

ALGORITMOS E COMPLEXIDADE

GUIÃO DAS AULAS PRÁTICAS

António Manuel Adrego da Rocha

**Departamento de Eletrónica, Telecomunicações e Informática
Universidade de Aveiro**

Ano Letivo de 2014/2015

Programa das aulas práticas de Algoritmos e Complexidade

- Análise Empírica da Complexidade de Algoritmos (3 aulas)
- Tipos de Dados Abstratos simples baseados em *arrays* e listas (3 aulas)
- Pesquisa (1 aula)
- Análise Empírica da Complexidade de Algoritmos Recursivos (2 aulas)
- Árvores Binárias ABP e AVL (2 aulas)
- Filas com Prioridade (1 aula)
- Grafos (2 aulas)

Bibliografia para as aulas práticas

- Estruturas de Dados e Algoritmos em C (3^a Edição Revista e Aumentada), António Adrego da Rocha, FCA Editora de Informática, 2014.
- Análise da Complexidade de Algoritmos, António Adrego da Rocha, FCA Editora de Informática, 2014.

Material para as aulas práticas

- O material necessário para as aulas práticas é disponibilizado na seguinte página da Internet:
<http://sweet.ua.pt/adrego/algoritmos/>.

Modo de Avaliação

- A nota final da disciplina é sempre calculada, em qualquer das épocas de exame, usando: a nota do exame escrito realizado na época de exames – com um peso de 50% – e a nota da componente prática cujos trabalhos são entregues ao longo do semestre – com um peso de 50%, sendo que em ambas as componentes há uma nota mínima de 8.0 valores (em 20).
- A nota da componente prática é calculada usando as notas de três trabalhos práticos – com um peso de 35% – e a nota das restantes aulas práticas – com um peso de 15%.
 - O primeiro trabalho prático tem um peso de 10% e é dedicado à análise da complexidade de algoritmos simples. Consiste na resolução dos guiões das aulas práticas 2 e 3 e da entrega dos respectivos guiões devidamente preenchidos.
 - O segundo trabalho prático tem um peso de 10% e é dedicado à implementação de tipos de dados abstratos. Consiste na resolução dos guiões das aulas práticas 4, 5 e 6 e da entrega dos respectivos guiões devidamente preenchidos e dos ficheiros de implementação dos tipos de dados abstratos polinómio e matriz (*polynomial.c* e *matrix.c*).
 - O terceiro trabalho prático tem um peso de 15% e é dedicado à implementação de algoritmos de dígrafos. Consiste na resolução dos guiões das aulas práticas 12, 13, 14 e do trabalho final sobre este tipo de dados abstrato e da entrega dos ficheiros de implementação dos tipos de dados abstratos fila com prioridade de Dijkstra e dígrafo (*pqueue_dijkstra.c* e *digraph.c*).
 - A resolução dos guiões das restantes aulas práticas tem um peso de 15% na nota prática, com especial incidência nas aulas 8 e 9 sobre recursividade (peso de 5%) e nas aulas 10 e 11 sobre árvores binárias (peso de 5%). A avaliação é feita através da entrega dos respectivos guiões devidamente preenchidos e do ficheiro de implementação do tipo de dados abstrato árvore binária de pesquisa (*abp.c*).

AULA 1 - ANÁLISE DA COMPLEXIDADE DE ALGORITMOS

Implemente cada um dos seguintes algoritmos e determine experimentalmente a complexidade do número de adições em função da dimensão da entrada n, considerando n par e potência de 2.

```
int Algorithm1 (int n)
{
    int i, j, sum = 0;
    for (i = 1; i <= n; i++)
        for (j = i; j <= n; j *= 2) sum += j;
    return sum;
}
```

```
int Algorithm2 (int n)
{
    int i, j, sum = 0;
    for (i = 2; i <= n; i *= 2)
        for (j = i-1; j <= i+1; j++) sum += j;
    return sum;
}
```

```
int Algorithm3 (int n)
{
    int i, j, sum = 0;
    for (i = 1; i <= n; i++)
        for (j = i; j <= n; j++) sum += j;
    return sum;
}
```

```
int Algorithm4 (int n)
{
    int i, j, sum = 0;
    for (i = 1; i <= n; i++)
        for (j = n; j > 1; j /= 2) sum += j;
    return sum;
}
```

Preencha a tabela com o valor da função e o número de adições para os sucessivos valores de entrada.

N	Algorithm1 (N)	Nº de Adições	Algorithm2 (N)	Nº de Adições	Algorithm3 (N)	Nº de Adições	Algorithm4 (N)	Nº de Adições
1								
2								
4								
8								
16								
32								
64								
O(N)								

Depois da simulação dos algoritmos responda às seguintes questões:

- Analisando os dados da tabela qual é a complexidade de cada algoritmo?
- Determine a complexidade formal de cada algoritmo obtendo uma expressão que corresponda aos valores obtidos experimentalmente. Faça as análises no verso da folha.

AULA 2 - ANÁLISE DA COMPLEXIDADE DE ALGORITMOS

Seja uma dada sequência (*array*) de números inteiros. Pretende-se determinar quantos elementos da sequência são iguais à soma dos elementos anteriores. Ou seja:

$$\text{array}[0] + \text{array}[1] + \dots + \text{array}[i-1] = \text{array}[i], \text{ para } i > 0$$

- Implemente uma função inteira eficiente e eficaz que determina quantos elementos (resultado da função) de uma sequência com n elementos respeitam esta propriedade.
Depois de validar o algoritmo apresente-o no verso da folha
- Determine experimentalmente a complexidade do número de adições efectuadas envolvendo elementos da sequência. Considere as seguintes dez sequências de dez inteiros todas diferentes e que cobrem todas as situações possíveis distintas de execução do algoritmo. Calcule para cada uma delas o número de elementos que obedecem à condição e o número de adições executadas.

10 3 15 7 9 20 11 25 27 29	Resultado	Nº de operações
10 3 15 7 35 33 20 55 27 29	Resultado	Nº de operações
10 3 15 7 33 68 20 156 99 27	Resultado	Nº de operações
1 6 3 10 33 20 73 146 99 27	Resultado	Nº de operações
1 6 3 10 33 20 73 146 -96 196	Resultado	Nº de operações
2 1 3 6 12 20 44 -14 74 16	Resultado	Nº de operações
2 1 3 6 12 24 48 -20 -18 58	Resultado	Nº de operações
2 1 3 6 12 24 48 96 -98 94	Resultado	Nº de operações
2 2 4 8 16 31 63 126 252 504	Resultado	Nº de operações
2 2 4 8 16 32 64 128 256 512	Resultado	Nº de operações

Depois da simulação do algoritmo responda às seguintes questões:

- Em termos do número de adições efectuadas podemos distinguir alguma variação na execução do algoritmo? Ou seja, existe a situação de melhor caso e de pior caso, ou estamos perante um algoritmo com caso sistemático?
- Qual é a complexidade do algoritmo?
- Determine a complexidade formal do algoritmo. Tenha em atenção que deve obter uma expressão matemática exacta e simplificada. Faça a análise no fim da folha
- Calcule a expressão para $N = 10$ e compare-a com os resultados obtidos experimentalmente.

AULA 3 - ANÁLISE DA COMPLEXIDADE DE ALGORITMOS

Seja uma dada sequência (*array*) de números inteiros ordenada por ordem não-decrescente. Pretende-se determinar se a sequência das diferenças entre os pares de elementos sucessivos é uma sequência crescente contínua de inteiros. Por exemplo, a sequência {12, 15, 19, 24, 30, 37} define a sequência de diferenças {3, 4, 5, 6, 7}, que é uma sequência crescente contínua de inteiros a começar em 3. E a sequência {10, 14, 19, 25, 32, 40} define a sequência de diferenças {4, 5, 6, 7, 8}, que é uma sequência crescente contínua de inteiros a começar em 4. Mas a sequência {11, 15, 20, 26, 32, 40} define a sequência de diferenças {4, 5, 6, 6, ...} que não é uma sequência crescente contínua de inteiros, porque a quarta diferença devia ser 7 e é 6.

- Implemente uma função inteira eficiente e eficaz que, sem utilizar um *array* auxiliar, verifica se uma sequência com n elementos define uma sequência de diferenças que é crescente e contínua. Considere o resultado da função do tipo: 1 para verdadeiro e 0 para falso.
Depois de validar o algoritmo apresente-o no verso da folha
- Determine experimentalmente a complexidade do número de operações aritméticas aditivas (subtrações e adições) envolvendo os elementos da sequência e o cálculo do incremento da diferença. Considere as seguintes nove sequências de dez inteiros todas diferentes e que cobrem todas as situações possíveis distintas de execução do algoritmo. Calcule para cada uma delas se se verifica a condição pretendida e o número de operações aritméticas aditivas executadas.

<table border="1"><tr><td>1</td><td>3</td><td>5</td><td>7</td><td>9</td><td>11</td><td>20</td><td>25</td><td>27</td><td>29</td></tr></table>	1	3	5	7	9	11	20	25	27	29	<table border="1"><tr><td>Resultado</td></tr></table>	Resultado	<table border="1"><tr><td>Nº de operações</td></tr></table>	Nº de operações
1	3	5	7	9	11	20	25	27	29					
Resultado														
Nº de operações														
<table border="1"><tr><td>1</td><td>3</td><td>6</td><td>9</td><td>11</td><td>13</td><td>20</td><td>25</td><td>27</td><td>29</td></tr></table>	1	3	6	9	11	13	20	25	27	29	<table border="1"><tr><td>Resultado</td></tr></table>	Resultado	<table border="1"><tr><td>Nº de operações</td></tr></table>	Nº de operações
1	3	6	9	11	13	20	25	27	29					
Resultado														
Nº de operações														
<table border="1"><tr><td>1</td><td>3</td><td>6</td><td>10</td><td>11</td><td>13</td><td>20</td><td>25</td><td>27</td><td>29</td></tr></table>	1	3	6	10	11	13	20	25	27	29	<table border="1"><tr><td>Resultado</td></tr></table>	Resultado	<table border="1"><tr><td>Nº de operações</td></tr></table>	Nº de operações
1	3	6	10	11	13	20	25	27	29					
Resultado														
Nº de operações														
<table border="1"><tr><td>1</td><td>3</td><td>6</td><td>10</td><td>15</td><td>17</td><td>20</td><td>25</td><td>27</td><td>29</td></tr></table>	1	3	6	10	15	17	20	25	27	29	<table border="1"><tr><td>Resultado</td></tr></table>	Resultado	<table border="1"><tr><td>Nº de operações</td></tr></table>	Nº de operações
1	3	6	10	15	17	20	25	27	29					
Resultado														
Nº de operações														
<table border="1"><tr><td>1</td><td>3</td><td>6</td><td>10</td><td>15</td><td>21</td><td>22</td><td>25</td><td>27</td><td>29</td></tr></table>	1	3	6	10	15	21	22	25	27	29	<table border="1"><tr><td>Resultado</td></tr></table>	Resultado	<table border="1"><tr><td>Nº de operações</td></tr></table>	Nº de operações
1	3	6	10	15	21	22	25	27	29					
Resultado														
Nº de operações														
<table border="1"><tr><td>1</td><td>3</td><td>6</td><td>10</td><td>15</td><td>21</td><td>28</td><td>30</td><td>37</td><td>39</td></tr></table>	1	3	6	10	15	21	28	30	37	39	<table border="1"><tr><td>Resultado</td></tr></table>	Resultado	<table border="1"><tr><td>Nº de operações</td></tr></table>	Nº de operações
1	3	6	10	15	21	28	30	37	39					
Resultado														
Nº de operações														
<table border="1"><tr><td>1</td><td>3</td><td>6</td><td>10</td><td>15</td><td>21</td><td>28</td><td>36</td><td>39</td><td>49</td></tr></table>	1	3	6	10	15	21	28	36	39	49	<table border="1"><tr><td>Resultado</td></tr></table>	Resultado	<table border="1"><tr><td>Nº de operações</td></tr></table>	Nº de operações
1	3	6	10	15	21	28	36	39	49					
Resultado														
Nº de operações														
<table border="1"><tr><td>1</td><td>3</td><td>6</td><td>10</td><td>15</td><td>21</td><td>28</td><td>36</td><td>45</td><td>49</td></tr></table>	1	3	6	10	15	21	28	36	45	49	<table border="1"><tr><td>Resultado</td></tr></table>	Resultado	<table border="1"><tr><td>Nº de operações</td></tr></table>	Nº de operações
1	3	6	10	15	21	28	36	45	49					
Resultado														
Nº de operações														
<table border="1"><tr><td>1</td><td>3</td><td>6</td><td>10</td><td>15</td><td>21</td><td>28</td><td>36</td><td>45</td><td>55</td></tr></table>	1	3	6	10	15	21	28	36	45	55	<table border="1"><tr><td>Resultado</td></tr></table>	Resultado	<table border="1"><tr><td>Nº de operações</td></tr></table>	Nº de operações
1	3	6	10	15	21	28	36	45	55					
Resultado														
Nº de operações														

Depois da simulação do algoritmo responda às seguintes questões:

- Qual é a sequência que corresponde ao melhor caso do algoritmo?
- Quais são as sequências que correspondem ao pior caso do algoritmo? O que é que as distingue?
- Determine o número de operações aditivas efectuadas no caso médio do algoritmo (para $N = 10$).
- Qual é a complexidade do algoritmo?
- Determine a complexidade formal do algoritmo nas situações do melhor caso, do pior caso e do caso médio, considerando uma sequência com dimensão N (com $N > 2$). Tenha em atenção que deve obter uma expressão matemática exacta e simplificada. Faça as análises no verso da folha
- Calcule as expressões para $N = 10$ e compare-as com os resultados obtidos experimentalmente.

AULAS 4, 5 E 6 - TIPOS DE DADOS ABSTRATOS

Para compreender a implementação de tipos de dados abstratos na linguagem C utilizando o paradigma modular, deve começar por ler o capítulo 2 do livro *Estruturas de Dados e Algoritmos em C*, analisando a implementação do tipo de dados abstrato COMPLEX (ponto 2.7 – Exemplo de um tipo de dados elemento matemático, páginas 50-57).

O tipo de dados abstrato VECTOR é constituído pelo ficheiro de interface `vector.h` e pelo ficheiro de implementação `vector.c` e implementa a criação de vectores e de operações sobre vectores. Este tipo de dados tem capacidade de múltipla instânciação e usa um controlo centralizado de erros.

Para armazenar as componentes de um vector é usado um *array*, permitindo assim, que os algoritmos matemáticos das operações sobre vectores sejam facilmente implementáveis sobre esta estrutura de dados indexada. A figura apresenta o armazenamento de um vector com 5 componentes num *array*.

$(v_4, v_3, v_2, v_1, v_0) = (2.5, 1.0, 0.0, 3.0, 5.5)$				
5.5	3.0	0.0	1.0	2.5
Vector[0]	Vector[1]	Vector[2]	Vector[3]	Vector[4]

Comece por ler com atenção os ficheiros e de seguida compile o módulo, usando para o efeito o comando `cc -c vector.c`, sendo que `cc` é um *alias* do compilador da linguagem C programado da seguinte maneira `gcc -ansi -Wall`. A compilação deve gerar o ficheiro objecto `vector.o`. De seguida compile e teste as aplicações `testvector.c` e `simvector.c`, que testam as operações do módulo `vector`. O primeiro programa é uma aplicação simples, enquanto que o segundo é uma aplicação gráfica. Não se esqueça que para compilar as aplicações, tem que mencionar o ficheiro objecto do tipo de dados (`vector.o`) no comando de compilação. Também é fornecida a *makefile* `mkvector`, para compilar o módulo e as aplicações. Teste convenientemente toda a funcionalidade do tipo de dados.

- Pretende-se desenvolver o tipo de dados abstrato POLY para processar polinómios de coeficientes reais, usando como estrutura de dados de suporte um *array*. O tipo de dados deve ter capacidade de múltipla instânciação e controlo centralizado de erros.

Para armazenar os coeficientes de um polinómio é usado um *array*, permitindo assim, que os algoritmos matemáticos das operações sobre polinómios sejam facilmente implementáveis sobre esta estrutura de dados indexada. A figura mostra o armazenamento de um polinómio de grau 4 (5 coeficientes) num *array*.

$3.5x^4 + 2.5x^3 + x^2 + 4.5$				
4.5	0.0	1.0	2.5	3.5
Poly[0]	Poly[1]	Poly[2]	Poly[3]	Poly[4]

A funcionalidade pretendida é especificada pelo ficheiro de interface `polynomial.h`, sendo também fornecido o esqueleto do ficheiro de implementação `polynomial.c`. São ainda fornecidas as aplicações `testpolynomial.c` e `simpolynomial.c` e a *makefile* `mkpolynomial`, para compilar o módulo e as aplicações.

Comece por ler com atenção os ficheiros e complete o módulo. Tenha em atenção que um polinómio de grau N tem N+1 coeficientes e que os polinómios devem ser sempre mantidos na sua forma mais reduzida.

Teste convenientemente toda a funcionalidade do tipo de dados. Determine os resultados das seguintes operações sobre os polinómios indicados:

$$\text{polinómio}[0] = 5x^3 + 4x^2 + 3x - 5 \quad \text{polinómio}[1] = 2x^2 + 5x + 2 \quad \text{polinómio}[2] = 5x^3 + 6x^2 + 3x - 7$$

$$\text{polinómio}[3] = \text{polinómio}[0] + \text{polinómio}[1] =$$

$$\text{polinómio}[4] = \text{polinómio}[1] + \text{polinómio}[0] =$$

$$\text{polinómio}[5] = \text{polinómio}[0] - \text{polinómio}[1] =$$

$$\text{polinómio}[6] = \text{polinómio}[1] - \text{polinómio}[0] =$$

$$\text{polinómio}[7] = \text{polinómio}[3] - \text{polinómio}[2] =$$

$$\text{polinómio}[8] = \text{polinómio}[1] \times \text{polinómio}[0] =$$

$$\text{polinómio}[9] = \text{polinómio}[0] \times \text{polinómio}[1] =$$

$$\text{polinómio}[0] = 5x^3 + 4x^2 + 3x - 5 \quad (\text{para } x=0.5) =$$

- Pretende-se desenvolver o tipo de dados abstrato MATRIX para processar matrizes de NLxNC elementos reais, usando como estrutura de dados de suporte um *array* bidimensional. O tipo de dados deve ter capacidade de múltipla instânciação e controlo centralizado de erros.

A funcionalidade pretendida é especificada pelo ficheiro de interface **matrix.h**, sendo também fornecido o esqueleto do ficheiro de implementação **matrix.c**. Comece por ler com atenção os ficheiros e complete o módulo.

As operações de transposição de uma matriz, de soma e de produto de matrizes implementam-se através das expressões indicadas de seguida:

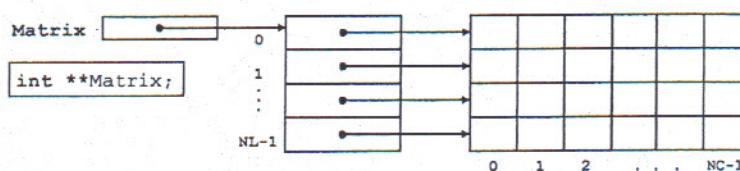
$$\text{Matriz_Transposta}[j,i] = \text{Matriz}[i,j], \text{ com } 1 \leq i \leq \text{NL} \text{ e } 1 \leq j \leq \text{NC}.$$

$$\text{Matriz_Soma}[i,j] = \text{Matriz_A}[i,j] + \text{Matriz_B}[i,j], \text{ com } 1 \leq i \leq \text{NL} \text{ e } 1 \leq j \leq \text{NC}.$$

$$\text{Matriz_Produto}[i,j] = \sum_{k=1}^{\text{NCA}} \text{Matriz_A}[i,k] \times \text{Matriz_B}[k,j], \text{ com } 1 \leq i \leq \text{NLA} \text{ e } 1 \leq j \leq \text{NCB}.$$

São ainda fornecidas as aplicações **testmatrix.c** e **simmatrix.c** e a *makefile* **mkmatrix**, para compilar o módulo e as aplicações.

Sugestão: Para poder manipular as matrizes com acesso indexado do tipo **Matriz[i][j]**, implemente na memória dinâmica uma estrutura de dados matricial, como se indica na figura. Os excertos de código necessários para criar e destruir uma sequência bidimensional com NLxNC elementos inteiros são apresentados na Figura 2.6 do capítulo 2 do livro *Estruturas de Dados e Algoritmos em C* (página 46).



Teste convenientemente toda a funcionalidade do tipo de dados. Determine os resultados das seguintes operações sobre as matrizes indicadas:

$$\begin{bmatrix} 2 & 3 & 4 \\ 1 & 2 & 5 \\ 7 & 2 & 1 \end{bmatrix} + \begin{bmatrix} 1 & 2 & 4 \\ 2 & 5 & 3 \\ 2 & 1 & 2 \end{bmatrix} = \begin{bmatrix} \quad & \quad & \quad \\ \quad & \quad & \quad \\ \quad & \quad & \quad \end{bmatrix}$$

$$\begin{bmatrix} 2 & 3 & 4 \\ 1 & 2 & 5 \\ 7 & 2 & 1 \end{bmatrix} - \begin{bmatrix} 1 & 2 & 4 \\ 2 & 5 & 3 \\ 2 & 1 & 2 \end{bmatrix} = \begin{bmatrix} \quad & \quad & \quad \\ \quad & \quad & \quad \\ \quad & \quad & \quad \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 1 & 2 & 1 & 0 \\ 3 & 4 & 2 & 1 \\ 1 & 3 & 0 & 5 \end{bmatrix} = \begin{bmatrix} \quad & \quad & \quad & \quad \\ \quad & \quad & \quad & \quad \\ \quad & \quad & \quad & \quad \end{bmatrix}$$

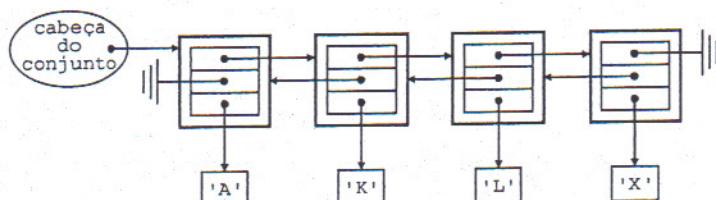
Determinante da matriz

$$\begin{bmatrix} 3 & 4 & 2 & 5 \\ 4 & 2 & 2 & 1 \\ 1 & 3 & 2 & 2 \\ 0 & 5 & 3 & 2 \end{bmatrix} =$$

- Pretende-se desenvolver o tipo de dados abstrato SET para processar conjuntos de caracteres alfabéticos maiúsculos, usando como estrutura de dados de suporte uma lista biligada para armazenar os elementos do conjunto de modo ordenado. Implemente um módulo, com capacidade de múltipla instânciação e com controlo de erros.

A funcionalidade pretendida é especificada pelo ficheiro de interface **set.h**, sendo também fornecido o esqueleto do ficheiro de implementação **set.c**. Comece por ler com atenção os ficheiros e complete o módulo. São também fornecidas as aplicações **testset.c** e **simset.c** e a *makefile* **mkset**, para compilar o módulo e as aplicações. Teste convenientemente toda a funcionalidade do tipo de dados.

Para armazenar os elementos de conjunto deve ser usada uma lista biligada, mantendo os seus elementos sempre ordenados. Desta forma optimiza-se os algoritmos associados às operações habituais sobre conjuntos. A figura apresenta o armazenamento do conjunto {A, K, L, X} numa lista biligada.



Sugestão: Para se familiarizar com as listas biligadas e os seus algoritmos de manipulação, comece por ler o ponto 3.3 – Listas biligadas do livro *Estruturas de Dados e Algoritmos em C*, páginas 96-102.

AULA 7 - PESQUISA

- Pretende-se determinar experimentalmente as complexidades das duas seguintes versões do algoritmo de pesquisa ternária e compará-las com a da primeira versão apresentada na aula teórica.

```

int TernarySearchV3 (int array[], int n, int value)
{
    int min, max, nelem, fpivot, pivot;
    min = 0; max = n-1;
    while (min <= max)
    {
        nelem = max - min + 1;
        if (nelem % 3 == 0)
            { fpivot = min + (nelem / 3) - 1; pivot = min + (2 * nelem / 3) - 1; }
        else
            { fpivot = min + (nelem / 3); pivot = min + (2 * nelem / 3); }
        if (array[fpivot] == value) return fpivot;
        else if (array[fpivot] > value) max = fpivot - 1;
        else
            {
                if (array[pivot] > value)
                    { min = fpivot + 1; max = pivot - 1; }
                else
                    {
                        if (array[pivot] == value) return pivot;
                        else min = pivot + 1;
                    }
            }
    }
    return -1;
}

```

```

int TernarySearchV4 (int array[], int n, int value)
{
    int min, max, nelem, fpivot, pivot;
    min = 0; max = n-1;
    while (min <= max)
    {
        nelem = max - min + 1;
        if (nelem % 3 == 0)
            { fpivot = min + (nelem / 3) - 1; pivot = min + (2 * nelem / 3) - 1; }
        else
            { fpivot = min + (nelem / 3); pivot = min + (2 * nelem / 3); }
        if (array[fpivot] >= value)
            {
                if (array[fpivot] == value) return fpivot;
                max = fpivot - 1;
            }
        else
            {
                if (array[pivot] >= value)
                    {
                        if (array[pivot] == value) return pivot;
                        min = fpivot + 1; max = pivot - 1;
                    }
                else min = pivot + 1;
            }
    }
    return -1;
}

```

Os resultados obtidos deverão ser analisados relativamente ao comportamento individual de cada um dos algoritmos implementados com o crescimento do número de elementos da sequência a pesquisar.

Os testes deverão ilustrar o comportamento dos algoritmos no caso médio. Neste caso, deverão ser considerados os seguintes tipos de situações e de testes:

- Pesquisa com 100% de sucesso. O elemento procurado está sempre presente no *array*. é sucessivamente procurado, um a um, cada um dos elementos do *array*;
- Pesquisa com 50% de sucesso. O elemento procurado pode não estar presente no *array*: são alternada e sucessivamente procurados elementos que pertencem e não pertencem ao *array*.

Faça a simulação dos algoritmos para sequências com $N = 2 \times 3^K - 1$ (com $5 \leq K \leq 10$) números inteiros pares ordenados por ordem crescente e determine o número médio de comparações efectuadas para os dois casos médios da pesquisa. Proceda da seguinte forma. Crie um *array* na memória dinâmica com capacidade para armazenar N inteiros (sendo N um parâmetro de entrada do programa de simulação) e preencha-o com os números pares de 2 até $2N$.

Para determinar a complexidade do caso médio com 100% de sucesso pesquise sucessivamente os valores inteiros pares de 2 até $2N$, acumulando o número de comparações efectuadas em cada pesquisa e calculando no final a média do número de comparações das N pesquisas.

Para determinar a complexidade do caso médio com 50% de sucesso pesquise sucessivamente os valores inteiros de 1 até $2N+1$, acumulando o número de comparações efectuadas em cada pesquisa e calculando no final a média do número de comparações das $2N+1$ pesquisas (N pesquisas com sucesso e $N+1$ pesquisas sem sucesso).

K	N	P – A(N) 100%	P – A(N) 50%
5	485		
6	1457		
7	4373		
8	13121		
9	39365		
10	118097		

- Faça a análise teórica (no verso da folha) dos casos médios destes dois algoritmos de pesquisa ternária para *arrays* com $N = 2 \times 3^K - 1$ elementos e compare os resultados experimentais com os resultados teóricos.

Sugestão: tenha em atenção a análise teórica dos casos médios da primeira versão do algoritmo de pesquisa ternária apresentada na aula teórica.

AULA 8 - ANÁLISE DA COMPLEXIDADE DE ALGORITMOS RECURSIVOS (ALGORITMOS SIMPLES)

Implemente os seguintes **algoritmos recursivos** e calcule o número de operações aritméticas (aditivas ou multiplicativas) executadas pelos algoritmos:

- Cálculo da potência x^n usando os seguintes métodos:

$$1^{\circ} \text{ método} \rightarrow x^n = x \times x^{n-1}$$

$$2^{\circ} \text{ método} \rightarrow x^n = \begin{cases} 1, & \text{se } n = 0 \\ (x^{n/2})^2, & \text{se } n \text{ é par} \\ x \times (x^{n/2})^2, & \text{se } n \text{ é ímpar} \end{cases}$$

- Preencha a tabela com o valor da função (para $x = 0.5$) e o número de multiplicações para os sucessivos valores de n .

N	1º método (N)	Nº de Multiplicações	2º método (N)	Nº de Multiplicações
1				
2				
3				
4				
5				
6				
7				
8				
9				
10				
11				
12				
13				
14				
15				
16				
O(N)				

- Analisando os dados da tabela qual é a complexidade de cada algoritmo?
- Determine a complexidade formal de cada algoritmo obtendo uma expressão que corresponda aos valores obtidos experimentalmente. Faça as análises no verso da folha.

- Cálculo do número de Fibonacci

$$F(n) = \begin{cases} 0, & \text{se } n = 0 \\ 1, & \text{se } n = 1 \\ F(n - 1) + F(n - 2), & \text{se } n \geq 2 \end{cases}$$

Implemente as várias versões (recursiva, dinâmica e repetitiva) do cálculo do número de Fibonacci e confirme as complexidades do número de adições de cada algoritmo apresentadas no capítulo 4 do livro *Análise da Complexidade de Algoritmos* (páginas 137 a 142).

AULA 9 - ANÁLISE DA COMPLEXIDADE DE ALGORITMOS RECURSIVOS (NÚMERO DE SCHRÖDER)

- Implemente uma função recursiva para calcular o número de Schröder, definido pela seguinte relação de recorrência:

$$\text{Schröder } (n) = \begin{cases} 1, & \text{se } n = 0 \\ \text{Schröder } (n - 1) + \sum_{i=0}^{n-1} \text{Schröder } (i) \times \text{Schröder } (n - i - 1), & \text{se } n > 0 \end{cases}$$

Construa um programa para executar a função para sucessivos valores de n e que permita determinar experimentalmente a complexidade deste algoritmo. Efectue a análise empírica da complexidade construindo uma tabela com o número de multiplicações efectuadas para diferentes valores de n . Qual é a ordem de complexidade da função recursiva?

Uma forma de resolver problemas recursivos de maneira a evitar o cálculo repetido de valores, consiste em calcular os valores de baixo para cima, ou seja, de Schröder (0) para Schröder (n) e utilizar um *array* para manter os valores entretanto calculados. Este método designa-se por programação dinâmica e reduz o tempo de cálculo à custa da utilização de mais memória para armazenar valores intermédios.

- Usando a técnica de programação dinâmica, implemente uma função repetitiva alternativa e efectue a análise empírica da sua complexidade. Qual é a ordem de complexidade da função repetitiva?

Se analisarmos a solução anterior verificamos que para cada produto $S(i) \times S(n-i-1)$ existe outro produto simétrico que é exatamente o mesmo, pelo que podemos calcular apenas metade dos produtos repetidos e duplicar o seu valor de forma eficiente, usando o deslocamento (*shift*) para a esquerda. Assim, é possível fazer uma implementação com aproximadamente metade das multiplicações. Tendo isto em conta desenvolva outra função repetitiva otimizada e efectue a análise empírica da sua complexidade. Qual é a ordem de complexidade desta implementação otimizada da função repetitiva?

- Faça a análise formal (no verso da folha) das três implementações da função e confirme as ordens de complexidade obtidas experimentalmente.

N	Recursivo (N)	Nº de Multiplicações	Dinâmico (N)	Nº de Multiplicações	Eficiente (N)	Nº de Multiplicações
0						
1						
2						
3						
4						
5						
6						
7						
8						
9						
10						
11						
12						

O(N)		
------	--	--

AULA 10 - ÁRVORE BINÁRIA DE PESQUISA

O tipo de dados abstrato ABP é constituído pelo ficheiro de interface `abp.h` e pelo ficheiro de implementação `abp.c` e implementa a manipulação de árvores binárias de pesquisa que armazenam números inteiros. Ele tem capacidade de múltipla instanciação e usa um controlo centralizado de erros.

- Comece por testar convenientemente toda funcionalidade do tipo de dados, usando para esse efeito o programa `testabp.c` e a *makefile* `mkabp` e os ficheiros de árvores disponibilizados: `arv0.txt`, `arv1.txt`, `arv2.txt` e `arv3.txt`.

Acrescente ao tipo de dados abstrato a seguinte funcionalidade:

- Uma função para verificar se a árvore está ou não vazia. Devolve 1 em caso afirmativo e 0 em caso contrário. A função deve ter o seguinte protótipo:

```
int ABPEmpty (PtABPNode proot);
```

- Uma função para obter o elemento armazenado na árvore, dado um ponteiro para o seu nó. A função deve ter o seguinte protótipo:

```
int ABPElement (PtABPNode pnode);
```

- Uma função repetitiva para obter um ponteiro para o nó do maior elemento armazenado na árvore. A função deve ter o seguinte protótipo:

```
PtABPNode ABPMaxRep (PtABPNode proot);
```

- Uma função repetitiva para determinar a soma dos elementos armazenados na árvore, fazendo uma travessia por níveis recorrendo a uma fila. A função deve ter o seguinte protótipo:

```
int ABPTotalSum (PtABPNode pnode);
```

- Uma função repetitiva para determinar o número de elementos múltiplos de um dado valor armazenados na árvore, fazendo uma travessia em profundidade recorrendo a uma pilha. A função deve ter o seguinte protótipo:

```
unsigned int ABPMultCount (PtABPNode proot, int pvalue);
```

- Uma função recursiva para determinar o número de elementos ímpares armazenados na árvore. A função deve ter o seguinte protótipo:

```
unsigned int ABPOddCount (PtABPNode proot);
```

- Uma função recursiva para determinar a soma dos elementos pares armazenados na árvore. A função deve ter o seguinte protótipo:

```
int ABPEvenSum (PtABPNode proot);
```

- Uma função repetitiva para determinar a soma dos elementos armazenados na árvore, com número de ordem ímpar. Ou seja, a soma do primeiro, terceiro, quinto, sétimo, etc. menores números inteiros armazenados na árvore, fazendo uma travessia em profundidade em ordem recorrendo a uma pilha. A função deve ter o seguinte protótipo:

```
int ABPOddOrderSum (PtABPNode proot);
```

O programa **simabp.c** permite testar estas operações. O programa constrói uma árvore binária de pesquisa através da inserção (opção **i**) e da remoção (opção **r**) de valores introduzidos pelo teclado, fazendo a visualização hierárquica da árvore na horizontal após cada operação bem sucedida. Quando termina a manipulação de elementos da árvore (opção **t**), apresenta no monitor o número de nós e a altura da árvore. Depois, o programa apresenta ainda o menor e o maior elementos da árvore, a soma de todos os elementos da árvore, quantos elementos da árvore são múltiplos de 7, quantos elementos da árvore são ímpares, a soma dos elementos pares da árvore, lista em ordem os elementos da árvore e finalmente apresenta a soma dos elementos da árvore com número de ordem ímpar.

- Comece por simular o programa não inserindo qualquer valor (opção **t**) e verifique se os seus algoritmos funcionam para uma árvore vazia. Depois simule o programa inserindo e removendo os seguintes elementos (**i 50 - i 60 - i 30 - i 25 - i 55 - i 80 - i 90 - i 5 - i 38 - i 35 - i 40 - r 38 - i 45 - i 15 - i 20 - i 10 - i 75 - i 85 - i 65 - i 70 - r 90 - r 40 - r 45 - t**).

- Depois desta simulação apresente a árvore na vertical e responda às seguintes questões:

- Quantos elementos tem a árvore e qual é a sua altura?
- Quais são o menor e o maior elementos da árvore?
- Qual é a soma dos elementos da árvore?
- Quantos elementos da árvore são múltiplos de 7?
- Quantos elementos da árvore são ímpares?
- Qual é a soma dos elementos pares da árvore?
- Apresente a listagem em ordem dos elementos
- Quais são os elementos com número de ordem ímpar da árvore?
- Qual é a sua soma?

AULA 11 - ÁRVORE DE ADELSON-VELSKII LANDIS

O tipo de dados abstrato AVL é constituído pelo ficheiro de interface `avl.h` e pelo ficheiro de implementação `avl.c` e implementa a manipulação de árvores de Adelson-Velskii Landis que armazenam números inteiros. Ele tem capacidade de múltipla instanciação e usa um controlo centralizado de erros.

- Comece por testar convenientemente toda funcionalidade do tipo de dados, usando para esse efeito o programa `testavl.c` e a *makefile* `mkavl` e os mesmos ficheiros de árvores disponibilizados para a árvore ABP (`arv0.txt`, `arv1.txt`, `arv2.txt` e `arv3.txt`).
- Como ficam as árvores em comparação com as da aula anterior?

Também é fornecido o programa `simavl.c` que simula a inserção e a remoção de elementos na árvore, fazendo a sua visualização hierárquica na horizontal após cada operação.

- Simule o programa inserindo e removendo os seguintes elementos (`i50-i60-i30-i25-i55-i80-i90-i5-i38-i35-i40-r38-i45-i15-i20-i10-i75-i85-i65-i70-r90-r40-r45-t`).
- Depois desta simulação apresente a árvore na vertical e responda às seguintes questões:

- Quantos elementos tem a árvore e qual é a sua altura?
- Quais são o menor e o maior elementos da árvore?

AULA 12 - FILAS COM PRIORIDADE

Comece por ler o Capítulo 9 – Filas com prioridade, mais concretamente o item 9.4 – Implementação com amontoado (páginas 439-443), para se familiarizar com a implementação de uma fila com prioridade baseada num amontoado binário (*binary heap*) e os respectivos algoritmos.

O tipo de dados abstrato PQUEUE_HEAP é constituído pelo ficheiro de interface pqueue.h e pelo ficheiro de implementação pqueue.c e implementa a manipulação de filas com prioridade baseadas em amontoados binários, orientadas aos máximos e que armazenam números inteiros. Ele tem capacidade de múltipla instânciação e as operações devolvem um código de erro relativo à execução da operação.

- Comece por testar convenientemente toda funcionalidade do tipo de dados usando os programas `testpqueue.c`, `simpqueue.c` e a makefile `mkpqueue`.

Para simular o algoritmo do caminho mais curto de Dijkstra usa-se habitualmente uma fila com prioridade implementada com um amontoado binário e organizada com prioridade orientada aos mínimos, que armazene elementos estruturados do tipo VERTEX, sendo a prioridade dos elementos estabelecida pelo campo Cost.

```

/* definição de um elemento da fila com prioridade */
typedef struct dijkstra
{
    unsigned int Vertex;      /* vértice */
    int Cost;                /* custo do caminho até ao vértice */
} VERTEX;

```

- Crie uma fila com prioridade para elementos estruturados do tipo VERTEX, cuja funcionalidade é descrita no ficheiro de interface `pqueue_dijkstra.h` e usando o esqueleto do ficheiro de implementação `pqueue_dijkstra.c`. Implemente as seguintes operações:

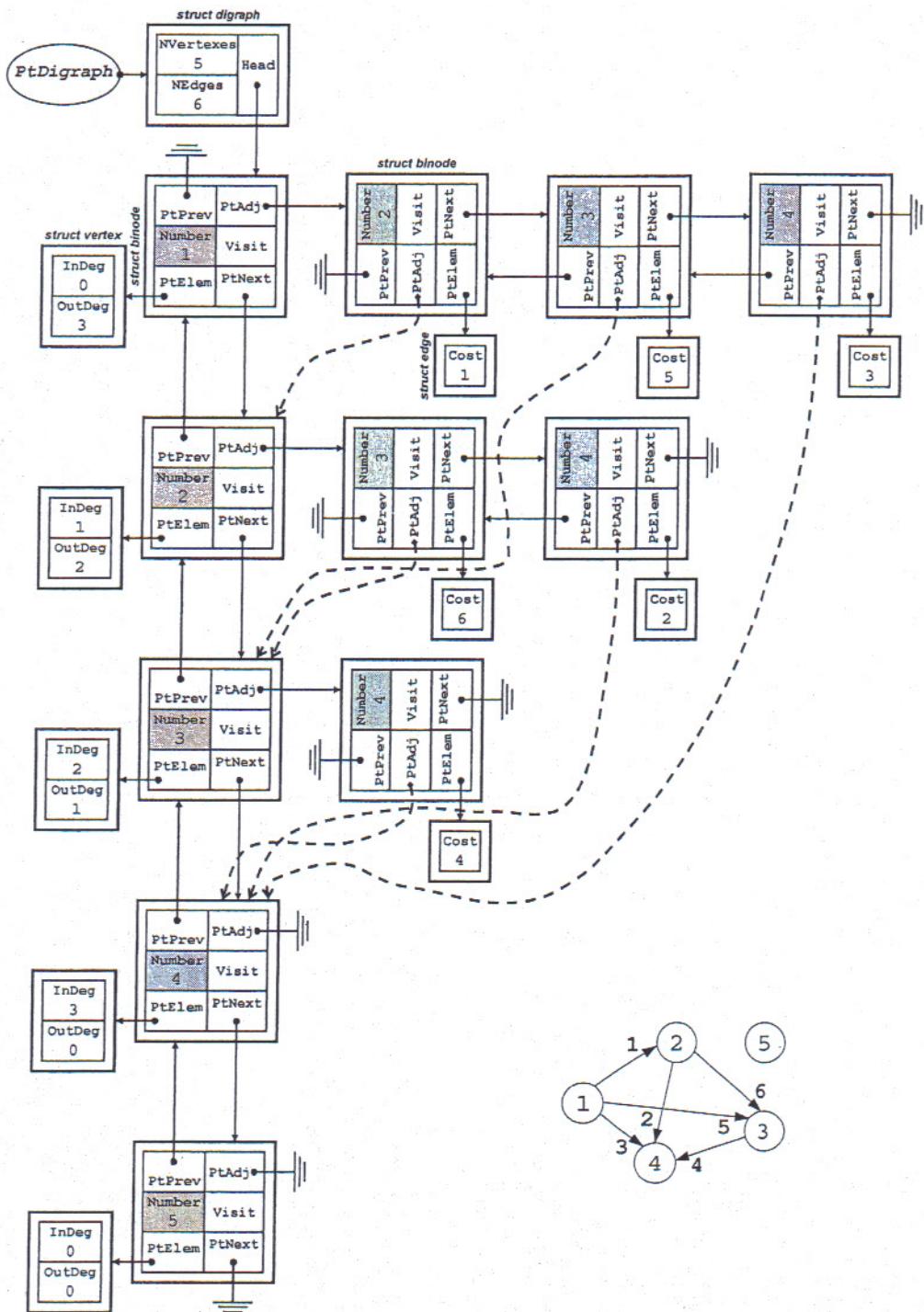
- Criação de uma fila com prioridade `PQueueCreate`;
- Destrução de uma fila com prioridade `PQueueDestroy`;
- Inserção de um elemento `PQueueInsert`;
- Remoção do elemento com menor chave `PQueueDeleteMin`;
- Promoção do elemento na fila com prioridade `PQueueDecrease`;
- Determinar se a fila com prioridade está ou não vazia `PQueueIsEmpty`.

Para testar a funcionalidade desta fila com prioridade pode usar o programa `teste1.c`, cuja execução é apresentada no ficheiro `teste1.txt`. Pode também usar o programa `teste2.c`, que simula o algoritmo de Dijkstra para o dígrafo apresentado na página 497 (sendo o custo infinito representado pelo valor 100), e cuja execução é apresentada no ficheiro `teste2.txt`.

AULAS 13 E 14 - DÍGRAFOS

Comece por ler o Capítulo 10 – Grafos (ver 10.3 – Implementação do Grafo, 10.4 – Caracterização do Grafo e 10.5 – Dígrafo/Grafo dinâmico, páginas 456-470). Estude o funcionamento da implementação do dígrafo/grafos dinâmicos e analise a sua implementação, para se familiarizar com os algoritmos.

O tipo de dados abstrato DIGRAPH_DYNAMIC é constituído pelo ficheiro de interface `digraph.h` e pelo ficheiro de implementação `digraph.c` e implementa a manipulação de dígrafos/grafos dinâmicos, usando listas biligadas genéricas para manter as listas dos vértices e das arestas ordenadas por ordem crescente, tal como se mostra na figura seguinte.



Para testar o tipo de dados – as operações básicas sobre dígrafos e as operações propostas para as aulas e projeto final – é fornecido o programa de simulação gráfica `simdigraph.c` e a `makefile` `mkdigraph`. Comece por testar convenientemente toda a sua funcionalidade básica.

Depois, acrescente-lhe a seguinte funcionalidade:

- Verificar de que tipo é um vértice. A função deve ter o seguinte protótipo:

```
int VertexType (PtDigraph pdig, unsigned int pv);
```

A função deve devolver: NO_DIGRAPH (se o dígrafo não existir); DIGRAPH_EMPTY (se o dígrafo estiver vazio); NO_VERTEX (se o vértice não existir); SINK se ele for um vértice sumidouro; SOURCE se ele for um vértice fonte; DISC se ele for um vértice desconexo; ou OK se for um vértice normal.

Teste a função para o `digrafo6.txt`, que é constituído por seis vértices e sete arestas, sendo que o vértice 5 é um vértice fonte, o vértice 3 é um vértice sumidouro e o vértice 6 é um vértice desconexo.

O tipo de dados providencia a função interna `DijkstraPQueue` que implementa o algoritmo de Dijkstra usando uma fila com prioridade baseada num amontoado binário. Esta função está simplificada, porque assume que as sequências foram previamente validadas pelas funções invocadoras. Acrescente ao tipo de dados as seguintes funções, que devem usar o algoritmo de Dijkstra:

- Determinar os caminhos mais curtos a partir de um dado vértice. Para esse efeito deve implementar uma função para executar de forma segura o algoritmo de Dijkstra. A função serve como *front-end* da função interna `DijkstraPQueue` e tem de validar as condições de execução da mesma. Ela deve ter o seguinte protótipo:

```
int Dijkstra (PtDigraph pdig, unsigned int pv, unsigned int pvpred[],  
              int pvcost[]);
```

A função deve devolver: NO_DIGRAPH (se o dígrafo não existir); DIGRAPH_EMPTY (se o dígrafo estiver vazio); NULL_PTR se algum dos ponteiros para as sequências for NULL; ou NO_VERTEX (se o vértice não existir). A função devolve a lista de vértices predecessores na sequência `pvpred` e os custos dos caminhos na sequência `pvcost`. A dimensão destas sequências é igual ao número de vértices do dígrafo.

- Determinar os vértices alcançáveis a partir de um dado vértice. Esta função deve usar a função interna `DijkstraPQueue` e depois devolver apenas a lista dos vértices alcançáveis. A função deve ter o seguinte protótipo:

```
int Reach (PtDigraph pdig, unsigned int pv, unsigned int pvlist[]);
```

A função deve devolver: NO_DIGRAPH (se o dígrafo não existir); DIGRAPH_EMPTY (se o dígrafo estiver vazio); NULL_PTR se o ponteiro para a sequência for NULL; NO_VERTEX (se o vértice não existir); ou NO_MEM se não existir memória para criar as sequências necessárias para invocar o algoritmo de Dijkstra. A função devolve os vértices alcançáveis na sequência `pvlist`, sendo que a posição 0 da sequência indica o número de vértices alcançáveis. A dimensão desta sequência é igual ao número de vértices do dígrafo.

Comece por determinar manualmente os vértices alcançáveis e os caminhos mais curtos para todos os vértices do `digrafo6.txt`. Depois, teste estas duas funções e compare os resultados.

TRABALHO FINAL SOBRE DÍGRAFOS

Adicionar ao tipo de dados abstrato DIGRAPH_DYNAMIC funções que permitam efectuar as seguintes operações:

- Verificar se um dado dígrafo G é **regular**. A função deve ter o seguinte protótipo:

```
int DigraphRegular (PtDigraph pdig, unsigned int *preg);
```

Um dígrafo diz-se regular se todos os seus vértices possuem o mesmo número de arcos incidentes (*indegree*) e também o mesmo número de arcos emergentes (*outdegree*). A função atribui a *preg* o valor 1, se o dígrafo for regular, e o valor 0, caso contrário. E devolve os seguintes valores de retorno: OK, NO_DIGRAPH (se o dígrafo não existir), DIGRAPH_EMPTY (se o dígrafo estiver vazio) ou NULL_PTR (se o ponteiro *preg* for NULL).

- Construir o dígrafo complementar \bar{G} de um dado dígrafo G. O dígrafo complementar \bar{G} é um dígrafo com os mesmos vértices, mas com as arestas que não existem no dígrafo G. A função deve ter o seguinte protótipo:

```
PtDigraph DigraphComplement (PtDigraph pdig);
```

A função começa por criar um dígrafo nulo e de seguida insere os vértices. Depois, insere as arestas que não existem no dígrafo e, finalmente, devolve a referência do dígrafo criado ou NULL, no caso de inexistência de memória.

- Verificar se um dado dígrafo G é **fortemente conexo**, isto é, se existe um caminho entre qualquer par de vértices do dígrafo G. A função deve ter o seguinte protótipo:

```
int DigraphStronglyConnected (PtDigraph pdig, unsigned int *pstrong);
```

A função atribui a *pstrong* o valor 1, se o dígrafo for fortemente conexo, e o valor 0, caso contrário. E devolve os seguintes valores de retorno: OK, NO_DIGRAPH, DIGRAPH_EMPTY, NO_MEM (se não existir memória para criar as sequências necessárias para invocar o algoritmo de Dijkstra) ou NULL_PTR (se o ponteiro *pstrong* for NULL).

- Implementar o **fecho transitivo** de um dado dígrafo G. Uma aresta (v_i, v_j) é inserida no dígrafo, se e só se, v_j é alcançável a partir de v_i e essa aresta ainda não existe no dígrafo. Considere que as novas arestas a inserir no dígrafo têm custo unitário. A função deve ter o seguinte protótipo:

```
int DigraphTransitiveClosure (PtDigraph pdig);
```

A função devolve os seguintes valores de retorno: OK, NO_DIGRAPH, DIGRAPH_EMPTY ou NO_MEM.

Atenção:

Apesar de habitualmente considerarmos que os vértices se encontram sequencialmente numerados, com início em 1, deve implementar os algoritmos de maneira o mais versátil possível. Ou seja, deve sempre varrer e processar a lista de vértices do dígrafo.

Para construir o fecho transitivo e verificar se um dígrafo é fortemente conexo deverá obrigatoriamente utilizar a função que determina os vértices alcançáveis (função **Reach**) que foi proposta no guião das aulas práticas.

Também deve respeitar os protótipos das funções propostos para poder simular toda a funcionalidade com o programa **simdigraph.c**.