

Universidade do Minho

Notas Algoritmos e Complexidade

Jorge Sousa Pinto
2020

C1. Introdução à Análise de Correcção dos Algoritmos

Correcção de um algoritmo

*Um algoritmo diz-se **correcto** se para todos os valores dos inputs (variáveis de entrada) ele pára com os valores esperados (i.e. correctos...) dos outputs (variáveis de saída). Neste caso diz-se que ele **resolve** o problema computacional em questão.*

Nem sempre a incorrecção é um motivo para a inutilidade de um algoritmo:

- Em certas aplicações basta que um algoritmo funcione correctamente para *alguns dos seus inputs*.
- Em problemas muito difíceis, poderá ser suficiente obter *soluções aproximadas* para o problema.

A análise da correcção de um algoritmo pretende determinar se ele é correcto, e em que condições.

A demonstração da correcção de um algoritmo cuja estrutura não apresente fluxo de controlo pode ser efectuada por simples inspecção. Exemplo:

```
int soma(int a, int b) {  
    int sum = a+b;  
    return sum;  
}
```

Em alguns casos a correcção advém da própria especificação. Considere-se por exemplo uma implementação recursiva da noção de *factorial* de um número. A implementação segue de perto a definição, pelo que a sua correcção é imediata —

uma vez que a própria definição é algorítmica, trata-se apenas de verificar se a sua codificação na linguagem de programação escolhida é correcta.

```
int factorial(int n) {  
    int f;  
    if (n<1) f = 1;  
    else f = n*factorial(n-1);  
    return f;  
}
```

No entanto, no caso geral esta análise poderá apresentar uma dificuldade muito elevada e deve por isso ser efectuada com algum grau de formalismo, possivelmente recorrendo a uma *lógica de programas*.

Especificação de programas

Pré-condições e pós-condições

A análise de correcção dos algoritmos baseia-se na utilização de *asserções*: proposições lógicas sobre o estado actual do programa (o conjunto das suas variáveis). Por exemplo,

- $x > 0$
- $a[i] < a[j]$
- $\forall i. 0 \leq i < n \Rightarrow a[i] < 1000$

Pré-condição:

É uma propriedade que se assume como verdadeira no estado inicial de execução do programa, i.e., só interessa considerar as execuções do programa que satisfaçam esta condição.

Pós-condição:

É uma propriedade que se deseja provar verdadeira no estado final de execução do programa.

Triplos de Hoare

Um triplo de Hoare escreve-se como $\{P\} C \{Q\}$, em que

- C é o programa cuja correcção se considera
- P é uma *pré-condição* e Q é uma *pós-condição*

O triplo $\{P\} C \{Q\}$ é *válido* quando todas as execuções de C partindo de estados iniciais que satisfaçam P , caso *terminem*, resultem num estado final do programa que satisfaz Q .

Exemplo

O triplo $\{x = 10 \wedge y = 20\} \text{ swap } \{y = 10 \wedge x = 20\}$ associa a um programa **swap** uma especificação que exprime o seguinte:

se as variáveis x e y tiverem inicialmente os valores 10 e 20 respectivamente, então no final da execução estes valores terão sido trocados.

Claramente, não é a especificação que esperamos de um típico programa **swap**, que deverá trocar os valores das duas variáveis *qualsquer que sejam esses valores iniciais*.

Para escrever uma especificação adequada necessitaremos de recorrer a *variáveis auxiliares*. Veja-se a seguinte versão:

$\{x = x_0 \wedge y = y_0\} \text{ swap } \{y = x_0 \wedge x = y_0\}$

As variáveis x_0, y_0 , ditas auxiliares, não podem ser utilizadas no programa **swap** — apenas podem ocorrer na especificação. É assim garantido que os seus valores não serão alterados durante a execução do programa, e por isso o valor de x_0 (resp. y_0) no final da execução é igual ao valor inicial da variável x (resp. y). Sendo assim, a especificação acima realmente capta a troca dos valores, tal como desejado.

Note-se que, depois de escrito o programa **swap**, ele terá que ser *mostrado correcto* (ou *verificado*) em ordem a esta especificação, o que estudaremos nos próximos módulos.

Exercício

Traduza por palavras suas o significado de cada uma das seguintes especificações para um programa C . Excepto quando indicado, considere que todas as variáveis são de tipo inteiro.

$$1. \{x \geq 0 \wedge y > 0\} \ C \ \{0 \leq r < y \wedge q * y + r = x\}$$

Esta especificação admite como solução programas triviais, que alteram os valores dos inputs. Por forma a ser possível referir na pós-condição os *valores iniciais* das variáveis, é necessário utilizar *variáveis auxiliares* como no exemplo seguinte.

$$2. \{x = x_0 \geq 0 \wedge y = y_0 > 0\} \ C \ \{0 \leq r < y_0 \wedge q * y_0 + r = x_0\}$$

As variáveis x_0, y_0 são auxiliares, não podendo ser utilizadas no programa C .

$$3. \{x = x_0 \geq 0 \wedge y = y_0 > 0\} \ C \ \{0 \leq r < y_0 \wedge (\exists_{q \geq 0}. q * y_0 + r = x_0)\}$$

$$4. \{x = x_0 \wedge y = y_0\} \ C \ \{(x = x_0 \vee x = y_0) \wedge x \geq x_0 \wedge x \geq y_0\}$$

$$5. \{x = x_0 \geq 0 \wedge e = e_0 > 0\} \ C \ \{|r * r - x_0| < e_0\}$$

(x, e, r variáveis de vírgula flutuante)

$$6. \{\forall_{0 \leq i < N}. A[i] = a_i\} \ C \ \{0 \leq p < N \wedge (\forall_{0 \leq i < N}. A[i] = a_i \wedge a_p \leq a_i)\}$$

(A um array de tipo inteiro)

Exercício

De forma inversa, escreva agora especificações formais (pré-condições e pós-condições) correspondentes às seguintes especificações informais:

1. Um programa que coloca na variável r a soma dos valores (iniciais) das variáveis x e y .
2. Um programa que, para valores não negativos da variável y , coloca na variável r o produto dos valores (iniciais) das variáveis x e y .
3. Um programa que, para valores não negativos da variável y , coloca na variável r o produto dos valores (iniciais) das variáveis x e y , sem alterar os valores dessas variáveis.
4. Um programa que coloca na variável r o mínimo múltiplo comum das variáveis x e y .
5. Um programa que recebe como input dois arrays A e B com N elementos, e calcula o comprimento do *prefixo mais longo* que os dois têm em comum.
6. Um programa que procura uma ocorrência de k entre os índices a e b de um array A , colocando o índice dessa ocorrência na variável r . Caso $k \notin A[a...b]$, r tomará o valor -1.
7. Um programa que soma todos os elementos de um array com N posições, guardando o resultado na variável $result$.

Exercício

Pronuncie-se sobre a validade dos seguintes triplos de Hoare. Corrija os triplos inválidos, modificando a *pós-condição* por forma a ser tão informativa quanto possível.

$$1. \{j = a\} \quad j := j + 1 \quad \{a = j + 1\}$$

$$2. \{i = j\} \quad i := j + i \quad \{i > j\}$$

3. $\{i = i_0\} \ j := i + 1; \ i := j + 1 \ \{i = i_0 + 1\}$
4. $\{i \neq j\} \text{ if } (i > j) \text{ then } m := i - j \text{ else } m := j - i \ \{m > 0\}$
5. $\{x = b\} \text{ while } (x > a) \ x := x - 1 \ \{x = a\}$

Prova de Correcção de Programas Sem Ciclos

No caso dos programas sem construções iterativas ou recursivas, é fácil determinar uma *condição de verificação*: uma fórmula lógica (de primeira ordem) tal que, se essa fórmula for válida, então o triplo será também ele de certeza válido.

Vejamos como exemplo o seguinte triplo:

$$\{i = i_0\} \ j := i + 1; \ i := j + 1 \ \{i = i_0 + 1\}$$

Calculamos a condição de verificação *propagando para trás a pós-condição*. Para isso questionamos: que condição terá de ser verdade no estado intermédio, entre as duas instruções, para que o triplo seja válido?

Para que seja $i = i_0 + 1$ depois da execução de $i := j + 1$, terá de ser verdade, antes da execução desta instrução, a condição $i = i_0 + 1[j + 1/i]$, ou seja, a pós-condição em que substituímos a variável atribuída pela expressão que lhe foi atribuída, resultando em $j + 1 = i_0 + 1$. Tecnicamente estamos a calcular uma noção conhecida pela *pré-condição mais fraca* da instrução $i := j + 1$ em ordem à condição $i = i_0 + 1$.

Podemos representar isto da seguinte forma:

$$\begin{aligned} \{\mathbf{i} = \mathbf{i}_0\} \\ j := i + 1 \end{aligned}$$

$\{j + 1 = i_0 + 1\}$

$i := j + 1$

$\{\mathbf{i} = \mathbf{i}_0 + \mathbf{1}\}$

E questionamos novamente: que condição terá de ser verdade no estado inicial do programa, entre as duas instruções, para que o triplo seja válido? Ou por outras palavras, qual será a pré-condição mais fraca de $j := i + 1$ em ordem à condição intermédia $\{j + 1 = i_0 + 1\}$?

Basta agora substituir, em $j + 1 = i_0 + 1$, a variável j pela expressão $i + 1$, obtendo-se $j + 1 = i_0 + 1[i + 1/j] \equiv i + 1 + 1 = i_0 + 1$. Representamos:

$\{\mathbf{i} = \mathbf{i}_0\}$

$\{i + 1 + 1 = i_0 + 1\}$

$j := i + 1$

$\{j + 1 = i_0 + 1\}$

$i := j + 1$

$\{\mathbf{i} = \mathbf{i}_0 + \mathbf{1}\}$

Note-se que não é possível propagar mais a condição, uma vez que o estado inicial foi já alcançado. A *condição de verificação* será então a seguinte implicação:

$i = i_0 \rightarrow i + 1 + 1 = i_0 + 1$, que é claramente inválida.

A justificação para o cálculo da condição de verificação é simples: se a pré-condição pode ser assumida no estado inicial, e foi calculada uma condição suficiente e necessária (por ser a mais fraca) que deve ser verdade no início do programa para que o triplo seja válido, então a pré-condição terá de ser mais forte do que a condição propagada.

Observe-se que se a pós-condição fosse $i = i_0 + 2$ a condição de verificação seria $i = i_0 \rightarrow i + 1 + 1 = i_0 + 2$, uma condição válida (e claro, também o triplo seria válido).

Execução Condicional

O cálculo da pré-condição mais fraca de um condicional exige um pouco mais de cuidado. Dado o comando **if** (b) C_1 **else** C_2 e uma pós-condição ψ , que condição terá de ser verdade no estado inicial para que a pós-condição seja verdade no estado final?

Começamos por calcular as pré-condições mais fracas ϕ_1 e ϕ_2 de cada um dos ramos do condicional em ordem a ψ . A pré-condição do condicional será então $(b \rightarrow \phi_1) \wedge (\neg b \rightarrow \phi_2)$.

Exercício

Prove a validade de cada um dos seguintes triplos de Hoare, começando por gerar as respectivas condições de verificação.

1. $\{i > j\} \quad j := i + 1; \quad i := j + 1 \quad \{i > j\}$
2. $\{s = 2^i\} \quad i := i + 1; \quad s := s * 2 \quad \{s = 2^i\}$
3. $\{\text{True}\} \quad \text{if } (i < j) \quad m := i \quad \text{else } m := j \quad \{m \leq i \wedge m \leq j\}$
4. $\{i \neq j\} \quad \text{if } (i > j) \quad m := i - j \quad \text{else } m := j - i \quad \{m > 0\}$

C2. Invariante de Ciclo

Notas prévias

Neste módulo assumiremos que na linguagem de programação utilizada não é possível atribuir valores às variáveis que são inputs do programa. Isto permitirá simplificar as respectivas especificações (porque torna desnecessária a utilização de variáveis auxiliares).

Na linguagem de programação da ferramenta [Dafny](#), que utilizaremos neste módulo para testar os invariantes de ciclo, não é possível atribuir valor a parâmetros.

A linguagem Dafny incorpora a **sintaxe** habitual de expressões da linguagem C (`==` como operador de comparação de igualdade, `&&` e `||` como operadores Booleanos, etc. No entanto, para a atribuição é utilizado o operador `:=`, muito comum em linguagens “brinquedo”.

Invariante de Ciclo

Estabelecer a correcção de algoritmos que incluem ciclos implica considerar qualquer número de iterações; no entanto, não é viável proceder a esta análise por casos, de forma exaustiva.

Recorremos por isso à noção de *invariante de ciclo* — uma propriedade (fórmula de primeira ordem) que se mantém verdadeira em todas as iterações, e que reflecte as transformações de estado efectuadas durante a execução do ciclo.

A ideia é que se o invariante se mantém verdadeiro ao longo da execução, ele será ainda verdadeiro à saída do ciclo, e deverá ser suficientemente forte para permitir provar a pós-condição desejada para o ciclo.

Raciocinar com um invariante de ciclo corresponde à ideia de prova indutiva no número de iterações de uma execução (que termina) do ciclo. Assim, para provar

que uma fórmula I é um invariante, devemos

1. **Inicialização:** Mostrar que I é verdade à entrada do ciclo (i.e. antes de se iniciar a primeira iteração) — *caso de base*
2. **Preservação:** Mostrar que, assumindo que I é verdade no início de uma iteração arbitrária (i.e. assumindo que a condição do ciclo é satisfeita), então será satisfeita no final dessa iteração — *caso induutivo*

Sendo I de facto um invariante, é ainda preciso mostrar que é útil:

3. **Utilidade:** Mostrar que o invariante I , juntamente com a negação da condição do ciclo (uma vez que terminou a sua execução), implica a verdade da pós-condição

Estas 3 propriedades podem por sua vez ser expressas por triplos de Hoare, envolvendo:

1. para a inicialização, o código que antecede o ciclo
2. para a preservação, o código que constitui o corpo do ciclo
3. para a utilidade, o código que sucede ao ciclo

Exemplo: Divisão Inteira

Especificação de um programa que calcula a divisão de x por y , colocando o quociente em q e o resto em r :

$$\{x \geq 0 \wedge y > 0\} \text{ divide } \{0 \leq r < y \wedge q * y + r = x\}$$

(especificação simplificada sem vars. auxiliares)

O seguinte programa está de acordo com esta especificação. Informalmente, conta “o número de vezes que y cabe em x ”.

```

r := x;
q := 0;
while (y <= r) {
    r := r-y;
    q := q+1;
}

```

Simulemos uma execução deste programa para $x = 14$ e $y = 3$, escrevendo o valor das variáveis r e q , alteradas pelo programa, à **entrada de cada iteração** do ciclo:

r	q	$q * y + r$
14	0	14
11	1	14
8	2	14
5	3	14
2	4	14

Constata-se que a expressão $q * y + r$ mantém o seu valor ao longo da execução, à entrada de cada iteração, e que este valor é igual a ao valor de x .

Por outro lado, o valor de r é não-negativo ao longo de toda a execução, pelo que temos o seguinte invariante de ciclo:

$$I \equiv \mathbf{0} \leq r \wedge q * y + r = x$$

Este exemplo encaixa num cenário típico bastante simples para o caso de programas que terminam com um ciclo:

A pós-condição é equivalente à conjunção do invariante e da negação da condição do ciclo.

Vejamos como estabelecer a correcção do programa com este invariante:

- *Inicialização do invariante.* Corresponde ao triplo de Hoare:
 $\{x \geq 0 \wedge y > 0\} \quad r := x; \quad q := 0; \quad \{I\}$

É imediato constatar que é válido, uma vez que $0 \leq 0 \wedge 0 * y + x = x$

- *Preservação do invariante.* Corresponde ao triplo de Hoare:
 $\{I \wedge y \leq r\} \quad r := r - y; \quad q := q + 1; \quad \{I\}$

Intuitivamente, admitimos que $I \equiv 0 \leq r \wedge q * y + r = x$ é verdade à entrada de uma iteração qualquer. Temos também $y \leq r$. Queremos mostrar que I voltará a ser verdade depois da execução da iteração (arbitrária) em causa.

Ora, os valores de r e q no início da próxima iteração serão respectivamente dados por $r - y$ e $q + 1$. Queremos então mostrar que a seguinte condição é verdadeira:

$$\begin{aligned} & 0 \leq (r - y) \wedge (q + 1) * y + (r - y) = x \\ & \equiv y \leq r \wedge q * y + r = x \end{aligned}$$

E efectivamente,

$$(0 \leq r \wedge q * y + r = x) \wedge y \leq r \rightarrow y \leq r \wedge q * y + r = x$$

Ou seja, o invariante é preservado por qualquer iteração do ciclo.

- *Utilidade do invariante.* Corresponde ao triplo de Hoare:

$$\{I \wedge \neg(y \leq r)\} \quad \{\} \quad \{0 \leq r < y \wedge q * y + r = x\}$$

A utilidade é neste caso trivial, uma vez que o programa termina com o ciclo, não sendo executadas quaisquer instruções subsequentes. Ora,

$I \wedge \neg(y \leq r)$ implica (neste caso é mesmo equivalente!) a pós-condição

$$0 \leq r < y \wedge q * y + r = x$$

Observe-se que para cada um dos triplos acima pode ser gerada uma *condição de verificação* usando a técnica descrita em [+C1. Introdução à Análise de Correcção – Especificação de Programas](#).

Verificação com a Ferramenta Dafny

Por forma a confirmar se um invariante serve ou não o nosso propósito, podemos recorrer a uma ferramenta automática de verificação de programas. A ferramenta Dafny, desenvolvida pela Microsoft Research, é um verificador de programas escritos numa linguagem simples como a que temos considerado acima. A unidade básica (rotina) de código é aqui o *método*.

Note-se que:

- O programa de divisão dado acima será escrito como um método tendo x e y como inputs, e q e r como outputs. Todas estas variáveis são de tipo nat
- A especificação deste método será escrita utilizando as palavras reservadas *requires* para a pré-condição, e *ensures* para a pós-condição, logo a seguir ao cabeçalho do método
- O invariante de ciclo será escrito logo a seguir à condição do ciclo, usando a palavra reservada *invariant*

Temos assim:

```
method divide(x: nat, y: nat) returns (r: nat, q:nat)
    requires y > 0
    ensures 0 <= r < y
    ensures q*y+r == x
{
    r := x;
    q := 0;
    while (y <= r)
        invariant q*y+r == x
    {
        r := r-y;
        q := q+1;
    }
}
```

O programa poderá ser verificado online [aqui](#). Obtemos como resultado:

```
Dafny 2.2.0.10923
Dafny program verifier finished with 1 verified, 0 errors
```

[[Permalink](#) para este exemplo em [rise4fun.com](#)]

Exemplo: Factorial

Vejamos como verificar a correcção de um programa de cálculo da noção de factorial de um número natural n . Isto corresponderá a provar a validade do seguinte triplo de Hoare:

$\{n \geq 0\}$ **ComputeFact** $\{f = n!\}$

O núcleo do programa é o seguinte:

```
f := 1;  
i := 1;  
while (i<=n) {  
    f := f*i;  
    i := i+1;  
}
```

O invariante I deverá ser tal que os triplos de Hoare seguintes sejam válidos:

1. **Inicialização:** $\{n \geq 0\} \quad f := 1; i := 1; \{I\}$
2. **Preservação:** $\{I \wedge i \leq n\} \quad f := f * i; i := i + 1; \{I\}$
3. **Utilidade:** $\{I \wedge \neg(i \leq n)\} \quad \{f = n!\}$

Para caracterizar o invariante apropriado, simulemos uma execução deste programa.

O par de variáveis (i, f) toma sucessivamente os seguintes valores:

$(1, 1), (2, 1), (3, 2), (4, 6), (5, 24), \dots$

É fácil observar que o valor de f à entrada de uma iteração corresponde ao factorial de $i - 1$.

É imediato ver que:

- este invariante é bem inicializado, uma vez que $0! = 1$
- ele é também preservado, uma vez que o corpo do ciclo multiplica f por i e incrementa esta variável

Notemos também que o valor de i varia entre 0 e $n + 1$. Esta informação deverá constar do invariante.

À saída do ciclo, sendo a condição $i \leq n$ falsa, teremos então que o valor final de i será $i = n + 1$, e o invariante implicará que $f = n!$ como desejado (*utilidade* do invariante para provar a pós-condição).

Vejamos agora como efectuar esta verificação em Dafny. As seguintes definições incorporam já a especificação e invariante discutidos:

```
function fact(n: int): int
    requires n >= 0
{
    if n == 0 then 1
    else n * fact(n-1)
}

method ComputeFact (n: int) returns (f: int)
    requires n >= 0
    ensures f == fact(n)
{
    f := 1;
    var i := 1;
    while (i<=n)
        invariant i <= n+1
        invariant f == fact(i-1)
    {
}
```

```
    f := f*i;
    i := i+1;
}
}
```

Note-se a presença de uma *função lógica*, recursiva, que capta a definição matemática de factorial, que não existe à partida na teoria do Dafny. Esta função *fact* é depois usada na especificação do programa, quer na pós-condição quer no invariante de ciclo.

Observe-se também que as variáveis locais que não sejam parâmetros ou outputs de um método devem ser declaradas com a palavra reservada `var`, como `i` no exemplo acima.

A verificação é levada a cabo com sucesso, o que mostra que a escolha do invariante foi apropriada, e também que o programa efectivamente calcula o factorial de n .

```
Dafny program verifier finished with 3 verified, 0 errors
```

[[permalink para este exemplo](#)]

Exercício

Uma outra noção que se pode definir recursivamente é a sequência de números de Fibonacci. Também aqui é necessário definir a sequência através de uma função lógica *fib*, que se pode depois utilizar na especificação e invariante.

1. Complete o invariante de ciclo na definição da função **ComputeFib**, que calcula o n -ésimo número de Fibonacci.
2. Escreva os triplos de Hoare correspondentes à inicialização, preservação, e utilidade do invariante, e argumente informalmente que são válidos
3. Verifique o programa recorrendo ao Dafny.

```
function fib(n: nat): nat
```

```

{
    if n == 0 then 0
    else if n == 1 then 1
        else fib(n - 1) + fib(n - 2)
}

method ComputeFib(n: nat) returns (b: nat)
    requires n > 0
    ensures b == fib(n)
{
    var i := 1;
    var a := 0;
    b := 1;
    while i < n
        invariant ...
    {
        b := b+a;
        a := b-a;
        i := i + 1;
    }
}

```

Exemplo: Contagem de ocorrências num array

O seguinte programa conta o número de ocorrências do valor guardado em k no array $vector$, entre as posições 0 e $n-1$.

```

result := 0;
i := 0;

```

```

while (i<n) {
    if (v[i] == k) result := 1+result;
    i := i+1;
}

```

A noção de contagem de ocorrências de um elemento num array não é primitiva, e terá de ser definida. Em Dafny definiremos uma função recursiva que exprima esta noção, como se segue:

```

function countn(v:array<int>, n:int, k:int) :int
    requires 0 <= n <= v.Length
    reads v
{
    if (n==0) then 0
    else if (v[n-1] == k) then 1+countn(v, n-1, k)
        else countn(v, n-1, k)
}

```

Note-se que, apesar de não se tratar de código, mas sim de uma definição ao nível lógico, esta função necessita de ser anotada com uma pré-condição que delimita o valor de n entre 0 e o limite alocado para o array ($v.Length$). É também necessária a cláusula `reads v`, que autoriza o acesso da função ao array (a discussão desta anotação está fora do nosso âmbito aqui).

Dispondo da função lógica `countn`, não teremos dificuldade em escrever uma especificação para o programa acima, a que chamaremos **Countn**:

$\{0 \leq n\} \text{ Countn } \{result = countn(vector, n, k)\}$

É fácil exprimir um invariante apropriado para este programa. No início de uma iteração, foram contadas as ocorrências entre os índices 0 e $i - 1$. Logo:

$$I \equiv (0 \leq i \leq n) \wedge result = countn(vector, i, k)$$

É bastante imediato entender que se trata de facto de um invariante do ciclo, sendo bem inicializado ($result = 0$ e $countn(vector, 0, k) = 0$ por definição) e preservado pelo corpo do ciclo, que actualiza a contagem olhando para a o índice i , que é depois incrementado.

A utilidade do invariante também é evidente, uma vez que, terminando o ciclo com $i = n$, a pós-condição é consequência de I .

Para verificar o programa escrevemos o seguinte método Dafny. Note-se que é necessário fortalecer a pré-condição, delimitando n com o tamanho alocado para o array.

```
method Countn(vector:array<int>, n:int, k:int) returns (result:int)
    requires 0 <= n <= vector.Length
    ensures result == countn(vector, n, k)
{
    result := 0;
    var i := 0;
    while (i < n)
        invariant 0 <= i <= n
        invariant result == countn(vector, i, k)
    {
        if (vector[i] == k) { result := 1+result; }
        i := i+1;
    }
}
```

A execução da ferramenta não identifica quaisquer erros, comprovando que o programa é correcto face à especificação:

```
Dafny program verifier finished with 3 verified, 0 errors
```

[[Permalink para este exemplo](#)]

Exercício

Considere-se o problema de somar todos os elementos de um *array* com N posições
+ $\{0 \leq n\}$ **Sum** { $result = \sum_{k=0}^{n-1} vector[k]$ }

Sendo Sum o seguinte programa:

```
result := 0;
i := 0;
while (i < n) {
    result := vector[i] + result;
    i := i+1;
}
```

$I : i \leq n \wedge result = \sum_{k=0}^{i-1} vector[k]$

Inic: $\{0 \leq n\}$ result := 0; i:=0 {I}

Preserv: $\{I \wedge i < n\}$ result := vector[i] + result; i := i+1 {I}

Util: $I \wedge i \geq n \rightarrow result = \sum_{k=0}^{n-1} vector[k]$

1. Defina um invariante para o programa acima
2. Escreva os triplos de Hoare correspondentes à inicialização, preservação, e utilidade do invariante, e argumente informalmente que são válidos
3. Verrifique o programa recorrendo ao Dafny. Para isso:
 - a. Escreva uma função lógica recursiva em Dafny que corresponda à noção de somatório entre dois índices de um *array*
 - b. Defina um método que lhe permita provar que o programa é correcto face à especificação acima.

Repita depois o exercício para as seguintes versões alternativas do programa (ambas correctas):

```

result := 0;
i := -1;
while (i < n-1) {
    i := i+1;
    result := vector[i] + result;
}

```

$$l : i \leq n - 1 \wedge result = \sum_{k=0}^i vector[k]$$

Inic: $\{0 \leq n\}$ result := 0; i:=-1 {I}

Preserv: $\{I \wedge i < n - 1\}$ i := i+1; result := vector[i] + result {I}

Util: $I \wedge i \geq n - 1 \rightarrow result = \sum_{k=0}^{n-1} vector[k]$

```

result := 0;
i := n;
while (i > 0) {
    i := i-1;
    result := vector[i] + result;
}

```

$$l : i \geq 0 \wedge result = \sum_{k=i}^{n-1} vector[k]$$

Inic: $\{0 \leq n\}$ result := 0; i:=n {I}

Preserv: $\{I \wedge i > 0\}$ i := i-1; result := vector[i] + result {I}

Util: $I \wedge i \leq 0 \rightarrow result = \sum_{k=0}^{n-1} vector[k]$

Exercícios

Multiplicação de números inteiros

Considere os três algoritmos seguintes de multiplicação de números inteiros, com a seguinte especificação:

$\{y \geq 0\}$ **mult** $\{r = x.y\}$

Para cada algoritmo,

1. Defina um invariante apropriado para o ciclo
2. Escreva os triplos de Hoare correspondentes à inicialização, preservação, e utilidade do invariante, e argumente informalmente que são válidos
3. Verifique o programa recorrendo ao Dafny, definindo métodos com as anotações que entender.

```
r := 0;  
i := 0;  
while (i < y)  
{  
    r := r + x;  
    i := i + 1;  
}
```

```
r := 0;  
i := y;  
while (i > 0)  
{  
    r := r + x;
```

```
i := i-1;
```

```
}
```

x = 2; y = 3

i	r
3	0 = 0*x
2	2 = 1*x
1	4 = 2*x
0	6 = 3*x

```
r := 0;
```

```
xx := x;
```

```
yy := y;
```

```
while (yy>0)
```

```
{
```

```
if (yy%2 != 0) {
```

```
    yy := yy-1;
```

```
    r := r+xx;
```

```
}
```

```
    xx := xx*2;
```

```
    yy := yy/2;
```

```
}
```

xx	yy	r
5	16	0
10	8	0
20	4	0
40	2	0
80	1	0
160	0	80

xx	yy	r
5	15	0
10	7	5
20	3	15
40	1	35
80	0	75

|: $yy \geq 0 \wedge xx^*yy + r = x^*y$

C3. Casos de estudo

Correcção de algoritmos de pesquisa num array

É interessante estudar a correcção (e invariantes) de algoritmos de pesquisa, por se tratar de algoritmos relativamente simples, que contêm um único ciclo, mas cujo número de iterações não é constante nem previsível, ao contrário de exemplos vistos anteriormente em que é efectuada uma travessia completa do array (como o somatório ou contagem de ocorrências).

Pesquisa linear

O programa seguinte procura k no array $vector$ entre os índices a e b , guardando o índice da primeira ocorrência na variável r , que ficará com o valor -1 caso não exista qualquer ocorrência.

O programa inclui 3 pós-condições. A primeira delimita os valores possíveis de r , e as duas seguintes caracterizam cada um desses casos.

```
method search(v:array<int>, a:int, b:int, k:int) returns (r:int)
    requires 0 <= a <= b < v.Length
    // ensures r == -1 || (a <= r <= b)
    // ensures r == -1 ==> forall x: int :: a <= x <= b ==> v[x] != k
    // ensures a <= r <= b ==> v[r] == k
    ensures (r == -1 && forall x: int :: a <= x <= b ==> v[x] != k) ||
           (a <= r <= b && v[r] == k)
{
    var i := a;
```

```

while (i<=b && v[i]!=k)
    invariant ...
{
    i := i+1;
}
if (i>b) { r := -1; }
else { r := i; }
}

```

Note-se que a comparação é feita na própria condição do ciclo!

Na escrita do invariante teremos de ter em conta que a condição poderá falhar agora, caso tenha sido atingido o índice b, ou ainda caso ($v[i] == k$), mas **de certeza que k não ocorreu nas posições anteriores do array**, porque nesse caso a execução não teria atingido o ponto actual.

Sendo assim um candidato a invariante será

$$I \equiv (0 \leq i \leq b + 1) \wedge (\forall x. a \leq x < i \rightarrow v[x] \neq k)$$

Os triplos de Hoare correspondentes às 3 propriedades do invariante são os seguintes:

1. **Inicialização:** $\{0 \leq a \leq b < v.Length\} \quad i := a \quad \{I\}$

2. **Preservação:** $\{I \wedge i \leq b \wedge v[i] \neq k\} \quad i := i + 1 \quad \{I\}$

3. **Utilidade:**

$$\{I \wedge (i > b \vee v[i] = k)\}$$

$\text{if } (i > b) \{r := -1\} \text{ else } \{r := i\}$

$$\{(r = -1 \vee (a \leq r \leq b)) \wedge (r = -1 \rightarrow \forall x. a \leq x \leq b \rightarrow v[x] \neq k) \wedge (a \leq r \leq b \rightarrow v[r] = k)\}$$

Note-se que I é claramente verdade imediatamente a seguir à avaliação da condição do ciclo, mesmo quando essa condição falha. Por outro lado, é fácil argumentar sobre a utilidade deste invariante para provar a pós-condição. A execução termina num dos seguintes cenários:

- $i = b + 1$ (foi atingido o final do array). Neste caso, o invariante I implica que k não se encontra nas posições $[a \dots b]$ de v , o que é coerente com a instrução $r := -1$ que será executada em seguida
- $i < b + 1$ e $vector[i] = k$ (k encontrado na posição actual). Neste caso, o invariante I implica que k não ocorre nas posições $[a \dots i - 1]$, pelo que i é o índice da primeira ocorrência, o que mais uma vez é coerente com a instrução $r := i$ que será executada em seguida

Exercícios

1. Considere a seguinte versão alternativa do algoritmo de procura linear, em que a comparação é agora feita no corpo do ciclo (note que a especificação é a mesma da versão anterior).
 - a. Defina um invariante para o ciclo
 - b. Escreva os triplos de Hoare correspondentes à inicialização, preservação, e utilidade do invariante, e argumente informalmente que são válidos
 - c. Verifique o programa recorrendo ao Dafny

```
method search(vector:array<int>, a:int, b:int, k:int) returns
(r:int)
    requires 0 <= a <= b < vector.Length
    ensures r == -1 || (a <= r <= b)
    ensures r == -1 ==> forall x: int :: a <= x <= b ==> vector
[x] != k
    ensures a <= r <= b ==> vector[r] == k
{
    r := -1;
    var i := a;
    while (i<=b && r == -1)
        invariant ...
    {
        if (vector[i] == k) { r := i; }
        i := i+1;
    }
}
```

```
}
```

Pesquisa num array ordenado

Se o *array* se encontrar à partida ordenado, o algoritmo de pesquisa linear poderá ser optimizado por forma a terminar a execução do ciclo antecipadamente, caso seja consultado um elemento *superior* ao procurado (assumindo que a ordenação do array é *crescente*).

Note que neste caso terá de ser incluída uma pré-condição exigindo que o *array* se encontre ordenado, como se segue:

```
method searchSorted(vector:array<int>, a:int, b:int, k:int) returns (r:int)
    requires 0 <= a <= b < vector.Length
    requires forall i1,i2 : int :: a <= i1 <= i2 <= b ==> vector[i1] <= vector[i2]
    ensures r == -1 || (a <= r <= b)
    ensures r == -1 ==> forall x: int :: a <= x <= b ==> vector[x] != k
    ensures a <= r <= b ==> vector[r] == k
{
    ...
}
```

Exercício

Complete o programa optimizado acima, incluindo o invariante de ciclo necessário para provar a sua correcção, e verifique-o com Dafny.

Pesquisa Binária

Podendo assumir-se que o array se encontra à partida ordenado, um método de pesquisa muito mais eficiente é a pesquisa binária, que em cada passo diminui para metade o comprimento da secção do array em que é feita a pesquisa, calculando o índice que se encontra a meio dessa secção e comparando-o com o elemento procurado.

Exercícios

Escreva invariantes apropriados e verifique cada uma das seguintes versões do algoritmo.

Dica: o invariante deverá afirmar que, caso k ocorra no array, terá de ocorrer na secção actualmente considerada para pesquisa, ou seja entre os índices l e u .

Versão 1 – comparação feita no corpo do ciclo:

```
method search(vector:array<int>, a:int, b:int, k:int) returns
(r:int)
    requires 0 <= a <= b < vector.Length
    requires forall i1,i2 : int :::
        a <= i1 <= i2 <= b ==> vector[i1] <= vector[i2]
    ensures r == -1 || (a <= r <= b)
    ensures r == -1 ==> forall x: int :: a <= x <= b ==> vector
[x] != k
    ensures a <= r <= b ==> vector[r] == k
{
    r := -1;
    var m; var l := a; var u := b;
    while (l <= u && r == -1)
        invariant ...
    {
        m := l + (u-l) / 2;
        // m := (l+u)/2;
        if (vector[m] < k) { l := m+1; }
        else { if (vector[m] > k) { u := m-1; } }
```

```

        else { r := m; }
    }
}
```

Versão 2 – comparação feita na condição do ciclo:

```

method search(vector:array<int>, a:int, b:int, k:int) returns
(m:int)
    requires 0 <= a <= b < vector.Length
    requires forall i1,i2 : int :::
        a <= i1 <= i2 <= b ==> vector[i1] <= vector[i2]
    ensures m == -1 || (a <= m <= b)
    ensures m == -1 ==> forall x: int :: a <= x <= b ==> vector
[x] != k
    ensures a <= m <= b ==> vector[m] == k
{
    var l := a; var u := b;
    m := l + (u-l) / 2;
    while (l < u && vector[m] !=k)
        invariant ...
    {
        if (vector[m] < k) { l := m+1; }
        if (vector[m] > k) { u := m-1; }
        m := l + (u-l) / 2;
    }
    if (m < l || vector[m] != k) { m := -1; }
}
```

Determinação de elementos repetidos num array

Considere agora o problema de se determinar se um array contém ou não algum elemento repetido. O seguinte programa faz isso de forma eficiente, colocando *r* com valor *true* caso exista um par de posições do array contendo o mesmo elemento, e parando a execução quando isso acontece.

```
r := false;
i := 0;
while (i < N - 1 && !r)
{
    j := i + 1;
    while (j < N && !r)
    {
        if a[i] == a[j] { r := true; }
        j := j + 1;
    }
    i := i + 1;
}
```

Em Dafny poderíamos escrever o seguinte método:

```
method dups(a: array<int>) returns (r: bool)
    requires a.Length > 0
    ensures r <==> (exists k, l :: 0 <= k < l < a.Length && a[k] == a[l])
{
    r := false;
```

```

var j; var N := a.Length;
var i := 0;
while (i<N-1 && !r)
    invariant ... <= i <= ...
    invariant !r ==>
    invariant r ==>
{
    j := i+1;
    while (j<N && !r)
        invariant ... <= j <= ...
        invariant !r ==>
        invariant r ==>
    {
        if a[i] == a[j] { r := true; }
        j := j+1;
    }
    i := i+1;
}

```

Escreva os invariantes necessários para provar a correcção do programa, procurando descrever o que é garantido no início de cada iteração dos ciclos interior e exterior, nos casos em que r seja verdadeiro e falso.

T1. Tempo de Execução de Algoritmos Iterativos

Contabilização de operações primitivas

Iremos:

1. identificar as operações primitivas que executam em *tempo constante*, i.e. tempo que não depende do tamanho do input
2. atribuir *custos (tempos) abstractos* c_1, c_2, \dots a estas operações primitivas, de forma independente de qualquer mecanismo de execução concreto
3. calcular o custo abstracto total do algoritmo em função do *tamanho do input* – $T(N)$

Exemplo: contagem de ocorrências num array

```
int conta (int k, int v[], int N) {  
    int i = 0, r = 0;                      c1      1  
    while (i < N) {                        c2      N+1  
        if (v[i] == k) r++;                c3      N  
        i++;                            c4      N  
    }  
    return r;                            c5      1  
}
```

Note-se que

- c_1 corresponde ao custo das duas operações de inicialização;
- a condição do ciclo é uma operação primitiva, avaliada $N+1$ vezes (falso na última vez), com custo c_2

Temos: $T(N) = c_1 + c_2(N + 1) + c_3N + c_4N + c_5$ ou simplificando:

$$T(N) = (c_2 + c_3 + c_4)N + c_1 + c_2 + c_5$$

Polinómio de primeiro grau em N: crescimento linear

Exemplo: identificação de duplicados num array

```
void dup (int *v, int a, int b) {  
    int i, j;  
    for (i=a ; i<=b ; i++) c1 N+1  
        for (j=a ; j<=b ; j++) c2 N*(N+  
1)  
            if (i!=j && v[i]==v[j]) c3 N*N  
                printf("%d igual a %d\n", i, j);  
}
```

- A função procura elementos repetidos entre os índices a e b. Sendo assim, o tamanho do input será dado por $N = b - a + 1$.
- c_1 é o custo agregado da condição $i < b$ (testada $N+1$ vezes) e do incremento $i++$ (feito N vezes)
- O ciclo exterior executa N iterações, e em cada iteração, o ciclo interior executa N iterações. Sendo assim:

O condicional (custo c_3) é executado no total N^2 vezes

Logo:

$$T(N) = c_1(N + 1) + c_2N(N + 1) + c_3N^2 \text{ ou simplificando:}$$

$$T(N) = (c_2 + c_3)N^2 + (c_1 + c_2)N + c_1$$

Polinómio de segundo grau em N : crescimento quadrático

Exemplo: identificação de duplicados num array

A versão anterior imprime cada par duas vezes! Pode ser optimizada:

```
void dup2 (int v[], int a, int b) {
```

```

int i, j;
for (i=a ; i<b ; i++)
    for (j=i+1 ; j<=b ; j++)
        if (v[i]==v[j])
            printf("%d igual a %d\n", i, j);
}

```

Sendo mais eficiente, esta versão é mais difícil de analisar, uma vez que, para cada valor de i fixado pelo ciclo exterior, o ciclo interior executará um número diferente de vezes!

A tabela seguinte detalha os valores tomados por j e o número de iterações do ciclo interior, para cada valor de i :

$i=a$	$j = a+1 \dots b$	$b-a$ iterações
$i=a+1$	$j = a+2 \dots b$	$b-a-1$ iterações
...
i (arbitrário)	$j = i+1 \dots b$	$b - (i+1) + 1 = b-i$
...
$i=b-1$	$j = b$	1 iteração

Somando todas estas iterações, obtemos $S_2 = \sum_{i=a}^{b-1} b - i$ e também $S_1 = \sum_{i=a}^{b-1} b - i + 1$.

Admitamos sem perda de generalidade que $a = 1$ e $b = N$. Então

$$S_2 = \sum_{i=1}^{N-1} (N - i) = (N - 1)N - \frac{(N-1)N}{2} = \frac{1}{2}N^2 - \frac{1}{2}N$$

$$S_1 = \sum_{i=1}^{N-1} (N - i + 1) = S_2 + N - 1$$

logo:

$$T(N) = c_1 N + c_2 S_1 + c_3 S_2$$

$$T(N) = k_2 N^2 + k_1 N + k_0$$

Polinómio de segundo grau em N: crescimento quadrático

(veremos que não é muito importante calcular os coeficientes k_0 a k_2)

EXERCÍCIO: Calcule o tempo de execução da seguinte versão alternativa desta função:

```
void dup (int v[], int a, int b) {  
    int i, j;  
    for (i=a+1 ; i<=b ; i++)  
        for (j=a ; j<i ; j++)  
            if (v[i]==v[j])  
                printf("%d igual a %d\n", i, j);  
}
```

$$T_{==}(n) = \sum_{i=a+1}^b i - 1 - a + 1 = \sum_{i=a+1}^b i - a = \sum_{i=a+1}^b \sum_{j=a}^{i-1} 1$$

$$T_{j < i}(n) = \sum_{i=a+1}^b \sum_{j=a}^i 1$$

Sem perda de generalidade, consideramos $a = 0$ e $b = n - 1$

$$T_{==}(n) = \sum_{i=1}^{n-1} i = \frac{(n-1)*n}{2} = \theta(n^2)$$

$$T_{j < i}(n) = \sum_{i=1}^{n-1} i + 1 = \sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} 1 = \frac{(n-1)*n}{2} + n - 1 = \theta(n^2)$$

+

$$T_{printf}^{pc}(n) = \frac{(n-1)*n}{2}, \text{ quando todos os elementos são iguais}$$

$$T_{printf}^{mc}(n) = 0, \text{ quando todos os elementos são diferentes}$$

$$T(n) = \theta(n^2) \quad (\text{sem casos})$$

Análise Assimptótica

Numa função polinomial como $T(N) = k_2N^2 + k_1N + k_0$,

para *inputs de tamanho elevado* o efeito dos termos de menor grau é anulado face ao crescimento do termo de maior grau.

E de facto, o interesse da constante multiplicativa k_2 é também pequeno: na *análise assimptótica* interessamo-nos apenas pela *ordem de crescimento* do tempo de execução dos algoritmos.

Escreveremos:

$$T(N) = \Theta(N^2)$$

Se o algoritmo A1 é *assimptoticamente melhor* do que A2, será melhor escolha do que A2 *excepto para inputs muito pequenos*.

Notação O ("big oh")

Para uma função g não-negativa de domínio \mathbb{N} ,
define-se $O(g)$ como o seguinte *conjunto de funções*:

$$O(g) = \{f \mid \text{existem } c, n_0 > 0 \text{ tais que } \forall n \geq n_0. 0 \leq f(n) \leq cg(n)\}$$

Para $n \geq n_0$, g é um limite superior de f a menos de um factor constante

Exemplos:

- $3n^2 + 7n \in O(n^2)$
- $4n - 5 \in O(n^2)$
- $7n^3 - 2n \notin O(n^2)$

Notação Ω (Omega)

Para uma função g não-negativa de domínio \mathbf{N} ,
define-se $\Omega(g)$ como o seguinte *conjunto de funções*:

$$\Omega(g) = \{f \mid \text{existem } c, n_0 > 0 \text{ tais que } \forall n \geq n_0. 0 \leq cg(n) \leq f(n)\}$$

| Para $n \geq n_0$, g é um limite inferior de f a menos de um factor constante

Exemplos:

- $3n^2 + 7n \in \Omega(n^2)$
- $4n - 5 \notin \Omega(n^2)$
- $7n^3 - 2n \in \Omega(n^2)$

Notação Θ (Theta)

Para uma função g não-negativa de domínio \mathbf{N} ,

$$\Theta(g) = O(g) \cap \Omega(g)$$

| Se $f \in \Theta(g)$, então para $n \geq n_0$ f tem um comportamento "igual" ao de g , a menos de factores constantes

Exemplos:

- $3n^2 + 7n \in \Theta(n^2)$
- $4n - 5 \notin \Theta(n^2)$
- $7n^3 - 2n \notin \Theta(n^2)$

Para os exemplos acima podemos escrever:

- Contagem de ocorrências num array: $T(N) = \Theta(N)$
- Identificação de duplicados num array: $T(N) = \Theta(N^2)$

Funções Úteis em Análise de Algoritmos

As funções tipicamente utilizadas são as *logarítmicas, polinomiais, e exponenciais*.
Alguns factos envolvendo estas funções:

A base das funções logarítmicas é irrelevante assintoticamente (desde que > 1):

$$\log_b n = O(\log_a n) \text{ se } a, b > 1$$

Já no caso das funções polinomiais, qualquer polinómio é limitado superiormente por qualquer outro polinómio de maior grau:

$$n^b = O(n^a) \text{ se } b \leq a$$

O mesmo acontece com as funções exponenciais relativamente à base:

$$b^n = O(a^n) \text{ se } b \leq a$$

Por outro lado, uma função logarítmica é limitada superiormente por qualquer função polinomial:

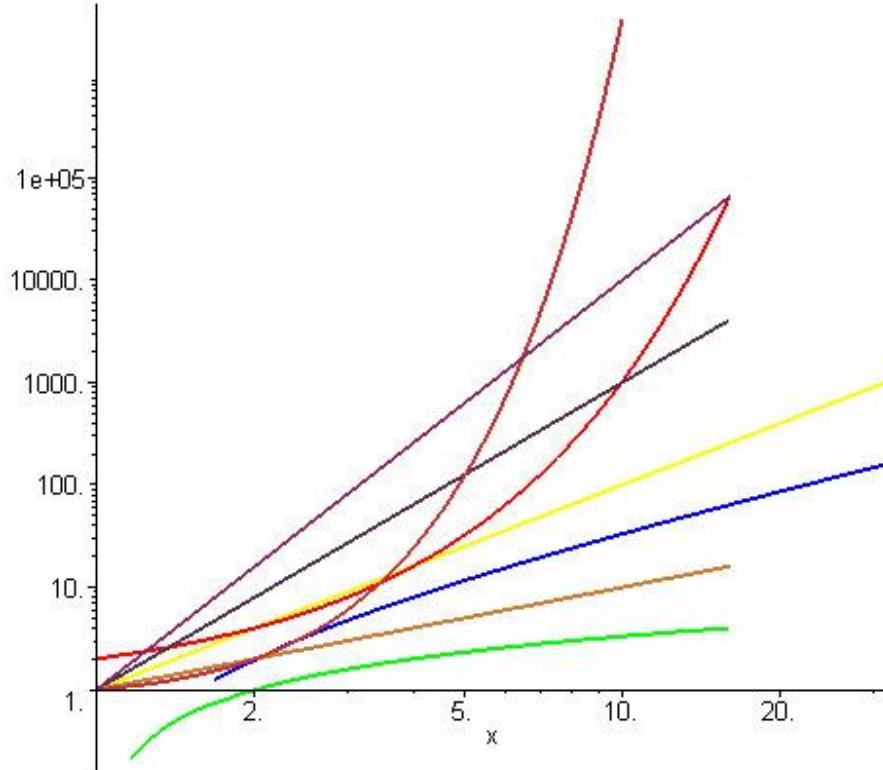
$$\log_b n = O(n^a)$$

e uma função polinomial é limitada superiormente por qualquer função exponencial de base > 1 :

$$n^b = O(a^n) \text{ se } a > 1$$

Crescimento de Algumas Funções Típicas

Tempo (μs)	$33n$	$46n \lg n$	$13n^2$	$3.4n^3$	
Tempo assimp.	n	$n \lg n$	n^2	n^3	2^n
Tempo de execução por tamanho do input	n=10 n=100 n=1000 n=10000 n=100000	.00033s .003s .033s .33s 3.3s	.0015s .03s .45s 6.1s 1.3m	.0013s .13s 13s 22m 1.5 dias	.0034s 3.4s .94h 39 dias 108 anos
Tamanho máx. do input para	1s 1m	3.10^4 18.10^5	2000 82000	280 2200	67 260
					20 26



$\log x$, funções polinomiais de diversos graus (rectas, incluindo $n \lg n$), 2^x , e $x!$

Identificação de Operações Relevantes

- Na prática, para analisar asymptoticamente um algoritmo não é necessário considerar custos abstractos associados às operações primitivas: basta contar o número de vezes que as operações são executadas.
- De facto não é sequer necessário contar *todas* as operações primitivas. Basta identificar as que são relevantes para a análise asymptótica.

Dadas duas operações op1 e op2 de um programa, se o número de execuções de op1 não for asymptoticamente superior ao número de execuções de op2, então

op1 pode ser descartada para efeitos de análise (assimptótica) de tempo de execução

Exemplo:

```
void dup2 (int v[], int a, int b) {  
    int i, j;  
    for (i=a ; i<b ; i++)  
        for (j=i+1 ; j<=b ; j++)           Soma S1  
            if (v[i]==v[j])                  Soma S2  
                printf("%d igual a %d\n", i, j);  
}
```

Neste programa basta considerar o condicional (corpo do ciclo interior) como única instrução asymptoticamente relevante. O tempo de execução pode ser analisado calculando simplesmente: $S_2 = \sum_{i=a}^{b-1} b - i = \Theta(N^2)$.

Apesar de a condição $j \leq b$ ser avaliada mais vezes ($S_1 = S_2 + N - 1$) do que o condicional (S2), asymptoticamente é equivalente considerar qualquer uma das duas para efeitos de análise.

Na prática contamos o número de operações de comparação efectuadas (entre elementos do array), o que é bastante intuitivo.

Tamanho e Representação

A noção apropriada de tamanho de um número corresponde ao número de caracteres necessários para o escrever: o tamanho de 3500 é 4 em notação decimal.

Um inteiro n em notação decimal ocupa aproximadamente $\log_{10} n$ dígitos; em notação binária (representação em máquina) ocupa $\log_2 n$ dígitos.

EXEMPLO:

Problema: Dado um inteiro positivo n , haverá dois inteiros $j, k > 1$ tais que $x = jk$? (i.e, será x um número primo ou não?)

Considere-se o seguinte algoritmo de “força bruta”:

```
found = 0;  
j = 2;  
while ((!found) && j < x) {  
    if (x mod j == 0) found = 1;  
    else j++;  
}
```

Este algoritmo executa em tempo $O(x)$, no entanto trata-se de um problema famoso pela sua dificuldade (e é por isso relevante em muitos algoritmos criptográficos).

Observe-se:

- se o algoritmo executa em tempo linear $O(x)$,
- e a representação de x utiliza $N = \log_k x$ dígitos (ou bits),
- então $x = k^N$,
- e o algoritmo executa por isso em tempo $T(N) = O(k^N)$.

Ou seja: *um algoritmo de tempo aparentemente linear é de facto de tempo exponencial no tamanho da representação do número!*

É importante identificar correctamente a noção adequada de tamanho do input de um problema, uma vez que disso depende a sua classificação como fácil ou difícil!

Exercícios Adicionais

1. Um dos algoritmos de *ordenação* mais simples é conhecido por *selection sort*. A ideia é, em cada iteração j do ciclo exterior (ciclo for), colocar na posição de índice j do array o j -ésimo menor elemento. O ciclo interior (while) determina o índice do menor elemento ainda não colocado na sua posição final, e a função swap é usada para trocar este elemento com o que se encontra na posição j .

```

void swap(int t[],int i,int j) {
    int tmp = t[i];
    t[i] = t[j];
    t[j] = tmp;
}

void selectionSort(int A[], int n) {
    int i, j, min, k;

    for (j=0; j < n-1; j++) {
        min = j;
        i = j+1;
        while (i < n) {
            if (A[min] > A[i]) min = i;
            i++;
        }
        if (j != min) swap (A, j, min);
    }
}

```

Note que a função swap executa em *tempo constante*, i.e. o seu tempo de execução não depende do tamanho do array. Escreveremos $T_{\text{swap}}(N) = \Theta(1)$. Tendo isto em conta, identifique uma ou mais operações relevantes e analise assintoticamente o tempo de execução da função.

T2. Análise de Pior Caso, Melhor Caso, e Caso Médio

Análise de pior e melhor caso

Exemplo: o seguinte algoritmo de **procura linear** encontra a primeira ocorrência de k no array v , entre as posições a e b .

```
int procura (int v[], int a, int b, int k) {  
    int i = a;  
    while ((i<=b) && (v[i]!=k))  
        i++;  
    if (i>b)  
        return -1;  
    else return i;  
}
```

A simples contagem do número de execuções das operações primitivas permite concluir que basta considerar como operação relevante a condição do ciclo, ou alternativamente a instrução `i++`. Consideraremos a primeira na nossa análise. Contaremos o número de comparações `(i<=b) && (v[i]!=k)` efectuadas.

Ora, o número de iterações do ciclo não é agora fixo: não depende apenas do tamanho do array, mas também do seu conteúdo concreto e do valor de k . Seja $N = b - a + 1$.

- **Pior Caso:** quando k não ocorre em $v[a..b]$
O corpo do ciclo é executado N vezes, e são feitas $N + 1$ comparações.
- **Melhor Caso:** quando $k == v[a]$, ocorre na primeira posição.
O corpo do ciclo é executado 0 vezes, e é feita apenas uma comparação.

Sendo assim, o algoritmo executa *no melhor caso em tempo constante*, e *no pior caso em tempo linear*. Escreveremos:

$$T(N) = \Omega(1), O(N)$$

É importante perceber que esta análise de casos extremos é limitada. Em particular, não sabemos o que acontece *em média*; nem sabemos qual a probabilidade de ocorrência de cada um destes casos extremos.

No entanto, a análise de pior caso é em geral considerada extremamente útil, pelas seguintes razões:

- Constitui uma *garantia*, uma vez que nos dá um limite superior para o tempo de execução, válido para qualquer input do algoritmo
- Em muitos cenários o pior caso ocorre *muito frequentemente*. Por exemplo no algoritmo de pesquisa ordenada, existem certamente muitos valores de k que não ocorrem no array, e sempre que for feita uma pesquisa com um destes valores estaremos em presença do pior caso
- Por esta razão, o pior caso frequentemente coincide com o caso *médio*

Exemplo: algoritmo **Insertion sort**.

A função seguinte ordena o array A de forma crescente entre as posições 0 e N-1.

```
void insertionSort(int A[], int N) {
    int i, j, key;
    for (j=1 ; j<N ; j++) {
        key = A[j];
        i = j-1;
        while (i>=0 && A[i]>key) {
            A[i+1] = A[i];
            i--;
        }
    }
}
```

```

    }
    A[i+1] = key;
}
}

```

Relembrando o funcionamento deste algoritmo, vai sendo construído um segmento inicial ordenado do array, satisfazendo o seguinte:

Invariante de ciclo: no início de cada iteração do ciclo `for`, o vector contém entre as posições 0 e $j-1$ os mesmos valores que lá estavam inicialmente, já ordenados.

Em cada passo (i.e. cada iteração do ciclo `for` exterior) é inserido no segmento ordenado um novo elemento (`key`, que está inicialmente na posição j). O ciclo interior `while` copia para a posição seguinte os elementos superiores ao que vai ser inserido, por forma a criar espaço para a inserção de `key`.

Compreender o funcionamento do algoritmo é importante para a análise do seu tempo de execução. Comecemos por observar que, havendo ciclos aninhados, as operações relevantes são as do *ciclo interior*. Consideraremos como operação relevante a condição `(i>=0 && A[i]>key)` .

Notemos em seguida que o número de iterações do ciclo exterior é fixo ($N-1$), mas o número de iterações do ciclo interior é variável. O número de iterações do ciclo interior varia entre 0 e j , logo o número de vezes que a condição é avaliada varia entre 1 e $j+1$.

- **Pior Caso:** quando o elemento `key` a inserir é menor do que todos os elementos do segmento ordenado, o que quer dizer que `key` será inserido na posição 0 do array. Neste caso a condição é testada inicialmente com $i = j - 1$ e pela última vez com $i = -1$, um total de $j + 1$ vezes.

Para que isto aconteça em todas as iterações do ciclo exterior, i.e. para todos os valores de j , é necessário que o array esteja inicialmente **ordenado por ordem decrescente** (inversa à pretendida). Neste caso o número total de vezes que a condição `(i>=0 && A[i]>key)` é avaliada será então

$$\sum_{j=1}^{N-1} j + 1 = \sum_{j=1}^{N-1} j + \sum_{j=1}^{N-1} 1 = \frac{(N-1)N}{2} + N - 1 = \frac{1}{2}N^2 + \frac{1}{2}N - 1.$$

- **Melhor Caso:** quando o elemento `key` a inserir é maior do que todos os elementos do segmento ordenado, o que quer dizer que `key` ficará na sua posição inicial (`j`). Neste caso a condição é testada uma única vez, falhando imediatamente. Para que isto aconteça em todas as iterações do ciclo exterior, i.e. para todos os valores de `j`, é necessário que o array esteja inicialmente **já ordenado por ordem crescente**. Neste caso o número total de vezes que a condição `(i>=0 && A[i]>key)` é avaliada será então $\sum_{j=1}^{N-1} 1 = N - 1$.

Sendo assim, o algoritmo executa *no melhor caso em tempo linear, e no pior caso em tempo quadrático*. Escreveremos:

$$T(N) = \Omega(N), O(N^2)$$

Nota: Não teria sido adequado optar por tomar como operação relevante (apenas) uma das atribuições no corpo do ciclo `while`, o que nos teria levado, incorrectamente, a um comportamento de tempo constante no melhor caso!!!

EXERCÍCIO: procura num array ordenado

A função seguinte pode ser utilizada para procurar a primeira ocorrência de `k` em `v`, desde que este array se encontre *ordenado* de forma crescente.

```
// requires v ordenado de forma crescente entre os índices a e b
int procuraOrd (int v[], int a, int b, int k) {
    int i = a;
    while ((i<=b) && (v[i]<k))
        i++;
    if (i>b || v[i]!= k)
        return -1;
}
```

```
    else return i;  
}
```

Identifique o melhor e o pior caso de execução desta função e analise os respectivos tempos de execução assintóticos. Tendo em conta essa análise como compara esta função com a função `procura` anterior?

EXERCÍCIO: algoritmo de incremento de um array de bits

Considere-se a operação de incremento de um número inteiro, representado como array de bits (*bitvector*). O primeiro 0 do array, isto é, aquele que se encontra na posição “menos significativa”, deverá passar a 1, sendo que todos os 1s que se encontram antes dele (i.e. em posições menos significativas) passarão a 0. Por exemplo:

- incrementar o bitvector **10101000** resulta em **10101001**
- incrementar o bitvector **01010111** resulta em **01011000**

Escreveremos o algoritmo em C considerando que os bitvectors são representados por arrays de números inteiros (com valor 0 ou 1), correspondendo o índice 0 ao bit menos significativo.

```
void inc (int b[], int N) {  
    int i = 0;  
    while ((i < N) && (b[i] == 1)) {  
        b[i] = 0;  
        i++;  
    }  
    if (i < N) b[i] = 1;  
}
```

Identifique o melhor e o pior caso deste algoritmo (i.e. em que circunstâncias ocorrem), e analise o seu tempo de execução em cada um desses casos.

EXERCÍCIO: algoritmo de procura binária

A procura num array ordenado pode ser feita de forma mais eficiente do que anteriormente pelo seguinte algoritmo de *procura binária*:

```
// v ordenado de forma crescente entre os índices a e b
int procuraBin (int v[], int a, int b, int k) {
    int m, result := -1;
    while (a <= b && result == -1) {
        m = a + (b-a) / 2;
        if (vector[m] < k) a = m+1;
        else if (vector[m] > k) b = m-1;
        else result = m;
    }
    return result;
}
```

Analise o tempo de execução assintótico do algoritmo no melhor e no pior caso.
Diga em que situações ocorrem esses casos.

Exercícios Adicionais

1. Considere os dois seguintes algoritmos de multiplicação de números inteiros:

```
INT prod (INT x, INT y) {
    int r = 0;
    while (y>0) {
        r = r+x;
```

```

        y = y-1;
    }
    return r;
}

INT bprod (INT x, INT y) {
    int r = 0;
    while (y>0) {
        if (y%2 != 0) r = r+x;
        x = x*2;
        y = y/2;
    }
    return r;
}

```

Como referido em [+T1. Tempo de Execução de Algoritmos Iterativos](#), na análise deste tipo de algoritmos é importante ter em conta que o tamanho de um número não é o próprio número: a análise deve ser feita em função do comprimento da representação dos números.

Analice então o número de vezes que a operação representativa correspondente à avaliação da expressão `y>0` é executada, no melhor e no pior caso. Efectue a análise em função do número de *bits necessários para representar y*, no melhor e no pior caso, ou seja o seu tamanho útil. A análise em função do tamanho do tipo INT não faz muito sentido, uma vez que este tamanho é fixo, e a análise assimptótica pressupõe a variação do tamanho.

RESOLUÇÃO

Antes de mais vejamos alguns exemplos do número de bits necessários para representar alguns números:

- para representar 2 = 10 são necessários 2 bits
- para representar 7 = 111 são necessários 3 bits
- para representar 17 = 10001 são necessários 5 bits

Colocando a questão ao contrário: quais são os números que podem ser representados com, por exemplo, 4 bits?

Esta gama abrange os números desde $1000 = 8$ até $1111 = 15$.

Generalizando, com n bits é possível representar os números desde 2^{n-1} até $2^n - 1$.

Na função `prod` o número de avaliações de $y > 0$ é $y + 1$, e a variação de y entre 2^{n-1} e $2^n - 1$ permite identificar o melhor e o pior caso:

$$T^{mc}(n) = 2^{n-1} + 1, \text{ quando } y = 10 \dots 0 = 2^{n-1}$$

$$T^{pc}(n) = 2^n, \text{ quando } y = 11 \dots 1 = 2^n - 1$$

Em termos assimptóticos, o tempo de execução é em ambos os casos exponencial, $T(n) = \theta(2^n)$.

Já na função `bprod` o número de avaliações é substancialmente inferior.

Considerando os números representáveis por 4 bits, temos para $y=8$ que y toma sucessivamente os valores 8, 4, 2, 1, 0, e para $y=15$ toma os valores 15, 7, 3, 1, 0.

Temos então em ambos os casos 5 avaliações, e é fácil ver que em geral teremos $T(n) = n + 1 = \theta(n)$.

Uma forma de entender a diferença de funcionamento entre ambas as funções passa por observar a evolução dos números em representação binária.

Assim com `bprod` para $y = 1000$ temos $1000 \rightarrow 100 \rightarrow 10 \rightarrow 1 \rightarrow 0$ (cada divisão por 2 corresponde a um *shift*), enquanto que com `prod` temos $1000 \rightarrow 111 \rightarrow 110 \rightarrow 101 \rightarrow 100 \rightarrow 11 \rightarrow 10 \rightarrow 1 \rightarrow 0$.

2. Algoritmo de ordenação **Bubblesort**

Este algoritmo, apesar de não ser considerado uma boa escolha (a análise empírica revela que de facto não é), é conceptualmente interessante. Tal como no **selection sort**, o passo básico coloca o i -ésimo elemento do vector na sua posição final, mas este passo inclui agora uma acção secundária, que vai

progressivamente contribuindo para ordenar os restantes elementos do array, o que significa que pode não ser necessário efectuar N passos básicos.

```
void bubbleSort (int v[], int N){  
    int i, j=0, ok=0;  
    while (!ok) {  
        ok = 1;  
        for (i=N-1; i>j; i--)  
            if (v[i-1]>v[i]) {  
                swap(v, i-1, i);  
                ok = 0;  
            }  
        j++;  
    }  
}
```

- a. Determine o melhor e o pior caso para o número de trocas (chamadas à função `swap`).
- b. Determine o melhor e o pior caso para o número de comparações entre elementos do array.
- c. Qual das operações deverá ser considerada a relevante do ponto de vista do tempo de execução do algoritmo? Efectue a análise asymptótica do tempo de execução no melhor e no pior caso.
- d. O princípio em que se baseia este algoritmo (e o invariante do ciclo exterior) é o mesmo do algoritmo *selection sort*, mas o i-ésimo elemento do vector é agora colocado na posição final i através de uma sequência de operações `swap` entre posições adjacentes do array. O algoritmo incorpora uma optimização que consiste em parar quando a colocação de um elemento é conseguida sem que seja necessário um único `swap`, o que significa que o vector está já ordenado.

O algoritmo pode ser mais optimizado. Para o array

[10,20,30,40,80,70,60,50] obtemos na primeira iteração do ciclo exterior [10,20,30,40,50,80,70,60], sendo que o último `swap` efectuado nesta iteração envolveu as posições 4 (80) e 5 (50). Isto significa que os elementos

nas posições 0 ... 4 estão já ordenados, podendo por isso ser deixados de fora da próxima iteração.

Implemente esta optimização e repita a análise de melhor e pior caso para o algoritmo optimizado.

3. Considere a seguinte função em C que determina se um vector de inteiros contém elementos repetidos.

```
int repetidos (int v[], int N) {  
    int i, j, rep = 0;  
    for (i=0; i<N-1 && !rep; i++)  
        for (j=i+1; j<N && !rep; j++)  
            if (v[i]==v[j]) rep = 1;  
    return rep;  
}
```

- a. Identifique o melhor e o pior casos da execução desta função no que respeita ao número de comparações (entre elementos do vector).
 - b. Quantas operações são executadas em cada caso? Efectue a análise assintótica do tempo de execução em ambos os casos.
-
4. Relembre o algoritmo selection sort de [+T1. Tempo de Execução de Algoritmos Iterativos](#):

```
void selectionSort(int A[], int n) {  
    int i, j, min, k;  
  
    for (j=0; j < n-1; j++) {  
        min = j;  
        i = j+1;  
        while (i < n) {
```

```
        if (A[min] > A[i]) min = i;  
        i++;  
    }  
    if (j != min) swap (A, j, min);  
}  
}
```

Claramente, a operação `swap` não é uma boa escolha para a análise assintótica do tempo de execução. Apesar disso, proceda agora à análise do melhor e do pior caso do número de swaps executados, tendo particular cuidado com a identificação de inputs que levam ao pior caso.

Análise de caso médio

Em qualquer dos exemplos anteriores o melhor e o pior caso identificam um espectro de execuções possíveis do algoritmo, limitado inferiormente e superiormente por aqueles dois casos especiais. No entanto, a simples identificação dos casos extremos não nos fornece qualquer indicação sobre qual o comportamento do algoritmo em termos médios.

A análise de caso médio procura responder a esta questão, calculando o **valor esperado** do número de execuções das operações relevantes. Trata-se de uma noção estudada em Teoria de Probabilidades, mas que é suficientemente simples para poder ser aplicada sem grandes noções teóricas daquela área.

EXEMPLO: jogo de dados

[origem: http://www.wikihow.com/Calculate-an-Expected-Value#Finding_the_Expected_Value_of_a_Dice_Game_sub]

Imagine-se um jogo de dados com os seguintes prémios monetários:

- 30€ caso se obtenha um 6
- 20€ caso se obtenha um 5

Cada jogada tem no entanto um custo fixo de 10€.

Para calcularmos o valor esperado de uma jogada, começamos por calcular o saldo real de cada caso (resultado de uma jogada):

- 1 -10€
- 2 -10€
- 3 -10€
- 4 -10€
- 5 10€
- 6 20€

Calculamos agora o valor esperado como a soma pesada destes valores, tomando como pesos as probabilidades de ocorrência dos diferentes casos. Tratando-se de um dado, a probabilidade de ocorrer cada face é a mesma: $\frac{1}{6}$. Logo a probabilidade de o saldo real ser -10€ é 4/6; a probabilidade de ser 10€ é 1/6, e a probabilidade de ser 20€ é também 1/6.

Sendo assim temos:

$$E = \frac{4}{6}(-10) + \frac{1}{6}10 + \frac{1}{6}20$$

Note-se que a soma das probabilidades de todos os casos é sempre igual a uma unidade. Neste caso $\frac{4}{6} + \frac{1}{6} + \frac{1}{6} = 1$.

O valor esperado é então $E = -1.67\text{€}$, pelo que se trata de um jogo que, em média, não compensa jogar.

Exemplo: algoritmo de procura linear

Relembremos o algoritmo estudado acima. O que pretendemos agora é calcular o valor esperado da operação que foi considerada relevante para a análise de melhor e

pior caso (a comparação $(i \leq b) \text{ } \&\& \text{ } (v[i] \neq k)$).

```
int procura (int v[], int a, int b, int k) {  
    int i = a;  
    while ((i <= b) && (v[i] != k))  
        i++;  
    if (i > b)  
        return -1;  
    else return i;  
}
```

Comecemos por simplificar a nossa análise admitindo que é certo que **k** ocorre **exactamente uma vez no array v**. Neste caso a condição $(i \leq b)$ será sempre verdadeira, uma vez que o ciclo terminará garantidamente antes de i ultrapassar o limite superior do array. Basta pois contar o número de comparações $(v[i] \neq k)$ entre k e elementos do array.

Admitimos também neste caso que k pode ocorrer *com igual probabilidade em qualquer posição do array*. Então, a probabilidade de esta ocorrência ser numa determinada posição i do array é de $\frac{1}{N}$, qualquer que seja i.

Naturalmente, o número de comparações $(v[i] \neq k)$ depende da posição em que k ocorrer pela primeira vez:

- 1 comparação se k ocorre na posição a
- 2 comparações se k ocorre na posição a+1
- ...
- N comparações se k ocorre na posição b

O *número esperado de comparações* pode então ser calculado escrevendo a soma pesada destes números, sendo o peso a probabilidade de cada caso, ou seja:

$$\frac{1}{N}1 + \frac{1}{N}2 + \dots + \frac{1}{N}N$$

$$= \frac{1}{N} \sum_{i=1}^N i = \frac{N(N+1)}{2N} = \frac{N+1}{2}$$

Este resultado está de acordo com o que poderíamos intuitivamente esperar: se k ocorre com igual probabilidade em qualquer posição do array, então *em média* o número de comparações feitas será o correspondente à situação em que k ocorre no meio do array.

Consideremos agora o que acontece no outro caso, quando k não ocorre no array v . Neste caso o número de vezes que a condição $(i \leq b) \ \&\& \ (v[i] \neq k)$ é avaliada é sempre $N+1$ (N vezes incluindo $(v[i] \neq k)$ e uma vez $(i \leq b)$), correspondente ao pior caso do algoritmo.

Como calcular agora o caso médio do tempo de execução no caso geral, combinando os dois casos anteriores? Mais uma vez se trata de uma combinação de casos, pelo que teremos que calcular a média pesada destes. Seja p a probabilidade de k ocorrer no array, ou seja a probabilidade do primeiro caso acima. Então, a probabilidade do segundo caso (k não ocorrer no array) será naturalmente $1 - p$.

Temos então que o valor esperado do número de comparações será, no caso geral:
$$T(N) = p \frac{N+1}{2} + (1 - p)(N + 1)$$

Simplificando vemos que se trata de uma expressão linear em N , qualquer que seja $p \in [0, 1]$.

$$T(N) = (1 - \frac{p}{2})N + 1 - \frac{p}{2} = \Theta(N)$$

Mas como pode ser calculada esta probabilidade p num caso concreto? Se se tratar de um array de números inteiros de m bits, existirão 2^m números diferentes, e assumindo aleatoriedade temos que a probabilidade de um destes números ocorrer num array com N números será $p = \frac{N}{2^m}$. Na prática em muitos cenários será $p \approx 0$, logo $T(N) \approx N + 1$, ou seja o comportamento de caso médio será igual ao do pior caso.

Em alternativa a este método baseado em contagem, quem tiver familiaridade com a noção de probabilidade condicionada pode alternativamente calcular da seguinte forma a probabilidade de serem feitas i comparações:

Para $i \leq N$, caso em que k ocorre no array, teremos a seguinte probabilidade:

probabilidade de k não ocorrer nas primeiras $i - 1$ posições

*

probabilidade de k ocorrer na posição i , condicionada pelo facto de não ocorrer nas primeiras $i - 1$ posições.

Ora, o efeito do condicionamento é aqui irrelevante (os valores guardados nas diversas posições do array são todos independentes), pelo que temos

$$(1 - \frac{1}{2^m})^{i-1} * \frac{1}{2^m}$$

Quanto ao caso em que são feitas $N + 1$ comparações, quando k não ocorre no array, temos probabilidade $(1 - \frac{1}{2^m})^N$ de ocorrência deste cenário.

Sendo assim, temos $T(N) = (\sum_{i=1}^N (1 - \frac{1}{2^m})^{i-1} * \frac{1}{2^m} * i) + (1 - \frac{1}{2^m})^N * (N + 1)$

em que o segundo termo da soma corresponde à situação em que são feitas $N + 1$ comparações porque k não ocorre em todo o array.

Este é o caso mais geral, em que nada é assumido sobre o número de ocorrências de k , que pode ser superior a 1. Para valores razoavelmente grandes de m , facilmente se vê que $T(N) \approx N + 1$.

De que forma pode a probabilidade de serem feitas i comparações ser calculada no cenário em que k ocorre exactamente uma vez no array, usando probabilidades condicionadas?

Continuamos a ter:

probabilidade de k não ocorrer nas primeiras $i - 1$ posições

*

probabilidade de k ocorrer na posição i , condicionada pelo facto de não ocorrer nas primeiras $i - 1$ posições.

Neste cenário, a probabilidade de não ocorrer nas $i - 1$ primeiras posições é igual à probabilidade de ocorrer nas $N - (i - 1)$ restantes, sabendo que lá ocorre de

certeza, ou seja $\frac{N-i+1}{N}$.

Atente-se no cálculo desta probabilidade: uma vez que k ocorre de certeza numa das N posições, existem N possibilidades diferentes, e a probabilidade de ocorrer num qualquer segmento do array de comprimento l é $\frac{l}{N}$.

Por outro lado, o **condicionamento agora é importante**: o facto de k não ocorrer nas $i - 1$ primeiras posições implica que ocorra necessariamente nas $N - i + 1$ últimas posições. A probabilidade condicionada de ocorrer na posição i é então $\frac{1}{N-i+1}$, pelo que a probabilidade de serem feitas i comparações é dada por

$$\frac{N-i+1}{N} * \frac{1}{N-i+1} = \frac{1}{N}$$

Note-se que esta probabilidade condicionada tem o valor $\frac{1}{N-i+1}$ apenas porque se sabe que k ocorre **exactamente** uma vez. O cenário em que k ocorre pelo menos uma vez não permite simplificação, sendo tratado no caso geral.

Exemplo: algoritmo de incremento de um array de bits

O exemplo que acabamos de ver ilustra uma situação comum, em que o caso médio do tempo de execução coincide com o pior caso. Vejamos agora um exemplo em que isso não se verifica.

Recorde a operação de incremento de um número inteiro, representado como array de bits. O primeiro 0 do array, isto é, aquele que se encontra na posição “menos significativa”, deverá passar a 1, sendo que todos os 1s que se encontram antes dele (i.e. em posições menos significativas) passarão a 0. Por exemplo:

- incrementar o bitvector 10101000 resulta em 10101001
- incrementar o bitvector 01010111 resulta em 01011000

```
void inc (int b[], int N) {  
    int i = 0;
```

```

while ((i < N) && (b[i] == 1)) {
    b[i] = 0;
    i++;
}
if (i<N) b[i] = 1;
}

```

Naturalmente, assumiremos que todos os bitvectors (inputs) possíveis ocorrem com igual probabilidade.

Consideremos como operação primitiva relevante o número de *bit flips* efectuados correspondentes a *passagens do valor de um bit de 1 a 0 ou de 0 a 1*.

Este número varia entre os casos extremos:

- 1 bit flip quando o *bit menos significativo (índice 0)* é 0, e
- N bit flips quando os *N-1 bits menos significativos têm valor 1*.

Para calcular o valor esperado do número de bit flips teremos que efectuar a habitual soma pesada, mas tendo em conta que **a probabilidade não é agora a mesma para os diversos casos**. Assim,

- metade dos bitvectors de comprimento N têm o bit menos significativo 0;
- dos restantes, metade têm o segundo bit menos significativo 0, i.e. termina em 01;
- dos restantes, metade têm o segundo bit menos significativo 0, i.e. termina em 001;
- e assim sucessivamente.

Temos então

$$T(N) = 1 \text{ flip} * \frac{1}{2} + 2 \text{ flips} * \frac{1}{4} + 3 \text{ flips} * \frac{1}{8} + \dots + N * \frac{1}{2^N} + N * \frac{1}{2^N}$$

Dos últimos dois termos iguais a $N * \frac{1}{2^N}$, o primeiro corresponde à situação em que todos os bits são 1 excepto o mais significativo, e o segundo à situação em que todos são 1. Em ambos os casos são feitos N bit flips. Por exemplo, com 4 bits, haverá 4 bit flips quer com 0111 → 1111 quer com 1111 → 0000.

$$T(N) = \sum_{k=1}^N k * \frac{1}{2^k} + N * \frac{1}{2^N}$$

E logo, uma vez que $\sum_{k=1}^{\infty} k/2^k = 2$,
 $T(N) < 2 = O(1)$

Trata-se pois de um algoritmo cujo comportamento é, no caso médio, assimptoticamente igual ao comportamento no melhor caso.

Em alternativa a este método baseado em contagem, quem tiver familiaridade com a noção de probabilidade condicionada pode alternativamente calcular a probabilidade $\frac{1}{2^k}$ de ocorrerem k bit flips da seguinte forma:

probabilidade de os $k - 1$ bits menos significativos serem 1

*

probabilidade de o bit k ser 0, condicionada ao facto de os $k - 1$ menos significativos serem 1

Ora, o efeito do condicionamento é aqui irrelevante (os valores dos bits são todos independentes), pelo que temos

$$(\frac{1}{2})^{k-1} * \frac{1}{2} = \frac{1}{2^k}$$

(ignora-se aqui o caso adicional de $k=N$ com todos os bits a 1, explicado em cima)

EXERCÍCIO: algoritmo **Insertion sort**

```
void insertionSort(int A[], int N) {  
    int i, j, key;  
    for (j=1 ; j<N ; j++) {  
        key = A[j];  
        i = j-1;  
        while (i>=0 && A[i]>key) {  
            A[i+1] = A[i];  
            i = i-1;  
        }  
        A[i+1] = key;  
    }  
}
```

```

    i--;
}
A[i+1] = key;
}
}

```

RESOLUÇÃO

Relembremos: o número de iterações do ciclo exterior é fixo ($N-1$), e para cada valor de j , o número de vezes que a condição $(i \geq 0 \text{ } \&\& \text{ } A[i] > key)$ do ciclo interior é avaliada (que é a operação considerada anteriormente para a análise) pode variar entre 1 e $j+1$. Se escrevermos n_j para designar este número, o número de operações relevantes efectuadas pode ser escrito como

$$T(N) = \sum_{j=1}^{N-1} n_j, \text{ com } 1 \leq n_j \leq j + 1.$$

Para calcularmos o *valor esperado* deste número, temos mais uma vez que assumir *aleatoriedade* no preenchimento do array, o que implicará que a probabilidade de a inserção ser feita em qualquer posição do segmento já ordenado do array é a **mesma**. Temos $j + 1$ situações diferentes, pelo que o valor deste probabilidade é $\frac{1}{j+1}$.

Sendo assim, o valor esperado do número de vezes que a condição $i \geq 0 \text{ } \&\& \text{ } A[i] > key$ é avaliada, para um determinado valor de j , é

$$n_j = \sum_{k=1}^{j+1} \frac{1}{j+1} k = \frac{1}{j+1} \sum_{k=1}^{j+1} k = \frac{1}{j+1} \frac{(j+1)(j+2)}{2} = \frac{j}{2} + 1.$$

Note-se que, este valor esperado a que chegamos vai de encontro à seguinte intuição:

Uma vez que assumimos que o array foi preenchido aleatoriamente, então em cada iteração do ciclo exterior, de entre os elementos do segmento ordenado do array (entre as posições 0 e $j-1$), metade dos elementos é superior a $A[j]$.

Ou seja, “em média”, a inserção será feita a meio do segmento já ordenado do array.

Considerando agora o tempo global de execução do algoritmo, temos assim

$$T(N) = \sum_{j=1}^{N-1} \left(\frac{j}{2} + 1\right)$$

$$T(N) = \frac{1}{2} \frac{(N-1)N}{2} + N - 1 = \frac{1}{4}N^2 + \frac{3}{4}N - 1 = \Theta(N^2)$$

O comportamento no caso médio é asymptoticamente quadrático (tal como no pior caso). Trata-se de uma conclusão importante, uma vez que o comportamento de melhor caso (linear) é muito promissor — em média, o comportamento do *insertion sort* não é melhor do que o do *selection sort*.

EXERCÍCIO: algoritmo de procura num array ordenado

```
// requires v ordenado de forma crescente entre os índices 0 e N-1
int procuraOrd (int v[], int N, int k) {
    int i = 0;
    while ((i<N) && (v[i]<k))
        i++;
    if (i>N-1 || v[i]!= k)
        return -1;
    else return i;
}
```

Calcule o caso médio do tempo de execução deste algoritmo, estimando o valor esperado do número de comparações (<). Para isso, assuma que o array ordenado está preenchido de forma aleatória com valores dentro de uma dada gama, e que k é escolhido aleatoriamente dentro dessa gama.

T3. Análise do Tempo de Execução de Algoritmos Recursivos

Algoritmos com uma chamada recursiva

Exemplo: contagem de ocorrências num array

Relembremos a função `conta` de [+T1. Tempo de Execução de Algoritmos Iterativos](#):

```
int conta (int k, int v[], int N) {  
    int i = 0, r = 0;           c1      1  
  
    while (i < N) {           c2      N+1  
        if (v[i] == k) r++;   c3      N  
        i++;                 c4      N  
    }  
    return r;                 c5      1  
}
```

É imediato escrever uma versão recursiva desta função:

```
int conta (int k, int v[], int N) {  
    int r;  
    if (N == 0) r = 0;  
    else {  
        r = conta(k, v+1, N-1);  
        if (v[0] == k) r++;  
    }  
    return r;  
}
```

Note-se que a invocação `conta(k, v+1, N-1)` chama recursivamente a função, passando-lhe o array de comprimento $N-1$ com início na posição 1 do array v (i.e., a "cauda" de v).

Ao analisarmos o tempo de execução desta função deparamo-nos com uma dificuldade: a própria função que caracteriza o tempo de execução terá que ter uma definição recursiva, uma vez que $T(N)$ dependerá necessariamente de $T(N - 1)$.

A análise de algoritmos recursivos requer pois a utilização de um instrumento que permita exprimir o tempo de execução sobre um input de tamanho N em função do tempo de execução sobre inputs de tamanhos inferiores. Esse instrumento são as *equações de recorrência*, estudadas pela primeira vez por Fibonacci, no início do século XIII.

A recorrência que caracteriza o tempo de execução do algoritmo acima é a seguinte:

$$T(N) = \Theta(1), \text{ se } N = 0$$

$$T(N) = T(N - 1) + \Theta(1), \text{ se } N > 0$$

A primeira cláusula corresponde ao caso de paragem da função, quando $N=0$, em que apenas são executadas operações de tempo constante. No caso recursivo, é feita uma invocação da função sobre um input de tamanho $N-1$, e ainda operações de tempo constante, em particular `if (v[i] == k) r++`.

Como resolver a recorrência acima? Podemos simplesmente expandir a definição:

$$\begin{aligned} T(N) &= T(N - 1) + \Theta(1) \\ &= T(N - 2) + \Theta(1) + \Theta(1) \\ &= \dots \\ &= T(0) + \Theta(1) + \dots + \Theta(1) \\ &= (N + 1) * \Theta(1) \\ &= \Theta(N) \end{aligned}$$

Como seria de esperar, o tempo de execução da versão recursiva do algoritmo de contagem de ocorrências é igual ao da versão iterativa.

Nota

Poderíamos igualmente escrever uma recorrência para o cálculo preciso (i.e. não assintótico) do número de execuções de uma determinada operação. Por exemplo a recorrência para o número de operações $v[0] == k$ executadas seria:

$$T(N) = 0, \text{ se } N = 0$$

$$T(N) = T(N - 1) + 1, \text{ se } N > 0$$

com solução $T(N) = N$.

EXERCÍCIO: Apresente uma versão alternativa da função `conta`, ainda recursiva, mas utilizando um *acumulador*. Será que o tempo de execução da função que escreveu pode ser descrito pela mesma recorrência da anterior?

EXERCÍCIO: Relembre a função de identificação de duplicados num array de [+T1. Tempo de Execução de Algoritmos Iterativos](#). Escreva uma versão recursiva desta função e uma recorrência que caracterize o seu tempo de execução. Resolva essa recorrência, expandindo-a como no exemplo acima.

```
void dup_rec (int v[], int a, int b) {
    if (a>=b) return;
    for (t=a+1; t<=b ; t++)
        if (v[a]==v[t]) printf("%d igual a %d\n", v, t);
    dup_rec (v, a+1, b);
}
```

$$T(N) = 0, \text{ se } N \leq 1$$

$$T(N) = N-1 + T(N-1), \text{ se } N > 1$$

$$T(N) = N-1 + N-2 + N-3 + \dots + 1 + T(1)$$

$$= 1 + 2 + 3 + \dots + N-2 + N-1$$

$$T(N) = \sum_{k=1}^{N-1} k = \frac{(N-1)N}{2} = \theta(N^2)$$

```
void aux (int v[], int i, int j, int x) {
```

```

for (t=i; t<=j ; t++)
    if (v[x]==v[t]) printf("%d igual a %d\n", x, t);
}

Taux (N) = N

void dup_rec_2 (int v[], int a, int b) {
    if (a<b) {
        aux(v, a+1, b, a);
        dup_rec_2 (v, a+1, b);
    }
}

```

$T(N) = 0$, se $N \leq 1$
 $T(N) = Taux(N-1) + T(N-1)$, se $N > 1$
 $= N-1 + T(N-1)$, se $N > 1$

```

void aux_rec (int v[], int i, int j, int x) {
    if (i>j) return;
    if (v[x]==v[i]) printf("%d igual a %d\n", i, t);
    aux_rec(v, i+1, j, x);
}

Taux_rec(N) = 0, se N=0
Taux_rec(N) = 1 + Taux_rec(N-1), se N>0

Taux_rec(N) = 1+1+...+1 = N

void dup_rec_3 (int v[], int a, int b) {

```

```

    if (a<b) {
        aux_rec(v, a+1, b, a);
        dup_rec_3 (v, a+1, b);
    }
}

T(N) = 0, se N<=1
T(N) = Taux_rec(N-1) + T(N-1), se N>1
      = N-1 + T(N-1), se N>1

```

O exemplo anterior caracteriza-se por ter o mesmo comportamento no melhor e no pior caso. Vejamos agora como analisar *algoritmos recursivos com diferentes casos* no que diz respeito ao tempo de execução.

Exemplo: procura linear num array

Relembremos o algoritmo de procura linear estudado em [+T2. Análise de Pior Caso, Melhor Caso, e Caso Médio:](#)

```

int procura(int v[], int a, int b, int k) {
    int i = a;
    while ((i<=b) && (v[i]!=k))
        i++;
    if (i>b)
        return -1;
    else return i;
}

```

Uma versão recursiva pode ser escrita como se segue, fazendo avançar o índice inferior `a` ao longo do array.

```

int procura(int v[], int a, int b, int k) {
    if (a > b) return -1;
    if (v[a]==k) return a;
    return procura(v, a+1, b, k);
}

```

Antes de mais note-se que este algoritmo recursivo tem dois casos de paragem:

- O caso $(a > b)$, que corresponde ao array vazio, e que será executado caso k não seja encontrado entre os índices a e b;
- O caso $(v[a]==k)$, que será executado quando k é encontrado na posição a do array.

Não é possível definir uma recorrência que entre em linha de conta com este segundo caso de paragem, uma vez que ele não depende do valor de N, mas sim do conteúdo do array. É no entanto possível escrever *recorrências específicas para a análise do tempo de execução no pior e no melhor caso*.

O **pior caso** de execução acontece quando k não ocorre no array, e é caracterizado pela seguinte recorrência (o caso de paragem é o primeiro que identificámos acima):

$$T_p(N) = \Theta(1), \text{ se } N = 0$$

$$T_p(N) = T_p(N - 1) + \Theta(1), \text{ se } N > 0$$

Em que $N = b - a + 1$. Note-se que no caso recursivo é feita uma invocação da função sobre um input de tamanho N-1, sendo também executadas operações de tempo constante. Esta recorrência, que é igual à do primeiro exemplo que vimos, tem solução

$$T_p(N) = \Theta(N).$$

Quanto ao **melhor caso**, ele ocorre quando k se encontra na posição a do array, e neste caso a recorrência “degenera” na seguinte definição não-recorrente, uma vez que o caso de paragem é imediatamente executado:

$$T_m(N) = \Theta(1)$$

A análise da versão recursiva revela que o algoritmo executa em tempo constante no melhor caso e linear no pior, tal como a versão iterativa.

Algoritmos de Divisão e Conquista

Os exemplos que vimos anteriormente são algoritmos muito simples que admitem versões iterativas e recursivas. No entanto, a recursividade é um instrumento fundamental na definição de algoritmos baseados numa estratégia algorítmica específica: os algoritmos de **divisão e conquista**. Trata-se aqui de algoritmos cuja definição sem a utilização de recursividade não é de todo trivial.

Um algoritmo recursivo de divisão e conquista tem a seguinte estrutura típica:

1. **Divisão** do problema em n sub-problemas
2. Resolução recursiva dos n sub-problemas (passo de **Conquista**)
3. **Combinação** das soluções dos sub-problemas para obter a solução do problema inicial

O caso de paragem ocorre normalmente no caso de inputs muito pequenos.

Em geral, o tempo de execução de um algoritmo de divisão e conquista será caracterizado por uma recorrência com a seguinte forma:

$$T(N) = \Theta(1), \text{ se } N \leq k$$

$$T(N) = D(N) + aT(N/a) + C(N), \text{ se } N > k$$

Em que cada *divisão* gera a sub-problemas, sendo o tamanho de cada sub-problema uma fração $1/a$ do original (ou próximo disso); k é o tamanho dos problemas com solução trivial; e D e C são funções que caracterizam o tempo das operações de *divisão* e *combinação*, respectivamente.

Exemplo: algoritmo Merge Sort

Neste algoritmo de ordenação bem conhecido a estrutura de divisão e conquista é instanciada da seguinte forma:

1. **Divisão** do array em duas partes de tamanho igual (a menos de uma unidade, no caso de o comprimento ser ímpar)
2. **Conquista**: ordenação recursiva dos dois vectores
3. **Combinação**: fusão dos dois vectores ordenados.

Este último passo será implementado por uma função auxiliar `merge`. A definição seguinte assume que são declarados globalmente dois arrays auxiliares `L` e `R` com tamanho suficiente para armazenar temporariamente os elementos das duas partes ordenadas de `A` que se pretende fundir. Será colocada uma *sentinela* (o valor do maior inteiro representável) no final dos arrays `L` e `R`, o que simplificará o algoritmo.

A primeira parte ordenada de `A` está contida entre os índices `p` e `q`; a segunda está contida entre os índices `q+1` e `r`.

```
void merge(int A[], int p, int q, int r) {
    int n1 = q-p+1, n2 = r-q;
    for (i=0 ; i<n1 ; i++) L[i] = A[p+i];      // L[], R[] globais
    for (j=0 ; j<n2 ; j++) R[j] = A[q+j+1];
    L[n1] = INT_MAX; R[n2] = INT_MAX;

    i = 0; j = 0;
    for (k=p ; k<=r ; k++)
        if (L[i] <= R[j]) {
            A[k] = L[i]; i++;
        } else {
            A[k] = R[j]; j++;
        }
}
```

O passo básico do algoritmo compara dois valores contidos nas primeiras posições de ambos os arrays, colocando o menor dos valores no array resultado. Este passo é executado em *tempo constante*, e são sempre executados N passos básicos, com $N = r - p + 1$. O algoritmo executa então em tempo $C(N) = \Theta(N)$.

Depois de definida a função de fusão ordenada, o algoritmo de ordenação é facilmente implementado através de uma função recursiva como a seguinte, que ordena o array entre os índices p e r .

```
void merge_sort(int A[], int p, int r) {  
    if (p < r) {  
        q = (p+r)/2;  
        merge_sort(A, p, q);  
        merge_sort(A, q+1, r);  
        merge(A, p, q, r);  
    }  
}
```

Sendo a invocação inicial, para ordenar um array de comprimento N ,

```
merge_sort(A, 0, N-1).
```

O tempo de execução pode ser caracterizado pela seguinte recorrência. Note-se a utilização dos operadores de arredondamento para captar o efeito da divisão inteira
 $q = (p+r)/2$.

$$T(N) = \Theta(1), \text{ se } N = 1$$

$$T(N) = T\lceil N/2 \rceil + T\lfloor N/2 \rfloor + \Theta(N), \text{ se } N > 1$$

em que o termo $\Theta(N)$ corresponde ao tempo de execução da função de fusão ordenada.

Mais uma vez resolveremos a recorrência começando por expandi-la. Mas vamos para isso considerar uma simplificação que permitirá simplificar a sua análise: admitiremos que N é uma potência de 2, o que significa que todos os valores tomados por esta variável na expansão da recorrência serão pares, e por essa razão os operadores de arredondamento podem ser dispensados. Além disso, escreveremos sem perda de generalidade o termo de tempo linear como cN e o termo constante como c .

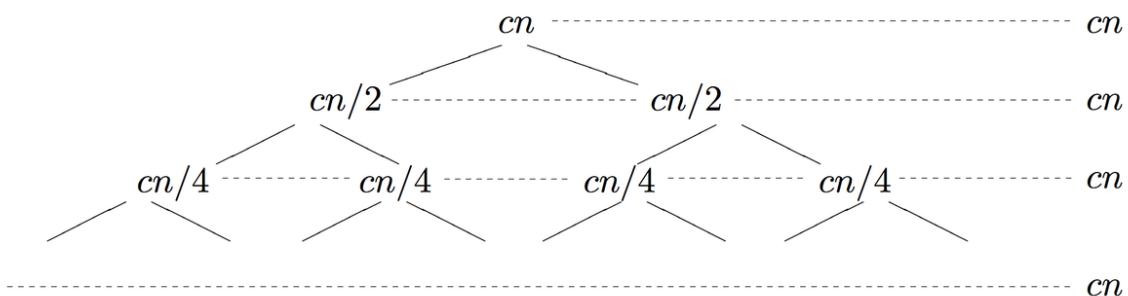
$$T(N) = c, \text{ se } N = 1$$

$$T(N) = 2T(N/2) + cN, \text{ se } N > 1$$

Efectuemos então a expansão:

$$\begin{aligned} T(N) &= 2T(N/2) + cN \\ &= 4T(N/4) + 2c(N/2) + cN = 4T(N/4) + 2cN \\ &= 8T(N/8) + 3cN \\ &= \dots \\ &= NT(1) + (\log N)cN \\ &= Nc + (\log N)cN \\ &= cN \log N + cN \\ &= \Theta(N \log N) \end{aligned}$$

Visualmente podemos compreender a execução do algoritmo através de uma árvore de recursividade em que todos os níveis contribuem para o tempo de execução com o mesmo custo cN , tendo a árvore $\log N + 1$ níveis (o último dos quais corresponde à execução dos casos de paragem, sobre os arrays singulares).



Em conclusão, o algoritmo merge sort executa em tempo $T(N) = \Theta(N \log N)$, sem distinção de casos.

Métodos de resolução de recorrências

Não existe um método universal para resolução de recorrências. No entanto, muitas das recorrências que caracterizam o tempo de execução de algoritmos de divisão e

conquista podem ser resolvidas pelo chamado *Master theorem*, que não estudaremos aqui.

https://en.wikipedia.org/wiki/Master_theorem

Adicionalmente, é possível *provar* indutivamente que uma determinada pseudo-solução (por exemplo calculada informalmente por expansão da recorrência, como nos exemplos anteriores) é de facto uma verdadeira solução de uma recorrência. O método de prova que se utiliza para isto designa-se por *método de substituição*, e veremos um exemplo da sua aplicação em [+T4. Tópicos sobre Algoritmos de Ordenação \(?\)](#).

Exercícios Adicionais

1. Para cada uma das seguintes recorrências, apresente um exemplo de um algoritmo cujo tempo de execução seja caracterizado por ela, e resolva-a. k é uma constante; assuma também que $T(1)$ é constante (caso de paragem).
 - a. $T(N) = k + T(N - 1)$
 - b. $T(N) = N + T(N - 1)$
 - c. $T(N) = k + T(N/2)$
 - d. $T(N) = k + 2 * T(N/2)$
 - e. $T(N) = N + 2 * T(N/2)$
 - f. $T(N) = N + T(N/2)$

```
int procuraBin (int v[], int a, int b, int k) {  
    if (a>b) result = -1;  
    else {  
        m = a + (b-a) / 2;  
        if (vector[m] < k) result = procuraBin (v, m+1, b, k);  
        else if (vector[m] > k) result = procuraBin(v, a, m-1, k);  
        else result = m;  
    }  
}
```

```

    }

    return result;
}

```

$$T(16) = k + T(8) = 2k + T(4) = 3k + T(2) = 4k + T(1) = 4k$$

$$T(N) = \sum_{i=1}^{\log N} k = \theta(\log N)$$

2. Considere o seguinte algoritmo para o problema conhecido por *Torres de Hanói*:

```

void Hanoi(int nDiscos, int esquerda, int direita, int meio)
{
    if (nDiscos > 0) {
        Hanoi(nDiscos-1, esquerda, meio, direita);
        printf("mover disco de %d para %d\n", esquerda, direita);
        Hanoi(nDiscos-1, meio, direita, esquerda);
    }
}

```

- a. Escreva uma relação de recorrência que exprima a complexidade deste algoritmo (por exemplo, em função do número de linhas impressas).
- b. Desenhe a árvore de recursão do algoritmo e obtenha a partir dessa árvore um resultado sobre a sua complexidade assintótica.

3. Considere a seguinte versão recursiva do algoritmo *insertion sort*.

```

void isort (int v[], int N) {
    int i; int t;
    if (N>1) {
        isort (v+1, N-1);
        i = 0;
    }
}

```

```

t = v[0];
while (i<N-1 && v[i]<t) {
    v[i] = v[i+1];
    i++;
}
if (i>0) v[i] = t;
}
}

```

- Identifique o melhor e pior casos de execução desta função.
- Para esses casos, apresente uma relação de recorrência que traduza o *número de comparações entre elementos do vector* em função do tamanho do vector.

$$T_{mc}(N) = 1 + T(N - 1) = N - 1$$

$$T_{pc}(N) = N - 1 + T(N - 1) = \frac{(N-1)N}{2}$$

- Considere o seguinte algoritmo para o cálculo dos números de Fibonacci.
Assuma que as operações aritméticas elementares se efectuam em tempo $\Theta(1)$.

```

int fib (int n)
{
    if (n==0 || n==1) return 1;
    else return fib(n-1) + fib(n-2);
}

```

- Escreva uma recorrência que descreva o comportamento temporal do algoritmo. Desenhe a respectiva árvore de recursão para $n = 5$.
- Efectue uma análise assimptótica do tempo de execução deste algoritmo.
- Apesar de traduzir exactamente a definição da sequência de números de Fibonacci, este algoritmo é muito ineficiente, como deverá ter concluído na alínea anterior. Escreva em C um algoritmo alternativo eficiente e analise o seu tempo de execução.

T4. Tópicos sobre Algoritmos de Ordenação

Algoritmo de ordenação Quicksort

Trata-se de um algoritmo de *divisão e conquista*, tal como o merge sort.

1. **Divisão:** *partição* do vector $A[p..r]$ em dois sub-vectores

- $A[p..q-1]$ e
- $A[q+1..r]$

tais que todos os elementos do primeiro (resp. segundo) são $\leq A[q]$ (resp. $\geq A[q]$).

Uma função auxiliar de partição recebe a sequência $A[p..r]$, executa a sua partição “in place” usando o último elemento do vector como *pivot*, e devolve o índice q . Note-se que um dos sub-vectores pode ser vazio.

2. **Conquista:** ordenação recursiva dos dois vectores

3. **Combinação:** nada a fazer!

Enquanto no *merge sort* o trabalho era feito na fase de combinação (fusão ordenada), aqui é feito na fase de divisão (partição), que claramente executa em tempo $\Theta(N)$. Vejamos uma implementação possível:

```
int partition (int A[], int p, int r)
{
    x = A[r];
    i = p-1;
    for (j=p ; j<r ; j++)
        if (A[j] <= x) {
            i++;
            swap(A, i, j);
        }
}
```

```
    swap(A, i+1, r);
    return i+1;
}
```

EXERCÍCIO: Identifique um invariante apropriado para o ciclo desta função de partição.

Tal como no merge sort, depois de definida a função auxiliar, o algoritmo de ordenação é facilmente implementado. A seguinte função ordena o array entre os índices p e r.

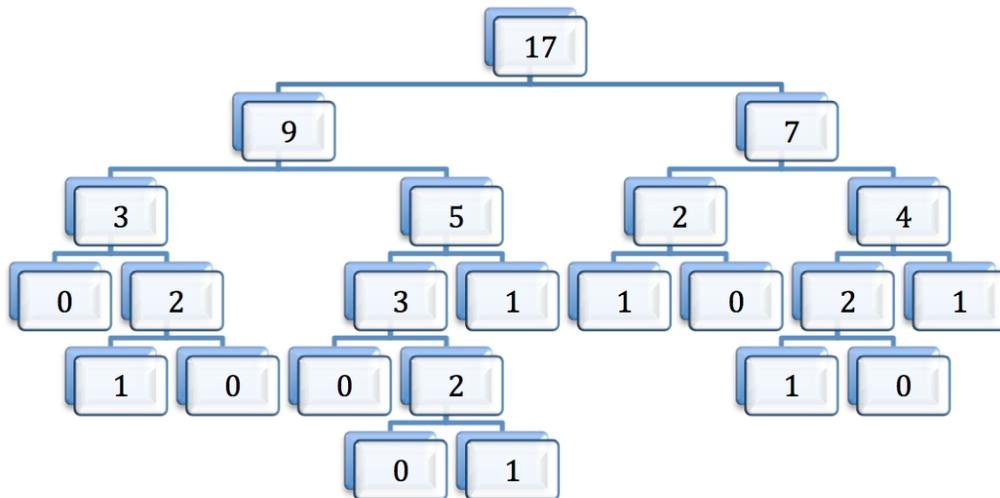
```
void quicksort(int A[], int p, int r)
{
    if (p < r) {
        q = partition(A, p, r);
        quicksort(A, p, q-1);
        quicksort(A, q+1, r);
    }
}
```

Vejamos uma simulação de execução do algoritmo. Mostra-se:

- a azul os elementos usados como pivots
- a verde, elementos que já foram pivot e foram colocados na posição final
- a vermelho, elementos que são casos de paragem (função chamada com um só elemento)

7	6	12	3	11	8	2	1	15	13	17	5	16	14	9	4	10
7	6	3	8	2	1	5	9	4	10	17	11	16	14	12	15	13
3	2	1	4	6	7	5	9	8	10	11	12	13	14	17	15	16
1	2	3	4	6	7	5	8	9	10	11	12	13	14	15	16	17
1	2	3	4	5	7	6	8	9	10	11	12	13	14	15	16	17
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Ao contrário do algoritmo merge sort, a árvore de recursividade do quicksort não tem a mesma forma para todos os arrays de entrada, dependendo completamente deste. Para o exemplo anterior teremos a seguinte árvore (os nós estão etiquetados com o **comprimento** do vector em cada invocação):



Análise intuitiva

Intuitivamente, o comportamento de pior caso do algoritmo ocorre quando a operação de partição produz o resultado *mais desequilibrado* possível, i.e. quando um dos vectores resultantes é vazio, e o outro contém $N-1$ elementos (todos excepto o pivot). Quando isto acontece em todas as invocações da função de partição, a execução é caracterizada pela seguinte recorrência, em que o termo $\Theta(N)$ corresponde ao tempo de execução da função de partição:

$$T_p(N) = \Theta(1), \text{ se } N \leq 1$$

$$T_p(N) = T_p(N - 1) + \Theta(N), \text{ se } N > 1$$

que tem como solução $T_p(N) = \Theta(N^2)$. Note-se que isto ocorre quando o array se encontra à partida ordenado de forma crescente (ou decrescente)!

[recordar que o algoritmo *insertion sort*, que executa também em tempo quadrático no pior caso, executa em tempo $\Theta(N)$ quando o array de entrada está já ordenado de forma crescente.]



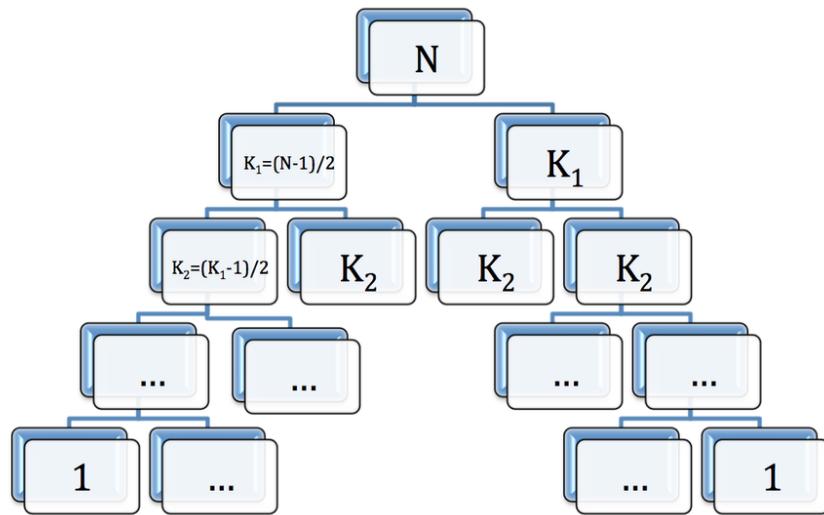
Árvore de recursividade correspondente ao pior caso de quicksort

Quanto ao melhor caso, ele ocorre quando, em todas as execuções da função de partição, ela produz o resultado *mais equilibrado* possível, i.e. quando ambos os vectores resultantes têm comprimento aproximado $\frac{N-1}{2}$. Em termos mais rigorosos a execução do algoritmo será então caracterizada pela seguinte recorrência:

$$T_m(n) = \Theta(n) + T_m(\lfloor n/2 \rfloor) + T_m(\lceil n/2 \rceil - 1)$$

Trata-se de uma recorrência muito semelhante à do algoritmo merge sort, com a mesma solução

$$T_m(N) = \Theta(N \log N).$$



Árvore de recursividade correspondente ao melhor caso de quicksort

Análise de pior caso

Como poderemos obter uma prova de que as nossas intuições acima estão de facto correctas? Ou seja, de que de facto o comportamento de *quicksort* é caracterizado por $T(N) = \Omega(N \log N), O(N^2)$?

Concentremo-nos na análise de pior caso. Observemos antes de mais que podemos descrever o tempo de execução no pior caso de uma forma rigorosa, utilizando para isso um operador de *maximização* sobre a soma do tempo das duas invocações recursivas, em ordem ao comprimento k de um dos vectores resultantes da partição (note-se que o outro vector terá comprimento $N-k-1$).

$$T_p(N) = \Theta(N) + \max_{k=0}^{N-1} (T_p(k) + T_p(N - k - 1))$$

Para mostrarmos que esta recorrência tem a mesma solução que a que escrevemos acima de forma intuitiva, utilizaremos o **método da substituição**.

Admitamos então que $T_p(N) \leq cN^2$ para uma determinada constante c . Então podemos aplicar as seguintes hipóteses de indução:

- $T_p(k) \leq ck^2$
- $T_p(N - k - 1) \leq c(N - k - 1)^2$

e logo,

$$T_p(N) \leq \Theta(N) + \max (ck^2 + c(N - k - 1)^2)$$

$$T_p(N) \leq \Theta(N) + c \max(P(k))$$

$$\text{com } P(k) = k^2 + (N - k - 1)^2 = 2k^2 + (2 - 2N)k + (N - 1)^2$$

Ora, por análise de $P(k)$ conclui-se que os seus máximos no intervalo $0 \leq k \leq N - 1$ se encontram nas extremidades, para $k = 0$ (primeiro vector é vazio) e $k = N - 1$ (segundo vector é vazio).

Para estes valores de k temos $P(0) = P(N - 1) = (N - 1)^2$, logo
 $T_p(N) \leq \Theta(N) + c(N - 1)^2$

E temos então que $T_p(N) = \Theta(N^2)$, o que conclui a prova pelo método de substituição.

Análise de Caso Médio

O caso médio do tempo de execução pode ser estimado através do valor esperado do número de comparações efectuadas entre elementos do vector:

$$T_{avg}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} P_{comp}(i,j)$$

Admitamos para simplificar que lidamos com sequências que não contêm elementos repetidos.

Comecemos por observar que, dados quaisquer dois elementos x e y , eles começam por ser mantidos no mesmo sub-vector, enquanto os pivots forem superiores ou inferiores a ambos.

Até que ocorrerá um de dois cenários:

1. Um dos elementos é usado como *pivot* na partição de um sub-array que contém o outro elemento. É o caso de (2,4) ou (4,7) no exemplo de execução. Neste caso os elementos **são comparados**.
1. Os elementos são *separados* por uma qualquer partição em que é usado um terceiro elemento z como pivot. É o caso do par (2,7) no exemplo.

Neste caso **não são comparados**. Note-se que o pivot z terá de ser um elemento do vector tal que $x < z < y$.

7	6	12	3	11	8	2	1	15	13	17	5	16	14	9	4	10
7	6	3	8	2	1	5	9	4	10	17	11	16	14	12	15	13
3	2	1	4	6	7	5	9	8	10	11	12	13	14	17	15	16
1	2	3	4	6	7	5	8	9	10	11	12	13	14	15	16	17
1	2	3	4	5	7	6	8	9	10	11	12	13	14	15	16	17
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Conhecendo o conjunto de elementos contidos no array, é possível calcular a probabilidade de qualquer par de elementos (x, y) ser ou não comparado durante a execução do algoritmo: sendo n o número de elementos z tais que $x < z < y$, a probabilidade de x e y serem comparados é $\frac{2}{2+n}$.

No exemplo, a probabilidade de 2 e 7 serem comparados é igual a $2/6 = 1/3$.

Ora, uma forma de conhecemos este número n de elementos contidos entre um par de elementos do array, consiste em observar o array final, ordenado. Se no array ordenado x e y se encontram nas posições i e j , então $n = j - i - 1$, e a probabilidade de terem sido comparados é dada por $\frac{2}{j-i-1+2} = \frac{2}{j-i+1}$.

Basta agora calcular a soma destas probabilidades para todos os pares de elementos:

$$T_{avg}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} \frac{2}{j-i+1}$$

Fazendo uma mudança de variável:

$$T_{avg}(n) = \sum_{i=0}^{n-2} \sum_{k=2}^{n-i} \frac{2}{k}$$

$$T_{avg}(n) \leq \sum_{i=0}^{n-2} \sum_{k=2}^n \frac{2}{k} = 2(n-1)(\sum_{k=1}^n \frac{1}{k} - 1)$$

Podemos agora utilizar o resultado seguinte:

$$\sum_{k=1}^n \frac{1}{k} \leq 1 + \ln n$$

Para obter

$$T_{avg}(n) = O(n \lg n)$$

Nesta análise assumimos, naturalmente, que todas as permutações dos elementos da sequência podem ocorrer com igual probabilidade (ou, o que é equivalente, que estamos a analisar uma versão *aleatorizada* do algoritmo).

Algoritmos de ordenação baseados em comparações

Todos os algoritmos estudados até aqui são baseados em **comparações**: dados dois elementos $A[i]$ e $A[j]$, é efectuado um teste (e.g. $A[i] \leq A[j]$) que determina a ordem relativa desses elementos, não sendo usado qualquer outro método para obter informação sobre o valor dos elementos a ordenar.

Admitamos que a sequência não contém elementos repetidos. O **conjunto de execuções** de um algoritmo baseado em comparações (sobre sequências de uma determinada dimensão) pode ser visto de forma abstracta como constituindo uma *Árvore de Decisão*: uma árvore binária cujos **caminhos descendentes**, desde a raiz até às folhas, correspondem às diferentes execuções do algoritmo, como se segue:

- cada nó contém uma *condição*, correspondente a uma **comparação** entre dois elementos, $A[i] \leq A[j]$
- os **caminhos descendentes que chegam** a este nó correspondem às execuções que efectuam esta comparação $A[i] \leq A[j]$
- os caminhos que continuam para a **sub-árvore esquerda** deste nó correspondem às execuções em que o teste $A[i] \leq A[j]$ teve resposta **verdadeira**
- os caminhos que continuam para a **sub-árvore direita** deste nó correspondem às execuções em que o teste $A[i] \leq A[j]$ teve resposta **falsa**

- assim, cada caminho da raiz até uma folha contém **a sequência de comparações efectuadas numa execução** concreta do algoritmo

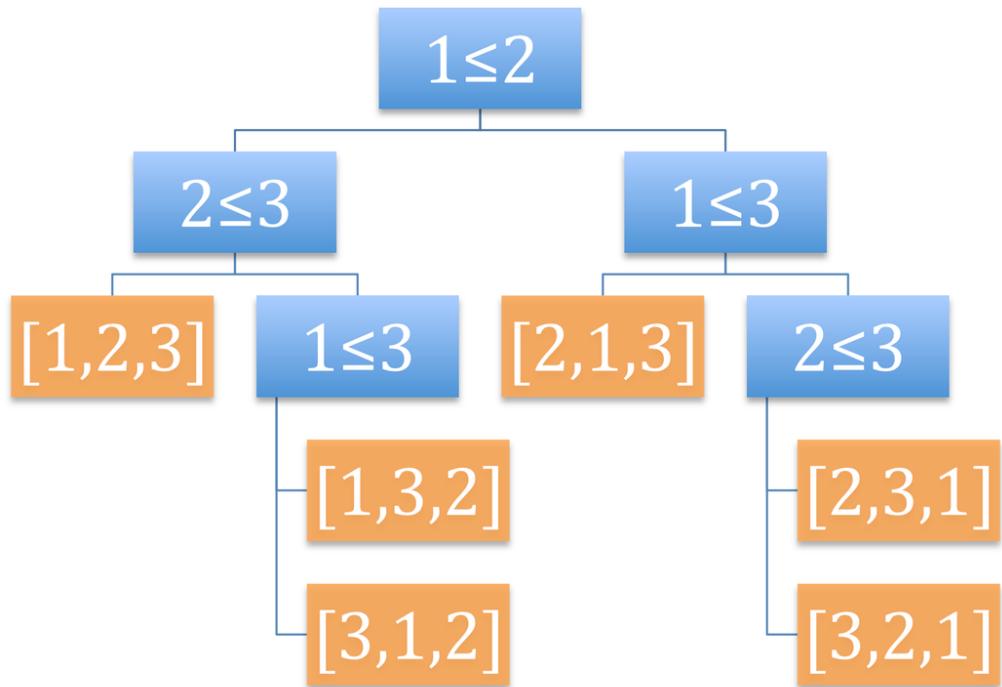
Note-se ainda que:

- Cada folha corresponde a uma ordenação possível do input, ou seja uma permutação possível da sequência inicial
- **Todas** as permutações da sequência devem aparecer como folhas, já que a árvore contempla todas as execuções
- Existem $N!$ permutações

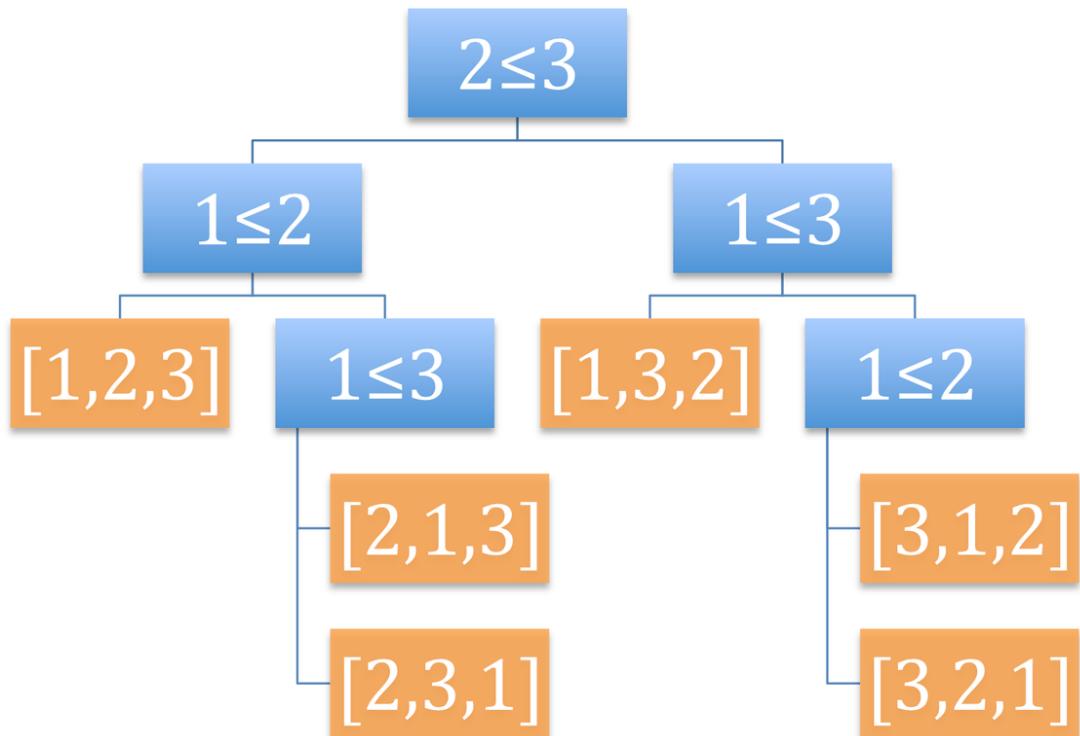
Assim:

O número de folhas da árvore de decisão de um algoritmo de ordenação de um array de comprimento N é igual a $N!$

Vejamos dois exemplos: a execução dos algoritmos **insertion sort** e **merge sort**, sobre inputs de comprimento 3 (o tamanho das árvores cresce exponencialmente com o comprimento dos array).



Árvore de Decisão insertion sort, $N=3$



Árvore de Decisão merge sort, $N=3$

Observe-se agora que:

- o número de operações de comparação efectuadas numa execução concreta é dado pelo **comprimento do caminho** descendente correspondente a essa execução;
- logo, o número de operações de comparação efectuadas no *pior caso* é dado pela **altura** da árvore de decisão

Teorema

A altura h de uma árvore de decisão tem o seguinte limite mínimo, em que N é o tamanho do input:

$$h \geq \lg(N!)$$

Prova

1. Em geral uma árvore binária de altura h tem **no máximo** 2^h folhas.
2. As árvores que aqui consideramos têm $N!$ folhas, correspondentes a todas as permutações do input
3. Assim, $N! \leq 2^h$
4. Logo, $\lg(N!) \leq h$

Ora, uma vez que num algoritmo de ordenação deste tipo o tempo de execução assintótico pode ser calculado tomando apenas em conta a operação de comparação, temos o seguinte

Corolário

Seja $T(N)$ o tempo de execução no pior caso de um qualquer algoritmo de ordenação baseado em comparações. Então $T(N) = \Omega(N \lg N)$

Prova

Basta usar o seguinte facto: $\lg(N!) = \Theta(N \lg N)$

Conclui-se assim que não é possível bater o comportamento de pior caso do algoritmo *merge sort*: pode-se dizer que é um algoritmo **assimptoticamente óptimo** uma vez que o seu tempo de execução no pior caso é $\Theta(N \lg N)$.

Algoritmos *Counting Sort* e *Radix Sort*

Na realidade, é possível ordenar vectores em tempo linear, batendo o limite $N \log N$, se se utilizar um método que não dependa de comparações entre elementos.

O algoritmo *counting sort* pode usado para ordenar sequências de números inteiros, se for conhecida à partida a gama de valores armazenados na sequência. Utiliza para isto um vector auxiliar para armazenar um histograma (uma contagem) dos elementos da sequência a ordenar.

A versão apresentada a seguir deste algoritmo:

- assume que os elementos do array A a ordenar estão contidos no conjunto $\{0 \dots k\}$, sendo o valor de k conhecido
- coloca a sequência ordenada no array B (não é pois um algoritmo de ordenação *in-place*).

```
void counting_sort(int A[], int B[], int N, int k) {  
    int C[k+1];  
    for (i=0 ; i<=k ; i++) /* inicialização de C[] */  
        */  
    C[i] = 0;  
  
    for (j=0 ; j<N ; j++) /* contagem ocorr. A[j] */  
        */  
    C[A[j]] = C[A[j]]+1;  
    for (i=1 ; i<=k ; i++) /* contagem dos <= i */  
        */  
    C[i] = C[i]+C[i-1];  
  
    for (j=N-1 ; j>=0 ; j--) /* construção do vector ordenado */  
        B[C[A[j]]-1] = A[j];  
        C[A[j]] = C[A[j]]-1;  
    }  
}
```

```
}
```

EXEMPLO:

```
k = 20
A = [10, 5, 20, 10, 17]
C = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0] // 1o. ciclo
C = [0, 0, 0, 0, 0, 1, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0] // 2o. ciclo
C = [0, 0, 0, 0, 0, 1, 1, 1, 1, 3, 3, 3, 3, 3, 3, 3, 4, 4, 5] // 3o. ciclo
B = [--, --, --, --, --]
B = [--, --, --, 17, --]
    // 4o. ciclo, 1 it.
C = [0, 0, 0, 0, 0, 1, 1, 1, 1, 3, 3, 3, 3, 3, 3, 3, 3, 4, 5]
B = [--, --, 10, 17, --]
    // 4o. ciclo, 2 it.
C = [0, 0, 0, 0, 0, 1, 1, 1, 1, 2, 3, 3, 3, 3, 3, 3, 3, 4, 5]
B = [--, --, 10, 17, 20]
    // 4o. ciclo, 3 it.
C = [0, 0, 0, 0, 0, 1, 1, 1, 1, 2, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4]
B = [ 5, --, 10, 17, 20]
    // 4o. ciclo, 4 it.
C = [0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 2, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4]
B = [ 5, 10, 10, 17, 20]
    // 4o. ciclo, 5 it.
C = [0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4]
```

Esta contagem permite colocar (último ciclo for) os elementos de A directamente nas suas posições finais.

A análise do tempo de execução é imediata: $T(N) = \Theta(N + k)$

Se $k = O(N)$, então $T(N) = \Theta(N)$

Propriedade de Estabilidade

Note-se que o array C:

- começa por guardar uma contagem do número de ocorrências de cada elemento no array A a ordenar
- depois do terceiro ciclo for, passa a guardar em cada posição i uma contagem do **número de elementos inferiores ou iguais a i**

Imaginemos que A contém duas ocorrências de um elemento x , nas posições a e b , sendo $a < b$, e que existem u elementos inferiores a x em A. Teremos então que $C[x] = u + 2$

O algoritmo preencherá o array B percorrendo A do final para o início, e:

1. ao passar pela posição b de A colocará x na posição $u + 2$ e decrementará $C[x]$ para $C[x] = u + 1$
2. mais tarde, ao passar pela posição a de A colocará x na posição $u + 1$ e decrementará $C[x]$ para $C[x] = u$

Sendo assim, o algoritmo counting sort observa a seguinte propriedade de *estabilidade*:

A ordem das ocorrências em A é preservada em B

Esta propriedade parece inútil quando se considera a ordenação de sequências de números, mas é de facto útil se estes números forem vistos como parte de estruturas contendo outros campos.

Considere-se por exemplo o problema de ordenação de datas. Os 3 campos presentes numa data devem ter prioridades diferentes na ordenação, tendo o *ano*

prioridade mais alta, seguindo-se o *mês*, e só depois o dia. Um método possível para ordenar datas consiste em fazer ordenações sucessivas usando cada um dos campos, *do menos significativo para o mais significativo*.

Consideremos por exemplo as datas:

- 21/8
- 26/3
- 14/4
- 21/3

Ordenando por dia:

- 14/4
- 21/8
- 21/3
- 26/3

Ordenando agora por mês:

- 21/3
- 26/3
- 14/4
- 21/8

As datas ficam ordenadas, mas é fundamental para isto que ordenação por mês tenha sido estável (foi preservada a ordenação anterior, pelo campo *dia*).

Se acrescentarmos o campo “ano” às datas:

- 21/3/2009
- 26/3/1975
- 14/4/1969
- 21/8/2009

Ordenando por ano é de novo preservada a ordenação anterior:

- 14/4/**1969**

- 26/3/**1975**
- 21/3/**2009**
- 21/8/**2009**

Mais uma vez a ordenação por ano preservou a ordenação anterior, pelo que as datas ficam ordenadas. Este algoritmo de ordenação de sequências de estruturas multi-campo, recorrendo a ordenações sucessivas usando um algoritmo de ordenação auxiliar estável (por exemplo *counting sort*), é conhecido por **radix sort**.

T5. Análise Amortizada de Algoritmos

Uma nova ferramenta de análise, que permite estudar o tempo necessário para se efectuar uma **sequência de operações sobre uma estrutura de dados**.

- Frequentemente, ao longo da vida de uma estrutura de dados, as operações que é possível realizar sobre ela são sujeitas a restrições que são ignoradas por uma análise de pior caso tradicional.
- A ideia chave é considerar o *pior caso da sequência* de N operações, em situações em que este é claramente mais baixo do que a soma dos tempos de pior caso das N operações singulares.
- Trata-se do estudo do custo médio (relativamente à sequência) de cada operação no pior caso, e não de uma análise de caso médio!
- Ao contrário da análise de caso médio, não envolve ferramentas probabilísticas nem assunções sobre os inputs.

3 técnicas:

1. Análise agregada
2. Método *contabilístico*
3. Método do *potencial*

Exemplo: Arrays Dinâmicos

Os vectores ou *arrays* são estruturas de dados com capacidade fixa e por isso limitada. Podem no entanto ser alocadas quer *estaticamente* quer *dinamicamente*. Por exemplo em C:

```
int u[1000];                                // estático
int *v = malloc(1000, sizeof(int));           // dinâmico
```

Uma solução comum para o problema do crescimento de um vector para além da sua capacidade é a utilização de vectores dinâmicos com *relocação*: quando se pretende inserir um elemento que já não cabe no vector, cria-se um novo vector, tipicamente com o *dobro* da capacidade do primeiro, copiando-se todos os elementos do primeiro para o segundo, antes de se inserir o novo elemento.

```
int* myrealloc(int* v, int n) {
    int* new = calloc(2*n, sizeof(int));
    for (i=0 ; i<n ; i++) new[i] = v[i];
    return new;
}
```

Considere-se agora uma *stack* implementada com um *array* dinâmico, e a sua operação *push*:

```
typedef struct stack {
    int *vec;
    int n;          // n. de elementos inseridos
    int cap;        // capacidade máxima
} Stack;

Stack push (Stack s, int x) {
    if (s.n == s.cap) {
        s.vec = myrealloc(s.vec, s.cap);
        s.cap *= 2;
    }
    (s.vec)[s.n] = x;
    (s.n)++;
    return s;
}
```

Como analisar uma sequência de operações *push*?

A análise de pior caso clássica da função *push* leva-nos a $T(S) = O(S)$, com S o tamanho actual da stack (no caso em que há lugar a realocação).

Note-se que de acordo com esta análise *uma sequência de N operações push executa no pior caso em tempo $T(N) = O(N^2)$* . No entanto, se examinarmos passo a passo a sequência em questão, vemos que a realocação será um evento raro, e por esta razão a análise acima é na verdade demasiado pessimista.

A partir de uma *stack* de capacidade c inicialmente vazia, teremos:

1. c inserções de custo (1),
2. seguidas de uma inserção de custo ($c + 1$),
3. novamente seguidas de $c - 1$ inserções de custo (1),
4. seguidas de uma inserção de custo ($2 * c + 1$),
5. seguidas de $2 * c - 1$ inserções de custo (1),
6. seguidas de uma inserção de custo ($4 * c + 1$),
7. seguidas de $4 * c - 1$ inserções de custo (1),
8. ...

Análise Agregada

A ideia desta forma de análise é simplesmente:

- Mostrar que para qualquer N , uma sequência de N operações tem no pior caso um *custo agregado* $O(P(N))$, que é inferior a N vezes o custo de pior caso de uma operação executada isoladamente;
- Então o *custo amortizado* por operação é $O(P(N))/N = O(P(N)/N)$.

Examinemos então com detalhe a sequência de operações *push*, calculando o custo de cada operação:

i	1	2	3	4	5	6	7	8	9	10	\dots
cap_i	1	2	4	4	8	8	8	8	16	16	\dots
t_i	1	1+1	2+1	1	4+1	1	1	1	8+1	1	\dots
$\log(i - 1)$		0	1		2				3		\dots

Pretendemos agora calcular o *custo agregado* $\sum_{i=1}^n t_i$.

Note-se que $1 + 2 + 4 + 8 + \dots = \sum_{k=0}^{\log(n-1)} 2^k$, logo

$$\begin{aligned}\sum_{i=1}^n t_i &= n + \sum_{k=0}^{\log(n-1)} 2^k \\ &= n + (2^{\log(n-1)+1} - 1) = n + 2(n-1) - 1 = 3n - 3\end{aligned}$$

O custo agregado da sequência é na verdade linear!

Consequentemente, o custo amortizado da operação *push* será constante:

$$T(n) = \frac{3n-3}{n} \leq 3 = O(1).$$

Com esta técnica (análise agregada),

Havendo diferentes operações sobre a estrutura de dados, considera-se que todas têm o mesmo custo amortizado

(as outras técnicas de análise amortizada diferem neste aspecto).

Método Contabilístico

Na análise agregada calculámos o custo amortizado a partir do custo agregado de uma sequência de operações. No método contabilístico vamos

- Arbitrar um custo amortizado **fixo** c para a operação, que não corresponderá ao real — será superior para algumas execuções da operação (acumulando *crédito*) e inferior para outras (gastando crédito acumulado)

- Em seguida teremos que argumentar que o custo amortizado individual que arbitrámos para a operação pode ser considerado válido para efeitos de análise de pior caso, ou seja
o custo amortizado total de uma qualquer sequência de N operações é um limite superior para o custo real dessa sequência no pior caso:
 $\sum_i t_i \leq Nc$
- Para isso basta calcular o *saldo* entre custo amortizado acumulado e custo real acumulado:
 $bal_{i+1} = bal_i - t_{i+1} + c_{i+1}$
partindo de um qualquer saldo inicial $bal_0 = K \geq 0$, e mostrar que para qualquer $i > 0$ se tem $bal_i \geq 0 \implies bal_{i+1} \geq 0$.

Arbitremos o valor $c = 2$ para custo amortizado da operação *push* e vejamos a evolução do saldo ao longo da sequência, com $bal_0 = 0$ e $bal_{i+1} = bal_i - t_{i+1} + c$.

i	1	2	3	4	5	6	7	8	9	10	...
cap_i	1	2	4	4	8	8	8	8	16	16	...
t_i	1	2	3	1	5	1	1	1	9	1	...
c_i	2	2	2	2	2	2	2	2	2	2	...
bal_i	1	1	0	1	-2						...

O custo amortizado escolhido não garante a condição fundamental de *saldo acumulado não-negativo* ao longo de toda a execução.

Façamos nova tentativa, arbitrando o valor $c = 3$ para o custo amortizado:

i	1	2	3	4	5	6	7	8	9	10	...
cap_i	1	2	4	4	8	8	8	8	16	16	...
t_i	1	2	3	1	5	1	1	1	9	1	...
c_i	3	3	3	3	3	3	3	3	3	3	...
bal_i	2	3	3	5	3	5	7	9	3	4	...

O custo amortizado de 3 é adequado para mostrar que a operação *push* é, em termos amortizados, de tempo constante no pior caso.

Note-se que a tabela acima constitui um argumento, mas **não uma prova formal**. É habitual, quando se aplica este método, apresentar também uma **interpretação intuitiva** para este custo arbitrado.

Intuitivamente, o custo amortizado 3 para a operação *push* corresponde a:

1. Inserção de um elemento na pilha, havendo espaço; note-se que este elemento ficará na *metade superior* da pilha.
2. Cópia posterior desse mesmo elemento para uma nova pilha, quando a capacidade da actual é excedida e há lugar a realocação; neste passo o elemento passa para a *metade inferior* da nova pilha
3. cópia de um outro elemento, da *metade inferior* da pilha antiga para a nova pilha.

Este terceiro custo tem a ver com a necessidade de cada elemento da metade superior da pilha actual (que foram colocados pela primeira vez depois da última realocação) “suportar” a cópia de um elemento na metade inferior, que já foi copiado para a actual na última realização. Cada elemento suporta a sua própria cópia quando está na metade superior da pilha; quando passar para a metade inferior as cópias passarão a ser pagas pela 3a. moeda de elementos da actual metade superior.

NOTA

Observe-se que na análise amortizada calculámos o custo $3n - 3 \leq 3n$ para uma sequência de n operações, o que reforça a escolha do custo amortizado $c = 3$.

Método do Potencial

Recapitulemos como foi calculado o custo amortizado:

- Na análise agregada, calculou-se o custo total da sequência de operações, e definiu-se o custo amortizado como sendo a média desse custo agregado;
- No método contabilístico, *arbitrou-se* o custo amortizado.

No método do potencial define-se uma *função de potencial* sobre o estado da estrutura de dados, e *calcula-se o custo amortizado* a partir desta função.

A ideia é que o potencial da estrutura deve *aumentar com operações de baixo custo*, e *diminuir com operações de alto custo*, tal como o saldo no método contabilístico. Mas enquanto naquele método o saldo é calculado a partir do custo amortizado, o potencial é definido à partida.

Seja Φ_i o potencial da estrutura de dados no estado i , ou seja depois de i operações da sequência. Esta função deve obedecer às condições

- $\Phi_0 = 0$
- $\Phi_i \geq 0$ para $i > 0$.

O **custo amortizado da operação i** será dado por $c_i = t_i - \Phi_{i-1} + \Phi_i$.

Compare-se com $bal_{i+1} = bal_i - t_{i+1} + c_{i+1}$ no método contabilístico.

O cálculo do **custo amortizado total** é *telescópico*:

$$\begin{aligned}\sum_{i=1}^n c_i &= \sum_{i=1}^n t_i - \Phi_{i-1} + \Phi_i \\ &= (t_1 - \Phi_0 + \Phi_1) + (t_2 - \Phi_1 + \Phi_2) + (t_3 - \Phi_2 + \Phi_3) + \dots + (t_n - \Phi_{n-1} + \Phi_n)\end{aligned}$$

Simplificando:

$$\sum_{i=1}^n c_i = \Phi_n - \Phi_0 + \sum_{i=1}^n t_i$$

Ora, uma vez que $\Phi_n \geq 0$, é necessariamente verdade que $\sum_i c_i \geq \sum_i t_i$, ou seja, o custo total amortizado constitui um limite superior para o tempo de execução da sequência de operações, como desejado.

Voltando ao nosso exemplo, tentemos sintetizar uma função de potencial a partir das propriedades da estrutura de dados.

Uma vez que já obtivemos, pelo método contabilístico, um bom candidato (3) para o custo amortizado, conhecemos à partida os valores de uma função de potencial possível — são os valores que o saldo toma ao longo da execução. Basta então encontrar a expressão simbólica correspondente a esta função.

Nesta perspectiva, o método do potencial é usado depois do método contabilístico, no sentido de *justificar* os custos amortizados arbitrados naquele método.

Observemos então de novo a tabela anterior, substituindo o saldo pelo potencial Φ_i no estado actual, e acrescentando uma coluna para o estado inicial, em que temos capacidade inicial 1, com ocupação 0.

$i = n_i$	0	1	2	3	4	5	6	7	8	9	10	...
cap_i	1	1	2	4	4	8	8	8	8	16	16	...
t_i		1	2	3	1	5	1	1	1	9	1	...
Φ_i	0	2	3	3	5	3	5	7	9	3	5	...

Tentemos chegar à definição de uma função de potencial adequada, esquecendo por momentos que conhecemos os saldos resultantes da aplicação do método contabilístico.

Procuramos uma expressão para Φ_i tal que:

- $\Phi_0 = 0$ e $\Phi_i \geq 0$ para $i > 0$
- que dependa apenas de n_i e de cap_i
- que aumente, a partir do valor inicial 0, à medida que a ocupação da pilha aumenta, diminuindo nos passos em que o array é duplicado.

Como este valor aumenta mesmo quando cap_i não varia, tem de crescer com n_i . Por outro lado, diminui quando cap_i aumenta, por isso deve decrescer com cap_i .

Observe-se que, com a excepção das duas primeiras configurações, em qualquer momento se tem mais de metade da capacidade sempre seguramente preenchida, uma vez que só nesse momento é feita a realocação, ou seja $n_i \geq cap_i/2 + 1$, logo $2 * n_i - cap_i \geq 2$.

Para $i = 0$ tem-se $2 * n_i - cap_i = -1$, e
para $i = 1$ tem-se $2 * n_i - cap_i = 1$

Em todas as configurações temos então $2 * n_i - \text{cap}_i \geq -1$,
logo $2 * n_i - \text{cap}_i + 1 \geq 0$.

Será esta expressão um bom candidato para definir a função de potencial? De facto sim, a função definida como

$$\Phi_i = 2 * n_i - \text{cap}_i + 1$$

produz os valores contidos na tabela acima (correspondentes ao saldo no método contabilístico).

Podemos confirmar isto, calculando o custo amortizado de *push* que resulta desta função de potencial.

Relembremos que $c_i = t_i - \Phi_{i-1} + \Phi_i$, logo

$$\begin{aligned} c_i &= t_i - (2 * n_{i-1} - \text{cap}_{i-1} + 1) + (2 * n_i - \text{cap}_i + 1) \\ c_i &= t_i - (2 * n_{i-1} - \text{cap}_{i-1} + 1) + (2 * (n_{i-1} + 1) - \text{cap}_i + 1) \end{aligned}$$

Temos dois casos distintos:

- Se $n_{i-1} < \text{cap}_{i-1}$ ($\text{cap}_i = \text{cap}_{i-1}$, não há realocação)
 $c_i = 1 - (2 * n_{i-1} - \text{cap}_{i-1} + 1) + (2 * (n_{i-1} + 1) - \text{cap}_{i-1} + 1) = 3$
- Se $n_{i-1} = \text{cap}_{i-1}$ ($\text{cap}_i = 2 * \text{cap}_{i-1}$, há realocação)
 $c_i = (n_{i-1} + 1) - (2 * n_{i-1} - n_{i-1} + 1) + (2 * (n_{i-1} + 1) - 2 * n_{i-1} + 1) = 3$

Obtemos o mesmo custo amortizado que tínhamos obtido pelo método contabilístico.

Temos agora certeza de que, em qualquer sequência de operações push, o tempo amortizado de execução da operação ao longo da sequência é constante.

Note-se que é possível utilizar o método do potencial de forma isolada, sem nenhuma ideia de quais os custos amortizados que serão obtidos, e nesse caso a síntese da função de potencial será mais desafiadora.

Exemplo: Fila de espera (queue) implementada com duas pilhas (stacks)

Uma implementação possível de uma fila de espera (Queue) utiliza duas pilhas A e B, por exemplo:

```
typedef struct queue {  
    Stack a;  
    Stack b;  
} Queue;
```

- A inserção (`enqueue`) de elementos é sempre realizada na pilha A
- para a saída de elementos (`dequeue`), se a pilha B não estiver vazia, é efectuado um `pop` nessa pilha;
- caso contrário, para todos os elementos de A, faz-se sucessivamente `pop` e `push` na pilha B. Faz-se depois `pop` da pilha B, devolvendo-se como resultado o elemento extraído.

```
enqueue 1; enqueue 2; enqueue 3
```

```
A      B
```

```
3
```

```
2
```

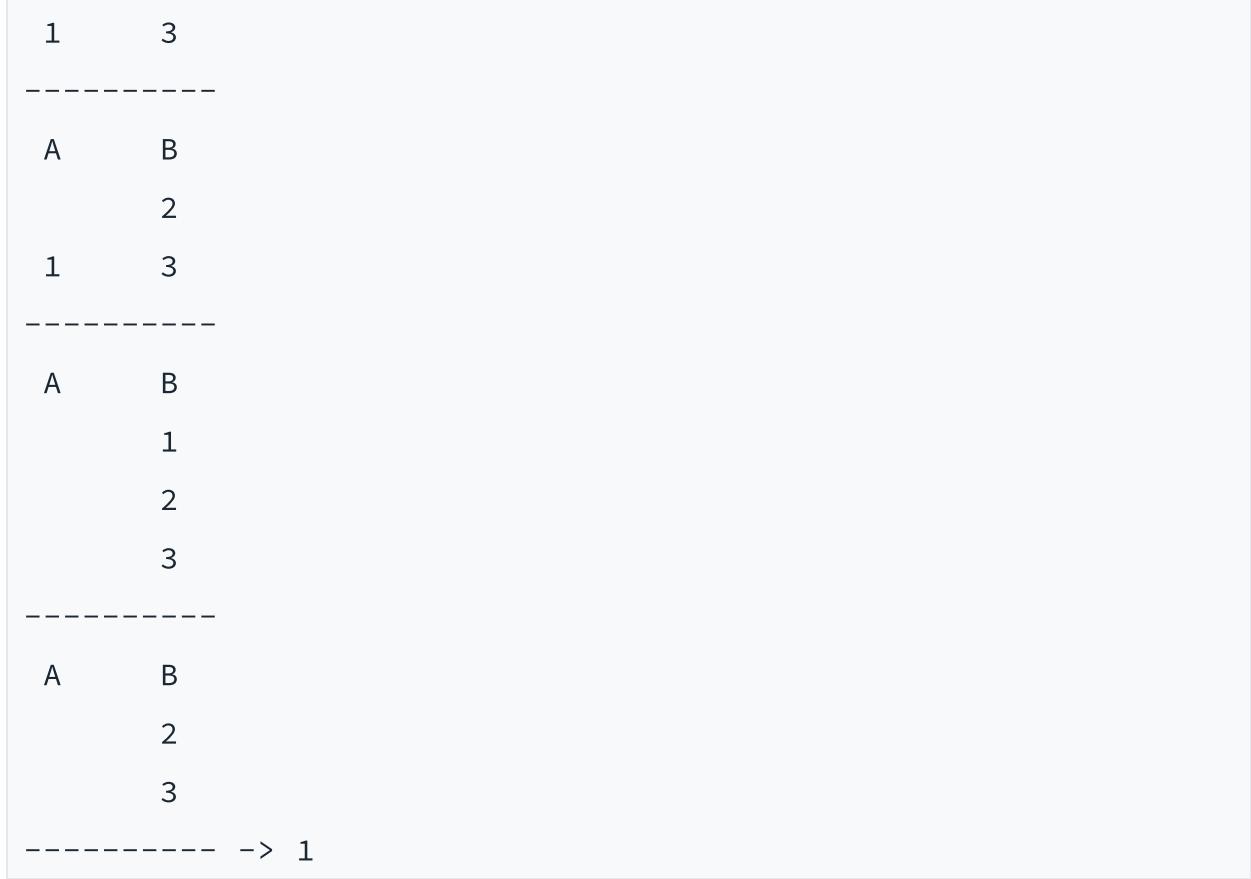
```
1
```

```
-----
```

```
dequeue
```

```
A      B
```

```
2
```



Pela análise tradicional de pior caso,

- `enqueue` executa em tempo $O(1)$
- `dequeue` executa em tempo $O(N)$

Veremos no entanto que **em termos amortizados, ambas as operações executam em tempo constante $O(1)$**

A análise amortizada é aqui mais desafiante, uma vez que uma sequência típica de operações será agora **heterogénea**, contendo ocorrências de `enqueue` e de `dequeue`.

Na análise agregada e pelo método contabilístico, apenas conseguiremos analisar as operações no contexto de **sequências concretas** que fixamos à partida.

Veremos que neste caso o método do potencial se revela superior, uma vez que é independente de qualquer sequência particular de operações que se considere.

Análise Agregada

Consideremos a seguinte **sequência concreta** de operações:

- n operações `enqueue` seguidas de n operações `dequeue`

i	0	1	2	...	n	$n+1$	$n+2$...	$2n$
n_A	0	1	2	...	n	0	0	...	0
n_B	0	0	0	...	0	$n-1$	$n-2$...	0
t_i		1	1	...	1	$2n+1$	1	...	1

Temos $\sum_{i=1}^n t_i = n + (2n + 1) + (n - 1) = 4n$

Globalmente a sequência executa em tempo $4n$, logo cada uma das $2n$ operações executa em tempo constante. Intuitivamente:

uma operação `dequeue` de custo linear $\Theta(n)$ só pode ser executada depois de n operações `enqueue` de tempo constante, que coloquem n elementos na pilha B . Os `enqueue`s amortizam o `dequeue` mais custoso.

O custo amortizado que resulta desta análise agregada é $\frac{4n}{2n} = 2$. Note-se que este valor resulta da análise feita para a sequência específica de operações considerada em cima, e é igual para ambas as operações. Veremos em seguida que os outros métodos permitirão chegar a custos diferentes, válidos para outras sequências.

Método Contabilístico

Considerando a mesma sequência concreta de operações, começemos por considerar um custo amortizado $c_i = 2$ para todas elas.

i	0	1	2	...	n	$n+1$	$n+2$...	$2n$
n_A	0	1	2	...	n	0	0	...	0
n_B	0	0	0	...	0	$n-1$	$n-2$...	0
t_i		1	1	...	1	$2n+1$	1	...	1
c_i		2	2	...	2	2	2	...	2
bal_i		1	2	...	n	1-n			

O custo amortizado considerado para a operação `enqueue` é insuficiente para cobrir o custo real da operação `dequeue` no pior caso. Consideremos então o custo amortizado $c_i = 3$:

i	0	1	2	...	n	$n+1$	$n+2$...	$2n$
n_A	0	1	2	...	n	0	0	...	0
n_B	0	0	0	...	0	$n-1$	$n-2$...	0
t_i		1	1	...	1	$2n+1$	1	...	1
c_i		3	3	...	3	3	3	...	3
bal_i		2	4	...	$2n$	2	4	...	$2n$

Este valor para o custo amortizado é suficiente.

Vemos no entanto que este custo é excessivo no caso da operação `dequeue`: basta para esta operação considerar um custo $c_i = 1$:

i	0	1	2	...	n	$n+1$	$n+2$...	$2n$
t_i		1	1	...	1	$2n+1$	1	...	1
c_i		3	3	...	3	1	1	...	1
bal_i		2	4	...	$2n$	0	0	...	0

Intuitivamente, o custo de 3 corresponde no caso do `enqueue` ao custo de efectuar:

1. push de um elemento na pilha A
2. pop desse elemento da pilha A
3. push do mesmo elemento na pilha B

Método do Potencial

Nas análises anteriores, agregada e pelo método contabilístico, escolhemos uma sequência que intuitivamente parecia levar ao pior caso de execução de `dequeue`.

No entanto as análises **não provaram** que os custos amortizados calculados são adequados para uma `qualquer` sequência das operações. Com o método do potencial isto será possível.

Para a aplicação do método consideremos a função de potencial seguinte:

$$\Phi_i = 2n_{A,i}.$$

A intuição para esta escolha vem da observação da tabela acima: a única operação custosa ($\text{custo} > 1$) é o primeiro `dequeue`, que tem um custo $2n + 1$, sendo n o número de elementos da primeira pilha; o potencial armazenado permite cobrir este custo (é fácil de ver que o potencial corresponde ao saldo na tabela acima).

Claramente temos $\Phi_i \geq 0$ para $i \geq 0$ e $\Phi_0 = 0$ (desde que se inicie a sequência com a stack A vazia). Simulemos a evolução do potencial ao longo da execução da sequência que temos vindo a considerar:

i	0	1	2	...	n	$n+1$	$n+2$...	$2n$
n_A	0	1	2	...	n	0	0	...	0
n_B	0	0	0	...	0	$n-1$	$n-2$...	0
t_i		1	1	...	1	$2n+1$	1	...	1
Φ_i	0	2	4	...	$2n$	0	0	...	0

É visível que o potencial cresce nas operações “leves” e decresce na operação pesada `dequeue`, de tempo linear.

Calculemos o custo amortizado de ambas as operações

- `enqueue`:

$$t_i - \Phi_{i-1} + \Phi_i = 1 - 2n_A + 2(n_A + 1) = 3$$

- `dequeue`. Teremos que considerar dois cenários:

- Se $n_B \neq 0$: $t_i - \Phi_{i-1} + \Phi_i = 1 - 2n_A + 2n_A = 1$

- Se $n_B = 0$: $t_i - \Phi_{i-1} + \Phi_i = (2n_A + 1) - 2n_A + 0 = 1$

É de realçar que

- Mais uma vez, o método de potencial vem confirmar e provar os custos amortizados a que se tinha chegado de forma intuitiva pela análise agregada e pelo método contabilístico.
(uma vez que a função de potencial cumpre os requisitos necessários)
- Os custos amortizados calculados com o método do potencial são garantidamente **válidos para qualquer sequência de operações**, ao contrário das análises anteriores.

EXERCÍCIO: Pilha com operação Multipop

[A resolver nas aulas TP]

Considere-se uma estrutura de dados do tipo `stack` com a habitual operação `push`, mas em que a operação `pop` é substituída por uma operação `multipop`, uma generalização que remove os k primeiros elementos, deixando a pilha vazia caso contenha menos de k elementos.

Uma implementação possível será

```
multiPop(S,k) {  
    while (!IsEmpty(S) && k != 0) {  
        pop(S);  
        k -= 1;  
    }  
}
```

Pela análise tradicional de pior caso,

- `push` executa em tempo $O(1)$
- `multiPop` executa em tempo $O(N)$

Utilize os 3 diferentes métodos estudados, para mostrar que em termos amortizados a operação `multiPop` executa também ela em tempo constante $O(1)$.

+

ED1. Implementação em C de uma Fila de Espera sobre um array, com circularidade

PROJECTO CODEBOARD DE SUPORTE A ESTE MÓDULO:

<https://codeboard.io/projects/47232>

1. O tipo abstracto de dados

Recorde-se que um tipo abstracto é definido através da sua *interface*, i.e. das operações disponibilizadas sobre ele, sendo que o código-cliente não necessita de ter acesso à implementação concreta que está a ser utilizada.

As duas operações mais importantes, e presentes em qualquer especificação de uma fila de espera, são as seguintes:

- **enqueue**: acrescenta um elemento na última posição da fila
- **dequeue**: remove (e devolve) o primeiro elemento da fila

A interface pode ser enriquecida com outras funções de consulta, por exemplo:

- **isEmpty**: testa se a fila está vazia
- **isFull**: testa se a fila está cheia (isto é se foi atingida a sua capacidade máxima)
- **front**: devolve o primeiro elemento da fila, sem o remover

Optaremos por uma versão mínima do tipo de dados, apenas com as duas primeiras funções, que deverão elas mesmas testar se é possível a sua execução (i.e., **enqueue** não assumirá que a fila não se encontra cheia, e **dequeue** não assumirá que ela não se encontra vazia).

Imaginemos uma fila de elementos do tipo Inteiro, inicialmente vazia, e na qual são inseridos por esta ordem os elementos 10, 20, 30, ..., 100.

[] \leftarrow enqueue \leftarrow 10
[10] \leftarrow enqueue \leftarrow 20

```
[ 10, 20 ] ← enqueue ← 30
[ 10, 20, 30 ] ← enqueue ← 40
...
[ 10, 20, 30, 40, 50, 60, 70, 80, 90, 100 ]
```

Imagine-se agora que serão “atendidos”, ou seja extraídos da fila, os primeiros 5 elementos:

```
10 ← dequeue ← [ 20, 30, 40, 50, 60, 70, 80, 90, 100 ]
20 ← dequeue ← [ 30, 40, 50, 60, 70, 80, 90, 100 ]
30 ← dequeue ← [ 40, 50, 60, 70, 80, 90, 100 ]
40 ← dequeue ← [ 50, 60, 70, 80, 90, 100 ]
50 ← dequeue ← [ 60, 70, 80, 90, 100 ]
```

Visivelmente, a estrutura de dados concreta a utilizar para a implementação deste tipo abstracto de dados deverá ser linear, e permitir o acesso em tempo constante a ambas as extremidades, uma vez que os elementos devem ser inseridos numa delas e removidos na outra.

2. Estruturas de dados em C e interface

Utilizaremos um *array* para implementar a fila de espera. Como em qualquer estrutura de dados implementada com base num *array*, é necessário guardar o número de elementos armazenados em cada momento. No entanto, isto não é suficiente: o exemplo anterior mostra que, por forma a executar a operação **dequeue** em tempo constante, é necessário guardar também o índice onde se encontra o primeiro elemento da fila. A operação **dequeue** será realizada alterando-se apenas este índice.

Considere-se que se implementa a fila de espera do exemplo anterior sobre um *array* com 10 posições. As primeiras operações (*enqueue*) aumentam o **tamanho** da fila de 0 para 10, sendo atingida a sua capacidade máxima:

10									
10	20								
10	20	30							

...
10	20	30	40	50	60	70	80	90	100	

O índice `inicio` do **primeiro** elemento é neste momento 0, correspondente à posição do elemento 10, o primeiro que foi inserido. As operações seguintes, de extracção (*dequeue*) de elementos, incrementam `inicio`.

	20	30	40	50	60	70	80	90	100	
		30	40	50	60	70	80	90	100	
			40	50	60	70	80	90	100	
				50	60	70	80	90	100	
					60	70	80	90	100	

O índice do primeiro elemento é neste momento 5, correspondente à posição do elemento 60. Note-se que os elementos 10 a 50 continuam armazenados nas primeiras posições, mas não são considerados parte da fila, uma vez que se encontram em posições do array inferiores a `inicio`.

Definiremos então uma estrutura com os 3 campos necessários, e uma função de inicialização que recebe (por referência) uma estrutura e inicializa o índice de `inicio` e o `tamanho` com o valor 0.

```
#define MAX 10

typedef struct queue {
    int inicio, tamanho;
    int valores [MAX];
} QUEUE;

// ensures queue (*q) is adequately initialized
void initQueue (QUEUE *q) {
    q->tamanho = 0;
    q->inicio = 0;
}
```

Uma vez que optamos por uma versão mínima deste tipo de dados abstracto, as funções **enqueue** e **dequeue** terão que detectar, respectivamente, se a fila se encontra cheia ou vazia. O valor de retorno das funções sinalizará estas situações, de acordo com o especificado nos contratos anotados nos protótipos seguintes:

```
// 1. assuming queue (*q) is full,  
//    ensures result == 1  
  
// 2. assuming queue is not full,  
//    ensures result == 0 and x is added at the back of queue  
int enqueue (QUEUE *q, int x);  
  
  
// 1. assuming queue (*q) is empty,  
//    ensures result == 1  
  
// 2. assuming queue is not empty,  
//    ensures result == 0 and  
//                  front element is written to (*x) and removed fro  
m queue  
int dequeue (QUEUE *q, int *x);
```

3. Implementação das funções **enqueue** e **dequeue**: circularidade

Numa primeira abordagem poderíamos escrever a seguinte definição para **enqueue**. Note o cálculo da posição onde será inserido x como sendo `inicio+tamanho`:

```
int enqueue (QUEUE *q, int x) {  
    if (q->tamanho == MAX) return 1;  
    (q->valores)[q->inicio+q->tamanho] = x;  
    (q->tamanho)++;  
    return 0;  
}
```

É no entanto altura de repararmos que esta utilização linear do array, que é necessariamente finito, não é eficiente do ponto de vista da utilização do espaço. Voltando ao exemplo anterior, observa-se que a última posição do array (9) está ocupada, pelo que não é possível inserir um novo elemento na posição `inicio+tamanho`. No entanto, as 5 primeiras posições encontram-se livres!

Uma solução que permite utilizar este espaço livre no início do array é armazenar a fila de espera *circularmente*. Imagine-se que pretendemos inserir agora os elementos 110, 120, ..., 150:

```
[ 60, 70, 80, 90, 100 ] ← enqueue ← 110
[ 60, 70, 80, 90, 100, 110 ] ← enqueue ← 120
...
[ 60, 70, 80, 90, 100, 110, 120, 130, 140, 150 ]
```

Numa implementação circular teremos sucessivamente:

					60	70	80	90	100
110					60	70	80	90	100
110	120				60	70	80	90	100
110	120	130			60	70	80	90	100
110	120	130	140		60	70	80	90	100
110	120	130	140	150	60	70	80	90	100

EXERCÍCIO: Corrija o código dado acima para a função `enqueue`, e defina a função `dequeue`, por forma a implementar correctamente o comportamento destas funções com uma utilização circular do array.

4. Código cliente

Por forma a testar a correcta implementação das funções escreveremos a seguinte função principal, que reproduz o exemplo acima. Note que o primeiro ciclo preenche completamente a fila; o segundo ciclo extrai a primeira metade dos elementos; o

terceiro ciclo volta a preencher totalmente a fila (ocupando a primeira metade do array); finalmente, o último ciclo esvazia a fila.

```
int main() {
    QUEUE q;
    int a, i, j, empty, full;

    initQueue(&q);
    i = 10;

    full = enqueue(&q, i);
    while (!full) {
        printf("enqueued %d; ", i);
        i += 10;
        full = enqueue(&q, i);
    }

    for (j=0; j<MAX/2; j++) {
        dequeue(&q, &a);
        printf("dequeued %d; ", a);
    }

    // first half of array now empty

    full = enqueue(&q, i);
    while (!full) {
        printf("enqueued %d; ", i);
        i += 10;
        full = enqueue(&q, i);
    }
}
```

```

    empty = dequeue(&q, &a);
    while (!empty) {
        printf("dequeued %d; ", a);
        empty = dequeue(&q, &a);
    }

    return 0;
}

```

Em cada operação **enqueue / dequeue** é impresso o elemento inserido ou extraído; a sequência esperada é a seguinte:

```

enqueued 10; enqueue 20; enqueue 30; enqueue 40; enqueue 50;
enqueue 60; enqueue 70; enqueue 80; enqueue 90; enqueue 100;
dequeued 10; dequeued 20; dequeued 30; dequeued 40; dequeued 50;
enqueue 110; enqueue 120; enqueue 130; enqueue 140; enqueue 150;
dequeued 60; dequeued 70; dequeued 80; dequeued 90; dequeued 100;
dequeued 110; dequeued 120; dequeued 130; dequeued 140; dequeued 150;

```

5. Outros Exercícios

1. Mesmo com uma utilização circular do array, sendo este aloçado estaticamente, existe uma capacidade máxima que não pode ser excedida. Pode-se utilizar em alternativa um array dinâmico, que pode ser realocado quando a sua capacidade é atingida.

Considere a seguinte estrutura, que inclui agora um apontador para o início do array, bem como um campo `cap` contendo a capacidade actual do mesmo, e reimplemente as anteriores, incluindo a de inicialização. A função **enqueue** deverá *duplicar* a capacidade do array quando necessário.

```

typedef struct queue {

```

```
int cap;
int inicio, tamanho;
int *valores;
} QUEUE;
```

2. Uma forma alternativa de se implementar uma fila de espera é recorrer a um outro tipo abstracto de dados, o das *pilhas* (*stacks*). Uma fila pode ser implementada com base em duas pilhas A e B, sendo que:
 - a. a operação **enqueue** será realizada mediante uma operação **push** na pilha A
 - b. a operação **dequeue** será realizada mediante uma operação **pop** na pilha B
 - c. ao tentar realizar uma operação **dequeue**, caso a pilha B se encontre vazia, *transferem-se todos os elementos de A para B, fazendo sucessivamente pop em A e push em B*, após o que será já possível realizar o **dequeue**Escreva todo o código necessário para esta implementação alternativa de uma fila de espera.
1. A solução anterior desperdiça espaço, uma vez que a pilha A pode ficar completamente preenchida, impossibilitando a realização de operações **enqueue**, havendo entanto espaço livre na pilha B. Diga como se pode resolver este problema de desperdício de espaço, recorrendo a uma terceira pilha C.

ED2: Filas com Prioridades e “Heaps”

[Projecto Codeboard de suporte a esta aula: <https://codeboard.io/projects/10165>]

O tipo abstracto de dados

Recorde-se que os tipos abstractos de dados (*Abstract Data Types*, ADTs) constituem um instrumento fundamental de abstracção, separando a **interface** de uma estrutura de dados (o conjunto de operações disponíveis sobre ela) da sua **implementação concreta**.

Uma *fila com prioridades* (*priority queue*) é um destes ADTs. Em particular, trata-se de uma estrutura do género *Buffer*, uma vez que dispõe de uma operação de **inserção** e outra de **extracção** de elementos, sendo a relação entre as duas operações regida por uma estratégia específica.

No caso das filas com prioridades a estratégia é mais complexa do que as bem conhecidas estratégias *Last-In, First-Out* (LIFO) e *First-In, First-Out* (FIFO) características dos buffers mais comuns, as **pilhas** e as **filas de espera**.

Numa fila com prioridades são associados valores numéricos aos elementos inseridos, que correspondem a valores de prioridade.

Arbitremos que as prioridades são dadas por números inteiros, correspondendo números pequenos a prioridades mais elevadas. Consideremos a seguinte sequência de inserções:

```
insert("AA", 10);
insert("BB", 20);
insert("CC", 5);
insert("DD", 15);
```

Se esta sequência for seguida de uma sequência de extracções (operação **pull**), os elementos serão extraídos pela seguinte ordem:

```
"CC"
"AA"
```

```
"DD"  
"BB"
```

Tratando-se de uma estrutura de dados definida a um nível abstracto, será necessário conceber uma implementação concreta.

Heaps

Uma *heap* é uma árvore binária, caracterizada por duas propriedades (invariantes de tipo):

- Invariante de **ordem**:

O valor associado a cada nó é *inferior ou igual* aos valores de todos os seus descendentes

- Invariante de **forma**:

- A árvore binária é *completa* (apenas o último nível pode não estar totalmente preenchido), e
- último nível é preenchido da esquerda para a direita, sem “lacunas”

Estas propriedades de forma implicam que a altura é necessariamente *logarítmica* no número de nós da árvore, logo as operações de inserção e de extracção de elementos podem ambas ser executadas em tempo $O(\log n)$.

Note-se que este invariante de ordem implica que o mínimo da estrutura se encontra na raiz da árvore, e por isso uma *heap* com esta propriedade designa-se por *min-heap*.

Substituindo

inferior ou igual por *superior ou igual* obtém-se uma *max-heap*.

A extracção devolve sempre o menor (resp. maior) elemento na *heap*, pelo que esta estrutura é adequada para a implementação de filas com prioridades.

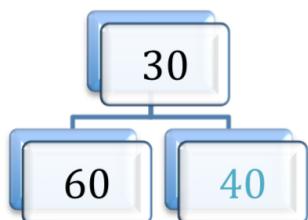
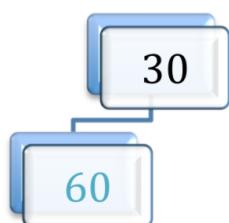
Algoritmo de Inserção:

1. Insere-se o novo elemento na primeira posição livre da heap, i.e. na posição mais à esquerda do último nível da heap;
2. Faz-se uma operação de **bubble-up**:
Enquanto o elemento inserido for de valor inferior o seu pai na árvore, troca-se sucessivamente (ao longo de um caminho ascendente da *heap*) estes dois elementos.

EXEMPLO

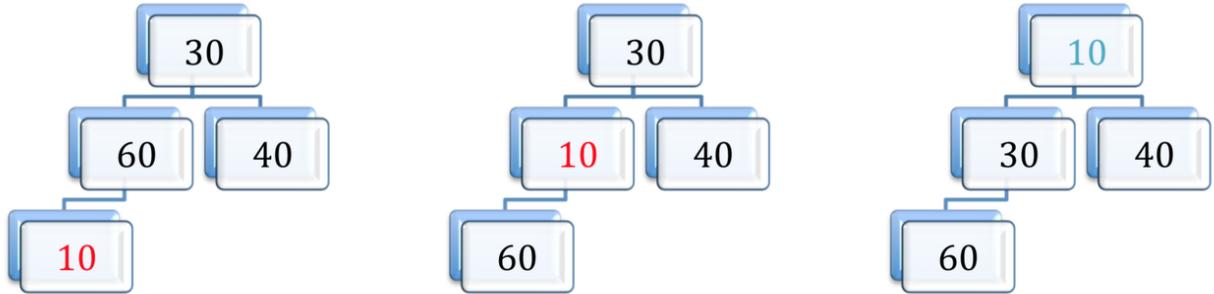
Consideremos uma sequência de inserções numa *min-heap*, começando com uma estrutura vazia.

```
Insert 30;  
Insert 60;  
Insert 40;
```



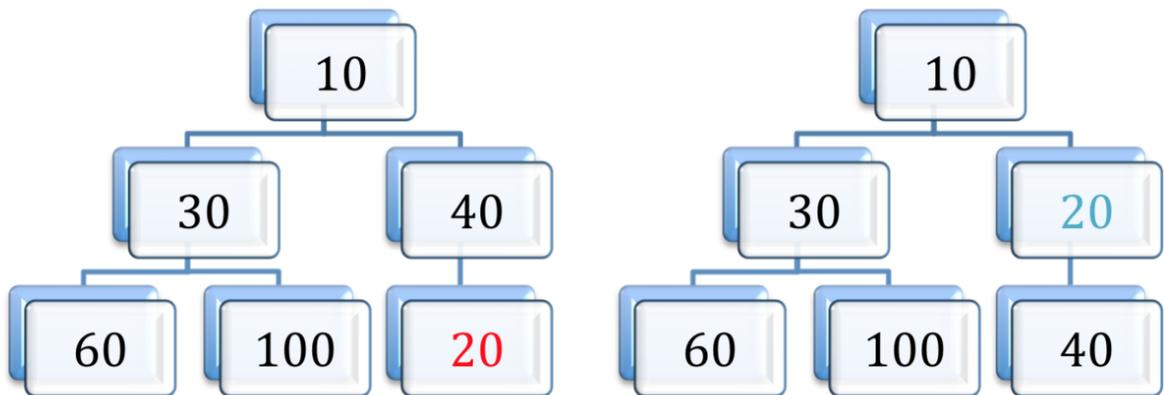
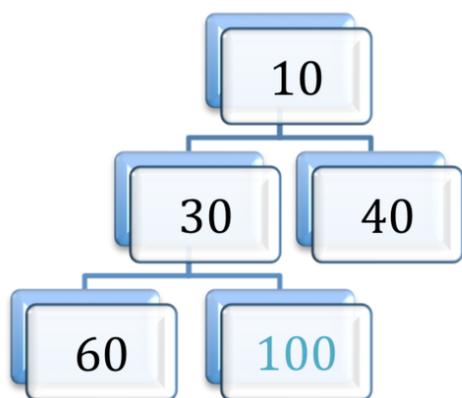
Nestes primeiros passos o invariante de ordem foi respeitado, pelo que não foi necessário executar *bubble-up*. No próximo passo isso já não será assim.

```
Insert 10;
```



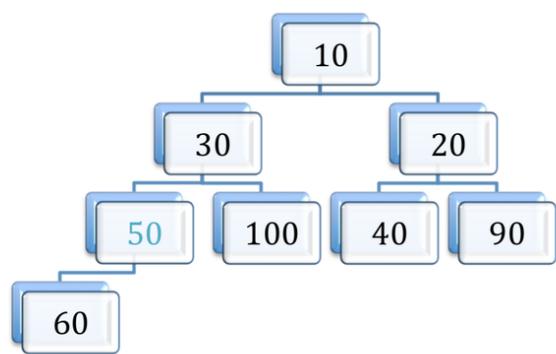
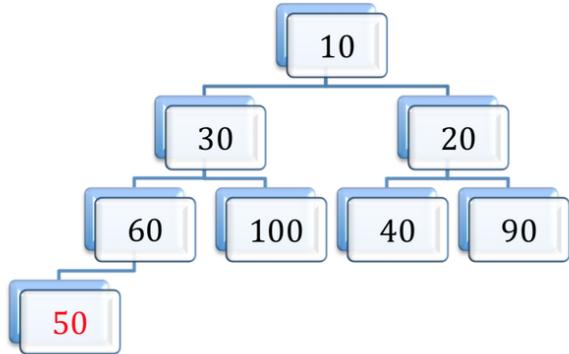
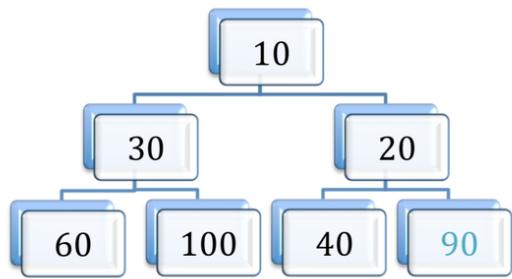
```
Insert 100;
```

```
Insert 20;
```



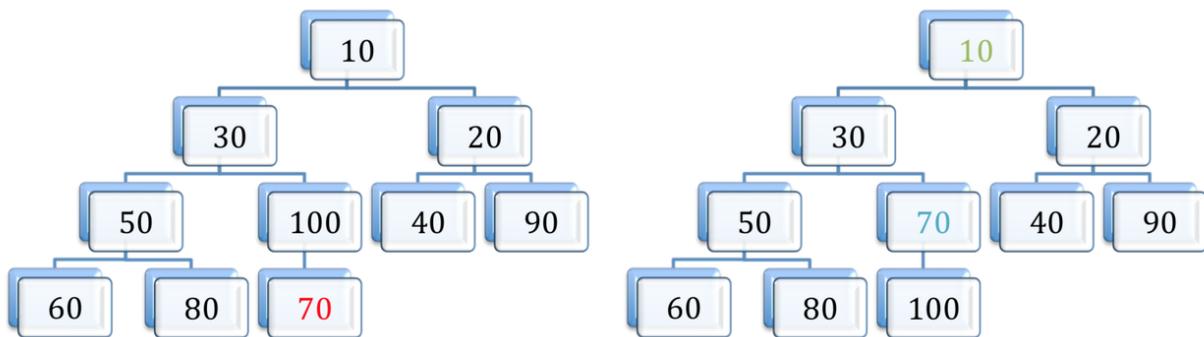
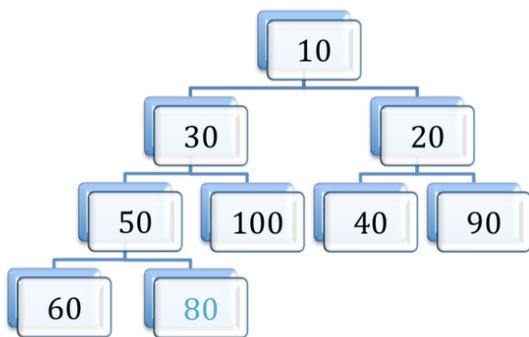
```
Insert 90;
```

```
Insert 50;
```



```
Insert 80;
```

```
Insert 70;
```



Observe-se que:

Se for possível o acesso directo ao pai de cada elemento da heap, o algoritmo de inserção (incluindo a op. de bubble-up) executará em tempo $O(\log N)$, uma vez que a altura da árvore é logarítmica em N

Algoritmo de Extracção (operação *pull*):

Naturalmente, o elemento a extrair será sempre a raiz da árvore (quer se trate de uma *min-heap* quer se trate de uma *max-heap*). A questão que se coloca é como reajustar a estrutura para eliminar a lacuna gerada na raiz, respeitando ainda todos os invariantes.

A intuição aponta no sentido de fazer subir o menor dos filhos da raiz, repetindo sucessivamente este passo. No entanto, é imediato constatar (por exemplo na heap construída acima) que este algoritmo não preserva os invariantes de forma de uma heap.

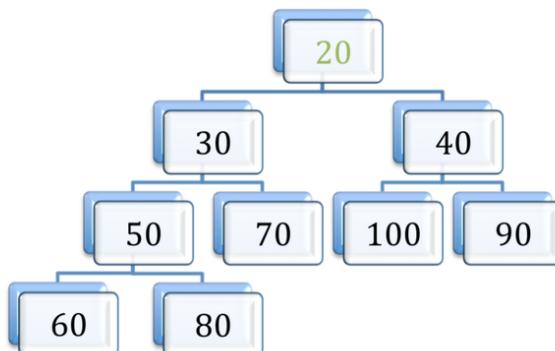
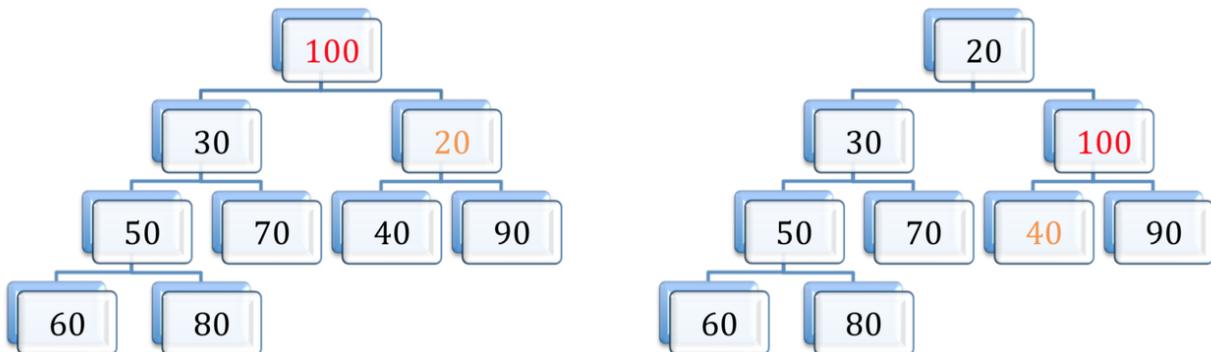
O algoritmo correcto é o seguinte:

1. Remove-se o elemento inserido na última posição da heap, i.e. na posição mais à direita do último nível da heap, e inscreve-se este mesmo elemento na raiz da heap, em substituição da raiz extraída.
2. Faz-se uma operação de **bubble-down** desta nova raiz:
Enquanto o nó actual for de valor superior a pelo menos um dos seus filhos, troca-se sucessivamente (ao longo de um caminho descendente da heap) o valor do nó com o do menor dos seus filhos

EXEMPLO

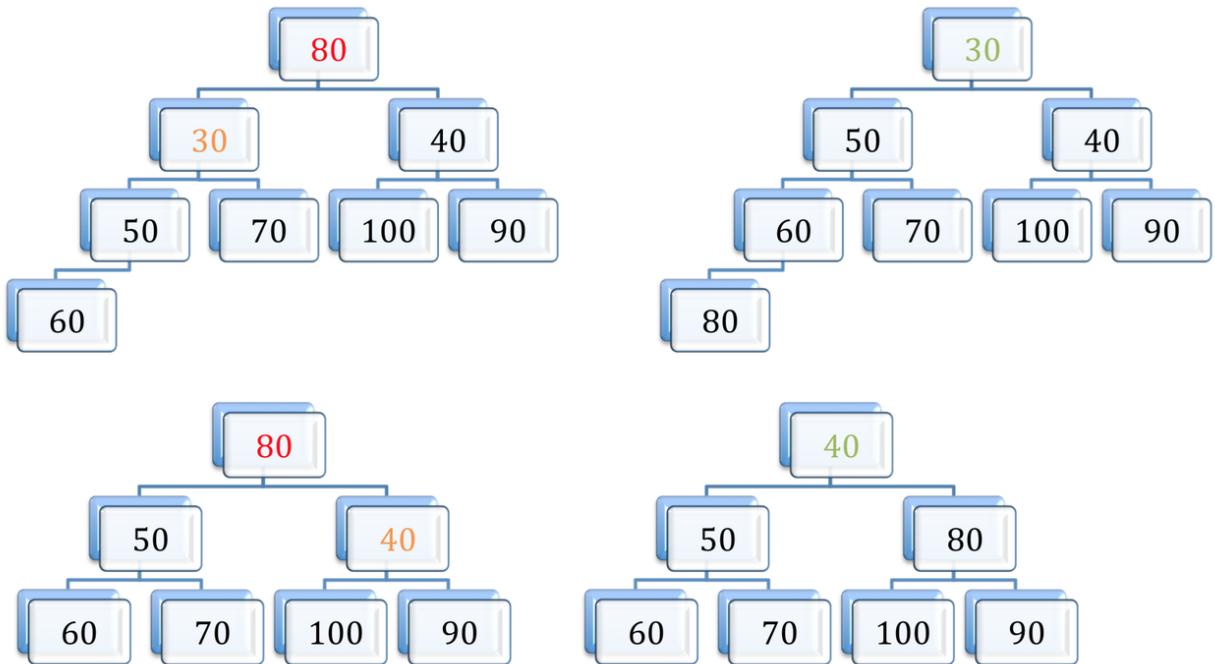
Executemos uma sequência de extracções a partir da *heap* do exemplo anterior.

```
Pull;
> 10
```

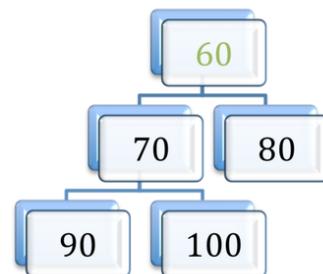
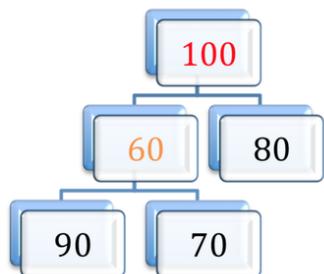
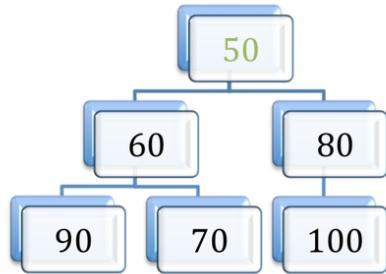
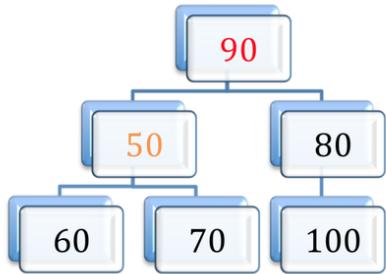


```
Pull;
> 20
```

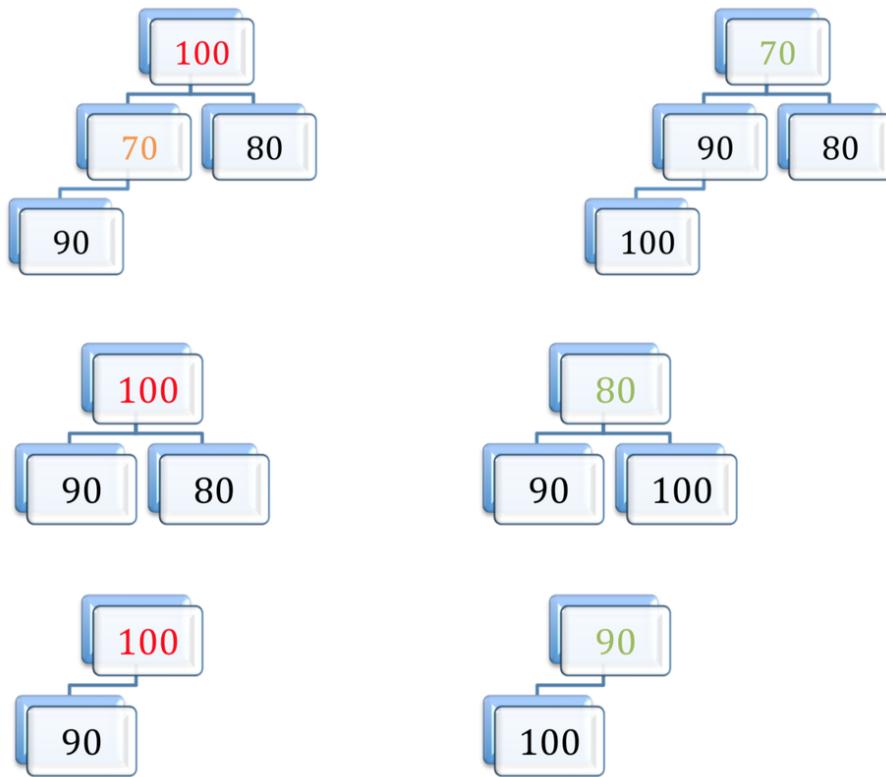
```
Pull;  
> 30
```



```
Pull;  
> 40  
Pull;  
> 50
```



```
Pull;  
> 60  
Pull;  
> 70  
Pull;  
> 80  
Pull;  
> 90
```



Heaps: Implementação Física

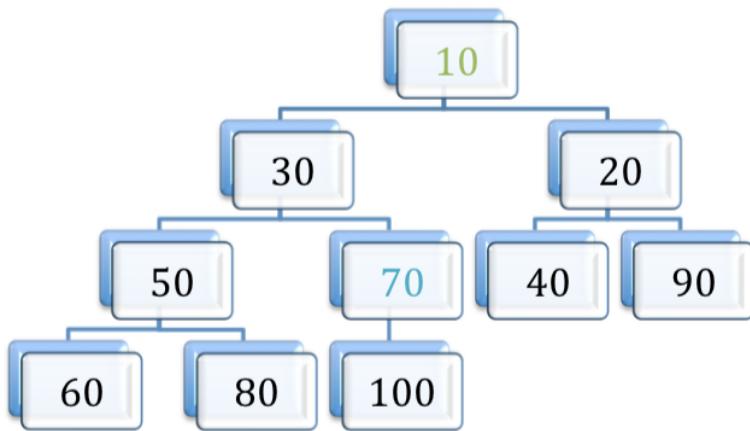
Tal como descrita acima, uma *heap* é uma estrutura de dados ao nível *lógico*.

Ao contrário do que acontece com uma árvore binária de pesquisa, que é tipicamente implementada por uma estrutura física ligada em memória dinâmica, as *heap* são tipicamente implementadas sobre *arrays* (podendo ser alocadas estática ou dinamicamente).

Basta dispor os elementos por ordem da raíz de árvore para as folhas, e percorrendo os níveis da esquerda para a direita

EXEMPLO

A *heap*:



pode ser implementada ao nível físico pelo seguinte vector:

i	0	1	2	3	4	5	6	7	8	9
$v[i]$	10	30	20	50	70	40	90	60	80	100
Nível	1	2	2	3	3	3	3	4	4	4

Observe-se que a implementação sobre um array permite o acesso directo (em tempo constante) não só aos filhos de um determinado nó, como também ao seu pai. Além disso, possibilita também o acesso em tempo constante ao último elemento da *heap*, o que é relevante para a execução dos algoritmos vistos atrás.

Uma consequência deste facto é que no melhor caso os algoritmos executam em tempo constante, o que não seria possível numa implementação ligada típica em que seria necessário localizar o último nó.

Os algoritmos de inserção e extração numa heap executam em tempo $\Omega(1)$, $O(\log N)$.

EXERCÍCIOS

[a resolver em <https://codeboard.io/projects/10165>]

Para a implementação de uma *min-heap* sobre um array dinâmico consideraremos as seguintes definições de tipos e protótipos de funções, em que `used` é o tamanho actual da *heap*, e `size` é a sua capacidade máxima (correspondente ao comprimento do array alocado).

```
typedef int Elem; // elementos da heap.
```

```

typedef struct {
    int    size;
    int    used;
    Elem  *value;
} Heap;

void initHeap (Heap *h, int size);
int insertHeap (Heap *h, Elem x);
int extractMin (Heap *h, Elem *x);
int minHeapOK (Heap h);

```

Implemente as 4 funções com os protótipos dados, notando o seguinte:

- A função `initHeap` inicializa uma *heap* (passada por referência), alocando para isso um *array* de comprimento `size`
- Se preferir, poderá começar por implementar a *heap* sobre um *array* estático
- Na implementação dinâmica, o comprimento do *array* deverá ser *duplicado* quando a capacidade se encontra completamente preenchida, por forma a assegurar que, em termos amortizados esta operação executa em tempo $\Omega(1)$, $O(\log N)$
- Os valores de retorno podem ser utilizados para um código de erro

O projecto Codeboard inclui uma função `main` que executa a sequência de inserções e extracções exemplificada acima.

ED3. Árvores AVL

PROJECTO CODEBOARD DE SUPORTE A ESTE MÓDULO:

<https://codeboard.io/projects/46752>

Tipos Abstractos de Dados e Estruturas de Dados (revisão de conceitos)

Os tipos abstractos de dados (*Abstract Data Types*, ADTs) constituem um instrumento fundamental de abstracção, separando a **interface** de uma estrutura de dados (o conjunto de operações disponíveis sobre ela) da sua **implementação** concreta.

Os tipos de dados abstractos são implementados com base em **estruturas de dados concretos**, como sejam por exemplo as *sequências*, as *árvores*, ou os *grafos*, a que está já associada uma forma ou organização interna particular, *linear*, *hierárquica*, ou *relacional*.

É comum definir-se estruturas de dados por **especialização** de outras. Por exemplo,

- Uma **árvore** é um caso particular de grafo (acíclico e com raíz).
- Uma **árvore binária** é um caso particular de árvore (cada nó tem no máximo dois descendentes).
- Uma árvore **binária de procura** (*Binary Search Tree*, BST) é um caso particular de árvore binária (com um invariante que estabelece uma relação de ordem *inorder*).

Cada estrutura de dados (lógica) pode ser implementada de diversas formas, a que correspondem diferentes **estruturas de dados físicas**. Por exemplo uma sequência de elementos de um mesmo tipo pode ter

- Implementação contígua: um *array* (estrutura indexada com acesso em tempo constante, podendo ser estático ou dinâmico);
- Implementação ligada: o acesso ao elemento seguinte é feito através de um campo “próximo”.

Note-se que se um *array* é já uma estrutura física, a noção de sequência ligada é algo que se encontra ainda ao nível lógico, podendo ser implementada também ela sobre um *array*, ou então como uma *lista ligada*, alocada dinamicamente, com utilização de apontadores.

O ADT Dicionário / Array Associativo / ou Mapeamento

Armazena pares chave \rightarrow valor, tendo a semântica de uma função finita. As operações básicas são

1. A **inserção** de um par chave \rightarrow valor;
2. A **alteração** do valor associado a uma chave;
3. A **consulta** com base numa chave, podendo obter-se como resultado um valor ou a indicação de que a chave não ocorre no dicionário;
4. A **remoção** de um par, dada a respectiva chave.

Note-se que as operações 1 e 2 podem ser implementadas pela mesma operação.

Se as chaves forem de *um tipo que admita uma noção de ordem*, um dicionário pode ser implementado por uma árvore binária de procura.

Árvores Binárias de Procura

Uma árvore binária de procura (“binary search tree”, BST) é uma estrutura de dados que pode ser utilizada para implementar *dicionários* ou simplesmente

(multi-)conjuntos, e cujas operações se caracterizam por um comportamento largamente dependente da *forma* da árvore.

Assim, temos os dois seguintes casos extremos:

- Uma árvore *totalmente desequilibrada* assume a forma de uma lista de elementos (cada nó tem sempre um descendente vazio); a operação de procura pode executar no melhor caso em tempo constante, e no pior caso em tempo linear: $T(N) = \Omega(1), O(N)$. O mesmo sucede com a operação de inserção.
- Já numa árvore *equilibrada*, os elementos inseridos ocupam um número de níveis próximo do mínimo possível, e o pior caso da procura passa para logarítmico: $T(N) = \Omega(1), O(\log N)$. As inserções executam *todas* em tempo logarítmico, $T(N) = \Theta(\log N)$.

É claramente desejável trabalhar com árvores equilibradas. Num cenário em que ocorram muito mais operações de procura do que inserções, uma solução possível é reequilibrar a árvore periodicamente (por exemplo, após cada 100 inserções). Mas no caso geral será preferível manter a árvore permanentemente equilibrada.

Árvores AVL

Uma árvore AVL (Adelson-Velskii & E.M. Landis) é uma árvore binária de procura em que todos os nós satisfazem adicionalmente o seguinte invariante estrutural:

As alturas da sub-árvore da esquerda e da sub-árvore da direita diferem no máximo numa unidade: $|h_e - h_d| \leq 1$.

10

10

\

20



Note-se que este invariante admite árvores que não são completas, i.e. é possível que um nível da árvore contenha elementos sem que o nível anterior esteja completamente preenchido.

No entanto, tal como nas árvores completas, a altura de uma árvore AVL é assintoticamente logarítmica, o que garante que a operação de procura executa em tempo $T(N) = O(\log N)$. Mas o que é mais interessante ainda é o seguinte:

O tempo de execução das operações de inserção e remoção, modificadas por forma a efectuarem o necessário ajuste das árvores para preservar o invariante AVL, é também $T(N) = O(\log N)$.

Veremos em seguida como modificar o algoritmo tradicional de inserção numa árvore binária de procura por forma a lidar com árvores AVL.

Algoritmo de inserção numa árvore AVL

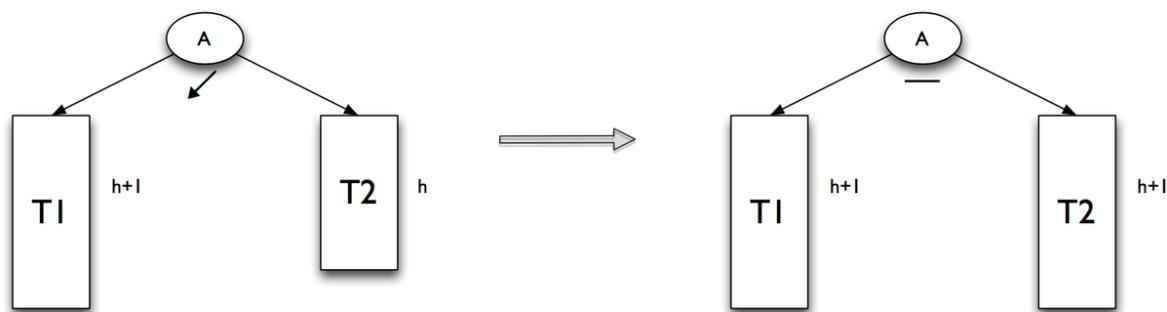
Em algumas situações, a inserção de um novo elemento numa árvore AVL preservará o invariante estrutural ($|h_e - h_d| \leq 1$) em todos os nodos. Consideremos agora em detalhe o que poderá suceder quando se faz, recursivamente, uma inserção à *direita da raiz* (naturalmente, o outro caso é simétrico).

Caso 0: inserção à direita não provoca aumento da altura da sub-árvore da direita.
Neste caso não há nada a fazer, a relação entre h_e e h_d mantém-se.

Nos restantes casos haverá aumento da altura da árvore da direita.

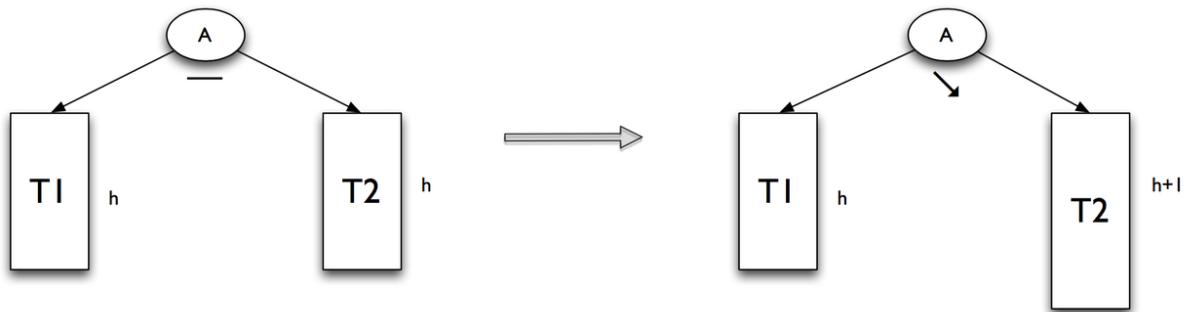
Caso 1: $h_e = h_d + 1$

A sub-árvore da esquerda é mais pesada, e neste caso passaremos a ter, depois da inserção à direita, $h_e = h_d$.



Caso 2: $h_e = h_d$

As duas sub-árvore têm à partida a mesma altura. Neste caso passaremos a ter, depois da inserção à direita, $h_d = h_e + 1$. A raiz (A) continua a satisfazer o invariante.



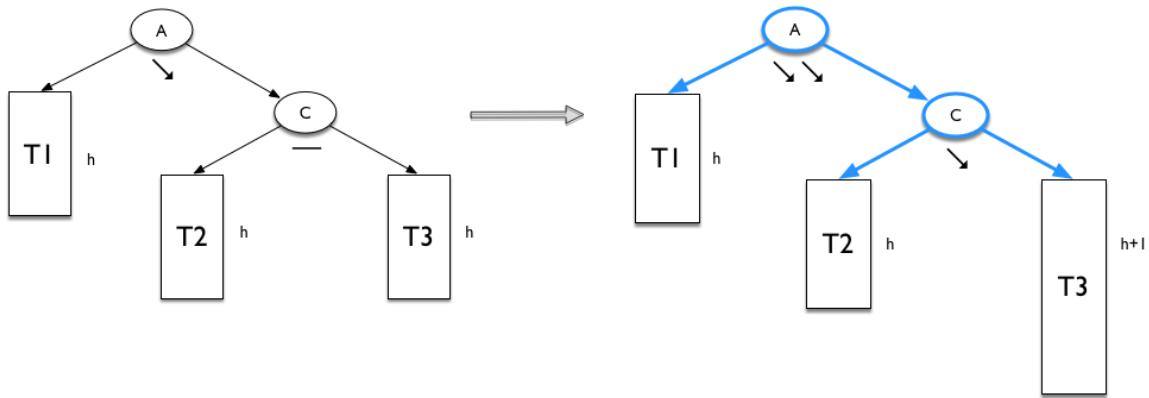
Já se adivinha que o caso problemático, que levará à necessidade de reajustar a árvore, ocorre quando a sub-árvore da direita já é à partida a mais pesada:

Caso 3: $h_d = h_e + 1$.

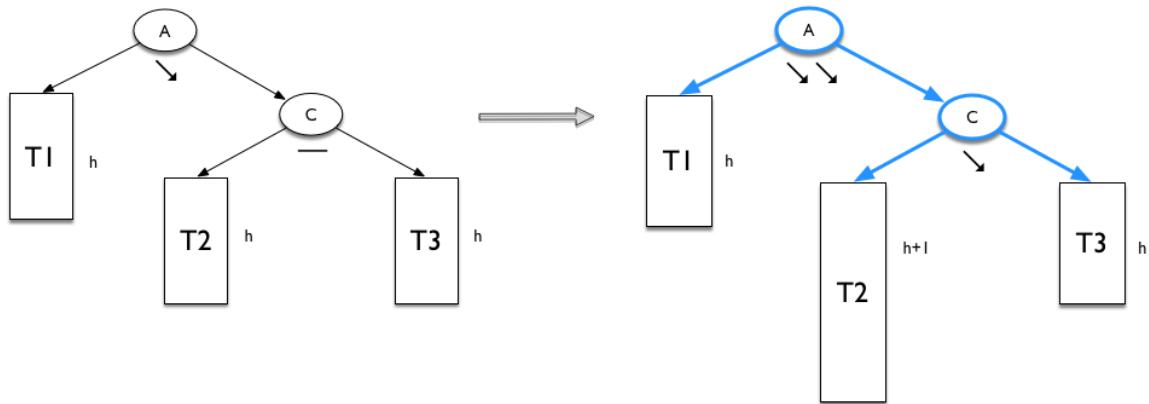
Note-se que neste caso a sub-árvore da direita tem necessariamente duas sub-árvore com a mesma altura h . Chamemos-lhes T2 e T3.

Então, podem agora surgir dois casos diferentes, consoante o aumento de altura ocorra em T2 ou em T3. Ambos os casos levam à violação do invariante na raiz (A), com $h_d = h_e + 2$.

Caso 3a: a inserção produziu um aumento da altura de T3

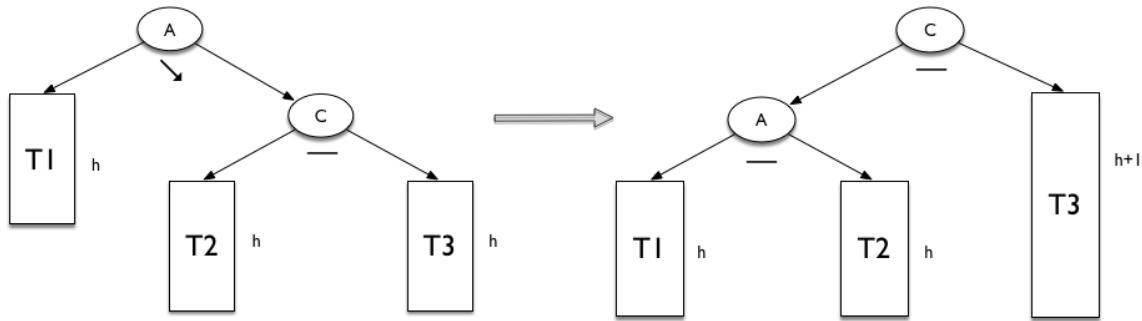


Caso 3b: a inserção produziu um aumento da altura de T2



Vejamos como será reposto o invariante em cada caso, ajustando-se a estrutura da árvore.

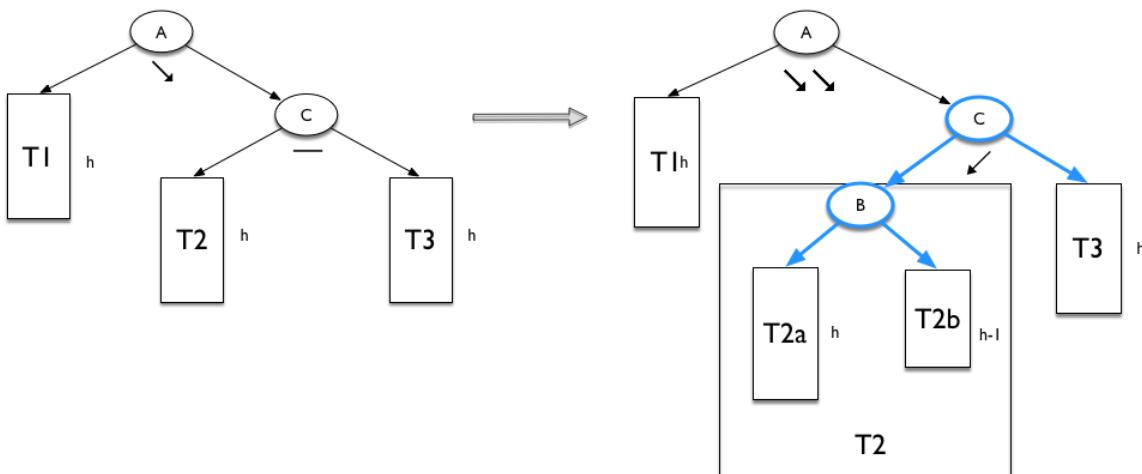
Comecemos pelo caso **Caso 3a**. Neste caso procede-se a uma operação de *rotação à esquerda* da árvore: o nó C tomará o papel da raiz, descendo o nó A para a esquerda de C. A árvore T2 passará a estar à direita de A.



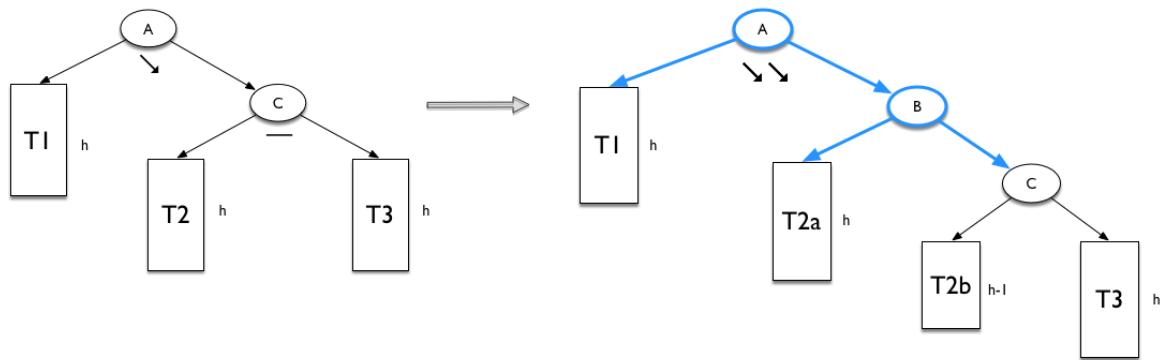
A figura acima mostra que as sub-árvore de A têm a mesma altura, e o mesmo acontece com as de C. Sendo assim, o invariante $|h_e - h_d| \leq 1$ é satisfeito em ambos os nós.

EXERCÍCIO: Mostre que o resultado da rotação continua a ser uma árvore binária de procura, i.e. que a ordem relativa dos elementos é preservada.

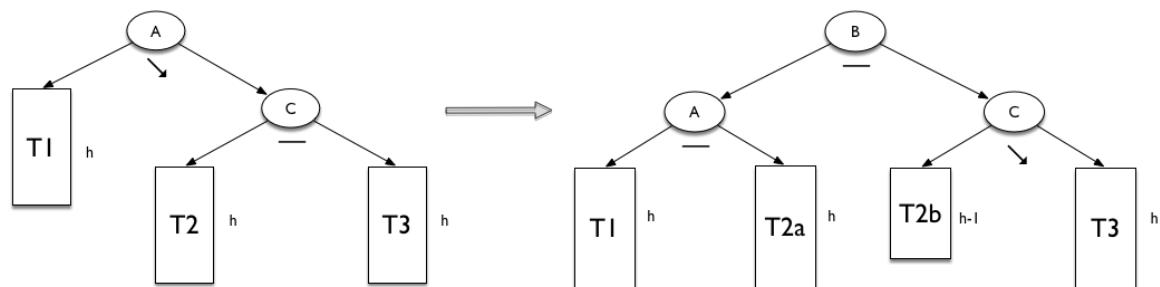
Caso 3b: a inserção produziu um aumento da altura de T2. Necessitamos neste caso de observar a estrutura de T2, com raiz B e sub-árvore T2a e T2b:



Uma simples rotação à esquerda não resolveria neste caso o problema. Antes disso é necessário efectuar uma *rotação à direita*, com eixo no nó C. B sobe para o lugar de C, que desce para a sua direita:



Em seguida faz-se a rotação à esquerda com eixo na raiz A, como no Caso 3A:



Mais uma vez as sub-árvore da nova raiz B têm a mesma altura, e o mesmo acontece com as de C. Note-se que este reajustamento repõe o invariante das árvores AVL, independentemente dos pesos relativos de T2a e T2b.

EXERCÍCIO: Represente graficamente a evolução de uma árvore AVL quando é efectuada a seguinte sequência de inserções: 10, 20, 30, 70, 40, 50. Não se esqueça de indicar os factores de balanceamento de cada nó.

$1\theta, E$

$1\theta, R$

\

$2\theta, E$

$1\theta, R$

\

$2\theta, R$

\

$3\theta, E$

$2\theta, E$

/

\

$1\theta, E$

$3\theta, E$

2θ, R
/ \
1θ, E 3θ, R
 \
7θ, E

2θ, R
/ \
1θ, E 3θ, R
 \
7θ, L
 /
4θ, E

2θ, R
/ \
1θ, E 3θ,
 \
4θ,
 \
7θ

2θ, R
/ \
1θ, E 4θ, E
 / \

```
30,E      70,E  
  
20,R  
/   \  
10,E      40,R  
/   \  
30,E      70,L  
/  
50,E  
  
  
  
  
40,E  
/   \  
20,E      70,L  
/   \  
10,E      30,E      50,E
```

Implementação em C do Algoritmo de Inserção

Tipos de dados

Consideremos uma árvore binária de números inteiros. A estrutura habitual dos nós de uma árvore binária comum será aumentada com um campo correspondente ao *estado de balanceamento* do nó, que poderá ser um de três valores:

- LH (sub-árvore da esquerda tem maior altura),
- RH (sub-árvore da direitura tem maior altura), ou
- EH (alturas iguais).

```
typedef int TreeEntry;
typedef enum balancefactor { LH , EH , RH } BalanceFactor;

struct treenode {
    BalanceFactor bf;
    TreeEntry entry;
    struct treeNode *left;
    struct treeNode *right;
};

typedef struct treenode *Tree;
```

Funções auxiliares

Vimos acima que o ajuste da estrutura das árvores tem por base operações simples de *rotação*.

A seguinte função realiza uma rotação simples à esquerda da árvore com raiz apontada por t, devolvendo o endereço da nova raiz. Note que a função não pode ser executada se a árvore da direita for vazia!

```
// requires:
```

```

// (t != NULL) && (t->right != NULL)
//
Tree rotateLeft(Tree t)
{
    Tree aux = t->right;
    t->right = aux->left;
    aux->left = t;
    t = aux;
    return t;
}

```

EXERCÍCIO: Implemente a função de rotação à direita.

Escreveremos em seguida uma função que, recebendo uma árvore cuja raiz deixou de satisfazer o invariante AVL e que está desequilibrada para a direita (i.e. $h_d = h_e + 2$), corriga a estrutura da árvore.

Para isso, a função:

- começa por determinar se o caso exige uma rotação simples ou dupla, examinando o indicador de balanceamento do nó à direita;
- no segundo caso (rotação dupla), e tendo em conta o que foi dito anteriormente, é necessário ajustar os indicadores de balanceamento dos nós à esquerda (A na figura) e à direita (C) da nova raiz da árvore (B), de acordo com o estado actual do indicador de balanceamento de B;
- finalmente ajusta-se o indicador da raiz.

Tal como a anterior, esta função não pode ser executada se a árvore da direita for vazia.

```

// requires:
// (t != NULL) && (t->right != NULL)
//
Tree balanceRight(Tree t)
{

```

```

if (t->right->bf==RH) {
    // Rotacao simples, caso 3a
    t = rotateLeft(t);
    t->bf = EH;
    t->left->bf = EH;
}
else {
    // Dupla rotação, caso 3b
    t->right = rotateRight(t->right);
    t=rotateLeft(t);
    switch (t->bf) {
        case EH:
            t->left->bf = EH;
            t->right->bf = EH;
            break;
        case LH:
            t->left->bf = EH;
            t->right->bf = RH;
            break;
        case RH:
            t->left->bf = LH;
            t->right->bf = EH;
    }
    t->bf = EH;
}
return t;
}

```

EXERCÍCIO: Implemente a função `balanceLeft`.

Função de inserção numa árvore binária de procura comum

O tipo esperado para uma função de inserção não deverá ser novidade: recebe um apontador para a raiz da árvore e o elemento a inserir, e devolve o endereço da raiz (que poderá ter sido alterado). O algoritmo recursivo é bem conhecido:

```
Tree insertTree(Tree t, TreeEntry e) {  
    if (t==NULL){  
        t = (Tree)malloc(sizeof(struct treenode));  
        t->entry = e;  
        t->right = t->left = NULL;  
    }  
    else if (e > t->entry)  
        t->right = insertTree(t->right, e);  
    else  
        t->left = insertTree(t->left, e);  
    return t;  
}  
+
```

Função de inserção numa árvore AVL

A função de inserção numa árvore AVL deve devolver, além do endereço da nova raiz da árvore, também *informação sobre se a altura da árvore cresceu ou não após esta inserção*. Para isso incluiremos no protótipo um parâmetro de tipo int passado por referência:

```
Tree insertTree(Tree t, TreeEntry e, int *cresceu);
```

A ideia é que depois da chamada `insertTree(t, e, &c)` com `c` de tipo `int`, `c` terá o valor 1 se a altura de `t` cresceu, e 0 em caso contrário.

O algoritmo de inserção numa árvore AVL segue a estrutura do anterior, com as seguintes diferenças:

- detecta os casos em que o invariante AVL é violado, e chama a função `balanceRight` ou `balanceLeft` para corrigir a estrutura da árvore
- reajusta os indicadores de balanceamento dos nós afectados pela inserção
- determina se a altura da árvore aumentou ou não, e atribui o valor adequado 0 ou 1 a `*cresceu`

EXERCÍCIO: Complete a definição da seguinte função de inserção:

```
Tree insertTree(Tree t, TreeEntry e, int *cresceu)
{
    if (t==NULL){
        t = (Tree)malloc(sizeof(struct treenode));
        t->entry = e;
        t->right = t->left = NULL;
        t->bf = EH;
        *cresceu = 1;
    }
    else if (e > t->entry) {
        t->right = insertTree(t->right, e, cresceu);
        if (*cresceu) {
            switch (t->bf) {
                case LH:
                    t->bf = EH;
                    *cresceu = 0;
                    break;
                case EH:
                    t->bf = RH;
                    *cresceu = 1;
            }
        }
    }
}
```

```

        *cresceu = 1;
        break;

    case RH:
        t = balanceRight(t);
        *cresceu = 0;
    }

}

else {
    t->left = insertTree(t->left,e,cresceu);
    if (*cresceu) {
        ...
    }
    return t;
}

```

Outros Exercícios

1. A seguinte função calcula a altura de uma qualquer árvore binária, em tempo $\Theta(N)$:

```

int nonAVL_treeHeight(Tree t) {
    int l, r;
    if (t==NULL) return 0;
    l = treeHeight(t->left);
    r = treeHeight(t->right);
    if (l>r) return l+1;
    else return r+1;
}

```

```
}
```

Redefina a função por forma a calcular a altura de uma árvore AVL em tempo $\Theta(\log N)$.

2. É possível testar se uma árvore binária é ou não AVL da seguinte forma:

```
int isAVL (Tree t) {  
    int l, r;  
    if (t == NULL) return 1;  
  
    l = treeHeight (t->left);  
    r = treeHeight (t->right);  
  
    return (abs (l-r) <= 1 &&  
            isAVL(t->left) &&  
            isAVL(t->right));  
}
```

- Analise o tempo de execução no pior caso desta função.
- É possível optimizar esta função alterando-a por forma a calcular simultaneamente a altura da árvore, dispensando assim a utilização da função `treeHeight`. Complete a seguinte definição e analise o seu tempo de execução no pior caso:

```
// altura da árvore será colocada em *p  
int isAVL_aux (Tree t, int *p) {  
    ...  
}  
  
int isAVL_opt (Tree a) {  
    int p;
```

```
    return (isAVL_aux (a, &p));  
}
```

ED4: Tabelas de “hash”

Projecto Codeboard:
<https://codeboard.io/projects/65989>

Arrays como Dicionários

Um simples *array* de comprimento N implementa um dicionário, com o conjunto $\{0 \dots N - 1\}$ como universo de chaves.

- O tipo do array deve ser o tipo dos valores que se pretende associar às chaves
- Um valor especial pode ser usado para sinalizar que uma chave não ocorre no dicionário.

No entanto, se o universo de chaves for muito grande, torna-se incompatível a utilização directa. Por exemplo, se as chaves forem inteiros de 32 bits, teríamos de utilizar um array de comprimento 2^{32} , superior a 4 mil milhões!!!

A técnica de *hashing* permite separar o universo de chaves do conjunto de índices do array. Por exemplo, se se pretende dimensionar um dicionário com chaves de 32 bits, mas que se prevê nunca tenha uma ocupação superior a 10000, basta usar uma função h que distribua, ou disperse (*hash*) as chaves pelas posições do array:

$$h : \{0 \dots 2^{32} - 1\} \rightarrow \{0 \dots 9999\}$$

Esta função é não-injectiva por natureza, o que significa que ocorrerão **colisões**: duas chaves poderão ser mapeadas para a mesma posição do array, e só uma pode ser inserida.

Uma característica desejável destas funções de *hash* é a **uniformidade**: todas as posições do array devem ter a mesma probabilidade de ser calculadas como resultado. Isto permitirá distribuir uniformemente a informação inserida no array, minimizando o número de colisões, bem como a formação de *clusters* em zonas específicas do array.

Quando `cap` é um número primo, a seguinte função básica comporta-se de forma razoavelmente uniforme:

```
int hash (int k, int cap) {  
    return k%cap;  
}
```

O desenho de uma tabela de hash implica a escolha de uma *estratégia para a resolução de colisões*, que permita inserir na tabela chaves que são à partidas mapeadas na mesma posição do array

Condicionamento de Chaves

Se as chaves não forem números naturais, devem ser previamente condicionadas, i.e. mapeadas em números naturais, novamente de forma determinista tão uniforme quanto possível.

Por exemplo, tratando-se de chaves de tipo *string* de caracteres:

```
int condition (char *s) {  
    int r = 0;  
    while (*s)  
        r += *s++;  
    return r;  
}
```

Tabelas de hash

Uma tabela de hash de capacidade cap é uma estrutura de dados física que implementa um dicionário de pares de tipo $K \rightarrow V$, e que consiste em:

1. Uma função de *hash* de tipo $h : K \rightarrow \{0, \dots, cap - 1\}$;
2. De acordo com a estratégia de resolução de colisões adoptada:
 - **Open addressing**: um array com posições $\{0 \dots cap - 1\}$ de pares (k, v) , ou
 - **Closed addressing**: um array com posições $\{0 \dots cap - 1\}$ de (apontadores para) listas ligadas de pares (k, v) .

O desenho de uma tabela de hash pretende sempre equilibrar a eficiência espacial e temporal.

- No limite, se $cap = \#(K)$, não há colisões e as operações de inserção, pesquisa, e remoção executam em tempo constante. Mas claro, desperdiça-se potencialmente muito espaço, porque o tamanho útil da tabela poderá ser muito inferior a $\#(K)$
- Ao diminuir substancialmente o tamanho cap do array, melhora-se a gestão do espaço, às custas de piorar a performance temporal das operações
- O princípio por que se rege o desenho das tabelas é que o tempo deverá sempre ser *tendencialmente constante*

Factor de Carga e Redimensionamento

Qualquer que seja a estratégia de resolução de colisões, terá de ser mantido num valor razoavelmente baixo a taxa de ocupação da tabela, ou *factor de carga*:

$$\alpha = \frac{\#\text{chaves inseridas}}{cap}$$

Ao desenhar uma tabela deve estipular-se um valor máximo para este factor, por exemplo $\alpha_{max} = 0.8$. Quando $\alpha = \alpha_{max}$ deverá redimensionar-se a tabela, que por esta razão deverá ser implementada por um *vector dinâmico*.

Relembre-se que as operações de redimensionamento (duplicação do tamanho da tabela) executam em *tempo amortizado constante*.

Closed Addressing

Uma solução possível para fazer “caber” vários pares chave → valor na mesma posição de um array é externalizar a informação, criando uma *lista ligada* cujo endereço inicial é guardado no array:

```
typedef struct node {  
    char key[MAXSTR];  
    ValueType info;  
    struct node * next;  
};  
typedef struct node *Hashtable[CAP];
```

Chama-se a esta implementação uma tabela encadeada (tabela com “chaining”)

A inserção na tabela faz-se agora mediante uma inserção na lista ligada apropriada.

Consideremos inserção dos pares (k_1, v_1) e (k_2, v_2) , com $h(k_1) = p$ e também $h(k_2) = p$.

Null		Null		Null	
Null		Null		Null	
Null		Null		Null	
Null	\xrightarrow{p}	$\rightarrow (k_1, v_1) \rightarrow \text{Null}$		\xrightarrow{p}	$(k_2, v_2) \rightarrow (k_1, v_1) \rightarrow \text{Null}$
Null		Null		Null	
Null		Null		Null	
Null		Null		Null	

Pontos a reter:

- Não é suposto que estas listas cresçaam indefinidamente: apesar de em teoria poder ser $\alpha > 1$, com um valor alto do factor de carga o tempo de execução das operações deixaria de ser “tendencialmente constante”
- As tabelas devem pois ser redimensionadas quando necessário, assegurando-se um factor de carga pequeno.
- O tempo de execução no pior caso de uma inserção ou consulta será $\Theta(1)$, desde que a função de hash seja uniforme e o factor de carga (que corresponderá ao comprimento médio das listas) pequeno.

Open Addressing

Ocorrendo uma colisão, procurar-se-á inserir a segunda chave numa outra posição do array, usando um método que possa ser reproduzido (nomeadamente quando se efectuar consultas)

Linear Probing

O método mais trivial de endereçamento aberto é conhecido por *linear probing*: a colisão resolve-se inserindo na posição seguinte (com circularidade) do vector.

Consideremos de novo a inserção dos pares (k_1, v_1) e (k_2, v_2) , por esta ordem, com $h(k_1) = p$ e $h(k_2) = p$.

	empty, -

⇒

	empty, -
	empty, -
	empty, -
p	k_1, v_1
	empty, -
	empty, -
	empty, -

⇒

	empty, -
	empty, -
	empty, -
p	k_1, v_1
$p + 1$	k_2, v_2
	empty, -
	empty, -

Ou:

	empty, -

⇒

	empty, -
p	k_1, v_1

⇒

$p + 1$	k_2, v_2
	empty, -
p	k_1, v_1

Além disso, “próxima posição” deve ser de facto interpretado como “próxima posição livre”. Seja $h(k_0) = p + 1$ e $h(k1) = h(k2) = p$, com k_0 inserido antes de k_1 :

	empty, -
$p + 1$	k_0, v_0
	empty, -
	empty, -

⇒

	empty, -
	empty, -
	empty, -
p	k_1, v_1
$p + 1$	k_0, v_0
	empty, -

⇒

	empty, -
	empty, -
	empty, -
p	k_1, v_1
$p + 1$	k_0, v_0
$p + 2$	k_2, v_2
	empty, -

Naturalmente, a operação de consulta deve reproduzir a mesma sequência de *probes* utilizada na inserção.

| Qual deverá ser o critério de paragem de uma operação de consulta?

Remoção de Chaves

Efectuemos agora as operações `insert(k1, v1)`, `insert(k2, v2)`, e `remove(k1)` por esta ordem, ainda com $h(k_1) = p$ e $h(k_2) = p$:

The diagram illustrates a hash table with 8 slots. On the left, the table has slots at indices 0 through 7. At index 0, the value is empty, _ . At index 1, the value is empty, _ . At index 2, the value is empty, _ . At index 3, the value is empty, _ . At index 4, the value is p , k_1, v_1 . At index 5, the value is $p + 1$, k_2, v_2 . At index 6, the value is empty, _ . At index 7, the value is empty, _ . An arrow points from this state to the right, indicating the transition. On the right, the table has slots at indices 0 through 7. At index 0, the value is empty, _ . At index 1, the value is empty, _ . At index 2, the value is empty, _ . At index 3, the value is empty, _ . At index 4, the value is p , empty, _ . At index 5, the value is $p + 1$, k_2, v_2 . At index 6, the value is empty, _ . At index 7, the value is empty, _ . The value at index 4 is highlighted in red as empty, _ , indicating it was removed.

	empty, _
p	k_1, v_1
$p + 1$	k_2, v_2
	empty, _
	empty, _

⇒

	empty, _
p	empty, _
$p + 1$	k_2, v_2
	empty, _
	empty, _

| O que sucede quando se efectuar agora uma consulta com a chave k_2 ?

Não é adequado marcar as posições onde ocorreram remoções como `empty`. Utiliza-se uma chave alternativa `removed`, que indica que uma pesquisa deve continuar para além daquela posição.

No entanto, a utilização desta chave `removed` contribui para a degradação da performance da operação de consulta. Ao fim de algum tempo já não haverá chaves `empty`, o que significa que as pesquisas de chaves inexistentes na tabela executarão todas em tempo linear.

É pois necessário proceder periodicamente a um “refrescamento” da tabela, reinicializá-la e voltando a inserir todos os pares, por forma a eliminar as chaves `removed`. Se a tabela for redimensionada frequentemente isto não será necessário, uma vez que ao redimensionar eliminam-se naturalmente as chaves `removed`.

Clustering

Um problema da estratégia de *linear probing* é a formação de clusters.

A probabilidade de cada posição ser preenchida é inicialmente dada por $r = \frac{1}{cap}$.

Revisitemos a sequência de inserções anterior, calculando a probabilidade de inserção em cada posição:

(r)	empty, -		(r)	empty, -		(r)	empty, -
(r)	empty, -		(r)	empty, -		(r)	empty, -
(r)	empty, -		(r)	empty, -		(r)	empty, -
(r)	empty, -		(0) p	k_1, v_1		(0) p	k_1, v_1
(r)	empty, -		(2*r)	empty, -		(0) p + 1	k_2, v_2
(r)	empty, -		(r)	empty, -		(3*r)	empty, -
(r)	empty, -		(r)	empty, -		(r)	empty, -

Este fenómeno de aumento da probabilidade de inserção em posições subsequentes às já preenchidas resulta na formação de “clusters”, que deterioram localmente o comportamento das operações sobre a tabela.

Este fenómeno pode ser mitigado com a utilização de outras técnicas de *open addressing*, como *quadratic probing*.

Quadratic Probing

Em vez de fazer os probes

$$p, p + 1, p + 2, p + 3 \dots$$

faz-se:

$$p, p + 1^2, p + 2^2, p + 3^2 \dots$$

Esta técnica reduz substancialmente a formação de clusters, às custas de priorar o potencial para aproveitamento de *caching* por apresentar menor grau de localidade.

Dados Empíricos e Comparação

O quadro seguinte contém o número de comparações efectuadas numa consulta de uma tabela com 900 chaves.

Factor de carga α	0.1	0.5	0.8	0.9	0.99	2.0
Consulta com sucesso, <i>chaining</i>	1.04	1.2	1.4	1.4	1.5	2.0
Consulta com sucesso, <i>quadratic probing</i>	1.04	1.5	2.1	2.7	5.2	–
Consulta com sucesso, <i>linear probing</i>	1.05	1.6	3.4	6.2	21.3	–
Consulta sem sucesso, <i>chaining</i>	0.10	0.5	0.8	0.9	0.99	2.0
Consulta sem sucesso, <i>quadratic probing</i>	1.13	2.2	5.2	11.9	126	–
Consulta sem sucesso, <i>linear probing</i>	1.13	2.7	15.4	59.8	430	–

Fonte:

Kruse R.L., Leung B.P., Tondo C.L., *Data Structures and Program Design in C*. Prentice-Hall, 2nd. ed., 1991

O que explica o facto de, com open addressing, o tempo da consulta sem sucesso ser substancialmente maior?

Conclusões

- Para factores de carga razoáveis (≤ 0.8), ambas as soluções alcançam número constante de comparações nas consultas. Mas esta contagem não é o único

aspecto relevante!

- *Open addressing* devidamente optimizado (*quadratic probing*, redimensionamento dinâmico para evitar factores de carga elevados) pode ser uma boa escolha, porque
 - não é penalizador em termos de espaço, ao contrário de implementações baseadas em *closed addressing / chaining*
 - apresenta vantagens em termos de *caching* (mesmo com *quadratic probing*). A grande desvantagem é a dificuldade em lidar com remoções.
- *Closed addressing / chaining* é eficiente e de programação simples, mas com um custo adicional de espaço relevante, e com as desvantagens inerentes à utilização de estruturas ligadas em termos de localidade.

G1. Representação de Grafos em Computador

PROJECTO CODEBOARD DE SUPORTE A ESTE MÓDULO:

<https://codeboard.io/projects/10725>

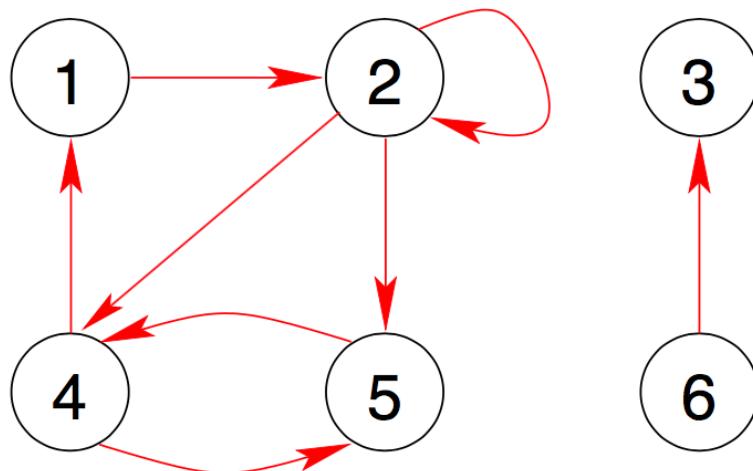
Grafos orientados

Um *grafo orientado* é um par (V, E) com V um conjunto finito de vértices ou *nós* e E uma relação binária sobre V – o conjunto de *arestas* ou *arcos* do grafo.

Exemplo:

$$V = \{1, 2, 3, 4, 5, 6\}$$

$$E = \{(1, 2), (2, 2), (2, 4), (2, 5), (4, 1), (4, 5), (5, 4), (6, 3)\}$$



Grafo orientado $G1$

- Se $(i, j) \in E$, j diz-se *adjacente a i*
 i e j são respectivamente os vértices de *origem* e *destino* da aresta (i, j)
- O *grau de entrada* de um vértice é o número de arestas com destino nesse vértice
- O *grau de saída* de um vértice é o número de arestas com origem nesse vértice

- Uma aresta (i, i) designa-se por **anel**
- O número máximo de arestas de um grafo com conjunto de vértices V é V^2
- Um grafo diz-se *esparso* se o número de arestas for muito inferior a V^2 , e *denso* em caso contrário

As aplicações dos grafos orientados são inúmeras; além da evidente modelação de redes (por exemplo de estradas), que discutiremos mais à frente, estes grafos têm aplicação por exemplo em *gestão de projectos*, fazendo corresponder os vértices a **tarefas**. As arestas podem exprimir uma relação de precedência: se $(t_1, t_2) \in E$, então a tarefa t_1 deve sempre ser realizada antes da tarefa t_2 .

Representação em Computador de Grafos Orientados

Representação por uma Matriz de Adjacências

Assumindo uma **ordenação natural** (enumeração) dos vértices do grafo, a representação mais imediata é por uma matriz de valores Booleanos (ou 0/1) tal que:

- a posição (i, j) contém o valor 1 sse $(i, j) \in E$

	1	2	3	4	5	6
1	0	1	0	0	0	0
2	0	1	0	1	1	0
3	0	0	0	0	0	0
4	1	0	0	0	1	0
5	0	0	0	1	0	0
6	0	0	1	0	0	0

Em **C** poderemos ter, por exemplo,

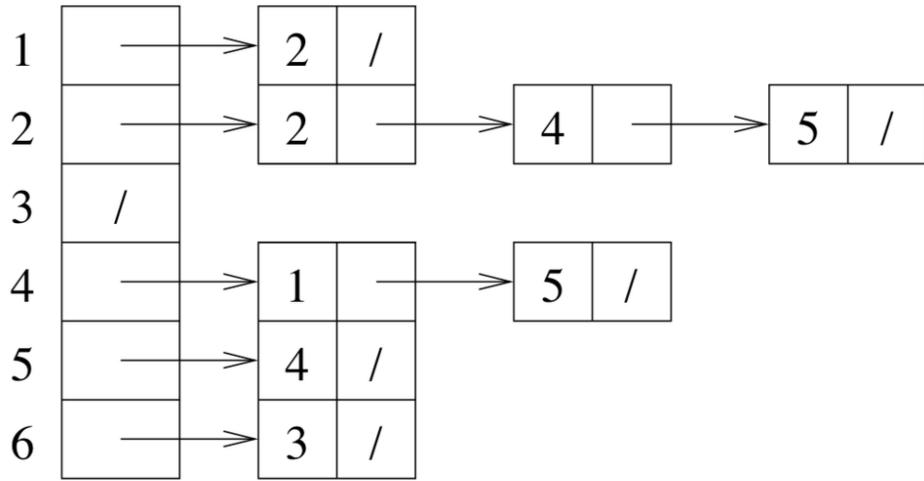
```
#define MAX 100
typedef char GraphM[MAX][MAX];
```

Representação por Listas de Adjacências

Uma representação alternativa consiste em associar a cada vértice do grafo uma lista contendo os vértices que lhe são adjacentes.

Em termos mais concretos, numa linguagem de programação como o C, assumindo uma **ordenação natural** (enumeração) dos vértices do grafo, podemos considerar

- um *array de apontadores*
- cujos índices correspondem aos vértices do grafo, e
- contendo em cada posição o (endereço do) primeiro elemento da lista de adjacência do vértice respectivo



Observe-se que a soma dos comprimentos das listas ligadas é $|E|$.

Uma possível definição de tipos em C será:

```
#define MAX 100
struct edge {
    int dest;
    struct edge *next;
};
typedef struct edge *GraphL[MAX];
```

Observe-se que o último `typedef` define o tipo `GraphL` como sendo o dos arrays, de comprimento `MAX`, de apontadores para estruturas `struct edge`.

Comparação das representações

A escolha de representação poderá passar pela consideração da **densidade** do grafo, e também por uma estimativa do número de **testes de adjacência** que serão efectuados durante a vida desta estrutura, e também da consulta de conjuntos de vértices adjacentes que serão efectuadas sobre o grafo

Com uma matriz de adjacências,

- O espaço de memória necessário para representar um grafo (V, E) é $\Theta(V^2)$, independente do número de arcos, e portanto da densidade do grafo;
- é possível verificar em tempo constante se dois vértices são adjacentes;
- para conhecer os adjacentes a um determinado vértice u , é necessário percorrer todos os vértices v do grafo, consultando para cada um a posição (u, v) da matriz.

Por outro lado, com listas de adjacências,

- O espaço de memória necessário para esta representação de um grafo (V, E) é $\Theta(V + E)$, o que a torna uma representação eficiente no caso de grafos esparsos;
- o teste de adjacência de dois vértices obriga à travessia de uma lista ligada, executada no pior caso em tempo $\Theta(V)$;
- a consulta dos adjacentes a um vértice u implica apenas percorrer a lista de adjacências de u , em vez de percorrer todos os vértices. Num grafo pouco denso, o impacto na implementação de um algoritmo que efectue esta operação repetidamente pode ser muito grande. É o caso por exemplo dos algoritmos de travessia de grafos.

Em resumo, se o espaço utilizado não for uma questão crítica, a escolha de representação passará pelo tipo de operação de consulta (adjacência isolada par a par VS. lista de adjacências de um vértices) preponderante nos algoritmos a implementar.

Grafos com Pesos

Em certos contextos é útil associar informação (**pesos**) às arestas de um grafo, em particular numérica. Uma utilização típica de grafos é na modelação de redes de infraestruturas, por exemplo:

- Uma rede de estradas (“mapa”), em que os vértices correspondem a localidades e as arestas a estradas entre elas. Os pesos das arestas neste caso podem corresponder ao comprimento dessas ligações / distância entre duas localidades ligadas por uma estrada directa.
- Uma rede de transporte ou abastecimento de algum recurso, como por exemplo água. Neste caso as ligações corresponderão a canais entre nós da rede, e o peso de uma aresta poderá ser a *capacidade* desse canal.

Representação em Computador de Grafos com Pesos

É imediato adaptar qualquer uma das duas representações acima para grafos com pesos numéricos associados às arestas.

Representação por uma Matriz de Adjacências

Basta armazenar na matriz:

- em vez do valor Booleano 1 para representar a existência de uma aresta, o peso dessa aresta;
- em vez do valor Booleano 0 para representar a inexistência de uma aresta, um valor especial **NE**

A escolha do valor **NE** depende da aplicação em questão. Considerando os dois exemplos anteriores:

- Num grafo que modela uma rede de estradas, a escolha natural para representar a ausência de uma ligação entre duas localizações será $\text{NE} = \infty$ (distância infinita)
- Já num grafo que modela uma rede, em que os pesos correspondem a capacidades de canais, a escolha natural para representar a ausência de um canal entre dois nós da rede será $\text{NE} = 0$

Em **C** poderemos ter por exemplo a seguinte definição de tipo:

```
#define NE 0
#define MAX 100
```

```
typedef int WEIGHT;
typedef WEIGHT GraphM[MAX][MAX];
```

Representação por Listas de Adjacências

As arestas do grafo são aqui representadas por nós das listas ligadas de adjacências. Sendo assim, basta criar um campo adicional nestas estruturas (nós das listas) para guardar o peso das arestas.

Note-se que, uma vez que nesta representação o teste de existência de uma aresta não é feito pela consulta do valor numérico do peso (como era o caso com uma matriz de adjacências), mas sim pela existência ou não de um nó com um determinado destino na lista ligada, não é agora necessária a utilização de um valor especial **NE** para representar o peso de uma aresta inexistente.

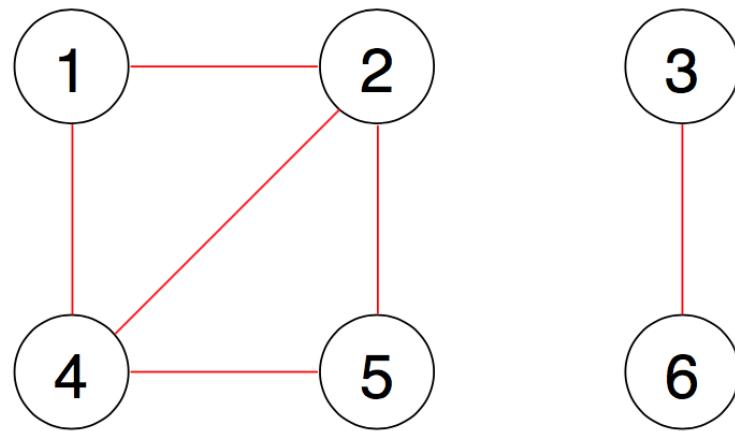
Em C:

```
#define NE 0
#define MAX 100
typedef int WEIGHT;
struct edge {
    int dest;
    WEIGHT weight;
    struct edge *next;
};
typedef struct edge *GraphL[MAX];
```

Grafos Não-orientados

Num grafo não-orientado as arestas são conjuntos com dois vértices $\{u, v\} \in E$ em vez de pares ordenados. Por outras palavras, as aresta são bi-direccionais, o que é

adequado, por exemplo, para modelar redes em que todas as ligações entre pares de vértices funcionam nos dois sentidos.



Num grafo não-orientado não se considera habitualmente a existência de anéis (arestas da forma $\{v, v\}$).

Tal como os grafos orientados, estes grafos podem ou não ter pesos associados às arestas.

Representação em Computador de Grafos Não-orientados

A representação típica de um grafo não-orientado passa pela sua conversão para um grafo orientado *simétrico*, em que se $\{u, v\} \in E$ então também $\{v, u\} \in E$.

Note-se que uma tal representação contém *redundância*:

- No caso da representação por uma matriz de adjacências poder-se-á eliminar esta redundância representando de forma eficiente apenas uma matriz triangular.
- No caso da representação por listas de adjacências a eliminação da redundância será quase de certeza uma má ideia. Se se representar a aresta $(u, v) \in E$ apenas por um nó, na lista de adjacência de u ou de v , então, para ter acesso a todos os vértices adjacentes a um qualquer nó não bastará percorrer a sua lista de adjacências; será necessário percorrer *todas* as listas de adjacências do grafo.

EXERCÍCIOS

No projecto codeboard, o ficheiro `main.c` contém código de teste das funções que deverá definir. Os protótipos das funções encontram-se no ficheiro `conversion.c` [<https://codeboard.io/projects/10725>]

Considere a representação em C de grafos com pesos, com os tipos de dados fornecidos acima.

1. Defina funções de conversão entre as duas representações (matrizes e listas de adjacências).
2. Para cada uma das representações defina funções de cálculo do **grau de entrada** e de **saída** de um vértice.
 - a. Analise o tempo de execução das 4 funções definidas.
3. Assuma agora uma representação por listas de adjacências.

A *capacidade* de um vértice v num grafo define-se como a diferença entre a soma dos pesos das arestas que têm v como destino e a soma dos pesos das arestas que têm v como origem. Defina uma função que calcula a capacidade de um vértice num grafo.

 - a. Analise o esforço computacional desta função.
4. Defina uma função `maxCap` que determina o vértice do grafo com maior capacidade. A função deverá executar em tempo $\Theta(|V| + |E|)$.

EXERCÍCIOS ADICIONAIS

1. Defina uma função `colorOK` (`GraphL g, int color[]`) que, dado um grafo não orientado `g` e um vector de inteiros `cor` verifica se essa coloração é válida. Diz-se que uma coloração é válida sse vértices adjacentes tiverem cores diferentes.

2. Defina em C uma função que calcula o inverso de um grafo (um grafo que tem uma aresta (u, v) se e só se existe uma aresta (v, u) no grafo original).

G2. Algoritmos de Travessia de Grafos

(por vezes também designados por algoritmos de **pesquisa**)

PROJECTO CODEBOARD DE SUPORTE A ESTE MÓDULO:

<https://codeboard.io/projects/10725>

Caminhos em grafos orientados

Num grafo (V, E) , um *caminho* do vértice v_0 para o vértice v_k é uma sequência de vértices

$$\langle v_0, v_1, \dots, v_k \rangle$$

tais que $v_i \in V$ para todo o $i \in \{0, \dots, k\}$, e $(v_i, v_{i+1}) \in E$ para todo o $i \in \{0, \dots, k-1\}$

Alternativamente, este caminho pode ser visto como uma sequência de arestas

$$(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$$

O *comprimento* deste caminho é o número k de arestas nele contidas.

Um vértice v é **alcançável** a partir do vértice s se existe um caminho de s para v .

Num grafo orientado, isto não implica que s seja alcançável a partir de v .

Um **ciclo** é um caminho de comprimento ≥ 1 com início e fim no mesmo vértice.
(Note-se que existe sempre um caminho de comprimento 0 de um vértice para si próprio, que não se considera ser um ciclo)

Um grafo diz-se **acíclico** se não contém ciclos. Um **grafo orientado acíclico** é usualmente designado por **DAG**, de *Directed Acyclic Graph*.

Florestas e Árvores

Uma **árvore** (G, V) (ou no caso mais geral uma **floresta**) é um caso particular de um DAG: um grafo orientado que, além de acíclico, satisfaz a seguinte restrição:

O grau de entrada de todo o vértice $v \in V$ é 0 ou 1.

Se $(u, v) \in E$ dizemos que o vértice u é o “pai” de v .

Os vértices com grau de entrada 0 são as **raízes** da floresta (vértices sem “pai”). Caso exista apenas uma raiz estamos em presença de uma árvore.

Travessia de Grafos

Um algoritmo de **travessia** de um grafo (V, E), a partir de um vértice inicial $s \in V$ dado, visita **todos os vértices alcançáveis** a partir de s , percorrendo caminhos seguindo uma determinada estratégia, **não passando mais do que uma vez por cada vértice**.

Os caminhos percorridos durante a travessia constituem um sub-grafo de (V, E) que é de facto uma árvore, designada por **árvore de travessia** do grafo. Diferentes estratégias produzirão árvores diferentes.

Note-se que não se trata aqui de um algoritmo para a resolução de um problema específico, mas antes de uma família de algoritmos. Para **resolver um problema específico** haverá que:

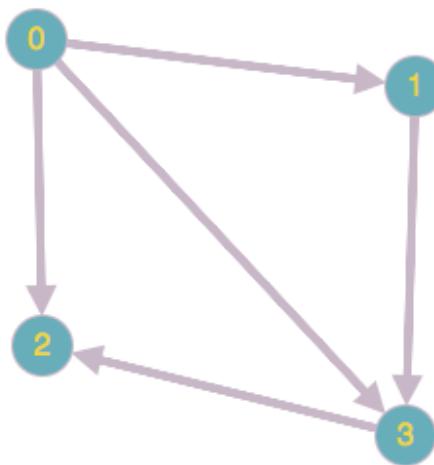
1. seleccionar a estratégia de travessia adequada para esse problema
2. adaptar o esquema geral da travessia para a resolução do problema em causa

Representação de árvores de travessia

Uma árvore de travessia pode ser representada por um simples array indexado pelos vértices do grafo, que associe a cada vértice v o seu pai na árvore, i.e. o vértice a partir do qual v foi alcançado durante a travessia. Utiliza-se por exemplo o valor -1 para identificar a raiz da árvore.

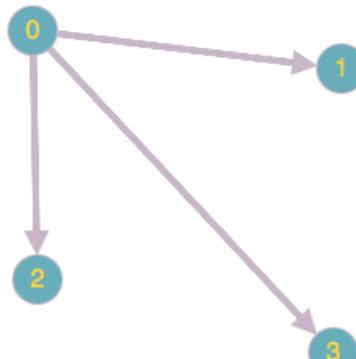
EXEMPLO

[O grafo seguinte foi desenhado em <http://graphonline.ru/en/>]

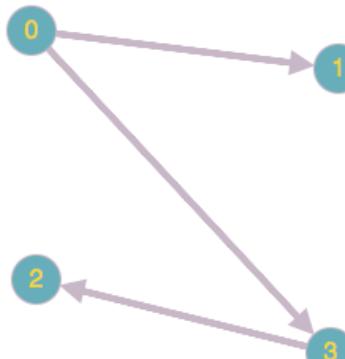


Grafo orientado G_2

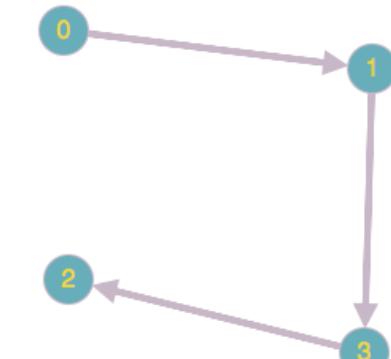
Existem várias travessias diferentes possíveis de G_2 com início no vértice 0. As árvores de travessia seguintes correspondem a 3 delas:



$pais = [-1, 0, 0, 0]$



$pais = [-1, 0, 3, 0]$



$pais = [-1, 0, 3, 1]$

Estado de um vértice e estruturas de dados auxiliares

Durante a execução de uma travessia, cada vértice de um grafo pode encontrar-se num de três estados, a que associaremos um código de cores:

- ainda não alcançado pela travessia [BRANCO]
- já alcançado, mas alguns dos seus vértices adjacentes ainda não alcançados [CINZENTO]
- já processado (todos os seus adjacentes foram alcançados) [PRETO]

Observe-se que

Os vértices cinzentos constituem uma fronteira entre os que já foram completamente tratados pela travessia e os que não foram ainda alcançados.

Para o controlo da travessia, um algoritmo deve guardar esta informação de estado (num array indexado pelos vértices do grafo).

Dependendo da aplicação, pode não ser importante distinguir entre os vértices CINZENTOS e PRETOS; neste caso a informação de estado será simplesmente Booleana (**não visitado / visitado**).

Travessia em Largura

Esta estratégia de travessia caracteriza-se pelo seguinte:

Todos os vértices à distância k de s são visitados antes de qualquer vértice à distância $k + 1$ de s .

Trata-se pois de uma travessia *por níveis* de distância desde a origem. Em qualquer ponto da execução, os vértices cinzentos poderão estar no máximo em dois níveis diferentes consecutivos.

*Como se define a noção de **distância** entre dois vértices?*

Esta estratégia exige a utilização de uma *fila de espera* como estrutura de dados auxiliar de controlo, onde serão armazenados os vértices que num determinado

momento se encontram cinzentos. A fila de espera é inicializada inserindo-se o vértice em que se pretende começar a travessia.

Passo básico

1. É retirado (*dequeued*) um vértice cinzento u da fila de espera, e processado de acordo com o problema concreto que se pretende resolver;
2. É percorrida a sua lista de adjacências, e para cada vértice v adjacente a u :
 - a. **caso esteja ainda branco**, v é inserido (*enqueued*) na fila de espera, passando a cinzento;
3. u é marcado com cor preta.

Nota: o passo “e processado de acordo com o problema concreto que se pretende resolver” pode ser executado em diferentes momentos: quando o nó entra para a fila de espera (fica cinzento); quando é retirado dessa fila (como ilustrado acima); ou finalmente quando passa a preto.

A execução terminará quando a fila de espera ficar vazia.

Código em Python

A seguinte função imprime as mudanças de estado dos vértices ao longo da execução, e constrói um vector ‘parent’ contendo a árvore de travessia.

```
for v in g:  
    color[v] = 'WHITE'  
    parent[v] = ''  
  
# Breadth-first traversal of graph 'g' taking as source the ve  
rtex 's'  
# Prints color changes and constructs traversal tree in 'paren  
t'.  
  
def bf_visit(g, s):  
    color[s] = 'GRAY'  
    print s, 'GRAY'  
    parent[s] = '(ROOT)'  
    q.enqueue(s)
```

```

while(not q.isEmpty()):
    u = q.dequeue()
    for v in g[u]:          # Em C: travessia de uma lista d
e adjacências!
        if color[v] == 'WHITE':
            color[v] = 'GRAY'
            print v, 'GRAY'
            parent[v] = u
            q.enqueue(v)
    color[u] = 'BLACK'
    print u, 'BLACK'

```

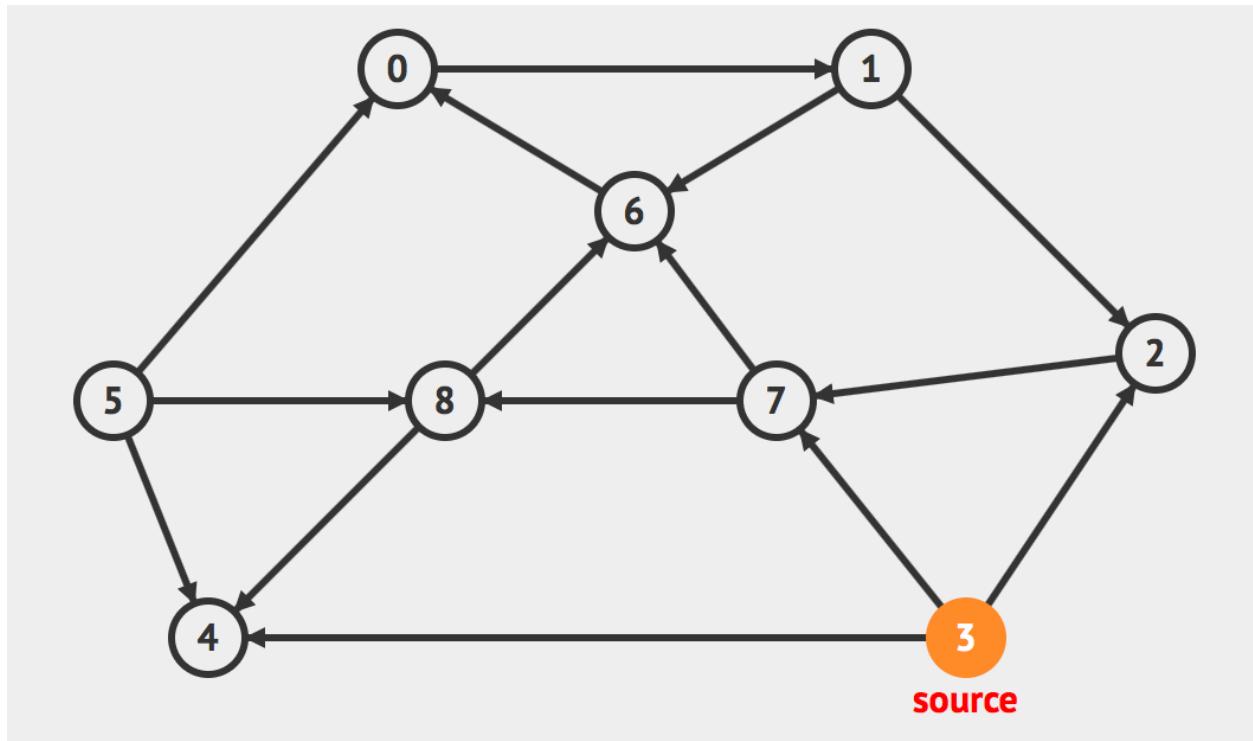
Observe-se que

A construção da árvore de travessia não é obrigatória: poderá ser ou não necessária, de acordo com o objectivo específico da travessia que se pretende implementar.

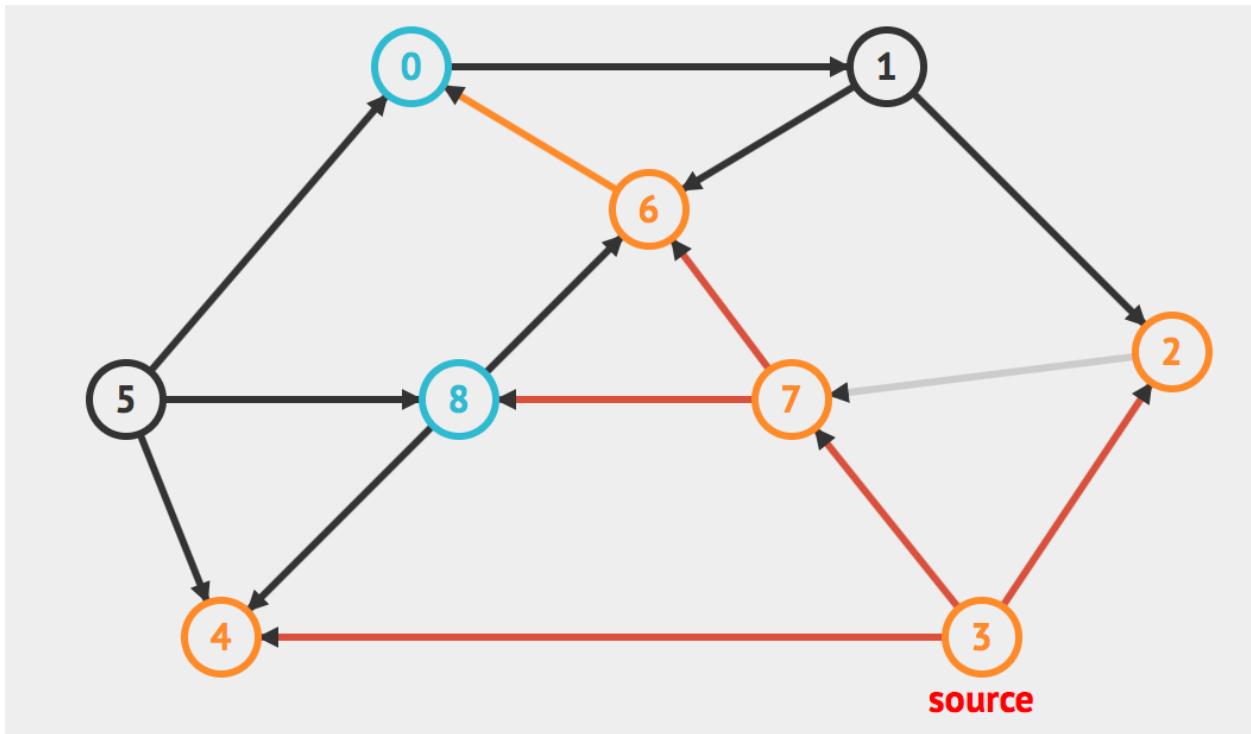
Exemplo

[As imagens foram obtidas a partir de uma animação em
<https://visualgo.net/en/dfsbfs>]

No grafo orientado seguinte, será efectuada uma travessia em largura com início no vértice 3



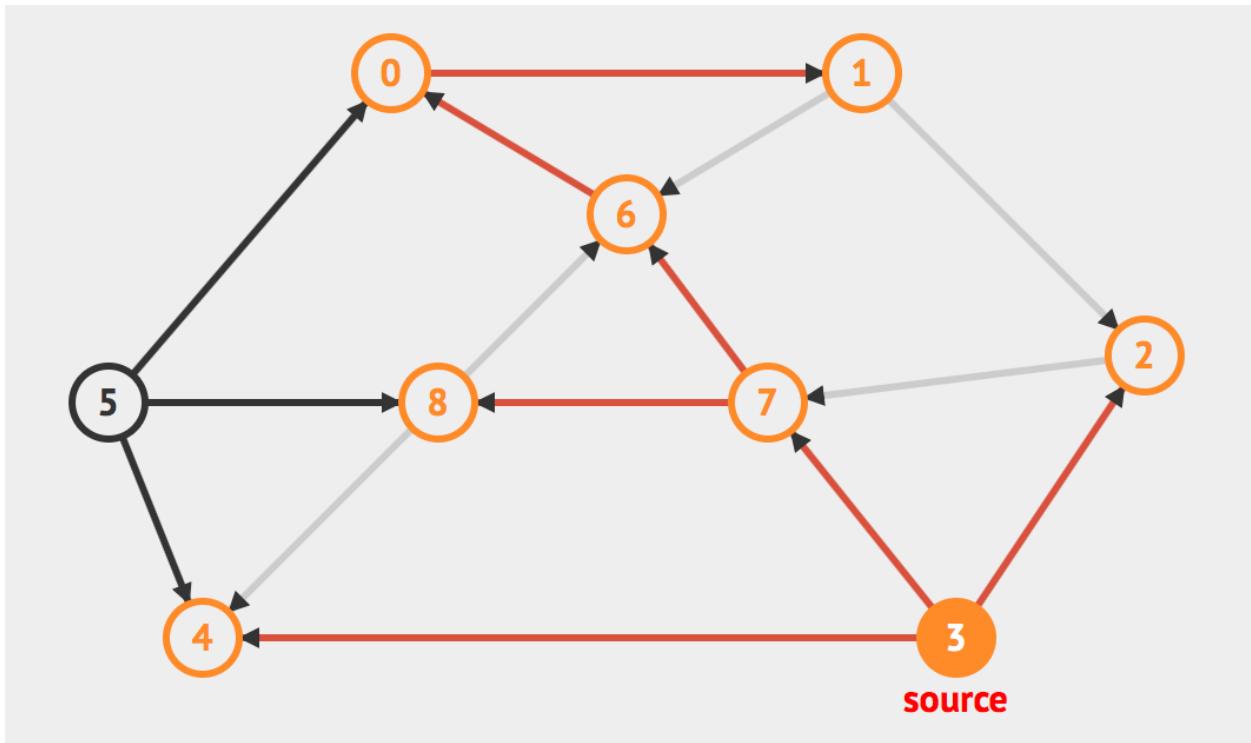
A meio da execução podemos observar que a fila de espera (vértices a azul) pode conter vértices que pertencem a dois níveis diferentes da árvore de travessia, i.e., a distância a que estão da origem não é a mesma (diferindo numa unidade).



Observe-se que, sendo a travessia feita em *largura*:

- 7 é visitado a partir de 3 e não de 2
- 8 é visitado antes de 0 e 1, uma vez que entra para a fila de espera logo a seguir a 6

Terminada a travessia temos a árvore final:



Análise do tempo de execução

- Uma vez que o passo básico envolve percorrer todos os vértices adjacentes a u , **assumiremos que o grafo é representado por listas de adjacências** — é a representação mais natural para a aplicação deste algoritmo
- Para o pior caso, assumimos que *todos os vértices do grafo são alcançados a partir de s*
- Cada vértice é *enqueued* e *dequeued* exactamente uma vez. Isto é garantido pelo facto de os vértices nunca serem pintados de branco depois da inicialização
- A lista de adjacência de cada vértice é percorrida exactamente uma vez (quando o vértice é *dequeued*), e o comprimento *total* das listas é $\Theta(|E|)$. Logo, o tempo total tomado pela travessia das listas de adjacência é $\Theta(|E|)$
- Não podemos no entanto concluir precipitadamente que o algoritmo executa em tempo $\Theta(|E|)$, uma vez que as operações de inicialização, e também o conjunto de operações de acesso à fila de espera, são executadas em tempo $\Theta(|V|)$

- Sendo assim, temos $T_p(V, E) = \Theta(|V| + |E|)$

Tempo linear no tamanho da representação por listas de adjacências

- No **melhor caso** o vértice s não tem adjacentes, e $T_m(V, E) = \Theta(|V|)$, devido ao tempo de inicialização do vector de cores

OBSERVAÇÃO

Na análise de algoritmos sobre grafos simples (i.e. que **não são multi-grafos**) leva-se por vezes a identificação do pior caso mais longe: tendo em conta que o valor máximo de $|E|$ é dado por $|V|^2$, quando o grafo é *completo*, podemos dizer que $T_p(V, E) = O(|V|^2)$.

Caminho Mais Curto e Distância entre dois vértices

Uma consequência da escolha desta estratégia é o seguinte resultado, que pode ser provado formalmente como parte da correcção do algoritmo:

*A árvore de travessia construída contém todos os **caminhos mais curtos** com origem em s e destino em cada um dos vértices alcançáveis a partir de s .*

Ou seja, para encontrar o caminho mais curto (i.e. o caminho com o menor comprimento) de s até um vértice d , basta:

1. efectuar uma travessia em largura com origem em s , e
2. identificar depois o caminho de s para d na árvore de travessia construída.

O comprimento do caminho mais curto entre s e d designa-se por *distância* entre os dois vértices. Uma travessia em largura pode ser usada para calcular distâncias.

No entanto, é importante perceber que o cálculo de distâncias / caminhos mais curtos é apenas uma aplicação possível de um algoritmo de travessia em largura.

Travessia em Largura Completa de um Grafo

Dependendo da aplicação, poderá ser importante alcançar *todos os vértices* de um grafo, sendo para isso necessário iniciar mais do que uma travessia. Basta percorrer todos os vértices do grafo, iniciando uma travessia em cada vértice que não tenha ainda sido alcançado pelas travessias anteriores:

```
# Travessia em largura completa, produzindo uma floresta de travessia
def bfs(g):
    for u in g:
        if color[u] == 'WHITE':
            bf_visit(g, u)
```

Este algoritmo executa agora necessariamente em tempo $T(V, E) = \Theta(|V| + |E|)$ em *todos os casos!*

Travessia em Profundidade

Esta estratégia de travessia caracteriza-se pelo seguinte:

Todos os vértices adjacentes a um vértice v são visitados imediatamente a seguir a v

Enquanto na travessia em largura os adjacentes a um vértice que acaba de ser visitado ficam em espera numa fila, e só serão por isso tratados depois de todos os que estão entretanto pendentes, na travessia em profundidade esses adjacentes serão os primeiros a ser visitados.

A árvore de travessia é pois construída em profundidade, ramo a ramo, e não por níveis como anteriormente. Por esta razão, uma travessia em profundidade **não pode ser utilizada**:

- para calcular distâncias entre vértices, nem
- para calcular caminhos mais curtos entre vértices.

Em termos de implementação, basta alterar a estrutura de dados auxiliar que armazena os vértices cinzentos (descobertos mas ainda não visitados), substituindo a fila de espera por uma **pilha**, mais uma vez inicializada inserindo-se o vértice em que se pretende começar a travessia.

Existe no entanto uma alternativa frequentemente utilizada, semelhante a um algoritmo típico de travessia de uma árvore, e que passa pela utilização de **recursividade**, substituindo-se assim a utilização explícita de uma pilha pela estrutura recursiva das chamadas de função.

Código em Python

A seguinte função imprime as mudanças de estado dos vértices ao longo da execução, e constrói um vector ‘parent’ contendo a árvore de travessia.

```
for v in g:
    color[v] = 'WHITE'
    parent[v] = ''

# Depth-first traversal of graph 'g' taking as source the vertex 's'

# Prints color changes and constructs traversal tree in 'parent'.

def df_visit(g, s):
    color[s] = 'GRAY'
    print s, 'GRAY'
    for v in g[s]:
        if color[v] == 'WHITE':
            parent[v] = s
            df_visit(g, v)
    color[s] = 'BLACK'
    print s, 'BLACK'
```

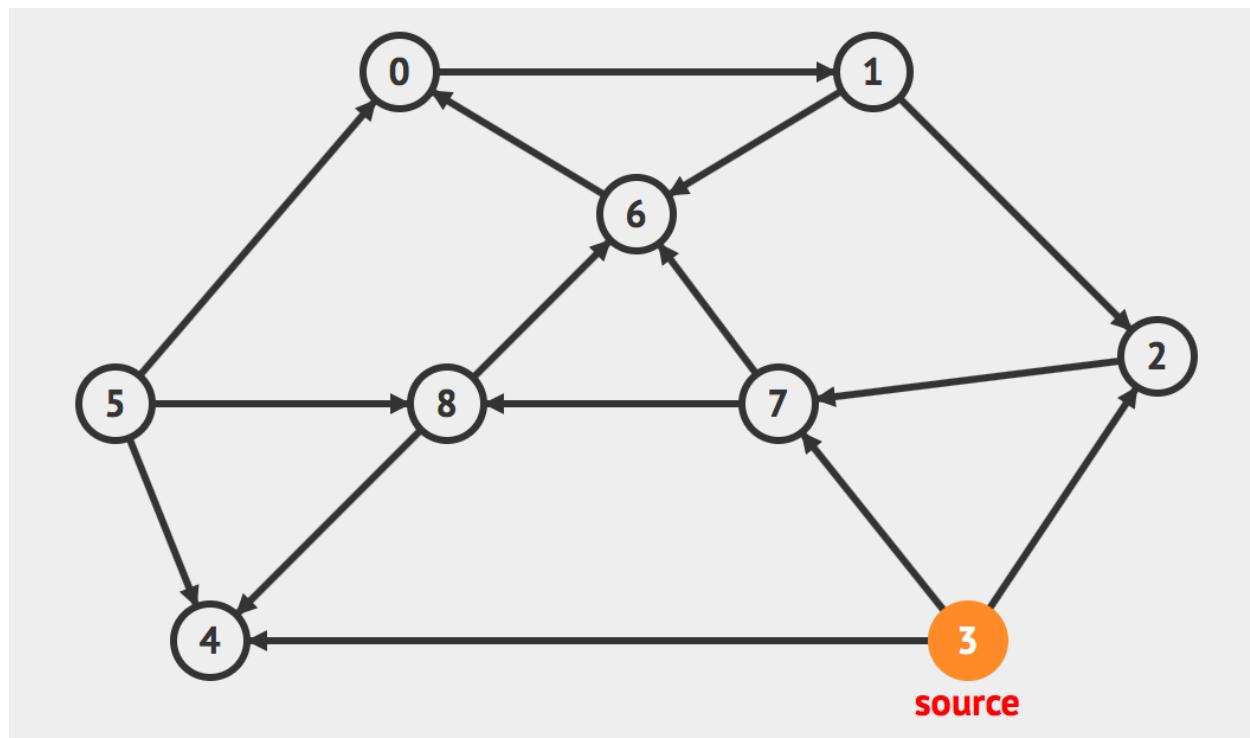
Tal como na travessia em largura, podemos facilmente implementar uma travessia completa iniciando uma travessia sucessivamente em todos os vértices do grafo, desde que não tenham ainda sido visitados por uma travessia anterior:

```
def dfs(g):
    for u in g:
        if color[u] == 'WHITE':
            df_visit(g, u)
```

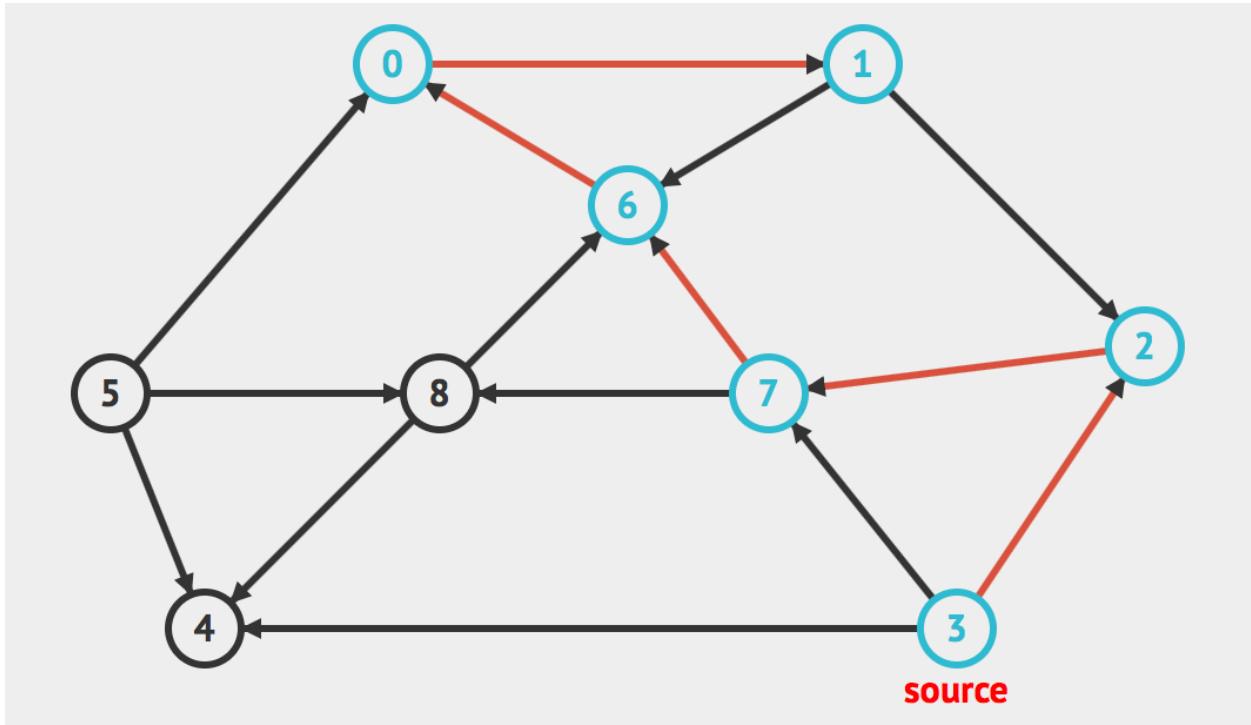
Exemplo

[As imagens foram obtidas a partir de uma animação em
<https://visualgo.net/en/dfsbfs>]

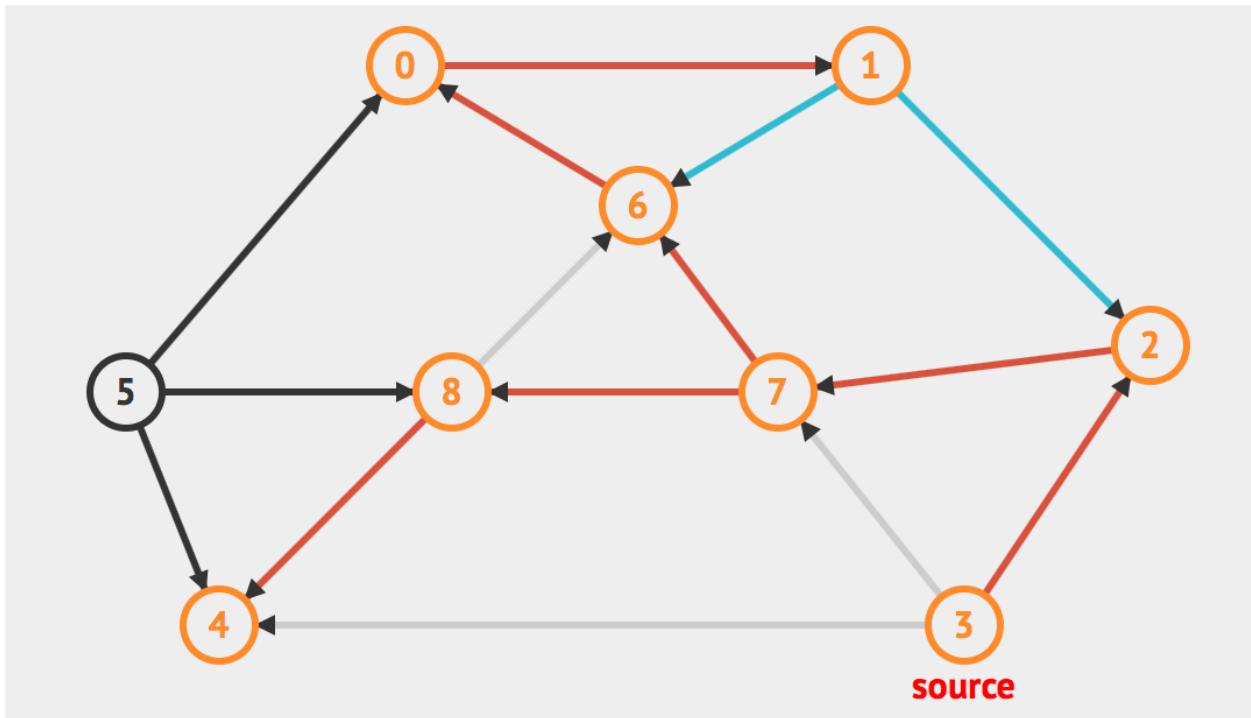
Novamente iniciaremos uma travessia no vértice 3 do seguinte grafo, mas desta vez em profundidade:



Na seguinte imagem intermédia é visível que os vértice 0 e 1 são visitados a partir de 6 antes de 8, que tal como 6 é adjacente a 7:



Na seguinte imagem final pode observar-se a vermelho a árvore de travessia em profundidade.



QUESTÃO: o que representarão as arestas a azul?

Análise do tempo de execução

- Mais uma vez, a análise do pior caso de `df_visit` coincide com a análise do caso único da travessia completa `dfs`
- Em `dfs`, o ciclo `for` executa em tempo $\Theta(|V|)$, excluindo o tempo tomado pelas execuções de `df_visit`. O mesmo acontece com os ciclos de inicialização dos vectores.
- `df_visit` é invocada *exactamente uma vez* para cada vértice do grafo (a partir de `dfs` ou da própria `df_visit`), uma vez que é invocada apenas com vértices brancos e a primeira coisa que faz é pintá-los de cinzento (e nenhum vértice volta a ser pintado de branco).
- Em `df_visit(g, u)`, o número de iterações do ciclo `for` é dado pelo número de vértices adjacentes de `u`. No total das invocações de `df_visit`, este ciclo é então

executado $\Theta(|E|)$ vezes

- Sendo assim, temos mais uma vez $T(V, E) = \Theta(|V| + |E|)$

Tempo linear no tamanho da representação por listas de adjacências

EXERCÍCIOS

[retomando <https://codeboard.io/projects/10725>]

1. Com base nas estruturas de dados que definiu anteriormente, implemente em C o algoritmo de travessia em largura.
 - a. Altere o algoritmo por forma a calcular as distâncias do vértice s a todos os vértices alcançáveis a partir dele, colocando-as num array.
 - b. Escreva uma função que imprime o caminho mais curto entre quaisquer dois vértices a e b de um grafo, com base numa travessia em largura.
 - c. Implemente igualmente em C o algoritmo de travessia completa.
1. Com base nas estruturas de dados que definiu anteriormente, implemente em C o algoritmo de travessia em profundidade.
 - a. Defina uma função que calcula se um grafo é ou não cíclico, com base numa travessia em profundidade.
 - b. Implemente igualmente em C o algoritmo de travessia completa em profundidade.

Código Python para teste dos algoritmos

```
class Queue:  
    def __init__(self):  
        self.items = []
```

```

def isEmpty(self):
    return self.items == []
def enqueue(self, item):
    self.items.insert(0,item)
def dequeue(self):
    return self.items.pop()
def size(self):
    return len(self.items)

g = {}
g[0] = [1]
g[1] = [2, 6]
g[2] = [7]
g[3] = [2, 4, 7]
g[4] = []
g[5] = [0, 4, 8]
g[6] = [0]
g[7] = [6, 8]
g[8] = [4, 6]

color = {}
parent = {}

q=Queue()

print "\nGraph:"
for v in g:
    print v, "->", ", ".join([str(u) for u in g[v]])

print "\nDFS:"
```

```
# bf_visit(g, 3)
# bfs(g)
# df_visit(g, 3)
dfs(g)
print "\nTraversal tree / forest:"
for j in sorted(parent.keys()):
    print j, "<-", parent[j]
```

G3. Algoritmos sobre Grafos Pesados: Estratégia Greedy

[Codeboard de apoio a este módulo: <https://codeboard.io/projects/10154>]

Árvores Geradoras Mínimas / *Minimum Spanning Trees* [MST]

Seja $G = (V, E)$ um grafo não-orientado, ligado (i.e. todos os vértices são alcançáveis a partir de qualquer outro vértice, não havendo componentes separados), e com pesos.

Uma **árvore geradora** de G é um sub-grafo (V, T) acíclico e ligado de G .

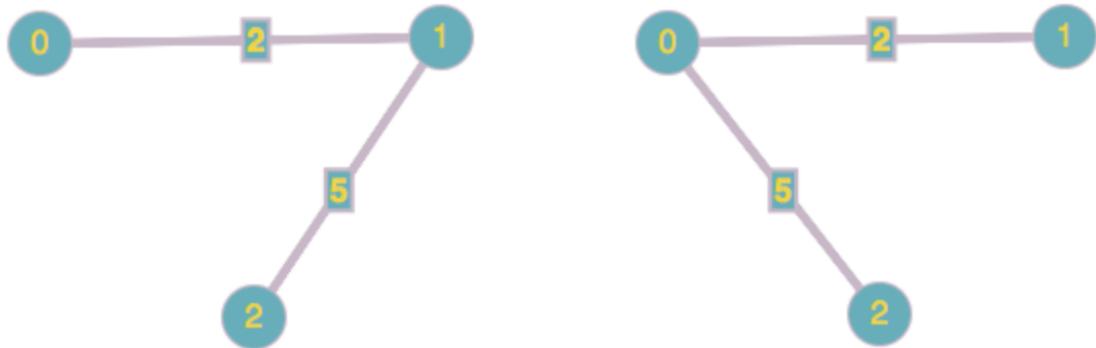
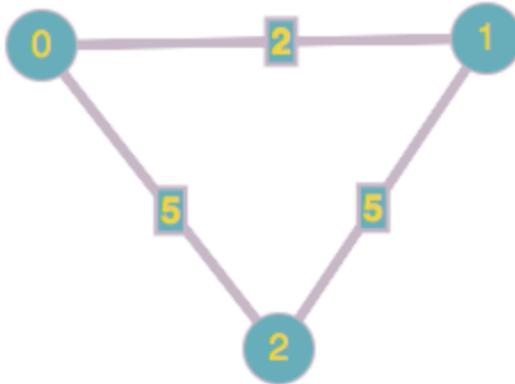
Note-se que, sendo sub-grafo acíclico de um grafo não-orientado, (V, T) é uma árvore que contém todos os vértices de G .

As **árvores geradoras mínimas** de G são aquelas para as quais o peso total $w(T) = \sum_{(u,v) \in T} w(u, v)$ é mínimo.

O problema da determinação de uma MST é pois um **problema de optimização**: trata-se de determinar um dos “melhores” objectos que satisfazem um conjunto de restrições (neste caso dadas pela definição de árvore geradora).

Exemplos

O grafo seguinte tem 3 árvores geradoras, das quais duas são mínimas, com peso 7.



No contexto de um grafo correspondente a uma rede de estradas, uma árvore geradora será uma sub-rede das estradas que permite deslocações entre todas as localidades sem redundância, i.e. existirá nesta árvore um único caminho entre cada par de localidades.

Uma árvore geradora mínima será então uma sub-rede que permitirá ligar todas as localidades entre si, com comprimento total mínimo.

Um outro exemplo será a ligação eléctrica de um número de pinos num circuito integrado. Cada fio liga um par de pinos; pretende-se minimizar a quantidade total de cobre utilizado nas ligações. Para isso constrói-se um grafo contendo as ligações possíveis entre pinos, com pesos correspondentes à quantidade de cobre utilizada

em cada ligação. O peso de uma MST corresponde à quantidade mínima de cobre necessária para o conjunto de ligações.

Trata-se de um problema que ocorre em muitos contextos, nomeadamente como sub-problema de outros.

Sub-estrutura Óptima

Diz-se que um problema de optimização possui *sub-estrutura óptima* se uma solução óptima para o problema contém soluções óptimas para sub-problemas do problema original.

O problema de determinação de árvores geradoras mínimas possui sub-estrutura óptima, uma vez que cada sub-árvore de uma MST de um grafo G é seguramente uma MST de um sub-grafo de G .

Um problema com estas características pode ser reformulado da seguinte forma:

dada uma solução parcial (solução de um sub-problema), pretende-se estendê-la até obter uma solução total (solução do problema original).

No caso do problema de determinação de MSTs:

dada uma MST de um sub-grafo de G , pretende-se estendê-la para obter uma MST de G

A estratégia greedy para resolução de problemas com sub-estrutura óptima

A estratégia de resolução greedy (“gananciosa”) é uma estratégia algorítmica para problemas com sub-estrutura óptima que se caracteriza da seguinte forma:

- É um método **top-down** (tal como a estratégia de divisão e conquista)
- Em cada passo, o algoritmo estende a solução actual efectuando uma escolha local

- Desta forma, o problema é reduzido a um problema mais pequeno, uma vez que a solução local cresce em cada passo, aproximando-se de uma solução para o problema inicial

Enquanto a estratégia de divisão e conquista gera problemas mais pequenos para resolver, e processa depois as respectivas soluções, a estratégia *greedy* transforma o problema num mais pequeno, estendendo localmente uma solução parcial.

A prova de correcção de um algoritmo baseado nesta estratégia terá que mostrar que o passo básico é correcto, ou seja:

se se parte de uma solução óptima de um sub-problema do problema original, então depois de se estender localmente esta solução teremos ainda uma solução óptima de um sub-problema do problema original.

Algoritmo de Prim para Construção de Árvores Geradoras Mínimas

Este algoritmo pode ser visto como um algoritmo de travessia, usando uma estratégia alternativa (nem em profundidade nem em largura). Trata-se aqui de uma estratégia **greedy**, que faz uma escolha local guiada pelos pesos das arestas.

Em cada passo do algoritmo partir-se-á de uma MST (V', T') de um sub-grafo de G , e acrescentar-se-á um vértice e uma aresta a esta árvore, obtendo-se uma árvore (V'', T'') que será ainda uma MST de um subgrafo (maior) de G .

Estrutura geral do algoritmo

Considera-se em cada instante da execução o conjunto de vértices de G dividido em 3 conjuntos disjuntos:

1. os vértices da **árvore** de travessia construída até ao momento;
2. os vértices na **orla** (adjacentes aos da árvore);
3. os restantes vértices.

Em cada passo selecciona-se uma aresta (*com origem na árvore e destino na orla*) para acrescentar à árvore. O vértice destino dessa aresta é também acrescentado. É esta a escolha local característica da estratégia *greedy*.

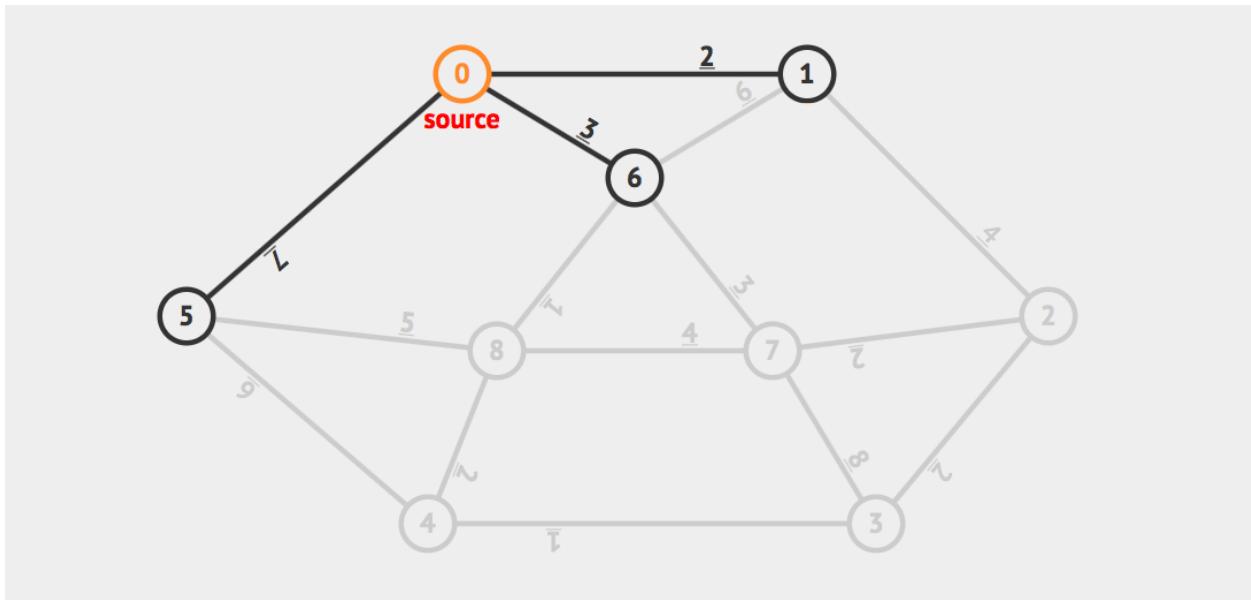
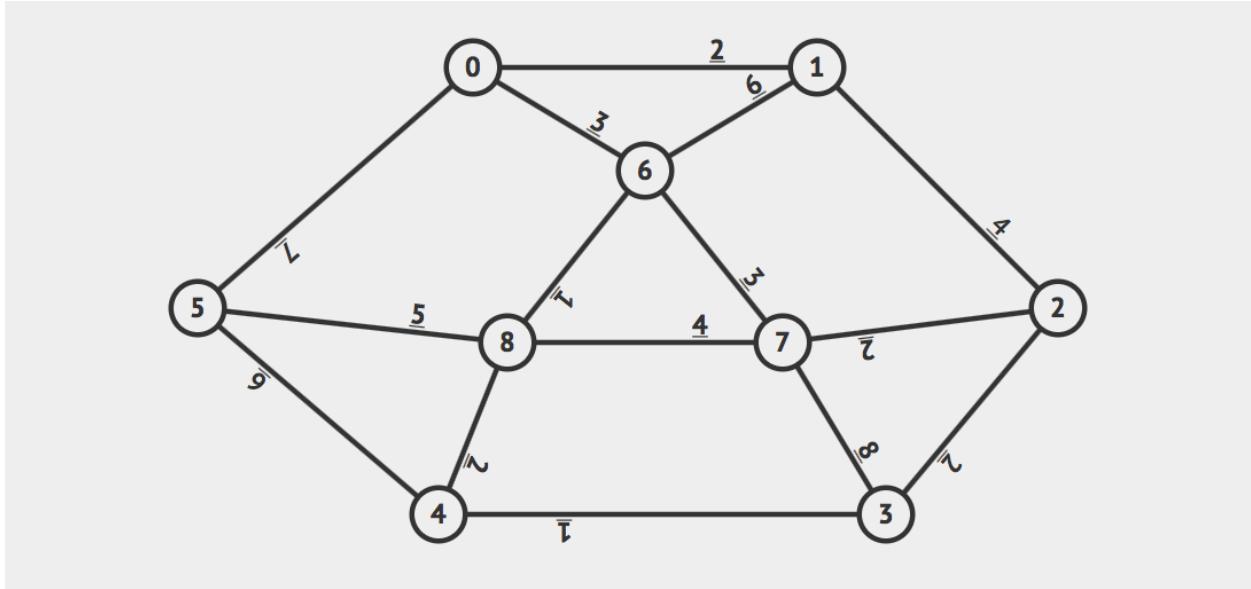
O algoritmo de Prim selecciona sempre o arco com **menor peso** nestas condições.

```
def mst_Prim(g):
    seleccionar vértice arbitrário x para inicio da árvore
    while (árvore não contém todos os vértices):
        actualizar orla em função do novo vértice x
        seleccionar aresta (u,v) de peso mínimo
            entre vértices da árvore e da orla
        x = v
        acrescentar vértice x e aresta (u,x) à árvore
```

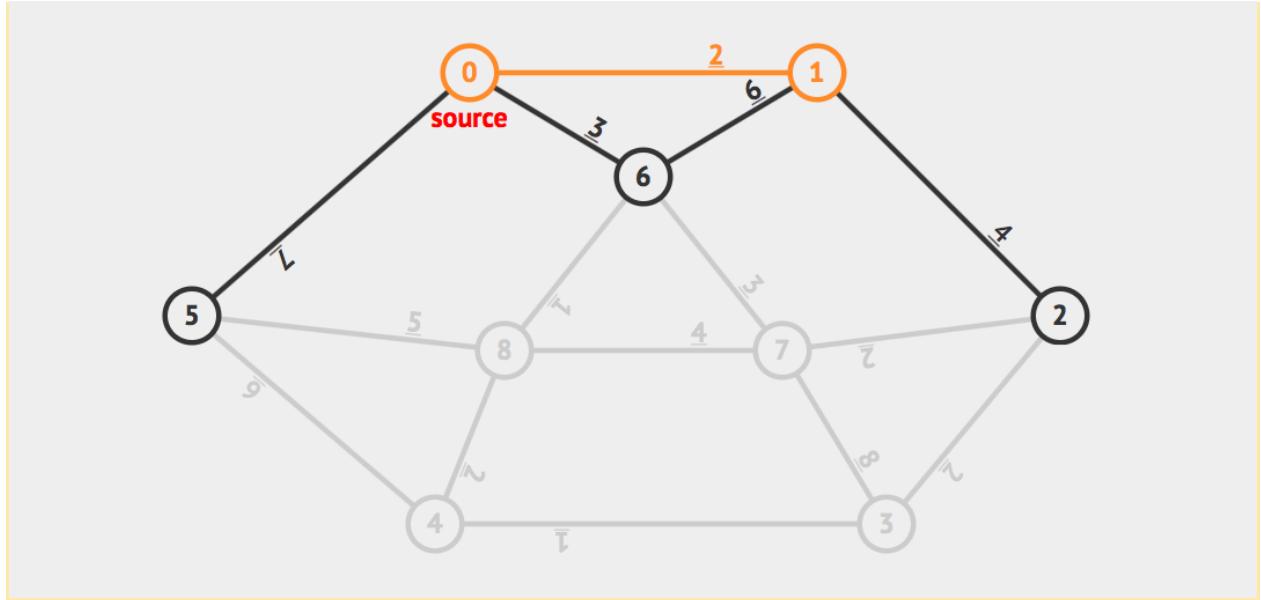
O exemplo seguinte ilustra a necessidade de se associar a cada vértice da orla o seu **arco candidato**. A selecção da aresta a acrescentar à árvore é feita escolhendo o **arco candidato de menor peso**.

Exemplo

[Animação realizada em <https://visualgo.net/en/mst>]

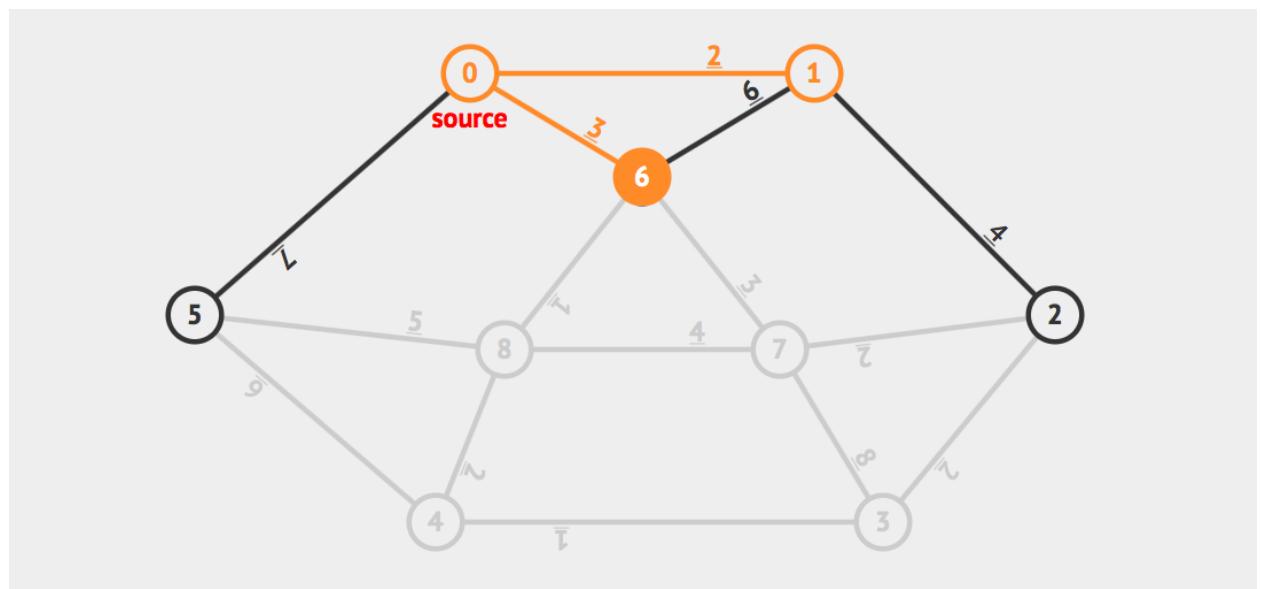


A orla contém neste momento os 3 vértices adjacentes à origem da árvore, 0.
Será seleccionado o arco candidato $(0, 1)$, uma vez que tem o menor peso:

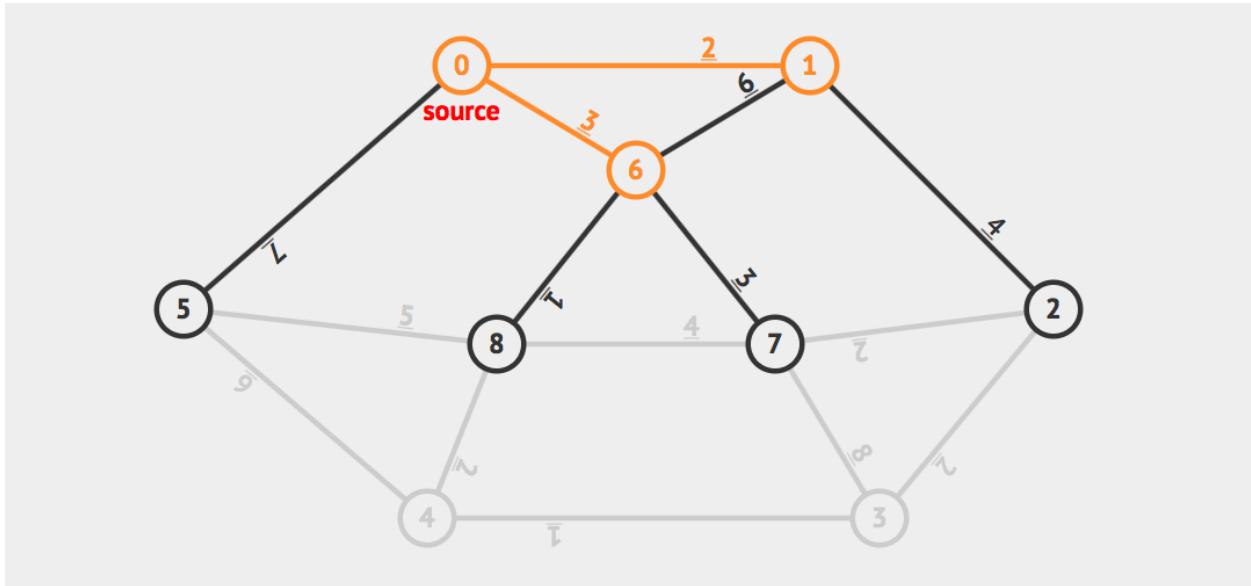


O vértice 2 foi acrescentado à orla, e existem agora dois arcos $(0, 6)$ e $(1, 6)$ que levam ao vértice 6. Mas o **arco candidato** deste vértice é único: naturalmente, $(0, 6)$.

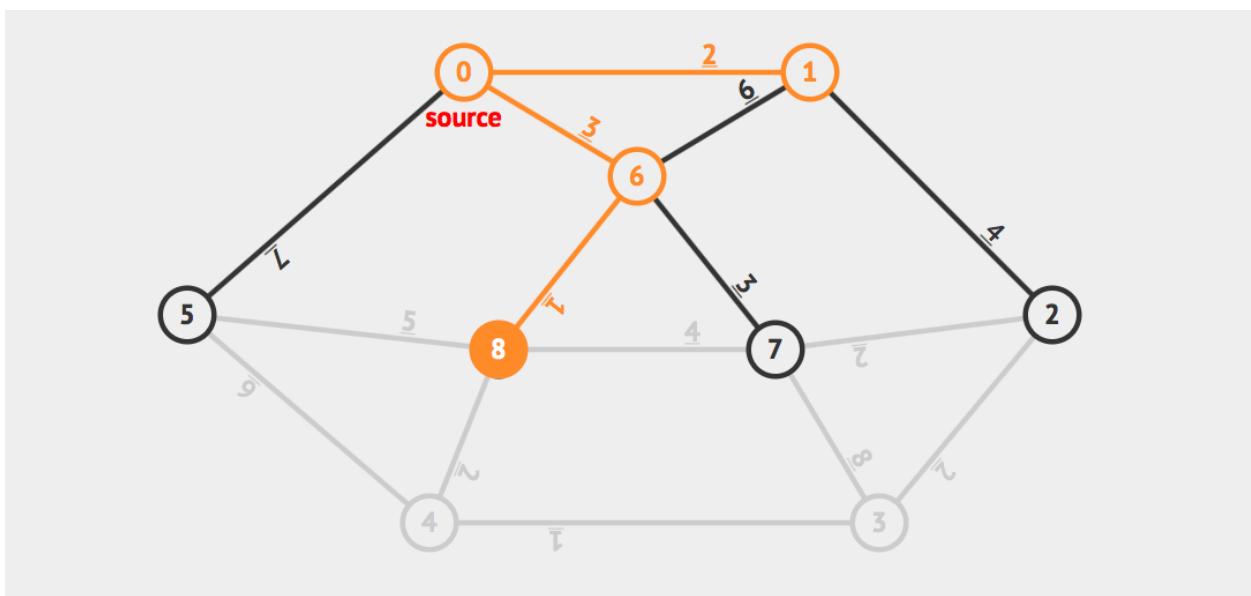
Durante a execução do algoritmo os arcos candidatos (a preto) podem já fazer parte da árvore de travessia construída. No instante acima, teremos `parent[6]==0`, e não `parent[6]==1`.



Depois de acrescentar um vértice à árvore há sempre que actualizar a orla:

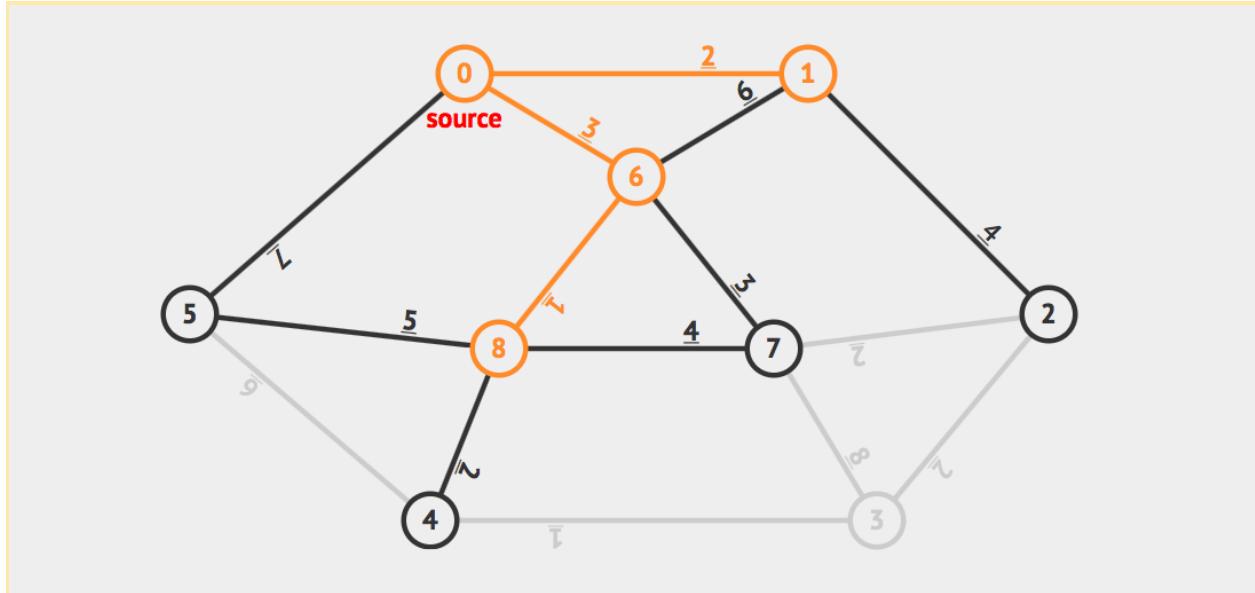


O próximo passo parece óbvio:

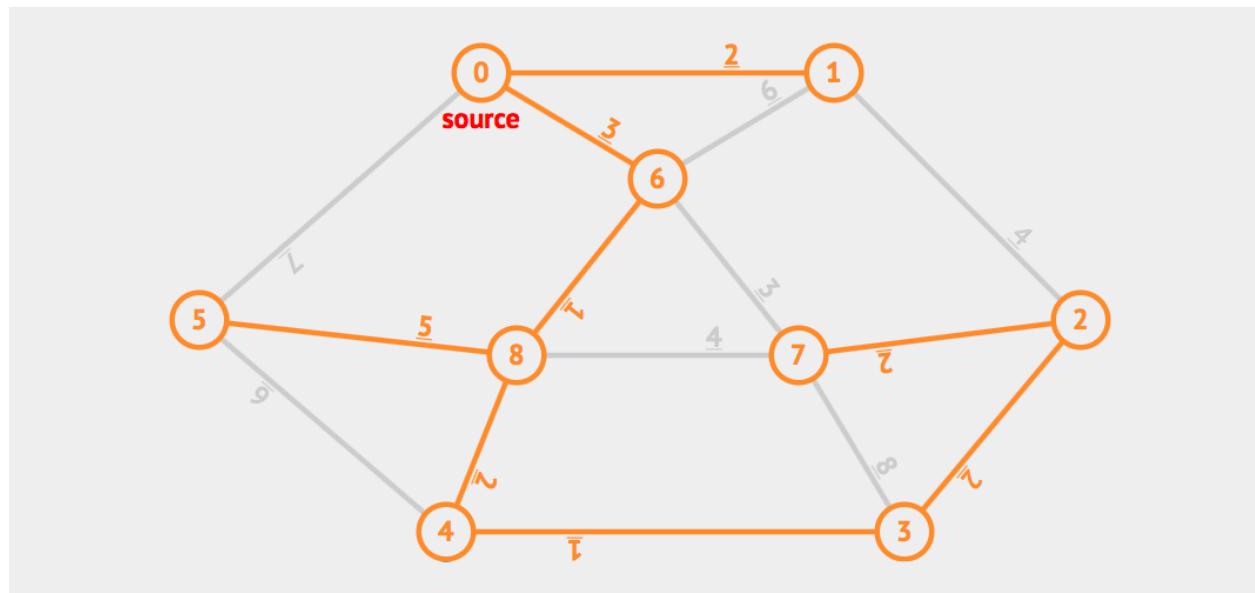


No entanto surge um fenómeno novo: ao actualizar a orla depois de acrescentar o vértice 8, surge um novo arco(8, 5) com destino no vértice 5, que **substitui o arco candidato anterior** (0, 5).

Os arcos candidatos fazem parte da árvore de travessia construída, no entanto, pela razão acima, poderão ser substituídos, não chegando nesse caso a pertencer à MST construída.



Continuando a aplicar este passo básico, obtemos a MST completa, com peso 18:



Algoritmo Detalhado em Python

Nota: a representação de grafos em Python como dicionários permite uma sintaxe para acesso ao peso de uma aresta que parece matricial, mas não é: `g[x]` designa a “lista” de adjacências do vértice `x`, mas que é aqui também ela um dicionário da forma **vértice destino \mapsto peso**, e não uma lista! Sendo assim `g[x][y]` é o peso da aresta (x,y)

Na implementação em Python representa-se por dicionários a árvore construída, a informação de status dos vértices, e a orla. Esta última em particular será representada por um dicionário cujo domínio são os vértices da orla, que são mapeados para o peso do respectivo arco candidato.

```
def mst_Prim(g):
    fringe = {}
    status = {}
    for i in g:
        status[i] = 'UNSEEN'
    edgeCount = 0

    x = 0;
    status[x] = 'INTREE'
    parent[x] = -1

    while (edgeCount < len(g)-1):
        for y in g[x]:
            wxy = g[x][y]           # weight of edge (x,y)
            if status[y] == 'UNSEEN':
                # add y to fringe with candidate edge (x,y)
                status[y] = 'FRINGE'
                parent[y] = x
                fringe[y] = wxy      # dictionary insertion
            elif status[y] == 'FRINGE' and wxy < fringe[y]:
                # replace candidate edge of y by (x,y)
```

```

        parent[y] = x
        fringe[y] = wxy      # dictionary update

# are we blocked? (non-connected graph)

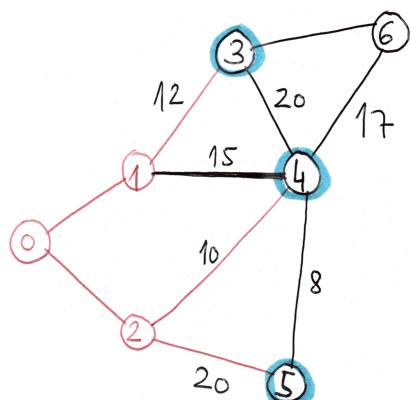
if len(fringe) == 0:
    return False

# select next candidate edge and vertex;
# remove them from fringe and add to tree
x = min(fringe, key=lambda x:fringe[x])
del fringe[x]
status[x] = 'INTREE'
edgeCount += 1

return True

```

Vejamos detalhadamente um exemplo de execução do passo básico do algoritmo.



$\text{status}[0] = \text{status}[1] = \text{status}[2] = \text{INTREE}$
 $\text{status}[3] = \text{status}[4] = \text{status}[5] = \text{FRINGE}$
 $\text{status}[6] = \text{UNSEEN}$

 $\text{parent}[1] = \text{parent}[2] = 0$ (definitivos)
 $\text{parent}[3] = 1$ and $\text{parent}[4] = \text{parent}[5] = 2$ (orla)

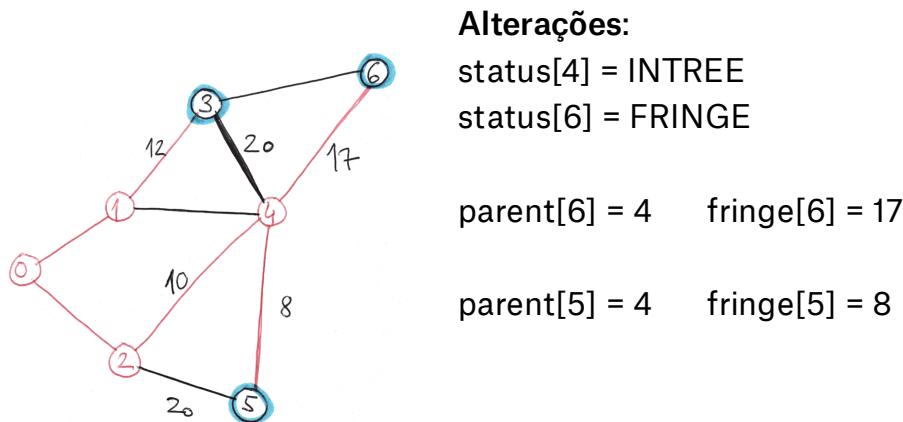
 $\text{fringe}[3] = 12$ $\text{fringe}[4] = 10$ $\text{fringe}[5] = 20$

a aresta (1,4) não é um arco candidato!

Será seleccionado o vértice 4 para ser acrescentado à árvore, e serão percorridos os seus adjacentes:

- 3 já pertencia à orla, mas não haverá alteração do arco candidato ($20 > 12$)

- 5 já pertencia à orla, e haverá alteração do arco candidato ($8 < 20$)
- 6 não pertencia à orla, passará a pertencer, com arco candidato (4,6)



Análise

O tempo de execução está claramente em $\Omega(|V| + |E|)$, uma vez que a estrutura básica do algoritmo é a de uma travessia (percorre todas as listas de adjacências). No entanto, este algoritmo executa outras operações potencialmente mais pesadas:

1. Em cada iteração do ciclo `while`, i.e. $|V|$ vezes, é feita a selecção do arco candidato de menor peso, e removido da orla o respectivo vértice
2. Em cada iteração dos ciclos `for`, i.e. $|E|$ vezes, pode ser feita uma das seguintes coisas:
 - a. é acrescentado um nó à orla, ou
 - b. é alterado o peso do arco candidato de um nó da orla

Tabela de hash

Se a orla for implementada como no código Python acima, por um dicionário (tabela de hash sobre um vector dinâmico), as operações de inserção/alteração do peso são executadas em tempo (tendencialmente) constante, mas a operação de selecção do

mínimo obriga a percorrer todos os vértices da orla, sendo executada em tempo linear.

O pior caso ocorrerá quando o primeiro vértice está ligado a todos os outros, tendo a orla sempre tamanho máximo, e teremos $T(N) = O(|V|^2 + |E|) = O(|V|^2)$, no caso de um grafo simples .

Lista ligada

Uma alternativa semelhante em termos de performance é implementar a orla como uma lista ligada (*não-ordenada*) contendo os vértices. Neste caso, para possibilitar que a alteração do peso do arco candidato seja feita em tempo constante, será preferível não incluir estes pesos na lista (orla), mas sim manter um *array* adicional indexado pelos vértices, e que associa a cada um o peso do respectivo arco candidato (este *array* pode ser visto como um complemento de *parent* que guarda o peso de todas as arestas da árvore construída). A selecção/remoção de mínimo exigirá no pior caso tempo linear, pelo que teremos ainda $T(N) = O(|V|^2 + |E|) = O(|V|^2)$.

Fila com Prioridades / Min-heap

A orla pode ainda ser implementada por uma *Fila* de vértices, tendo os pesos dos arcos candidatos como *prioridades*. Neste caso as operações de inserção, alteração de peso, e remoção do mínimo serão todas executadas em tempo logarítmico, pelo que teremos $T(N) = O(|V| \cdot \log |V| + |E| \cdot \log |V|) = O(|E| \cdot \log |V|)$

EXERCÍCIO

[<https://codeboard.io/projects/10154>]

Com base nas definições fornecidas no projecto Codeboard, implemente em C o algoritmo de Prim, escolhendo para a orla uma das estruturas sugeridas acima.

Caminhos Mais Curtos

O problema da determinação de caminhos mais curtos num grafo pesado é uma generalização do mesmo problema em grafos sem pesos. Pretende-se agora

minimizar não o número de arestas dos caminhos, mas sim o peso (comprimento, na analogia geográfica) desses caminhos.

A definição de peso de um caminho $P = v_0, v_1, \dots, v_k$ é a esperada:

$$w(P) = \sum_{i=0}^{k-1} w(v_i, v_{i+1})$$

Algumas notas:

- O problema está intimamente ligado com o do cálculo da *distância* entre dois vértices, que é precisamente definida como o peso do caminho mais curto.
- O problema que consideraremos aqui é o da construção de todos os caminhos mais curtos com origem num determinado vértice e destino em todos os vértices alcançáveis a partir dele, conhecido por ***single-source shortest paths***.
- O problema simétrico, ***single-destination shortest paths***, pode ser reduzido a este sobre o grafo simétrico do original.
- O problema ***single-pair shortest paths*** (caminho mais curto ponto a ponto) é naturalmente um problema mais simples, mas não existe um algoritmo específico: resolve-se utilizando um algoritmo de ***single-source shortest paths*** (podendo a sua execução ser interrompida antecipadamente).

Algoritmo de Dijkstra para o problema ***single-source shortest paths***

Trata-se de um algoritmo extremamente parecido com o de Prim, igualmente baseado numa travessia guiada por uma estratégia específica, e com o mesmo tempo assintótico de execução.

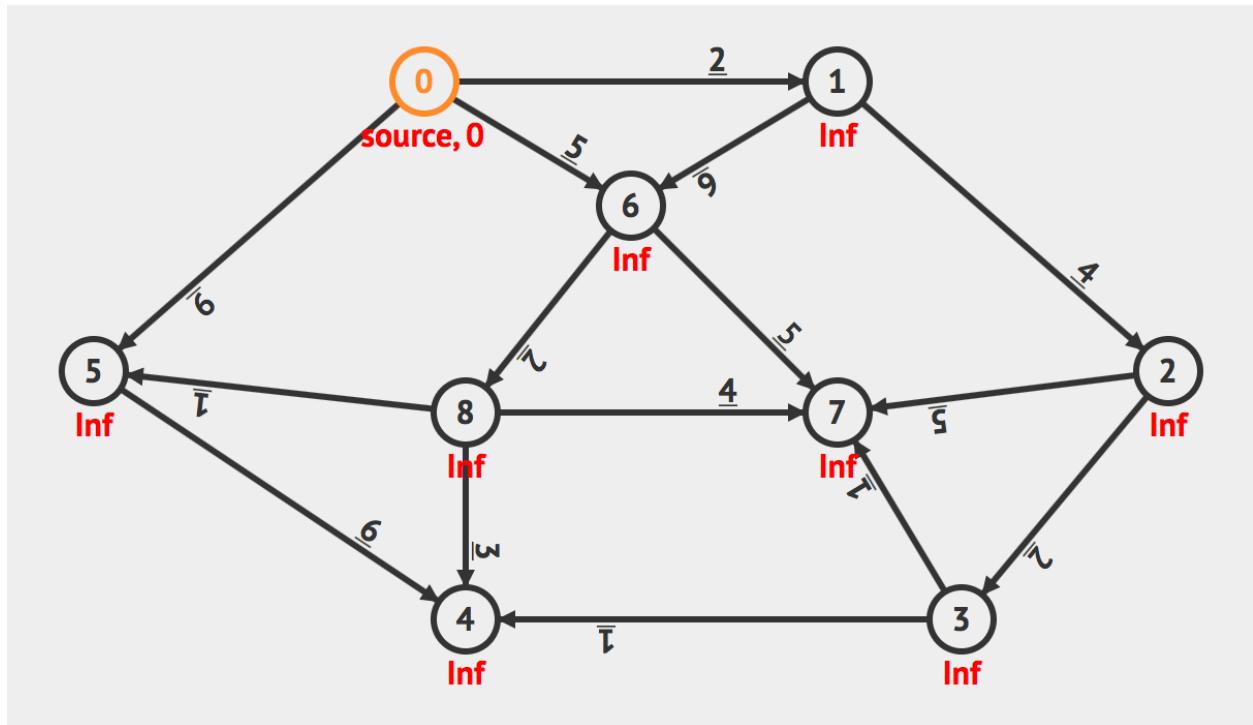
A árvore de travessia construída, com origem num vértice s , contém todos os caminhos mais curtos com origem em s e destino em vértices alcançáveis a partir de s .

As alterações a introduzir são as seguintes:

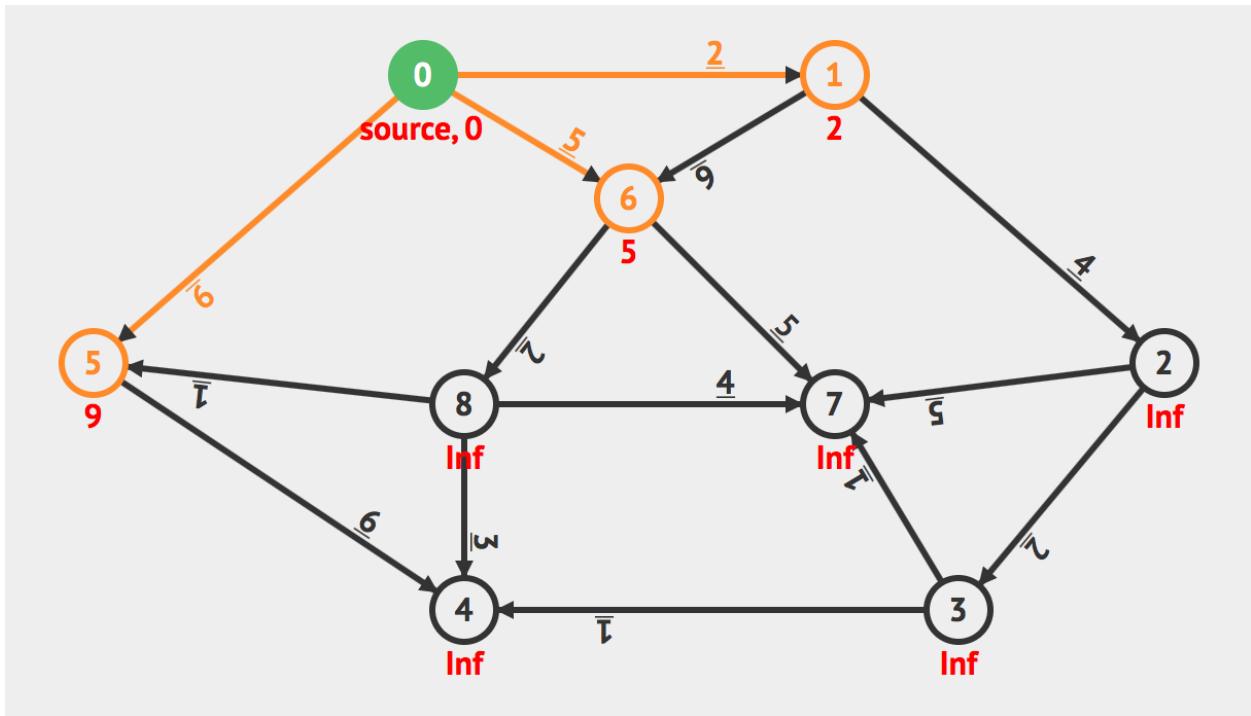
- À semelhança do que acontece no algoritmo de travessia em largura, utiliza-se um vector $\text{dist}[]$ para armazenar a distância desde a origem até aos vértices da árvore de travessia
 - $\text{dist}[y] = \delta(s, y)$ se y pertence à árvore de travessia
neste caso o valor $\text{dist}[y]$ não se alterará mais durante a execução do algoritmo, contém já o valor real da distância entre s e y
- Mas tal como acontece com o vector $\text{parent}[]$, o vector $\text{dist}[]$ estará definido também para os vértices da orla:
 - $\text{dist}[z] = \delta(s, y) + w(y, z)$ se z está na orla e (y, z) é o seu arco candidato
neste caso esta informação é provisória e poderá ser modificada caso se altere o arco candidato
- O critério de selecção do vértice da orla a acrescentar à árvore em cada passo é simplesmente a minimização do valor $\text{dist}[z]$, ou seja, *acrescenta-se o vértice cuja distância desde a origem é a menor.*

Exemplo

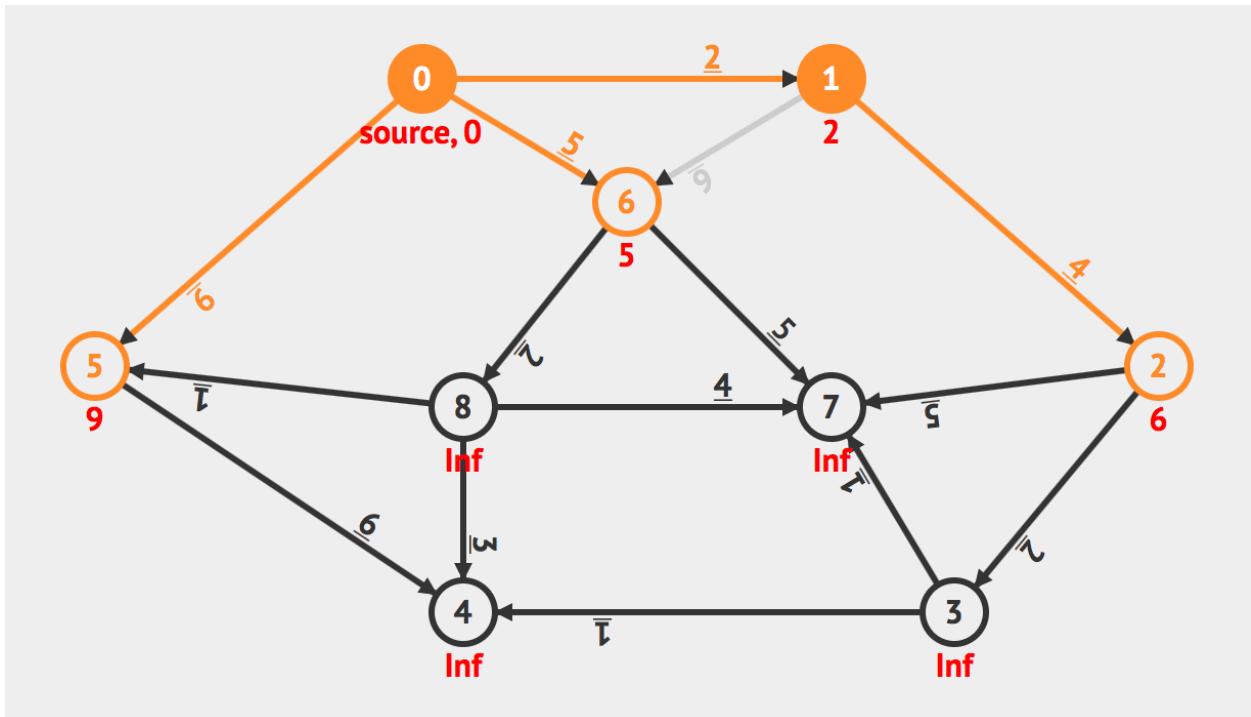
[Animação construída em <https://visualgo.net/en/sssp>]

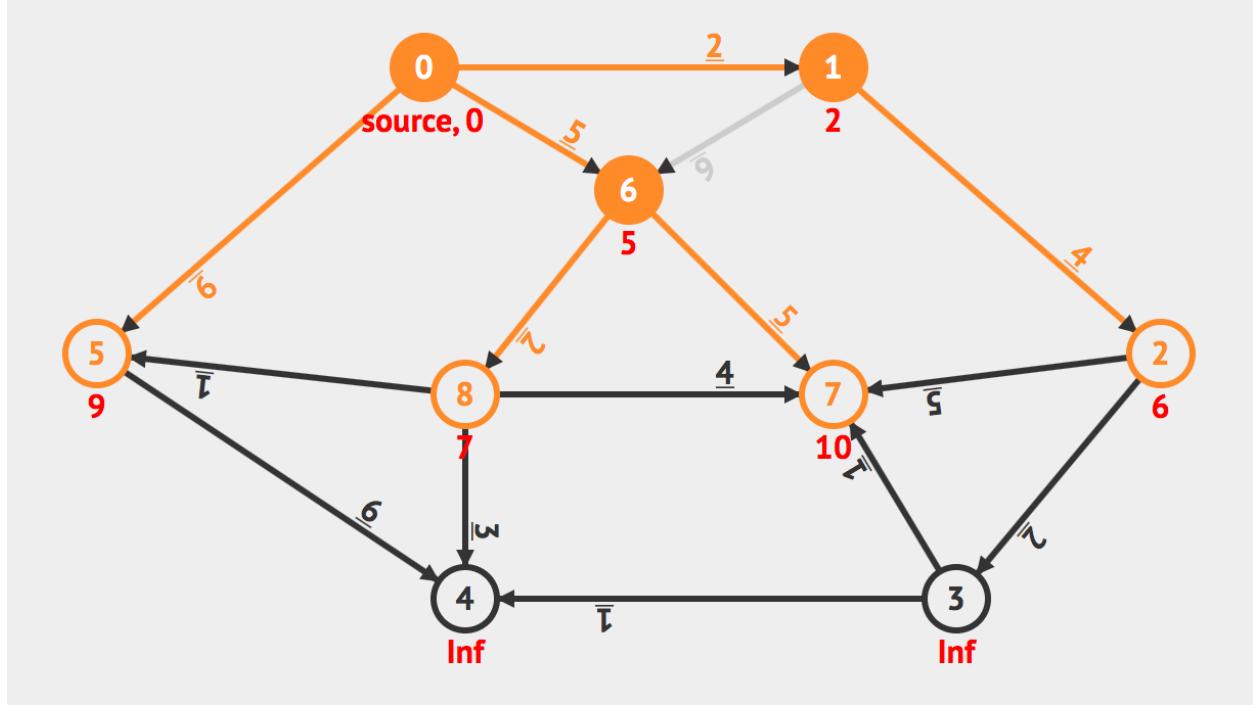


Anotaremos (a vermelho) nos vértices da árvore e da orla a distância desde a raiz até cada um deles. Para os vértices da orla estes valores não são definitivos, podendo ser alterados se houver mudança do arco candidato.



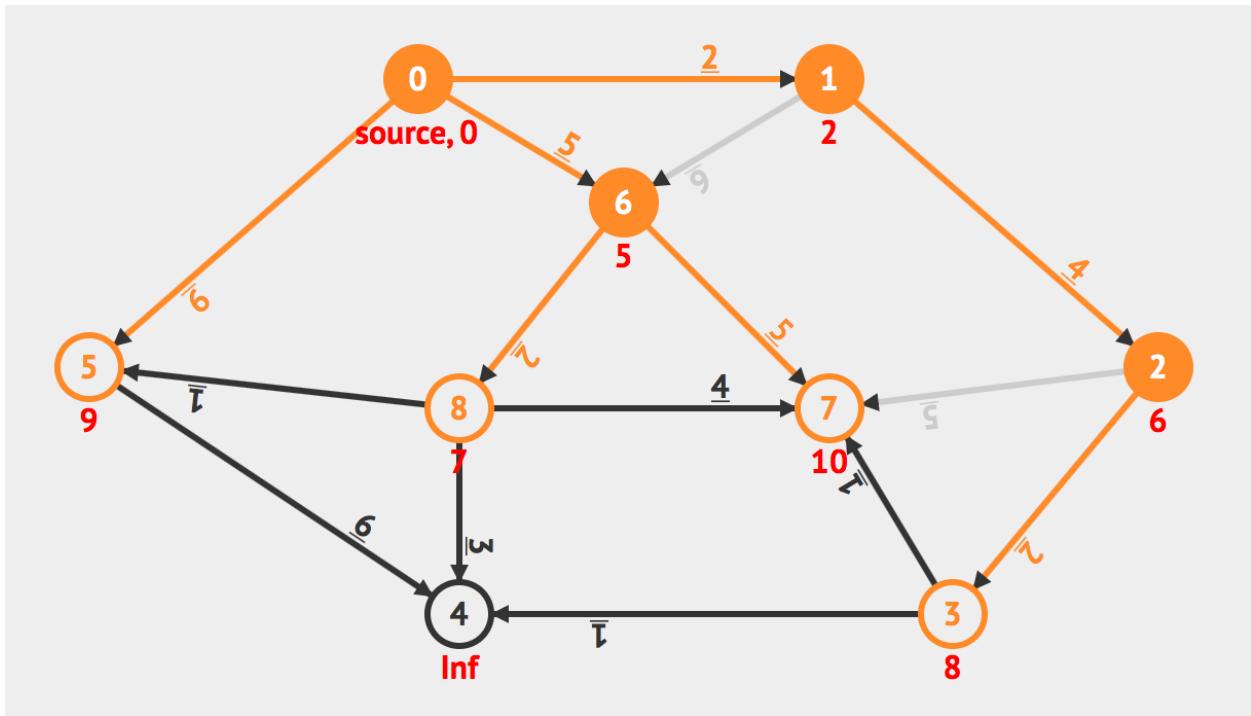
Será escolhido o vértice 1. Note-se que apesar de haver duas arestas que levam ao vértice 6, não haverá alteração do arco candidato, uma vez que $2+6 > 5$.



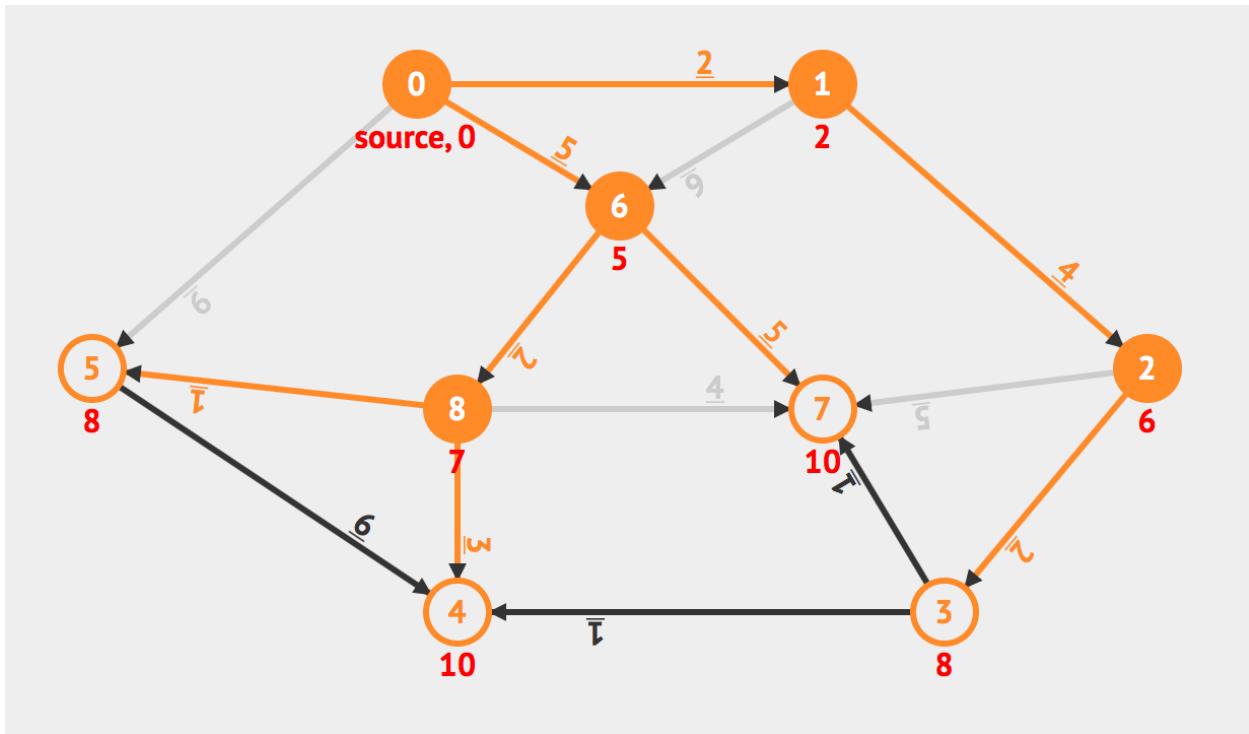


Note-se que, ao acrescentar vértices à orla, a distância desde a raiz é calculada somando a distância da origem ao pai com o peso das arestas acrescentadas à árvore.

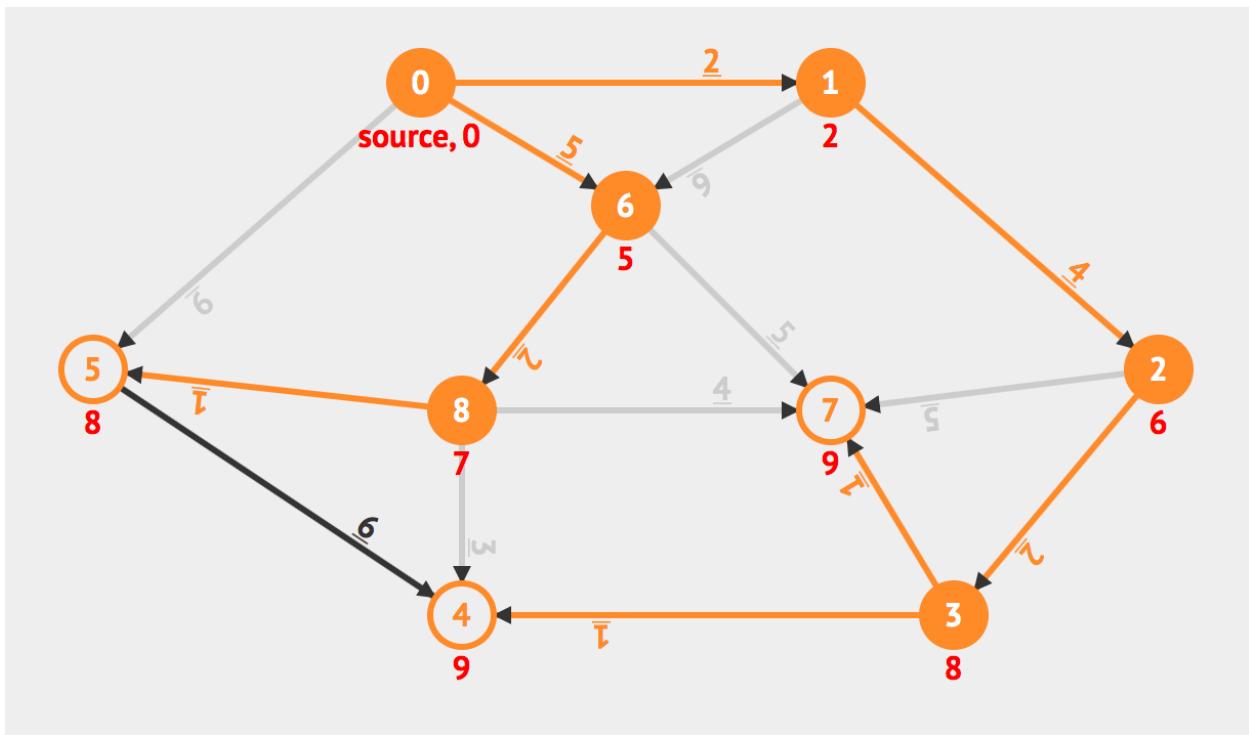
No próximo passo, a distância até 8 será $5+2=7$, e até 7 será $5+5=10$.



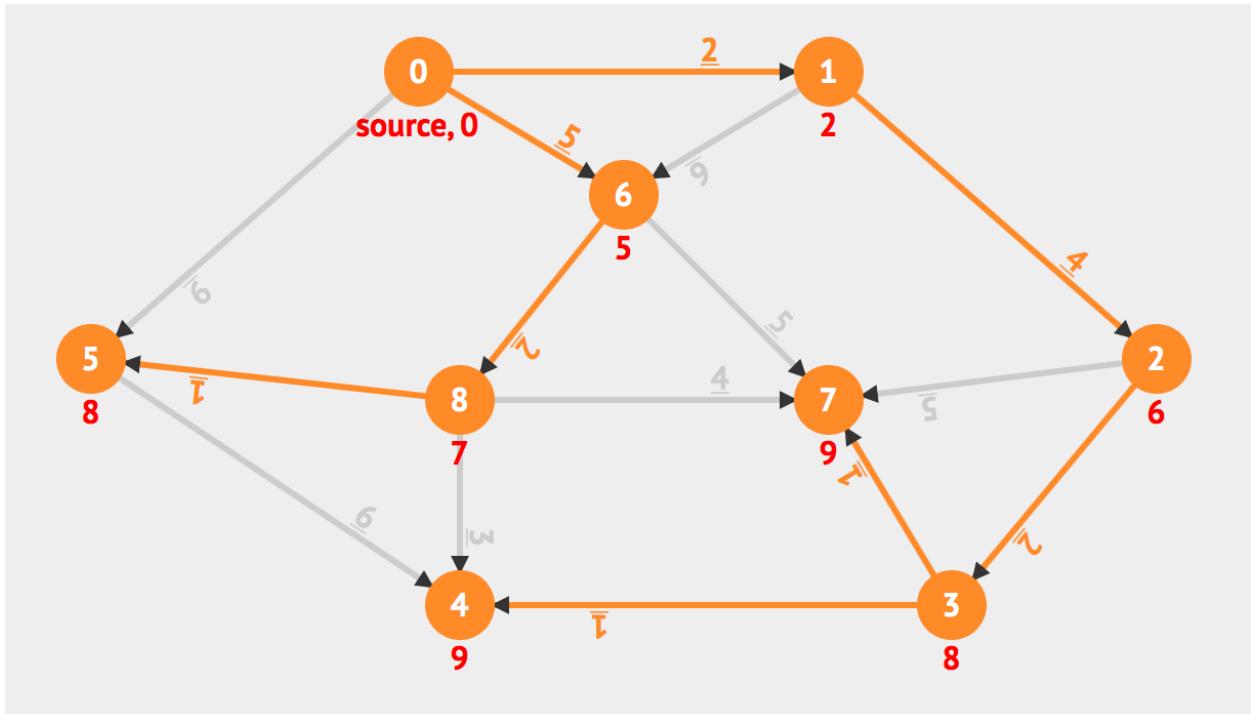
No próximo passo acrescentaremos o vértice 8 à árvore, o que levará a uma alteração do arco candidato de 5, uma vez que $7+1 < 9$. Também a distância desde a raiz, anotada no vértice, será alterada.



No próximo passo haverá nova alteração de arco candidato e distância, desta vez do vértice 7.



Os restantes passos não trazem novidades. A árvore de travessia construída contém todos os caminhos mais curtos com origem no vértice 0.



Algoritmo Detalhado em Python

```
def sp_Dijkstra(g, s):
    fringe = []
    status = []
    ## NEW IN DIJKSTRA
    dist = []
    for i in g:
        status[i] = 'UNSEEN'
    edgeCount = 0

    x = s;
```

```

status[x] = 'INTREE'
parent[x] = -1
## NEW IN DIJKSTRA
dist[x] = 0

while (edgeCount < len(g)-1):
    for y in g[x]:
        wxy = g[x][y]
        if status[y] == 'UNSEEN':
            # add y to fringe with candidate edge (x,y)
            status[y] = 'FRINGE'
            parent[y] = x
            fringe[y] = wxy
            ## NEW IN DIJKSTRA
            dist[y] = dist[x] + wxy
            ## MODIFIED FOR DIJKSTRA
        elif (status[y] == 'FRINGE'
        and dist[x] + wxy < dist[y]):
            # replace candidate edge of y by (x,y)
            parent[y] = x
            fringe[y] = wxy
            ## NEW IN DIJKSTRA ##
            dist[y] = dist[x] + wxy

    # are we blocked? (non-connected graph)

    if len(fringe) == 0:
        return False

    # select next candidate edge and vertex

```

```

    # remove them from fringe and add to tree
    ## MODIFIED FOR DIJKSTRA
    x = min(fringe, key=lambda x:dist[x])
    del fringe[x]
    status[x] = 'INTREE'
    edgeCount += 1

return True

```

Código Python para teste dos algoritmos

Prim:

```

graph = {
    0: { 1: 2, 5: 7, 6: 3 },
    1: { 0: 2, 2: 4, 6: 6 },
    2: { 1: 4, 3: 2, 7: 2 },
    3: { 2: 2, 4: 1, 7: 8 },
    4: { 3: 1, 5: 6, 8: 2 },
    5: { 0: 7, 4: 6, 8: 5 },
    6: { 0: 3, 1: 6, 7: 3, 8: 1 },
    7: { 2: 2, 3: 8, 6: 3, 8: 4 },
    8: { 4: 2, 5: 5, 6: 1, 7: 4 }
}

```

```

def printGraph(g):
    for i in g:
        print"[%s]"%(i),

```

```

        for j in sorted(g[i].keys()):
            print "-> (%s, %s)"%(j,g[i][j]),
        print

printGraph(graph)
print

parent = []
ok = mst_Prim(graph)
if ok:
    sum = 0;
    print "\nMST:\n",
    for i in graph:
        if parent[i]>=0 :
            print"%1s--%1s"%(parent[i], i)
            sum += graph[i][parent[i]]
    print "Total weight = ", sum
else:
    print "UNCONNECTED GRAPH, CANNOT BUILD MST!"

```

Dijkstra:

```

graph = {
    0: { 1: 2, 5: 9, 6: 5 },
    1: { 2: 4, 6: 6 },
    2: { 3: 2, 7: 5 },
    3: { 4: 1, 7: 1 },
    4: { },
    5: { 4: 6 },

```

```

6: { 7: 5, 8: 2 },
7: { },
8: { 4: 3, 5: 1 }
}

def printGraph(g):
    for i in g:
        print"[%s]%(i),
        for j in sorted(g[i].keys()):
            print "-> (%s, %s)"%(j,g[i][j]),
        print

printGraph(graph)
print

parent = {}
ok = sp_Dijkstra(graph, 0)
if ok:
    sum = 0;
    print "\nShortest Paths Tree:\n",
    for i in graph:
        if parent[i]>=0 :
            print"%1s--%1s"%(parent[i], i)
else:
    print "UNCONNECTED GRAPH, CANNOT REACH ALL VERTICES!"

```


G4: Algoritmos sobre Grafos Pesados: Programação Dinâmica

Nota inicial

Neste módulo consideraremos, sem perda de generalidade, que $V = \{0, \dots, |V| - 1\}$. Esta indexação dos vértices de um grafo por números naturais será útil uma vez que os algoritmos que discutiremos se baseiam na representação de grafos por matrizes de adjacências.

Fecho Transitivo de um Grafo Orientado

A noção de *fecho transitivo* de uma relação binária E é bem conhecida: trata-se da relação obtida adicionando-se à relação os pares necessários por forma a obter uma relação transitiva.

Para calcular o fecho transitivo de uma relação basta iterar o seguinte passo básico:

- Se $(i, k) \in E$ e $(k, j) \in E$ então $E := E \cup (i, j)$

Se a relação em questão for a relação de adjacência de um grafo $G = (V, E)$, o seu fecho transitivo será um grafo $G' = (V, E')$, em que E' é o fecho transitivo de E .

Note-se que:

| Se o vértice v é **alcançável** a partir de u em G , então v será **adjacente** a u em G' .

Aparentemente bastará iterar o passo identificado acima, percorrendo sucessivamente todos os pares de vértices i, j e todos os vértices intermédios k , por forma a determinar se j é alcançado a partir de i passando por k .

```
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
```

```

R[i][j] = G[i][j];

for (i=0 ; i<n ; i++)
    for (j=0 ; j<n ; j++)
        for (k=0 ; k<n; k++)
            if (R[i][k] && R[k][j]) R[i][j] = 1;

```

No entanto esta solução apresenta problemas. Basta observar que se tivermos as seguintes 3 adjacências:

$0 \rightarrow 1 \rightarrow 3 \rightarrow 2$

então claramente o algoritmo falhará na inclusão da aresta $0 \rightarrow 2$, uma vez que nem $0 \rightarrow 3$ nem $1 \rightarrow 2$ terão sido acrescentadas em R quando $i = 0$ e $j = 2$:

- $0 \rightarrow 3$ só será acrescentada quando $i = 0$ e $j = 3$, o que acontecerá na próxima iteração do ciclo interior
- $1 \rightarrow 2$ só será acrescentada quando $i = 1$ e $j = 2$, o que acontecerá ainda mais tarde, só na próxima iteração do ciclo exterior

O problema pode ser resolvido iterando o algoritmo acima até não haver mais pares acrescentados, o que levaria a um algoritmo de tempo $O(N^4)$. Existe no entanto uma solução mais simples, que consiste em simplesmente trocar a ordem relativa dos ciclos, passando o vértice intermédio a ser iterado no ciclo mais exterior. O algoritmo resultante é conhecido por algoritmo de Warshall, e executa em tempo $\Theta(N^3)$.

```

void warshall (GraphM G, GraphM R, int n)
{
    int i, j, k;
    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
            R[i][j] = G[i][j];

    for (k=0 ; k<n; k++)

```

```

    for (i=0 ; i<n ; i++)
        for (j=0 ; j<n ; j++)
            if (R[i][k] && R[k][j]) R[i][j] = 1;
}

```

Será talvez mais fácil entender o seu funcionamento observando o seguinte invariante do ciclo exterior.

Seja $S_k = \{0, \dots, k-1\}$, para $k \in V$. Então,

No início de cada iteração do ciclo mais exterior, $R[i][j] == 1$ se existe um caminho (não vazio) de i para j contendo além destes apenas vértices pertencentes ao conjunto $\{0, \dots, k-1\}$

Para a prova de preservação deste invariante basta observar que a iteração k vai testar se existe caminho de i para j passando *adicionalmente* por k , e efectuará $R[i][j] = 1$ se for esse o caso (claro que este valor podia já ser 1 antes!).

Quando $k == n$, na terminação do ciclo, teremos calculado o fecho transitivo, como desejado.

Memoization e Programação Dinâmica

Números de Fibonacci:

Recordemos a definição da sequência de Fibonacci:

$$\begin{aligned} \text{fib}(1) &= 1 \\ \text{fib}(2) &= 1 \\ \text{fib}(n) &= \text{fib}(n-1) + \text{fib}(n-2) \end{aligned}$$

É bem sabido que a implementação recursiva “**top-down**” desta definição é proibitiva, por efectuar muitos cálculos redundantes. Basta expandir $\text{fib}(n-1)$ na definição acima para observar este fenómeno:

$$\text{fib}(n) = 2 * \text{fib}(n - 2) + \text{fib}(n - 3)$$

Por forma a partilhar estes resultados intermédios, basta efectuar o cálculo “bottom-up” armazenando (*memoizing*) os resultados num vector para poderem ser utilizados quando necessário. Isto permite o cálculo em tempo $\Theta(n)$:

```
int Fibonacci (int n) {  
    int k, fib[n];  
  
    fib[1] = 1;  
    fib[2] = 1;  
    for (k=3 ; k<=n; k++)  
        fib[k] = fib[k-1] + fib[k-2];  
    return fib[n];  
}
```

Quando são necessários, os valores de `fib[k-1]` e `fib[k-2]`, em vez de recalculados recursivamente, são simplesmente consultados no vector de resultados.

A estratégia algorítmica conhecida por *programação dinâmica* consiste simplesmente em optimizar a estratégia de cálculo *bottom-up* com memorização de resultados intermédios, observando que, em cada passo de cálculo, são necessários apenas os dois últimos resultados calculados, o que dispensa o armazenamento, neste caso, de toda a série de Fibonacci, bastando guardar em duas variáveis os dois últimos números da sequência.

O resultado é um algoritmo que utiliza apenas espaço em $\Theta(1)$:

```
int Fibonacci (int n) {  
    int k, a=1, b=1, t;  
    for (k=3 ; k<=n; k++) {  
        t = a+b;
```

```

    b = a;
    a = t;
}
return a;
}

```

Distância entre vértices de um grafo:

Consideremos o problema do cálculo da distância entre vértices num grafo pesado.

Naturalmente, uma vez que a distância é o peso do caminho mais curto, podemos aplicar o algoritmo de Dijkstra para identificar o caminho mais curto e calcular então o seu peso. Mas tentemos abordar o problema de outra forma.

Seja $d_k(i,j)$ o peso do caminho mais curto de i para j passando apenas pelos vértices de $S_k = \{s_0, \dots, k-1\}$. Esta noção tem uma definição simples, recursiva em k :

$$d_0(i,j) = w_{i,j} \quad [\text{peso da aresta } (i, j)]$$

$$d_{k+1}(i,j) = \min(d_k(i,j), d_k(i,k) + d_k(k,j))$$

É imediato que a distância entre os vértices é $\delta(i,j) = d_{|V|}(i,j)$

O que é interessante é que este cálculo recursivo apresenta um padrão (**top-down**) muito ineficiente, mas que, tal como no caso do cálculo da sequência de Fibonacci, tem muito potencial para armazenamento de resultados intermédios, se se optar por uma estratégia de cálculo **bottom-up**, calculando e armazenando, por esta ordem, d_1, d_2, \dots, d_n .

Utilizaremos um vector de matrizes (i.e., uma matriz tri-dimensional) tal que $D[k][i][j] = d_k(i,j)$.

É imediato escrever o algoritmo com *memoization* com base na definição anterior:

```

void memoDistances (GraphM G, GraphM D[MAX], int n)
{

```

```

int i, j, k;
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        D[0][i][j] = G[i][j];

for (k=0 ; k<n; k++)
    for (i=0 ; i<n ; i++)
        for (j=0 ; j<n ; j++)
            D[k+1][i][j] = min(D[k][i][j], D[k][i][k]+D[k][k]
[j]);
}

```

Mais uma vez, o passo decisivo para se obter um algoritmo baseado em programação dinâmica é a observação de que é possível dispensar o armazenamento de matrizes intermédias, usando uma única matriz D que armazena, no início da iteração k , o equivalente à matriz $D[k]$ no algoritmo anterior.

```

void dynDistances (GraphM G, GraphM D, int n)
{
    int i, j, k;
    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
            D[i][j] = G[i][j];

    for (k=0 ; k<n; k++)
        for (i=0 ; i<n ; i++)
            for (j=0 ; j<n ; j++)
                D[i][j] = min(D[i][j], D[i][k] + D[k][j]);
}

```

Observe-se que $d_{k+1}(i, k) = d_k(i, k)$ e $d_{k+1}(k, j) = d_k(k, j)$, e por isso o conteúdo da linha k e a coluna k não são alteradas. É isto que torna desnecessário armazenar a matriz anterior ($k-1$).

All-Pairs Shortest Paths: algoritmo de Floyd-Warshall

O algoritmo de Floyd-Warshall é uma variante deste que calcula, além das distâncias, os próprios *caminhos mais curtos entre todos os pares de vértices do grafo*.

Os caminhos são armazenados de forma compacta, numa matriz bi-dimensional de vértices intermédios.

```
void FloydWarshall (GraphM G, GraphM D, int P[MAX][MAX], int n)
{
    int i, j, k;

    for (i=0; i<n; i++)
        for (j=0; j<n; j++) {
            D[i][j] = G[i][j];
            P[i][j] = -1;
        }

    for (k=0 ; k<n; k++)
        for (i=0 ; i<n ; i++)
            for (j=0 ; j<n ; j++) {
                if (D[i][k] + D[k][j] < D[i][j]) {
                    D[i][j] = D[i][k] + D[k][j];
                    P[i][j] = k;
                }
            }
}
```

```
        }
    }
}
```

Para compreender a interpretação da informação de vértices intermédios calculados na matriz P, considere-se o seguinte exemplo (cfr. grafo usado como exemplo em [+G3. Algoritmos sobre Grafos Pesados: Estratégia Greedy](#)):

Para saber qual é o caminho mais curto entre os vértices 0 e 5, consulta-se $P[0][5] == 8$. Isto significa que o caminho passa pelo vértice 8. Consulta-se então recursivamente:

- $P[0][8] == 6$. Então consulta-se ainda recursivamente:
 - $P[0][6] == -1$
 - $P[6][8] == -1$
- $P[8][5] == -1$

Os valores -1 são casos de paragem, significando que o caminho mais curto corresponde nestes casos a arestas directas. Sendo assim, o caminho mais curto entre 0 e 5 é (0, 6, 8, 5).

O tempo de execução deste algoritmo é em termos assintóticos semelhante ao do algoritmo de Dijkstra (repetido a partir de todos os vértices do grafo).

Código para teste dos algoritmos:

```
int main(int argc, const char * argv[]) {

    GraphM gm1 = {
        { 0, 0, 1, 0 },
        { 1, 0, 0, 0 },
        { 0, 0, 0, 1 },
        { 0, 1, 0, 0 }
    };
}
```

```

GraphM gm1;
int n1 = 4;

GraphM gm2 = {
    { NE, 2, NE, NE, NE, 9, 5, NE, NE},
    { 2, NE, 4, NE, NE, NE, 6, NE, NE},
    { NE, 4, NE, 2, NE, NE, NE, 5, NE},
    { NE, NE, 2, NE, 1, NE, NE, 1, NE},
    { NE, NE, NE, 1, NE, 6, NE, NE, 3},
    { 9, NE, NE, NE, 6, NE, NE, NE, 1},
    { 5, 6, NE, NE, NE, NE, NE, 5, 2},
    { NE, NE, 5, 1, NE, NE, 5, NE, 4},
    { NE, NE, NE, NE, 3, 1, 2, 4, NE}
};

GraphM gm2[MAX];
int paths[MAX][MAX];
int n2 = 9;

warshall (gm1, gm1, n1);
printGraphM(gm1, n1);

memoDistances(gm2, gm2, n2);
printGraphM(gm2[n2], n2); printf("\n");

// dynDistances(gm2, gm1, n2);

FloydWarshall(gm2, gm1, paths, n2);
printGraphM(gm1, n2); printf("\n");
printGraphM(paths, n2);

}

```

