOP (3)

Exemplo

Custo amortizado das operações

	Custo real (c_i)	Dif. potencial ($s_i - s_{i-1}$)	Custo amortizado ($\widehat{c_i}$)	
PUSH	1	1	2	$O(1)$
POP	1	-1	0	$O(1)$
MULTIPOP	$\min(s_{i-1}, k)$	$-min(s_{i-1}, k)^*$	0	$O(1)$

* Se $s_{i-1} \geq k$, são desempilhados k elementos, se $s_{i-1} < k$, são desempilhados s_{i-1} elementos, donde $s_i = s_{i-1} - \min(s_{i-1}, k)$ e

$$\begin{aligned} \Phi(D_i) - \Phi(D_{i-1}) &= s_i - s_{i-1} \\ &= s_{i-1} - \min(s_{i-1}, k) - s_{i-1} \\ &= -\min(s_{i-1}, k) \end{aligned}$$

Pilha com operação MULTIPOP (4)

Exemplo

Uma sequência de operações

	Operação	Estado			Custo real		Custo amortizado	
i		Pilha	s_i	$s_i - s_{i-1}$	c_i	Total	$\widehat{c_i}$	Total
		$\emptyset$	0			0		0
1	PUSH(a)	[a]	1	1	1	1	2	2
2	PUSH(b)	[a b]	2	1	1	2	2	4
3	PUSH(c)	[a b c]	3	1	1	3	2	6
4	PUSH(d)	[a b c d]	4	1	1	4	2	8
5	MULTIPOP(2)	[a b]	2	-2	2	6	0	8
6	POP()	[a]	1	-1	1	7	0	8
7	MULTIPOP(2)	$\emptyset$	0	-1	1	8	0	8

O custo amortizado total não é inferior ao custo real em nenhum momento

Contador binário (1)

Exemplo

Contador binário com k bits, `C[0..k-1]`

- Bit menos significativo na posição 0
- Operação INCREMENT

INCREMENT(C)

```
1 i <- 0
2 while i < |C| and C[i] = 1 do
3   C[i] <- 0
4   i <- i + 1
5 if i < |C| then
6   C[i] <- 1
```

Bit	3	2	1	0	Oper.	Custo c_i	Total
	0	0	0	0			0
+1	0	0	0	1	1	1	1
+1	0	0	1	0	2	2	3
+1	0	0	1	1	3	1	4
+1	0	1	0	0	4	3	7
+1	0	1	0	1	5	1	8
+1	0	1	1	0	6	2	10

O custo de uma operação é o número de bits que mudam de valor

No pior caso, todos os k bits passam de 1 a 0

Contador binário (2)

Exemplo

Numa sequência de n operações

- O bit 0 muda a cada operação
- O bit 1 muda a cada 2 operações (*i.e.*, em metade)
- O bit 2 muda a cada 4 operações (*i.e.*, num quarto)
- ...
- O bit i muda a cada i operações (*i.e.*, em $n \times 2^{-i}$)

Partindo de todos os bits a 0, o número total de mudanças de valor de um bit é

$$n + \left\lfloor \frac{n}{2} \right\rfloor + \left\lfloor \frac{n}{4} \right\rfloor + \ldots = \sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$$

Contador binário (3)

Exemplo

Sejam C_i o estado do contador a seguir à i-ésima operação e b_i o número de bits a 1 em C_i

Função potencial

$$\Phi(C_i) = b_i$$

$\Phi(C_0) = 0$	Inicialmente, todos os bits estão a 0
$\Phi(C_i) = b_i \geq \Phi(C_0)$	O número de bits a 1 nunca é negativo

Seja t_i o número de bits que mudam de 1 para 0 no i-ésimo incremento

Se $b_{i-1} < k$, então $b_i = b_{i-1} - t_i + 1$ (há um bit que passa de 0 a 1)

Se $b_{i-1} = k$, então $t_i = k$ e $b_i = b_{i-1} - t_i = 0$ (todos os bits passam a 0)

Contador binário (4)

Exemplo

Diferença de potencial

$$\Phi(C_i) - \Phi(C_{i-1}) = b_i - b_{i-1}$$

$$= \begin{cases} b_{i-1} - t_i + 1 - b_{i-1} = 1 - t_i & \text{se } b_{i-1} < k \\ b_{i-1} - t_i - b_{i-1} = -t_i = -k & \text{se } b_{i-1} = k \end{cases}$$

Custo amortizado de INCREMENT

$$\widehat{c_i} = c_i + \Phi(C_i) - \Phi(C_{i-1})$$

	Custo real (c_i)	Dif. potencial $(b_i - b_{i-1})$	Custo amortizado $(\widehat{c_i})$	
$b_{i-1} < k$	$t_i + 1$	$1 - t_i$	2	$O(1)$
$b_{i-1} = k$	k	$-k$	0	$O(1)$

Contador binário (5)

Exemplo

Bit	3	2	1	0	Operação	Custo real c_i	Custo real Total	Custo amortizado $\widehat{c_i}$	Custo amortizado Total
	0	0	0	0			0		0
+1	0	0	0	1	1	1	1	2	2
+1	0	0	1	0	2	2	3	2	4
+1	0	0	1	1	3	1	4	2	6
+1	0	1	0	0	4	3	7	2	8
+1	0	1	0	1	5	1	8	2	10
+1	0	1	1	0	6	2	10	2	12
+1	0	1	1	1	7	1	11	2	14
+1	1	0	0	0	8	4	15	2	16
			...						
+1	1	1	1	1	15	1	26	2	30
+1	0	0	0	0	16	4	30	0	30

O custo amortizado total nunca é inferior ao custo real

Caminho mais curto

Num grafo pesado, com pesos w, o peso do caminho

$$p = v_0\, v_1\, \ldots\, v_k$$

é a soma dos pesos dos arcos que o integram

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

O caminho p é mais curto que o caminho p' se o peso de p é menor que o peso de p'

Cálculo dos caminhos mais curtos

3 algoritmos

1. Cálculo dos caminhos mais curtos num grafo orientado acíclico (DAG), com pesos possivelmente negativos

2. Algoritmo de Dijkstra, só aplicável a grafos sem pesos negativos

3. Algoritmo de Bellman-Ford, aplicável a qualquer grafo pesado

Estes algoritmos calculam os caminhos mais curtos de um nó s para os restantes nós do grafo (*single-source shortest paths*)

Caminhos mais curtos

Funções comuns aos diversos algoritmos

O peso do caminho mais curto de s a qualquer outro nó é inicializado com ∞

INITIALIZE-SINGLE-SOURCE(G, s)

```
1 for each vertex v in G.V do
2     v.d <- INFINITY    // peso do caminho mais curto de s a v
3     v.p <- NIL         // predecessor de v nesse caminho
4 s.d <- 0
```

Se o caminho de s a v, que passa por u e pelo arco (u, v), tem menor peso do que o mais curto anteriormente encontrado, encontrámos um caminho mais curto

RELAX(u, v, w)

```
1 if u.d + w(u,v) < v.d then
2     v.d <- u.d + w(u,v)
3     v.p <- u
```

Caminhos mais curtos a partir de um vértice

Algoritmo para DAGs

$G = (V, E)$ – DAG pesado (pode ter pesos negativos)

DAG-SHORTEST-PATHS(G, w, s)

```
1 topologically sort the vertices of G
2 INITIALIZE-SINGLE-SOURCE(G, s)
3 for each vertex u, taken in topologically sorted
                                                order do
4     for each vertex v in G.adj[u] do
5         RELAX(u, v, w)
```

Caminhos mais curtos a partir de um vértice

Algoritmo de Dijkstra

$G = (V, E)$ – grafo pesado orientado (sem pesos negativos)

DIJKSTRA(G, w, s)

```
1 INITIALIZE-SINGLE-SOURCE(G, s)
2 Q <- G.V                      // fila com prioridade, chave u.d
3 while Q != EMPTY do
4     u <- EXTRACT-MIN(Q)
5     for each vertex v in G.adj[u] do
6         RELAX(u, v, w)                    // pode alterar Q!
```

Quando é encontrado um novo caminho mais curto para um vértice (na função RELAX), é necessário reorganizar a fila `Q` (DECREASE-KEY)

Caminhos mais curtos a partir de um vértice

Algoritmo de Bellman-Ford

$G = (V, E)$ – grafo pesado orientado (pode ter pesos negativos)

BELLMAN-FORD(G, w, s)

```
1 INITIALIZE-SINGLE-SOURCE(G, s)
2 for i <- 1 to |G.V| - 1 do
3     for each edge (u,v) in G.E do
4         RELAX(u, v, w)
5 for each edge (u,v) in G.E do
6     if u.d + w(u,v) < v.d then
7         return FALSE
8 return TRUE
```

Caminhos mais curtos entre cada dois vértices de um grafo

All-pairs shortest paths

Problema

Como calcular os caminhos mais curtos entre cada dois vértices de um grafo pesado (orientado ou não)

Soluções

- Aplicar um dos algoritmos anteriores a partir de cada um dos vértices
- . . . ?

Caminhos mais curtos entre cada dois vértices de um grafo

Algoritmo de Floyd-Warshall

Os vértices intermédios de um caminho simples $v_1\ v_2\ \ldots\ v_l$ são os vértices $\{v_2, \ldots, v_{l-1}\}$

Seja $G = (V, E)$ um grafo pesado, com $V = \{1, 2, \ldots, n\}$

Seja p um caminho mais curto do vértice i para o vértice j, cujos vértices intermédios estão contidos em $\{1, 2, \ldots, k\}$

- Se k não é um nó intermédio de p, os nós intermédios de p estão contidos em $\{1, 2, \ldots, k-1\}$
- Se k é um nó intermédio de p, então p pode decompor-se num caminho p_1 de i para k e num caminho p_2 de k para j
  - Os nós intermédios de p_1 e de p_2 estão contidos em $\{1, 2, \ldots, k-1\}$ (porque p é um caminho simples)
  - p_1 e p_2 são caminhos mais curtos de i para k e de k para j, respectivamente

Caminhos mais curtos entre cada dois vértices de um grafo

Função recursiva

w_{ij}: matriz de adjacências do grafo

$$w_{ij} = \begin{cases} 0 & \text{se} \quad i = j \\ w(i,j) & \text{se} \quad i \neq j \wedge (i,j) \in E \\ \infty & \text{se} \quad i \neq j \wedge (i,j) \notin E \end{cases}$$

$d_{ij}^{(k)}$: peso de um caminho mais curto de i para j com nós intermédios contidos em $\{1, 2, \ldots, k\}$

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{se} \quad k = 0 \\ \min \left\{ d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \right\} & \text{se} \quad k \geq 1 \end{cases}$$

Caminhos mais curtos entre cada dois vértices de um grafo

Cálculo iterativo de $d_{ij}^{(k)}$

FLOYD-WARSHALL-1(w)

```
1 n <- w.rows
2 d^(0) <- w
3 for k <- 1 to n do
4   let d^(k)[1..n,1..n] be a new matrix
5   for i <- 1 to n do
6     for j <- 1 to n do
7       d^(k)[i,j] <-
          min(d^(k-1)[i,j], d^(k-1)[i,k] + d^(k-1)[k,j])
8 return d^(n)
```

Complexidade temporal $\Theta(V^3)$

Complexidade espacial $\Theta(V^3)$

Caminhos mais curtos entre cada dois vértices de um grafo

Cálculo iterativo melhorado

FLOYD-WARSHALL(w)

```
1 n <- w.rows
2 d <- w
3 for k <- 1 to n do
4   for i <- 1 to n do
5     for j <- 1 to n do
6       if d[i,k] + d[k,j] < d[i,j] then
7         d[i,j] <- d[i,k] + d[k,j]
8 return d
```

Complexidade temporal $\Theta(V^3)$

Complexidade espacial $\Theta(V^2)$

Caminhos mais curtos entre cada dois vértices de um grafo

Predecessores

O predecessor de v_j no caminho $q = v_i \ldots v_j$

- Não existe, se $q = v_j$
- É v_i, se $q = v_i\ v_j$
- É o predecessor de v_j no caminho $v_k \ldots v_j$, se

$$q = v_i \ldots v_k \ldots v_j$$

π_{ij}: predecessor de v_j num caminho mais curto de v_i para v_j

$$\pi_{ij} = \begin{cases} \text{NIL} & \text{se } i = j \\ i & \text{se um caminho mais curto de } i \text{ para } j \text{ é } v_i\ v_j \\ \pi_{kj} & \text{se um caminho mais curto de } i \text{ para } j \text{ é } v_i \ldots v_k \ldots v_j \\ \text{NIL} & \text{se } d_{ij} = \infty \end{cases}$$

Caminhos mais curtos entre cada dois vértices de um grafo

Inclusão do cálculo dos predecessores

FLOYD-WARSHALL(w)

```
 1 n <- w.rows
 2 d <- w
 3 let p[1..n,1..n] be a new matrix        // p[i,j] ≡ π_ij
 4 for i <- 1 to n do
 5   for j <- 1 to n do
 6     if i = j or w[i,j] = ∞ then
 7       p[i,j] <- NIL
 8     else
 9       p[i,j] <- i
10 for k <- 1 to n do
11   for i <- 1 to n do
12     for j <- 1 to n do
13       if d[i,k] + d[k,j] < d[i,j] then
14         d[i,j] <- d[i,k] + d[k,j]
15         p[i,j] <- p[k,j]
16 return d and p
```

Caminho mais curto entre dois vértices

Reconstrução do caminho

PRINT-ALL-PAIRS-SHORTEST-PATH(p, i, j)

Exercício

Complexidade dos algoritmos

$G = (V, E)$

	Compl. Temporal
Percurso em largura	$O(V + E)$
Percurso em profundidade	$\Theta(V + E)$
Ordenação topológica (ambos os algoritmos)	$\Theta(V + E)$
Grafo transposto	$\Theta(V + E)$
Cálculo das componentes fortemente conexas	$\Theta(V + E)$
Algoritmos de Prim e de Kruskal	$O(E \log V)$
Caminhos mais curtos num DAG	$\Theta(V + E)$
Algoritmo de Dijkstra	$O(E \log V)$
Algoritmo de Bellman-Ford	$O(VE)$
Algoritmo de Floyd-Warshall	$\Theta(V^3)$

Pressupostos

Grafo representado através de listas de adjacências (excepto algoritmos de Kruskal, de Bellman-Ford e de Floyd-Warshall)

Algoritmos de Prim e de Dijkstra recorrem a uma fila tipo *heap* binário com actualização (EXTRACT-MIN e DECREASE-KEY com complexidade temporal logarítmica no número de elementos da fila)

Algoritmo de Kruskal usa Partição com compressão de caminho

Problema do fluxo máximo

Redes de fluxos (1)

Modelam redes de ligações, por onde flui algo:

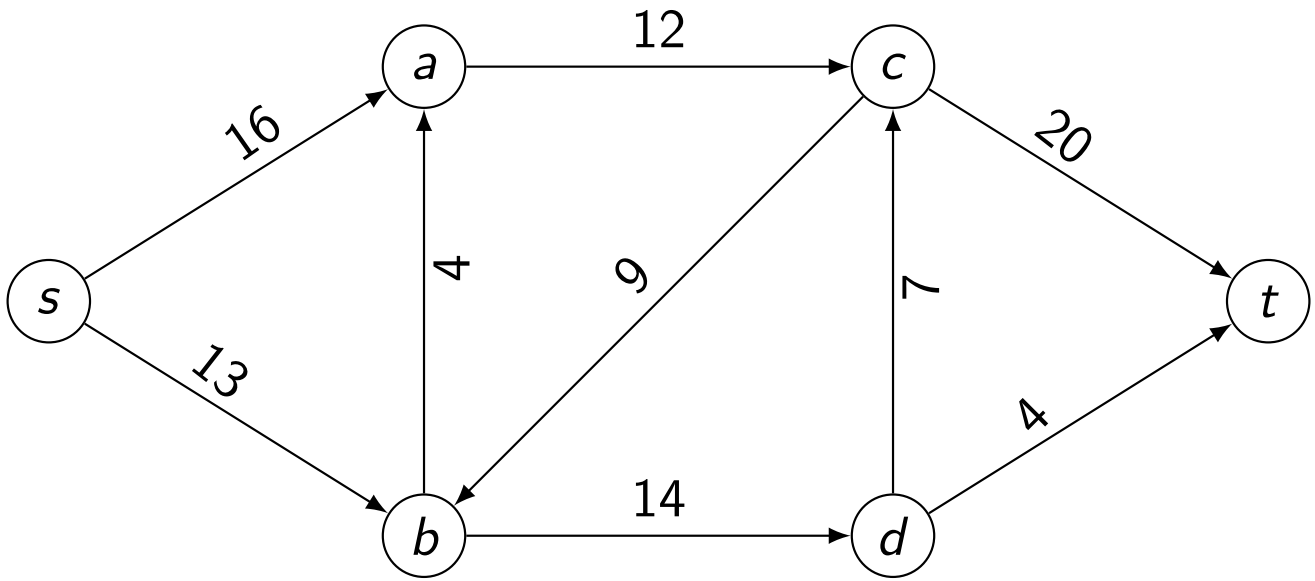
- Líquidos
- Gases
- Trânsito automóvel
- Comunicações
- ...

Cada ligação liga dois pontos da rede, tem uma direcção e uma capacidade

Em cada rede de fluxos existem dois pontos especiais:

- Fonte (*source*) Origem de tudo o que flui na rede
- Dreno (*sink*) Destino final de tudo o que flui na rede

Redes de fluxos (2)



c é a capacidade da ligação (u, v)

Redes de fluxos (3)

Rede de fluxos (*Flow network*)

- Modelada através de um grafo orientado $G = (V, E)$
- $c(u, v) > 0$ é a capacidade do arco (u, v)
- $s \in V$ é a fonte (*source*) da rede
- $t \in V$ é o dreno (*sink*) da rede ($s \neq t$)
- Se $(u, v) \in E$, então $(v, u) \notin E$
- Assume-se que, qualquer que seja o vértice $v \in V$, existe um caminho $s \ldots v \ldots t$

  (Logo, $|E| \geq |V| - 1$)

Fluxos (1)

Fluxo

- Um fluxo numa rede de fluxos é uma função $f : V \times V \to \mathbb{R}$, que satisfaz:

  Capacidade O fluxo que passa numa ligação não pode exceder a sua capacidade

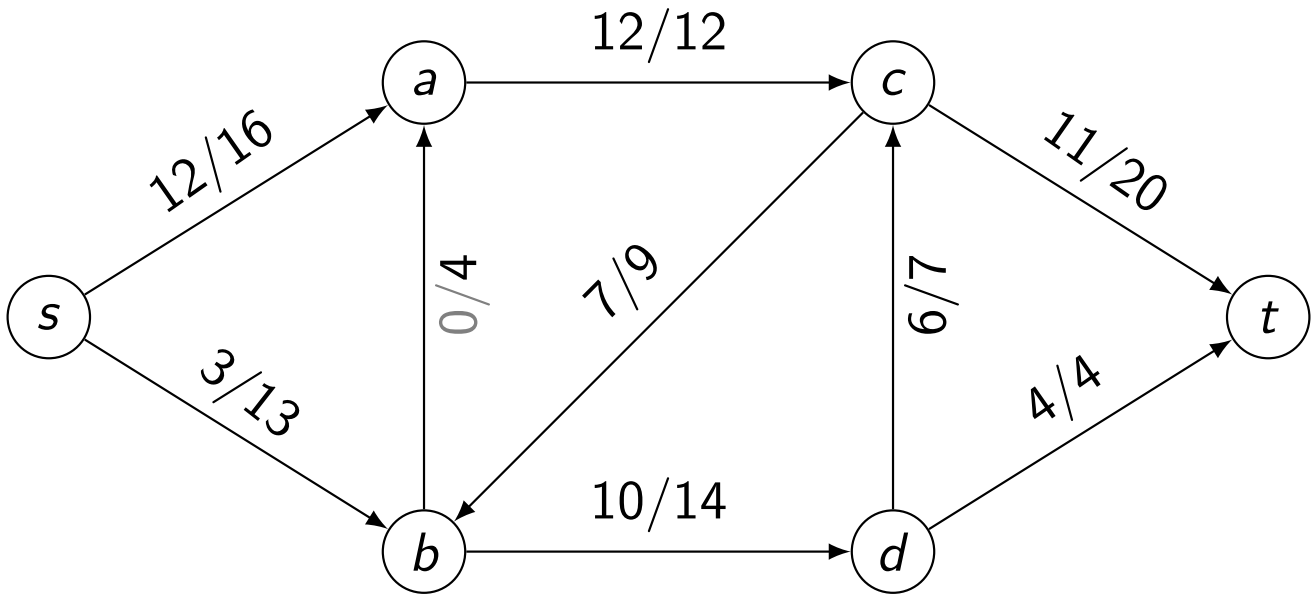
  $$0 \leq f(u, v) \leq c(u, v)$$

  Conservação do fluxo O fluxo que entra num vértice (diferente de s e de t) é o fluxo que sai do vértice

  $$\forall u \in V \setminus \{s, t\}, \quad \sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$$

- $(u, v) \notin E \to f(u, v) = 0$

Fluxos (2)



f é o fluxo que passa pela ligação (u, v), com capacidade c

NOTA: Quando f é 0, por vezes, omite-se o '0/'

Fluxos (3)

Valor do fluxo

O valor de um fluxo é o fluxo produzido pela fonte s

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$$

Rede residual (1)

Dado um fluxo f, a capacidade residual da rede $G = (V, E)$ é

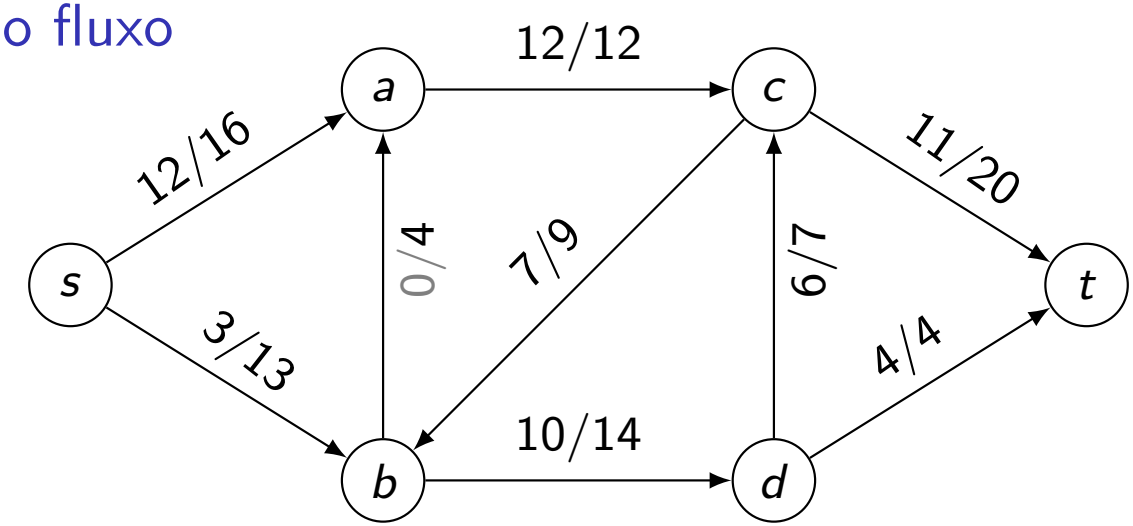
$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v) & \text{se } (u, v) \in E \\ f(v, u) & \text{se } (v, u) \in E \\ 0 & \text{caso contrário} \end{cases}$$

A rede residual resultante é $G_f = (V, E')$, com

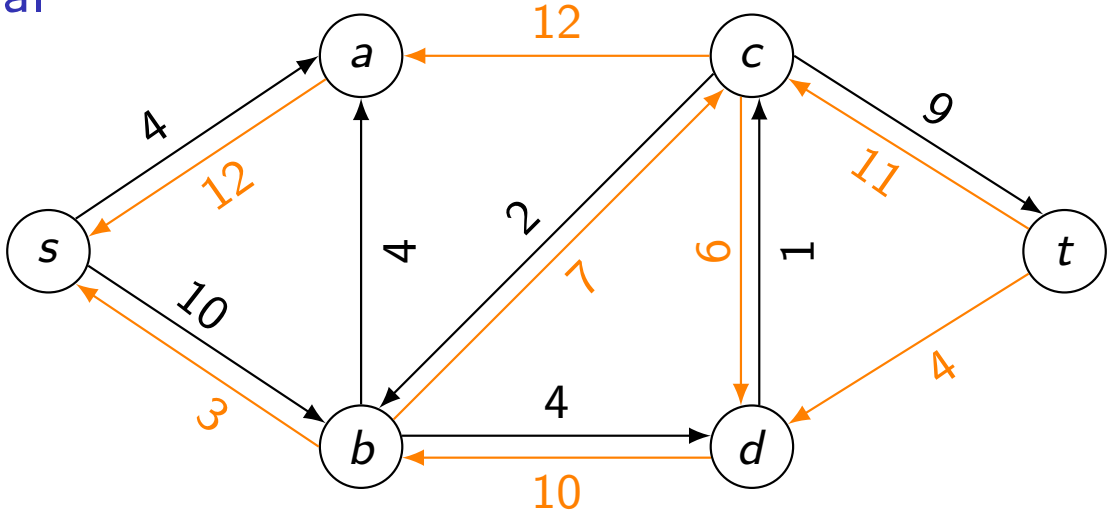
$$E' = \{ (u, v) \mid c_f(u, v) > 0 \}$$

Rede residual (2)

Rede com o fluxo



Rede residual



Rede residual (3)

Numa rede residual

- A capacidade dos arcos comuns à rede original corresponde à capacidade não utilizada pelo fluxo
- A capacidade dos arcos com orientação oposta à dos da rede original corresponde à quantidade de fluxo que pode ser cancelada

Uma rede residual indica os limites das alterações que podem ser feitas a um fluxo

Problema do fluxo máximo

Dada uma rede de fluxos, qual é o valor máximo de um fluxo?

Incremento de um fluxo (1)

Seja G_f uma rede residual e seja $p = v_1\, v_2\, \ldots\, v_k$, com $v_1 = s$ e $v_k = t$, um caminho simples em G_f, da fonte s para o dreno t

A capacidade residual de p é

$$c_f(p) = \min_{1 \leq i < k} \left\{ c_f(v_i, v_{i+1}) \right\} > 0$$

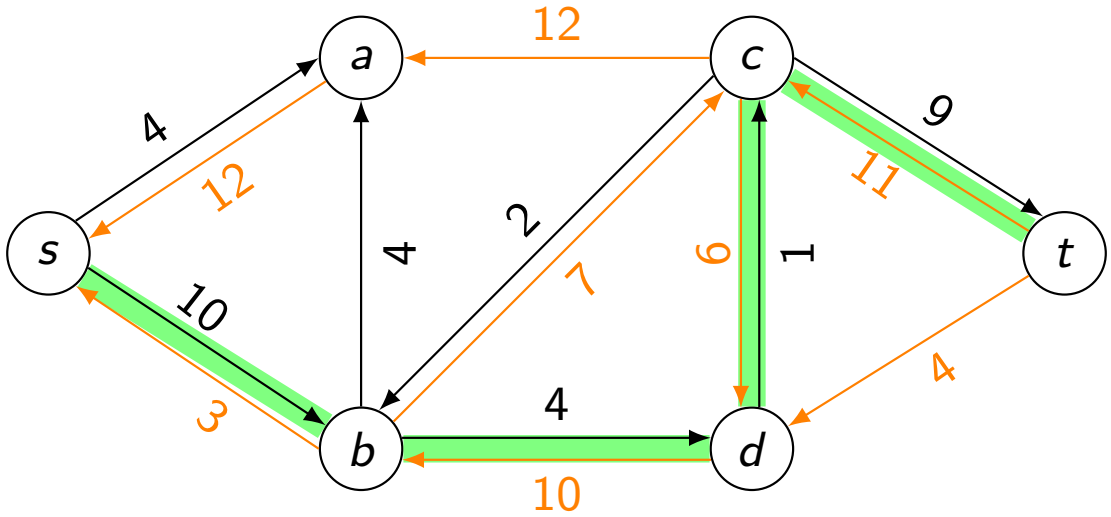
O fluxo aumentado por p é

$$f'(u,v) = \begin{cases} f(u,v) + c_f(p) & \text{se } (u,v) \in E \text{ está em } p \\ f(u,v) - c_f(p) & \text{se } (v,u) \in E \text{ está em } p \\ f(u,v) & \text{caso contrário} \end{cases}$$

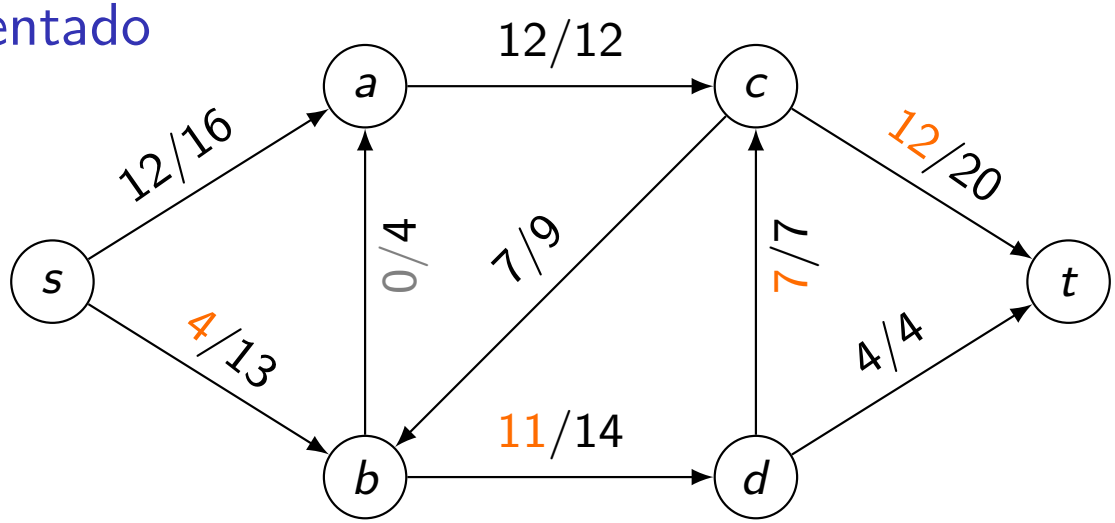
O valor de f' é $|f'| = |f| + c_f(p)$

Incremento de um fluxo (2)

Caminho $s...t$ na rede residual



Fluxo aumentado



Incremento de um fluxo (3)

Numa rede residual

- A capacidade dos arcos comuns à rede original corresponde à capacidade não utilizada pelo fluxo
- A capacidade dos arcos com orientação oposta à dos da rede original corresponde à quantidade de fluxo que pode ser cancelada

Uma rede residual indica os limites das alterações que podem ser feitas a um fluxo

Método de Ford-Fulkerson

1. Inicializar $f(u, v) = 0$, para todo $(u, v) \in E$
2. Enquanto houver um caminho simples $p = s \ldots t$ na rede residual
   a. Seja $c_f(p) = \min \{ c_f(u, v) \mid (u, v) \text{ está em } p \}$
   b. Para cada arco (u, v) em p
      - Se $(u, v) \in E$, o fluxo no arco (u, v) é aumentado em $c_f(p)$ unidades

$$f(u, v) = f(u, v) + c_f(p)$$

      - Senão, então $(v, u) \in E$ e são canceladas $c_f(p)$ unidades de fluxo no arco (v, u)

$$f(v, u) = f(v, u) - c_f(p)$$

Algoritmo de Edmonds-Karp

EDMONDS-KARP(G, s, t)

```
 1  for each edge (u,v) in G.E do
 2      (u,v).f <- 0                    // fluxo f(u,v) = 0
 3  Gf <- RESIDUAL-NET(G)
 4  while (cf <- BFS-FIND-PATH(Gf, s, t)) > 0 do
 5      v <- t
 6      while v.p != NIL do
 7          if edge (v.p,v) is in G.E then
 8              (v.p,v).f = (v.p,v).f + cf
 9          else // edge (v,v.p) is in G.E
10              (v,v.p).f = (v,v.p).f - cf
11          v <- v.p
12      UPDATE(Gf, G)
```

Complexidade temporal $O(VE^2)$

Complexidade temporal do algoritmo de Edmonds-Karp

Grafo representado através de listas de adjacências

Linhas

1–2 Ciclo executado $|E|$ vezes

3 Construção da rede residual: $\Theta(V + E)$

4–12 Ciclo executado $O(VE)$ vezes

4 Percurso em largura no grafo: $O(V + E)$

6–11 Ciclo executado $O(V)$ vezes

12 Actualização da rede residual: $O(V)$

Complexidade temporal do algoritmo

$$\Theta(E) + \Theta(V + E) + O(VE(V + E)) = O(VE^2)$$

($\forall_{v \in V}$, existe um caminho $s \ldots v \ldots t$, pelo que $|E| \geq |V| - 1$)

Restantes operações com complexidade temporal constante

Cortes (1)

Um corte (*cut*) (S, T), numa rede de fluxos $G = (V, E)$, é uma partição tal que

- $s \in S$
- $t \in T$
- $T = V - S$

A capacidade do corte (S, T) é soma das capacidades das ligações que o atravessam, de S para T

$$c(S, T) = \sum_{u \in S} \sum_{v \in T} c(u, v)$$

Considera-se que $c(u, v) = 0$, se $(u, v) \notin E$

Cortes (2)

Dado um fluxo f, o fluxo (líquido) que atravessa o corte (S, T) é a diferença entre o fluxo que o atravessa de S para T e o que o atravessa de T para S

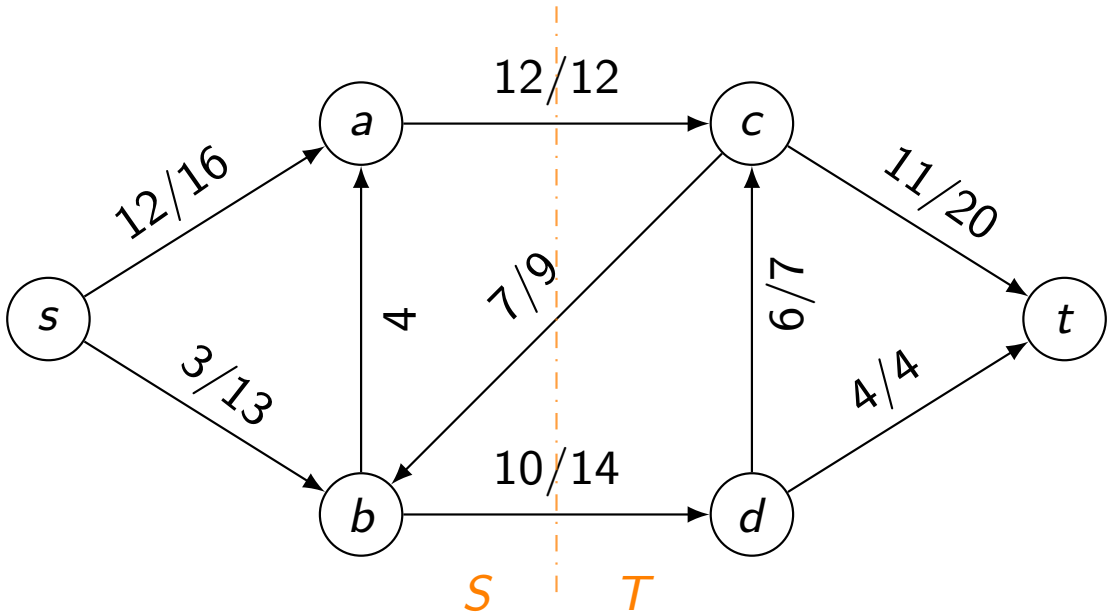
$$f(S,T) = \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u)$$

O fluxo (líquido) que atravessa o corte (S, T) não pode ser superior à capacidade do corte

$$f(S,T) \leq c(S,T)$$

Cortes (3)

Corte (S, T) numa rede de fluxos



$S = \{s, a, b\}, T = \{t, c, d\}$

Capacidade do corte: $c(S, T) = 12 + 14 = 26$

Fluxo que atravessa o corte: $f(S, T) = 12 + 10 - 7 = 15$

Corte mínimo

Um corte é mínimo se não existe nenhum corte com capacidade inferior

Nenhum fluxo pode ter um valor superior à capacidade de um corte mínimo

Teorema do fluxo-máximo corte-mínimo (*Max-flow min-cut theorem*)

Seja $G = (V, E)$ uma rede de fluxos e f um fluxo em G

f é um fluxo máximo sse existe um corte (S, T) de G tal que

$$|f| = c(S, T)$$

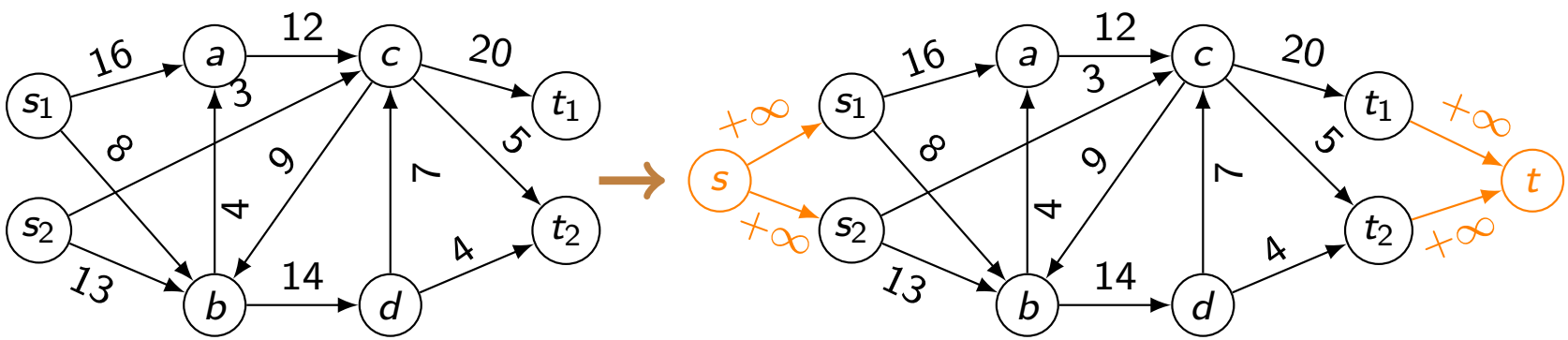
Variações (1)

Arcos anti-paralelos

Podem eliminar-se acrescentado um novo vértice intermédio

Múltiplas fontes e/ou múltiplos drenos

Acrescenta-se uma nova fonte s, ligada às anteriores por arcos de capacidade $+\infty$, e/ou um novo dreno t, a que os anteriores são ligados por arcos de capacidade $+\infty$



Variações (2)

Vértices com capacidade

Divide-se o vértice u em dois

1. O vértice de entrada u_e, que será o destino de todos os arcos que tinham destino u
2. O vértice de saída u_s, que será a origem de todos os arcos que tinham origem u

Acrescenta-se o arco (u_e, u_s), com a capacidade de u