Algoritmos e Complexidade – 8 de Novembro de 2017 – Duração: 90 min

1. Considere a seguinte variante da função `partition` usada no algoritmo de quick sort, que particiona o array em três zonas, em que na do meio todos os elementos são iguais (ao pivot).

```
void partition (int v[], int N,
                int *ep, int *eq){
  int i, p, q, t;
  // PRE: N > 0
  i=p=q=0;
  // ...
  while (i<N-1) {
      // ...
      if (v[i] < v[N-1]) {
         t = v[i]; v[i] = v[q];
         v[q] = v[p]; v[p] = t;
         p++; q++;}
      else if (v[i] == v[N-1]) {
         t = v[i]; v[i] = v[q];
         v[q] = t; q++;
      }
      i++;
  }
  t=v[i]; v[i] = v[q];
  v[q] = t; q++;
  // POS: p < q
  *ep = p; *eq = q;
}
```

   (a) Complete a anotação da função de forma a provar que no final, a zona do meio tem pelo menos um elemento (i.e. que `p < q`).

   Apresente as condições de verificação correspondentes à prova da correcção **parcial** do código entre `//PRE:` e `//POS:`, **omitindo as que dizem respeito à preservação do invariante**.

   (b) Apresente uma definição da função `qsort` que tira partido do resultado desta variante da função `partition`.

   Mostre que essa função tem um melhor caso que é linear no tamanho do array argumento. Assuma para isso que a função `partition` tem um comportamento linear no tamanho do array argumento.

2. Considere a seguinte função que *remove* de um array de inteiros os elementos consecutivamente repetidos.

```
int noRep (int v[], int N) {
  int i;
  i=0;
  while (i<N-1)
     if (v[i] == v[i+1]) {
         shift(v+i, N-i);
         N--;
     } else i++;
  return N;
}
```

```
void shift (int v[], int N) {
  int i;
  for (i=0;i<N-1;i++)
     v[i]=v[i+1];
}
```

   (a) Apresente um variante do ciclo da função `nRep` e mostre que ele decresce em cada iteração.

   (b) Identifique o melhor e pior casos da execução da função `noRep` em termos do número de escritas no array.

   Para o pior caso identificado calcule qual o número de escritas no array em função do tamanho do array argumento.

3. Considere a definição recursiva da função `minSort` ao lado.
Admitindo que num array aleatório com N elementos a probabilidade de o elemento na posição 0 ser o menor de todos é $1/N$, apresente uma recorrência que traduza o número médio de trocas (`swap`) efectuado por esta função.
Apresente uma solução dessa recorrência (não precisa de simplificar os *somatórios* envolvidos nessa solução).

```
void minSort (int v[], int N) {
  int m;
  if (N>0) {
     m = minInd (v,N);
     if (m!=0) swap (v,0,m);
     minSort (v+1,N-1);
  }
}
```

Algoritmos e Complexidade – 6 de Janeiro de 2018 – Duração: 90 min

1. A função apresentada ao lado testa se um array tem números repetidos em tempo quadrático no tamanho do array argumento.
   Usando uma estrutura de dados auxiliar à sua escolha, apresente uma definição alternativa desta função que execute em tempo linear no tamanho do array argumento.
   Mostre, mesmo que informalmente, que a complexidade da função apresentada é de facto linear.

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

2. Considere o grafo não orientado e ligado representado na matriz à direita. Considere que as células não preenchidas correspondem a arestas que não existem.
   Apresente a evolução da função `int primMST (Grafo g, int pesos[], int ant[])` que implementa o algoritmo de Prim para o cálculo de uma árvore geradora de custo mínimo.
   Nomeadamente, para cada iteração do algoritmo, apresente o estado do vector de pesos e dos antecessores.

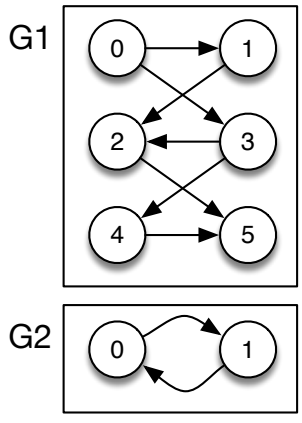
	0	1	2	3	4	5	6	7
0		3	5					
1	3			6	7			
2	5			1		2		
3		6	1		8	3		
4		7		8			6	7
5			2	3			2	
6					6	2		5
7					7		5	

3. Considere a definição ao lado para representar (as arestas de) grafos em listas de adjacência.
   O **grau de entrada** (*indegree*) de um vértice define-se como o número de arestas do grafo que têm esse vértice como destino.
   Um grafo diz-se uma **árvore** sse (1) existe um vértice chamado raiz cujo grau de entrada é zero e (2) para todos os outros vértices existe um único caminho da raiz para esse vértice.

```
#define NV ...
typedef struct aresta {
  int destino, peso;
  struct aresta *prox;
} *Grafo [NV];
```

   Defina uma função `int isTree (Grafo g)` que testa se um dado grafo orientado é um árvore. A função deverá retornar `-1` se o grafo `não` for uma árvore. No outro caso deverá retornar a raiz da árvore.

4. Considere que se usam arrays de inteiros para representar transformações dos vértices de um grafo.
   Por exemplo o array $f =$ | 0 | 1 | 2 | 3 | 4 | 5 |
   |---|---|---|---|---|---|
   | 1 | 0 | 1 | 0 | 1 | 0 |
   transforma os vértices pares em 1 e os ímpares em 0.
   Uma transformação de vértices f diz-se um **homomorfismo** de um grafo g num grafo h sse para cada aresta (a, b) do grafo g existe uma aresta $(f(a), f(b))$ no grafo h. Por exemplo, a transformação F acima é um homomorfismo entre os grafos G1 e G2 ao lado.
   O problema de, dados dois grafos, determinar se existe um homomorfismo entre eles é NP-completo.



   - Descreva um algoritmo não deterministico polinomial que resolva este problema.
   - Usando a parte deterministica do algorritmo anterior, defina uma função `int homomorphic (Grafo g, Grafo h, int N)` que, usando *força bruta* determina se existe um homomorfismo entre os grafos argumento (assuma que ambos os grafos têm `N` vértices).