Algoritmos e Complexidade (2° Ano – LEI)
Métodos de Programação II (2° Ano LESI)

Exames

1 Ano Lectivo de 2007/2008

1.1 Frequência

Parte I

1. Faça a análise assimptótica do tempo de execução da função `minpairs`.

```
typedef struct {int p; int s;} Pair;

void minpairs(Pair a[], char v[], int n) {
   int i,j;
   for (i=0; i<n, i++) v[i]=1;
   for (i=n-1; i>=0; i--)
      for (j=i-1; j>=0; j--)
         if (a[i].s <= a[j].s) v[j]=0;
}
```

2. Determine as condições de verificação necessárias para provar a correcção parcial do seguinte programa (anotado em comentários) usado para calcular o máximo divisor comum entre dois números positivos.

```
// (a = a0 > 0) /\ (b = b0 > 0)
while (a != b)
    // mdc(a0,b0) = mdc(a,b)
    if (a > b) a = a - b;
    else b = b - a;
// a = mdc(a0,b0)
```

3. Considere a seguinte função recursiva:

```
void exemplo(int a[], int n) {
   int x = n/2;
   if (n>0) {
      exemplo(a,x);
      processa(a,n);
      exemplo(a+x,n-x);
   }
}
```

Sabendo que $T_{processa}(n) = \Theta(n)$, escreva uma recorrência que descreva o comportamento temporal da função `exemplo` e indique, justificando, a solução dessa recorrência.

4. Pretende-se implementar um dicionário de sinónimos usando uma *tabela de hash*. Esta tabela pode ser definida da seguinte forma, onde a cada palavra está associada uma lista ligada com os seus sinónimos.

```c
typedef struct s {
   char *sin;
   struct s *next;
} Sin;

typedef struct p {
   char *pal;
   Sin *sins;
   struct p *next;
} Pal;

#define TAM ...
typedef Pal *Dic[TAM];

int hash(char *pal);
```

Implemente a seguinte função que, dada uma palavra, imprime todos os seus sinónimos:

```c
void sinonimos(Dic d, char *pal);
```

5. Considere o seguinte tipo de dados que permite representar grafos orientados e não pesados usando listas de adjacências.

```c
typedef struct arco {
  int dest;
  struct arco *seg;
} Arco;

#define TAM ... // número de vértices do grafo
typedef Arco *Grafo[TAM];
```

Implemente a seguinte função que determina se há caminho entre dois vértices:

```c
int haCaminho(Grafo g, int o, int d);
```

6. Relembre o conceito de *min-heap*. Implemente uma função que converte uma *min-heap* representada num array para uma *min-heap* representada como uma árvore do tipo indicado.

```c
#define TAM ...

typedef int Heap[TAM];

typedef struct nodo {
   int val;
   struct no *esq, *dir;
} Nodo, *Tree;
```

Parte II

1. O diâmetro de um grafo pesado define-se como a distância (minima) entre os seus nodos mais afastados. Dado um grafo pesado representado numa matriz de adjacências implemente uma função que calcule o seu diâmetro.
2. Justifique a seguinte afirmação:

   > *Na inserção de um elemento numa AVL, no caso em que o balanceamento se faz com uma rotação simples para a direita, o factor de balanço dos dois nodos envolvidos fica zero.*

3. Defina uma função (o mais eficiente possível) que receba a árvore resultado de uma travessia de um grafo (vector dos pais) e faça uma travessia pre-order dessa árvore (imprimindo os índices correspondente). Considere que o grafo tem `N` vértices e que a raíz da árvore tem pai `-1`.

1.2 Exame da Época de Recurso

Parte I

1. Determine as condições de verificação necessárias para provar a correcção parcial da seguinte função anotada.

```
int bubble(int a[], int n) {
   int i, k;
   // n > 0
   i = n-1; k = 0;
   // n > 0 && i = n-1 && k = 0
   while (i > 0) {// i >= 0 && k < n
      if (a[i] < a[i-1]) {swap(a,i,i-1); k++}
      i--;
   }
   // k < n
   return k;
}
```

2. Analise a complexidade da seguinte função identificando o melhor e pior casos.

```
void bsort(int a[], int n) {
   while (bubble(a,n));
}
```

3. É possível armazenar um grafo orientado não pesado usando dois arrays: o primeiro armazena as arestas ordenadas por vértice de origem, indicando para cada uma delas qual o vértice destino; o segundo indica em que posição do primeiro array começam as arestas com origem em cada vértice; a última posição do array de vértices é usada para indicar qual é a primeira posição livre do array de arestas. Assumindo que o grafo tem exactamente `V` vértices e no máximo `A` arestas, este tipo de dados (`Grafo1`) pode ser definido da seguinte forma:

```
typedef struct {
   int arestas[A];
   int vertices[V+1];} Grafo1;
```

Implemente uma função `void converte (Grafo1 *g, Grafo2 h)` que permita converter um grafo do tipo `Grafo1` num grafo implementado com listas de adjacência (`Grafo2`).

```
typedef struct arco {
   int destino;
   struct arco *next;} Arco;
typedef Arco *Grafo2 [V];
```

4. Pretende-se usar uma tabela de *hash* para armazenar o conjunto das matrículas dos carros com acesso aos parques reservados da universidade. Assumindo que o tratamento das colisões é feito usando *chaining*, esta tabela pode ser implementada da seguinte forma:

```
#define SIZE 1009
typedef struct no {
   char matricula[6];
   struct no *next;
} No;
typedef No *Tabela[SIZE];
```

- Implemente uma função de *hash* razoável para este problema.

  `int hash(char matricula[6]);`

- Implemente a função de inserção de uma matrícula na tabela, garantindo que não se armazenam matrículas repetidas.

```
int insert(Tabela t, char matricula[6]);
```

5. Considere o seguinte tipo para implementar uma árvore AVL de inteiros.

```
typedef struct no {
   int info;
   int bal;
   struct no *esq, *dir;
} No;
typedef No *Arvore;
```

   Implemente a função `Arvore rr(Arvore arv)` que faz uma rotação simples para a direita numa determinada sub-árvore. Não se esqueça de actualizar o factor de balanço nos nós envolvidos na rotação.

6. Analise a complexidade da seguinte função que calcula os factores de balanço de uma árvore. Assuma que árvore está balanceada e que a função `altura` executa em tempo linear no tamanho da árvore de entrada.

```
void bals(Arvore a) {
   if (!a) return;
   a->bal = altura(a->dir) - altura(a->esq);
   bals(a->esq);
   bals(a->dir);
}
```

Parte II

1. O diâmetro de um grafo pesado define-se como a distância (minima) entre os seus nodos mais afastados. Dado um grafo pesado representado com **listas de adjacências**, compare em termos de complexidade as seguintes alternativas para efectuar esse cálculo: (1) Uso repetido do algoritmo de Dijkstra ou (2) conversão para matrizes de adjacência e uso do algoritmo de Warshall.

2. Num grafo não orientado, um *click* é um subconjunto de vértices em que quaisquer dois elementos estão ligados por uma aresta.

   (a) Defina uma função que dado um grafo e um conjunto de vértices determina se esse conjunto é um *click*.

   (b) Um *click* diz-se maximal se qualquer conjunto que o contenha não for um *click*. Defina uma função que, dado um grafo e um vértice calcula um *click* maximal que contenha o vértice dado.

3. Demonstre que a função `bubble` apresentada na questão 1 da primeira parte, satisfaz as seguintes propriedades: (1) o número de trocas efectuadas é menor do que `n` e (2) coloca na primeira posição do vector o menor dos elementos do array.

1.3 Exame da Época Especial

Parte I

1. Determine as condições de verificação necessárias para provar a correcção parcial da seguinte função anotada.

```
int factorial(int n) {
   int i, k;
   // n >= 0
   k = 1; i=0;
   // n > 0 && k = 1
   while (i < n) {// i <= n 0 && k = i!
      i++; k*=i;
   }
   // k = n!
   return k;
}
```

2. Analise a complexidade da seguinte função que usa a função da alínea anterior:

```
void factoriais (int a[], int n) {
   int i;
   for (i=0; (i<n); i++) a[i] = factorial (i);
}
```

3. É possível armazenar um grafo orientado não pesado usando dois arrays: o primeiro armazena as arestas ordenadas por vértice de origem, indicando para cada uma delas qual o vértice destino; o segundo indica em que posição do primeiro array começam as arestas com origem em cada vértice; a última posição do array de vértices é usada para indicar qual é a primeira posição livre do array de arestas. Assumindo que o grafo tem exactamente `V` vértices e no máximo `A` arestas, este tipo de dados (`Grafo1`) pode ser definido da seguinte forma:

```
typedef struct {
   int arestas[A];
   int vertices[V+1];} Grafo1;
```

   Defina uma função que, para esta representação, calcule o grau de entrada de um dado vértice (i.e., o número de antecessores desse vértice).

4. Analise a complexidade da seguinte função que calcula quantos elementos de um vector de inteiros são menores do que um dado inteiro:

```
int menores (int a [], int n, int x) {
  int u, l, m;
  l = 0; u=n-1;
  while (l<u) {
     m = (l+u) / 2;
     if (a[m] < x) l = m+1;
     else u=m;
  }
  return l
}
```

5. Para o ciclo da função da alínea anterior, determine um variante que lhe permita mostrar que a função termina. Indique além disso as condições que devem ser verificadas para completar essa demonstração.

6. Relembre o algoritmo de inserção balanceada (AVL). Identifique pelo menos um dos casos em que o desbalanceamento não pode ser resolvido por uma rotação simples, dizendo, para esse caso qual a estratégia para voltar a balancear a árvore.

Parte II

1. Mostre, apresentando um exemplo (pre-condição, pós-condição e programa), que nem todos os programas parcialmente correctos estão correctos.

2. O problema de coloração de grafos é conhecido como sendo um problema difícil. O que pretendemos aqui é validar soluções para esse problema.

   Considere o tipo `Grafo` para armazenar as arestas de um grafo.

```
typedef struct arco {
   int destino;
   struct arco *next;} Arco;
typedef Arco *Grafo [V];
```

   Uma coloração de um grafo com `V` vértices pode ser guardada num vector com `V` inteiros, cada um deles identificando a cor do vértice respectivo. Defina então uma função `int corOK (Grafo g, int cor[V])` que determina se a coloração `cor` é apropriada, i.e., que não existem vértices adjacentes com a mesma cor. A função deve retornar `0` se tal **não** for o caso ou então deve retornar o número de cores diferentes usadas.

3. Analise a complexidade da função apresentada na alínea anterior, em função do número de vértices e arestas do grafo. Indique se necessário o melhor e o pior casos.

4. Com base no resultado da alínea anterior, justifique a afirmação *"o problema da coloração de grafos é um problema NP"*.

2 Ano Lectivo de 2006/2007

2.1 Exame da 1ª Chamada

Parte I

1. Relembre a noção de *min-heap* binária.

   (a) Descreva, por palavras suas, as propriedades que esta estrutura de dados deve ter.

   (b) Desenhe as três *heaps* de inteiros que resultam de: **(1)** inserir consecutivamente os números 10, 20, 30, 22, 25, 35, 40, 24; **(2)** inserir o número 15 na *heap* resultante, e finalmente **(3)** remover o mínimo da *heap*.

2. Considere o seguinte programa:

```
while (e < n) {
  y = y * x;
  e = e+1;
}
```

   Prove que este ciclo termina e que preserva o seguinte invariante: $y = x^e \ \wedge \ e \leq n$

3. Com base na seguinte implementação de uma árvore de procura, defina a função `treeToArray` que preenche um array com os elementos da árvore de procura, ordenados por ordem crescente. Note que esta função tem ainda um parâmetro (de entrada e saída) que indica qual a primeira posição livre do array.

```
typedef struct Node {
  int elem;
  struct Node *esq, *dir;
} *Tree;

void treeToArray (Tree t, int A[], int *i);
```

4. Considere a função `maxSort` que faz a ordenação de um array de tamanho `n`.

```
void maxSort(int A[], int n) {
  int i,j;

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

   Sabendo que `swap(A,j,i)` troca os valores contidos nas posições `i` e `j` do array `A` em tempo constante, efectue a análise assimptótica do comportamento da função `maxSort`, no pior caso de execução.

5. Indique, justificando, a solução da seguinte recorrência:

$$T(n) = \begin{cases} c & \text{se} \ \ n \leq 1 \\ 4\,T(n/2) & \text{se} \ \ n > 1 \end{cases}$$

6. Assuma que a função `ssSP` faz o cálculo dos caminhos mais curtos com origem num dado vértice, segundo o algoritmo de Dijkstra, num grafo pesado.

```
void ssSP (Grafo g, int v, int pai[], int dist[]);
```

Note que, dados o grafo `g` e o vértice de origem `v`, esta função guarda a árvore dos caminhos mais curtos em `pai`, e as respectivas distâncias em `dist`.

Usando a função `ssSP`, defina a função `camMaisCurto` que dado o grafo `g`, o vértice origem `v` e o vértice destino `d`, imprime no `stdout` a sequência de vértices do caminho mais curto (da origem para o destino), um vértice por linha. Esta função devolve 1 se não houver caminho, e 0 se existir.

```
int camMaisCurto (Grafo g, int v, int d);
```

Parte II

Considere o problema de determinar o k-ésimo menor elemento de um vector (possivelmente desordenado) com N elementos. Trata-se de uma generalização do problema de determinar o menor elemento de um vector. Uma forma de resolver este problema consiste em ordenar o vector (com uma complexidade limitada inferiormente por $N. \log N$) e seleccionar o elemento do índice **k**.

Uma forma alternativa, mais eficiente, consiste em adaptar o algoritmo de ordenação *quicksort*, cuja peça fundamental é a operação de partição. Essa operação de partição, linear no tamanho do vector, apesar de não ordenar o vector, retorna um índice `p` que satisfaz a seguinte propriedade: *todos os elementos do vector antes dessa posição são menores (ou iguais) e todos os seguintes são maiores (ou iguais).* Daí que corresponda de facto ao p-ésimo elemento mais pequeno. Assim sendo, pode-se usar esta operação de partição iteradamente, com um tamanho de vector cada vez mais pequeno, até que seja determinado o valor pretendido.

Assuma que existe definida a função `particao` cujo tipo é

```
int particao (int v[], int a, int b);
```

linear em `(b-a)` que retorna um índice entre `a` e `b` segundo a especificação informal acima.

1. Apresente uma definição da função que calcula o k-ésimo menor elemento de um array com N elementos não ordenado.

   ```
   int kEsimo (int a[], int N, int k){ ... }
   ```

2. Apresente um argumento de terminação do programa que apresentou. Se não resolveu a alínea anterior, baseie o seu raciocínio na descrição informal feita acima.

3. Apresente uma relação de recorrência que traduza a complexidade do algoritmo descrito. Apresente ainda a solução dessa relação de recorrência.

4. Considere agora os problemas (de decisão) Π_1 e Π_2

   Π_1**:** determinar se o k-ésimo **menor** elemento de um vector é **maior** que um dado número.

   Π_2**:** determinar se o k-ésimo **maior** elemento de um vector é **menor ou igual** a um dado número.

   Escreva (em **C**) uma redução polinomial de Π_2 a Π_1.

   Sabendo que Π_1 é da classe **P**, que pode concluir sobre Π_2? Justifique.

2.2 Exame da 2ª Chamada

Parte I

1. Considere as seguintes declarações de tipos:

   ```
   typedef struct node {
     int elem;
     struct node *esq, *dir;
   } *Tree;

   int maximo(Tree t, int *x);
   ```

   Escreva em C a função `maximo` que recebe uma *árvore binária de procura* e que devolve na variável `x` o elemento de maior valor da árvore. A função deve retornar 1 se não existir máximo, e 0 caso contrário.

2. Considere o seguinte programa:

```
i = 0;
s = 0;
while (i < n) {
  s = s + a;
  i = i + 1;
}
```

Prove que este programa está <u>parcialmente</u> correcto em relação à especificação Pré-condição: $n \geq 0$
Pós-condição: $s = n * a$

3. Considere a seguinte função:

```
void example(int A[], int N) {
  int i;
  Node *p;

  for (i = 1; i <= N; i++)
      p = insert(A,N,i,p);
  convert(p,A,1);
}
```

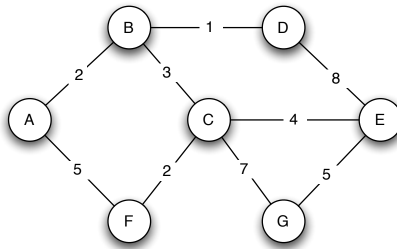
Sabendo que $T_{insert}(N) = \mathcal{O}(\log N)$ e que $T_{convert}(N) = \mathcal{O}(N^2)$, faça a análise assimptótica do tempo de execução da função `example` no pior caso de execução.

4. Declare o tipo de dados `Tabela` para implementar *Tabelas de Hash* (com chave de tipo `int` e valor do tipo `char*`) com tratamento de colisões por encadeamento (*chaining*), e defina a função `int insere(Tabela tab, int k, char *val);` de inserção na tabela de hash, que deve devolver 1 caso a chave já exista na tabela. Assuma que já está definida uma função de hash (indique apenas o seu tipo).

5. Indique, justificando, a solução da seguinte recorrência:

$$T(n) = \begin{cases} c_1 & \text{se} \quad n \leq 1 \\ c_2 + 5\ T(n/3) & \text{se} \quad n > 1 \end{cases}$$

6. Considere o seguinte grafo:



Relembre o algoritmo de Prim para o cálculo da *Árvore Geradora de Custo Mínimo*. Ilustre as diversas etapas de execução deste algoritmo, sobre o grafo da figura. Identifique os conjuntos dos nós da árvore, dos nós da orla e dos arcos candidatos, ao longo da execução do algoritmo. Considere que o nó A é o primeiro nó a ser incluído na árvore.

Parte II

1. Relembre o algoritmo de Dijkstra para cálculo do caminho mais curto de um dado vértice para todos os outros, num grafo pesado e orientado (assuma que os pesos são números inteiros positivos). Assuma que a função `dijkstraSP`, definida como

```
dijkstraSP (Grafo *g, int v, int st[], int pesos[]);
```

devolve nos vectores `st` e `pesos` a árvore construída pelo algoritmo e o peso do caminho mais curto para cada um dos vértices. Assuma ainda que o número de vértices é dado pela constante `NV`.

(a) Considere que após se invocar a função `dijkstraSP (Grafo *g, int v, int st, int pesos)`, se pretende determinar qual a aresta de maior peso usada. Escreva uma função que determine qual o peso dessa aresta.

(b) Usando a função `dijkstraSP` escreva uma função que preencha uma matriz quadrada (NV×NV) com os pesos dos caminhos mais curtos entre todos os vértices (assuma que um peso de -1 corresponde a não existir caminho). Por outras palavras, defina a função de cálculo do fecho transitivo de um grafo pesado usando a função de caminho mais curto.

(c) Assumindo que a função `dijkstraSP` executa em tempo $\Theta(\mathtt{NV}\log(\mathtt{NV}))$, determine um limite superior para o tempo de execução da função referida na alínea anterior.

2. O problema (de optimização) da coloração de grafos consiste em determinar o menor número de cores a atribuir aos vértices de um grafo, de forma a que dois vértices adjacentes tenham cores diferentes. O problema de decisão associado à coloração de grafos é da classe **NP**. Mostre que tal é verdadeiro, definindo um algoritmo não determinístico polinomial que testa se um dado grafo pode ser colorido com **k** cores.

2.3 Exame da Época de Recurso

Parte I

Esta parte do exame representa 12 valores da cotação total. Cada uma das 6 alíneas está cotada em 2 valores.
A obtenção de uma classificação abaixo de 8 valores nesta parte implica a reprovação no exame.

1. Declare o tipo de dados `HashTable` para implementar *Tabelas de Hash* (com chave de tipo `char*` e valor do tipo `float`) com tratamento de colisões por encadeamento, e defina a funçãode remoção na tabela de hash.

```
void remove(HashTable tab, char *key);
```

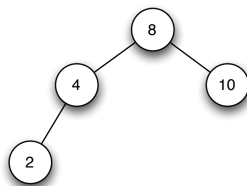
Assuma que já está definida uma função de hash (indique apenas o seu tipo).

2. Considere o seguinte programa:

```
i = 0;
p = 1;
while (i < n) {
  p = x * p;
  i = i + 1;
}
```

Prove que este programa está *totalmente* correcto em relação à especificação Pré-condição: $n \geq 0$
Pós-condição: $p = x^n$

3. Considere a seguinte árvore AVL:



Descreva o conceito de árvore AVL, e ilustre o resultado de se inserir sucessivamente na árvore da figura os nós com chave 1 e 3, preservando sempre o invariante destas árvores.

4. Considere a seguinte função:

```
void exemplo(int A[], int N) {
  int i;

  processa(A,N);
  i = 0;
  while (i < N) {
```

```
    altera(A,i);
    i = i+1;
  }
  apresenta(A,N);
}
```

Sabendo que $T_{processa}(N) = \mathcal{O}(N.\log N)$, $T_{altera}(i) = \Theta(i)$ e $T_{apresenta}(N) = \Theta(N)$ faça a análise assimptótica do tempo de execução da função `exemplo`.

5. Considere o seguinte função para o problema das torres de Hanoi:

```
int Hanoi(int N, int esq, int dir, int meio) {
  if (N > 0) {
     Hanoi(N-1, esq, meio, dir);
     printf("Mover disco de %d para %d\n", esq, dir);
     Hanoi(N-1, meio, dir, esq);
  }
}
```

Escreva uma recorrência que descreva o comportamento temporal da função `Hanoi` e indique, justificando, a solução dessa recorrência.

6. Considere a seguinte declaração de tipos para implementar um grafo orientado e pesado:

```
typedef struct arco {
  int dest;
  int peso;
  struct arco *seg;
} Arco;

typedef Arco* Grafo[MAX];

int minPeso(Grafo g, int n);
```

Implemente a função `minPeso` que devolve o peso do arco de menor peso do grafo;

Parte II

1. Relembre o algoritmo de Prim para cálculo de uma árvore geradora de custo mínimo. Este algoritmo pode ser optimizado modificando a forma como os vértices estão armazenados na orla. As alternativas possíveis são:

   - lista ordenada por ordem de chegada;
   - lista ordenada por ordem crescente do peso que os liga à árvore até aí construída;
   - numa min-heap cujo critério de ordenação é o peso que os liga à árvore até aí construída.

   (a) Para cada uma das alternativas apresentadas, diga qual é o esforço total de manutenção da orla, assumindo que cada nodo do grafo será acrescentado e retirado da orla exactamente uma vez.

   (b) Para que a terceira alternativa seja viável é necessário que dado um vértice, seja possível determinar em tempo constante qual a sua posição na orla (para poder actualizar o seu peso). Diga como tal pode ser feito e apresente a definição de uma função que, dada a orla (min-heap representada num array), um vértice e um peso, actualize a localização desse vértice na orla (acrescentando-o caso ele não exista).

   (c) Defina uma função que calcula a profundidade da árvore geradora de custo mínimo resultante do algoritmo de Prim. Declare apenas a assinatura da função que cálcula a àrvore geradora.

2. Um isomorfismo entre dois grafos G e H não pesados é uma bijecção f entre os seus vértices que verifica a seguinte propriedade: quaisquer dois vértices v e u são adjacentes em G se e só se $f(u)$ e $f(v)$ são adjacentes em H. Justifique que o problema de decisão de verificar se dois grafos são isomorfos se trata de um problema NP, isto é, mostre que existe um algoritmo não-determinístico, limitado polinomialmente, que o resolve.

2.4 Exame da Época Especial

Parte I

1. Considere a seguinte declaração de tipos para implementar uma *tabela de Hash* com resolução de colisões por *open addressing*

```
#define HASHSIZE ...
#define EMPTY    ...
#define DELETED  ...
#define USED ...
```

```
typedef struct entry {
  int    status;
  int    key;
  float  value;
} Entry;

typedef Entry Table[HASHSIZE];
```

Defina a função de inserção na tabela de hash (de uma chave `k` e valor `v`) pelo método de pesquisa linear (*linear probing*).

```
int insert(Table tab, int k, float v);
```

Esta função deve devolver 1 caso a chave já exista na tabela, -1 se a tabela já estiver cheia e 0 se tudo correr bem. Assuma que já está definida uma função de hash (indique apenas o seu tipo).

2. Considere o seguinte programa:

```
x = 0;
i = 0;
while (i < n) {
  x = x + (2 * i) + 1;
  i = i + 1;
}
```

Prove que este programa está *totalmente* correcto em relação à especificação
Pré-condição: $n \geq 0$
Pós-condição: $x = n^2$

3. Considere a seguinte relação de recorrência que traduz a complexidade de um determinado algoritmo, em função do tamanho N do *input*.

$$T(N) = \begin{cases} c & \text{se} \quad N < 2 \\ \\ N + 2.T(N/3) & \text{se} \quad N > 1 \end{cases}$$

Apresente, justificando, uma expressão que descreva o comportamento assimptótico desse algoritmo.

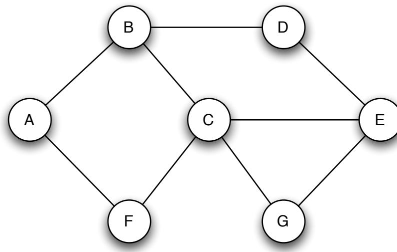
4. Considere a função `minimo` que devolve na variável `x` o elemento de menor valor de uma árvore binária de pesquisa.

```
typedef struct node {
  int elem;
  struct node *esq, *dir;
} *Tree;
```

```
int minimo(Tree t, int *x) {
  if (t) {
    while (t->esq)
       t = t->esq;
    *x = t->elem;
    return 0;
  }
  else return 1;   // erro
}
```

Efectue a análise assimptotica do tempo de execução da função `minimo` em função do número de elementos da árvore: para isso, identifique o melhor e o pior caso, justificando para cada um deles uma medida do tempo de execução.

5. Considere o seguinte grafo não pesado:



Desenhe uma árvore de antecessores passível de ser produzida pelo algoritmo de travessia em largura. Indique também a ordem porque os vértices são alcançados. Considere que o vértice E é utilizado como ponto de partida da travessia.

6. Assuma que tem definida uma função que implementa o algoritmo da questão anterior (onde `NV` é o número de vértices do grafo).

```
void travDF (Grafo g, int pai [NV])
```

O array preenchido por esta função pode ser usado para determinar o comprimento do caminho mais curto (i.e., com menos arestas) entre dois vértices do grafo. Defina a função que calcula esse comprimento, devolvendo -1 caso não exista caminho.

```
int distancia (Grafo g, int origem, int destino)
```

Parte II

Suponha definidos o tipo `VERTICE` e uma função `Vert2Int` **injectiva** de conversão deste tipo de dados num número inteiro (sendo injectiva, quando aplicada a vértices diferentes dá resultados diferentes).

Pretende-se armazenar um grafo pesado com `N` destes vértices usando tabelas de hash (open addressing) para armazenar os vértices e listas de adjacência para representar as arestas.

1. Defina os tipos de dados necessários, bem como a função de *hash* que, dado um vértice retorna o índice que lhe corresponde, usando a função `Vert2Int` como auxiliar. Note que esta função pode dar resultado fora da gama de definição da tabela. Note ainda que como não temos disponível qualquer função de igualdade entre vértices, esse teste tem de ser feito também usando a função `Vert2Int`.

2. Apresente definições de funções que permitam:
   (a) Calcular o peso de uma aresta (se existir).
   (b) Calcular o número de antecessores de um vértice.
   (c) Calcular o número de vértices que não são acessíveis de um dado vértice.

3 Ano Lectivo de 2005/2006

3.1 Exame da 1ª Chamada

Questão 1 Considere o seguinte código (C)

```
r = 0;
while (x>= y){
    x=x-y; r=r+1;
   }
```

que calcula (em `r`) a divisão inteira de `x` por `y`.

1. Prove a correcção parcial deste algoritmo (sem provar que o ciclo termina) segundo a seguinte especificação:

   pré-condição: $x = x_0 \land y = y_0$

   pós-condição: $r * y_0 + x = x_0 \land x < y_0$

2. Modifique a especificação acima, fortalecendo a pré-condição, de forma a garantir a terminação do ciclo. Mostre então que, face a essa nova especificação, o algoritmo está correcto.

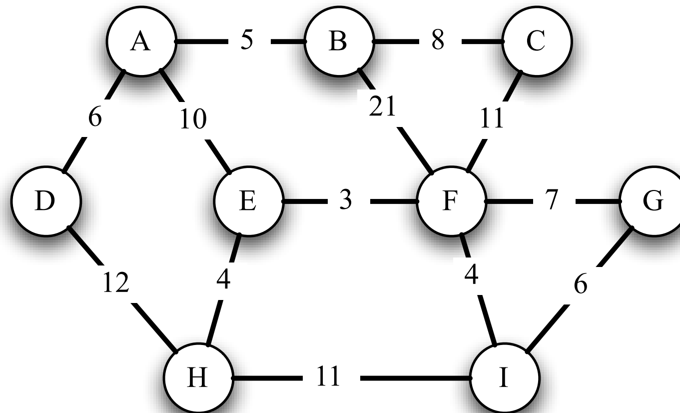
Questão 2 Numa árvore AVL inicialmente vazia, foram inseridas as chaves 50, 20, 80, 15, 35 e 90, por esta ordem.

1. Desenhe a árvore obtida, não se esquecendo de, para cada nodo explicitar o factor de balanço correspondente.
2. Determine uma sequência de chaves a inserir no resultado acima para que a árvore resultante tenha na raíz a chave 35. Explicite as rotações efectuadas.

Questão 3 Uma forma de determinar se um grafo orientado é cíclico é tentar calcular uma ordenação topológica desse grafo. Se tal for possível o grafo é acíclico.

1. Escreva uma definição em C de uma função (representado em listas de adjacência) que testa se um grafo é ou não cíclico.
2. Descreva o tempo de execução da função apresentada em função do número de vértices e arestas do grafo.
3. Descreva ainda o tempo de execução desse mesmo algoritmo assumindo que o grafo estava representado em matrizes de adjacência.

Questão 4 Considere o seguinte grafo (ligado, pesado e não orientado).



1. Mostre qual o comportamento do algoritmo de Dijkstra de cálculo do caminho mais curto de um vértice a todos os outros quando aplicado a este grafo a partir do vértice A. Nomeadamente, diga em cada iteração, qual o valor da fronteira e da árvore geradora construída até à altura.
2. Mostre, usando por exemplo a árvore obtida na alínea anterior, que a árvore geradora construída pelo algoritmo de Dijkstra não é necessariamente uma árvore geradora de custo mínimo.
3. Relembre que um dos resultados do algoritmo de Dijkstra é a árvore geradora dos caminhos mais curtos, representada por um vector de *ascendentes*. Defina uma função que, dado um destes vectores de ascendentes e um vértice imprima, caso exista, o caminho até esse vértice.

3.2 Exame da 2ª Chamada

Questão 5 Considere a seguinte implementação de uma tabela de hash com resolução de colisões utilizando *open addressing* com pesquisa linear:

```
#define EMPTY    -1
#define NULL     -2

typedef struct {
int key;
```

```
int value;
} HashEntry;

typedef struct {
HashEntry table[HASH_SIZE];
} HashTable;

int hash(int key);
```

1. Justifique a necessidade de reservar dois valores (`NULL` e `EMPTY`) no espaço das chaves de acesso à tabela de hash para assinalar entradas não existentes. Implemente a seguinte função de pesquisa de um valor na tabela de hash. A função deverá retornar 1 e preencher a referência `entry` no caso de a pesquisa ser bem sucedida, e retornar 0 no caso contrário.

   ```
   int retrieveTable(HashTable *table, int key, HashEntry *entry);
   ```

2. Implemente a seguinte função que conta o número de entradas na tabela de hash com valores da chave na gama $low \leq key \leq high$.

   ```
   int countRange(HashTable *table, int low, int high);
   ```

   Compare, utilizando notação assimptótica, a eficiência no melhor e no pior caso com que é possível resolver o problema anterior para os seguintes casos:

   - Tabela de hash utilizando *open addressing*.
   - Tabela de hash utilizando *chaining*.

Questão 6 Considere o problema de calcular a potência (inteira) de um número. A especificação de tal procedimento pode ser dada pelos seguintes predicados.

pré-condição: $x = x_0 \wedge y = y_0 \geq 0$

pós-condição: $r = {x_0}^{y_0}$

Considere o seguinte fragmento de C que pretende satisfazer a especificação acima.

```
r = 1;
while (y!=0){
   if (y%2 != 0) r = r*x;
   x = x * x
   y = y/2; // divisão inteira
   }
```

1. Use o seguinte invariante na prova da correcção. $I \doteq x^y * r = {x_0}^{y_0}$

   $$\frac{\{\,P \wedge c\,\}\; S\; \{\,Q\,\} \quad (P \wedge \neg c) \Rightarrow Q}{\{\,P\,\}\ \texttt{if}\ \ c\; S\; \{\,Q\,\}}\ \text{(if)} \qquad \frac{\{\,I \wedge c \wedge 0 \leq V = v_0\,\}\; S\; \{\,I \wedge 0 \leq V < v_0\,\}}{\{\,I\,\}\ \texttt{while}\ \ c\; S\; \{\,I \wedge \neg c\,\}}\ \text{(while-1)}$$

2. Identifique o melhor e pior casos deste fragmento de código. Escreva relações de recorrência que exprimam o tempo de execução deste fragmento de código em função do valor do expoente (`y`) no melhor e pior caso. Assuma, como simplificação, que as operações de teste, multiplicação e divisão executam em tempo constante. Resolva as recorrências.

Questão 7

```
typedef struct sAVL {
   void *valor;
   char balanco; // 'E' (esquerdo) 'D' (Direito) ou 'B' (Balanceado)
   struct sAVL *esq, *dir;
   } Nodo, *AVL;
```

1. Defina uma função em C que, dada uma árvore binária, calcula, para cada nodo da árvore o seu factor de balanço, isto é, a diferença entre os pesos das suas duas sub-árvores.

   ```
   void calcula_Bals (AVL a);
   ```

2. Descreva o tempo de execução dessa função em função do número de elementos da árvore. Relembre o tempo de execução da inserção em AVLs (em que os factores de balanço são guardados para cada nodo da árvore) e explique o porquê deste gasto adicional de memória (a guardar uma quantidade que pode ser calculada sempre que necessária)

Questão 8 Dado um grafo orientado, o algoritmo de Tarjan calcula as componentes fortemente ligadas do grafo em tempo linear no número de arestas e de vértices ($\Theta(E+V)$). A versão apresentada nas aulas teóricas, constrói essa informação num vector com uma componente por vértice cujo valor é o número de ordem da componente a que pertence esse vértice.

```
#define N ...
typedef ... VERTICE;
typedef struct sAdjList {
    int destino;
    struct sAdjList *next;
    } *AdjList;
typedef struct sGrafo {
   int nvert;
   VERTICE Vertice [N];
   AdjList Arestas [N];
   } *Grafo;
void tarjan (Grafo g, int componentes[]);
```

1. Defina uma função que, dado um grafo orientado, calcula um novo grafo, com um vértice por cada componente ligada do grafo original e com uma aresta a ligar dois vértices sempre que exista um caminho no grafo original dos vértices de uma das componentes para a outra.

   ```
   Grafo graphComp (Grafo g);
   ```

2. Descreva o tempo de execução desse algoritmo em função do número de arestas e vértices do grafo original.

3.3 Exame da Época de Recurso

Questão 9 Considere o seguinte algoritmo de multiplicação de dois números que toma partido de que a divisão e multiplicação por 2 são operações muito eficientes (trata-se, em representação binária, de *shift*s).

```
resultado = 0;
while (y>0){
   if (y % 2 != 0) { y=y-1resultado = resultado + x;
   x = x*2;
   y = y/2;
   }
```

1. Escreva predicados que descrevam a pré e pós condição deste algoritmo. Não se esqueça que a pré-condição deve garantir que o algoritmo termina. Apresente ainda um invariante de ciclo que lhe permita provar a correcção do algoritmo em causa.

2. Assumindo que as operações de adição, multiplicação por 2 e divisão por 2 executam em tempo constante `c`, calcule o tempo de execução deste algoritmo como uma função do valor de `y`. Sugestão: Exprima o número de iterações do ciclo como uma relação de recorrência sobre o valor de `y`.

Questão 10 Recorde o algoritmo Merge Sort que estudou nas aulas de MP2.

1. Implemente uma versão do algoritmo Merge Sort que, em vez de utilizar sub-listas de tamanho aproximadamente $N/2$, utiliza sub-listas de tamanho aproximadamente $N/3$. Sugestão: comece por implementar a função de combinação de sub-listas ordenadas

```
void merge(int A[],int esq,int div1,int div2,int dir);
```

em que `esq` e `dir` representam os limites da zona válida do *array* A e os sub-*arrays* a serem combinados estão nas regiões $[esq \ldots div_1]$, $[div_1 + 1 \ldots div_2]$ e $[div_2 + 1 \ldots dir]$.

2. Escreva uma equação de recorrência que descreva o tempo de execução $T(N)$ do algoritmo anterior e desenhe a árvore correspondente para um caso em que a
3. É possível exprimir o comportamento assimptótico de $T(N)$ na notação Θ? Justifique a sua resposta e, em caso afirmativo, apresente esse resultado.

Questão 11 Considere o problema de determinar se um dado grafo orientado é ou não simétrico.

```
#define N ...
typedef int MAdj [N][N]
typedef struct sAdj {
    int destino; int peso;
    struct sAdj *next;
    } LAdj [N];
```

1. Defina em C uma função para resolver este problema. Apresente duas soluções, uma para cada uma das representações estudadas.
2. Para cada uma das funções apresentadas atrás, descreva o tempo de execução $T(N, E)$ e a memória extra usada $M(N, E)$ em função do número de vértices (N) e de arestas (E).

Questão 12 Dado um grafo não orientado, não pesado e ligado, o **diâmetro do grafo** define-se como o mais longo dos caminhos mais curtos do grafo (relembre que o comprimento de um caminho é o número de arcos desse caminho).

1. Usando os algoritmos estudados nas aulas, apresente uma algoritmo para calcular o diâmetro de um grafo.
2. Baseado ainda nas análises de complexidade das várias componentes usadas na solução apresentada na alínea anterior, caracterize a complexidade dessa solução.

3.4 Exame da Época Especial

Questão 13 Sejam $G_1 = \langle V_1, E_1 \rangle$ e $G_2 = \langle V_2, E_2 \rangle$ grafos. Uma função $f : V_1 \to V_2$ é um *homomorfismo de grafos* quando preserva adjacências (i.e. para quaisquer $x, y \in V_1$, temos que $(x, y) \in E_1 \Rightarrow (f(x), f(y)) \in E_2$). Se f possuir uma inversa que seja um homorfismo de grafos, então dizemos que f é um *isomorfismo de grafos*. Determinar se existe um isomorfismo entre dois grafos é um exemplo conhecido de um problema que se sabe pertencer a **NP** e que se julga não pertencer a **P**.

1. Defina tipos de dados adequados para representar grafos e homomorfismos de grafos.
2. Defina uma função polinomial que, dados dois grafos e um homomorfismo de grafos, determine se esse homomorfismo é um isomorfismo de grafos.
3. Justifique porque é que a função polinomial que apresentou na alínea anterior não contradiz a afirmação apresentada atrás onde se afirma que o problema de determinar se dois grafos são isomorfos não deverá pertencer à classe **P**.

Questão 14 Considere o seguinte código (C)

```
resto = x;
while (resto>= y){
    resto=resto-y;
   }
```

que calcula o resto da divisão inteira de `x` por `y`.

1. Prove a correcção parcial deste algoritmo, segundo a seguinte especificação:

   pré-condição: $x = x_0 \geq 0 \wedge y = y_0$

pós-condição: $(\exists_{r \geq 0} . r * y_0 + resto = x_0) \wedge 0 \leq resto < y_0$

2. Apesar de parcialmente correcto (com respeito à especificação acima), o algoritmo apresentado não está correcto (porque não conseguimos mostrar a sua terminação). Modifique a especificação acima, fortalecendo a pré-condição, de forma a garantir tal propriedade. Mostre então que, face a essa nova especificação, o algoritmo está correcto.

Questão 15 Considere agora o seguinte excerto de código que pretende obter o mesmo objectivo do da questão anterior – o cálculo do resto da divisão inteira de `x` por `y`.

```
my=y; resto = x;
while (my < resto) my = my * 2;
while (my>= y){
        if (resto>= my) resto -= my;
my = my / 2;
}
```

Compare a complexidade (assimptótica) das das versões, em função do número de bits usados para representar `x` e `y`.

4 Ano Lectivo de 2004/2005

4.1 Exame da 1ª Chamada

Questão 16 Considere a função `min` definida da seguinte forma

```
int min(int A[], int a, int b)
{
 int i, m;

 m = A[a];
 i = a+1;
 while (i <= b)
   if (m > A[i]) {
        m = A[i];
        i = i+1;
   }
   else i = i+1;

 return m;
}
```

1. Mostre que a função `min(A,a,b)` calcula correctamente o menor inteiro presente entre as posições `a` e `b` do array `A`. Isto é, prove que

$$\{ \ a \leq b \ \} \ \mathbf{F} \ \{ \ \forall a \leq p \leq b. \ \ m \leq A[p] \ \}$$

   F representa todo o corpo da função `min` excepto a instrução `return m`.

2. Mostre que `min(A,a,b)` determina o elemento mínimo do array `A` entre as posições `a` e `b`, em tempo linear em ordem a `b-a`.

Questão 17 Considere a seguinte declaração de dados para implementar tabelas de hash com tratamento de colisões por encadeamento.

```
typedef Entry   *HashTable[HASHSIZE]

typedef struct entry {
  int           key;
  char          *value;
  struct entry  *next;
} Entry;
```

1. Explique o funcionamento de uma tabela deste tipo, referindo-se nomeadamente; à estratégia de resolução de colisões de chaves; dimensionamento da tabela (como deverá ser o valor de `HASHSIZE`); complexidade em termos de tempos de execução das operações básicas de inserção, remoção e consulta, para uma tabela bem dimensionada para o volume da informação a guardar.

2. Considere agora que para fazer a listagem da informação ordenada por chave, se pretende indexar a informação da tabela utilizando árvores binárias de procura. Para isso, definiram-se os seguintes tipos de dados.

```
typedef struct node {
  int          key;
  Entry        *info;;
  struct node  *left;
  struct node  *right;
} Node;

typedef Node   *Tree
```

   Defina uma função que recebe uma tabela de hash e gera uma árvore binária de procura que indexa a informação dada nessa tabela. A função terá o seguinte protótipo:

```
Tree geraArv (HashTable table);
```

3. Compare, em termos gerais, a complexidade da operação básica de consulta na tabela de hash, com uma hipotética operação de consulta que utilize a árvore binária de procura. Refira-se ao melhor e ao pior caso de execução.

Questão 18 As duas travessias de grafos estudadas nas aulas diferem apenas na estrutura de dados usada para armazenar os próximos vértices a serem visitados. Enquanto que na travessia *Depth-first* se usa uma *stack*, na *Breadth-first* usa-se uma *queue*. Considere agora uma outra alternativa de travessia em que a estrutura de dados utilizada para armazenar estes nodos é uma lista, ordenada por ordem crescente do peso da aresta que liga esse nodo aos já visitados.

Apresente uma codificação desta variante, assumindo que o grafo se encontra armazenado como listas de adjacência.

Questão 19 Considere que para um determinado problema se desenvolveram dois programas: P1 e P2:

P1 é um programa que determina uma solução para o problema. O seu tempo de execução (em função da dimensão N do seu input) é dado pela função T_1.

P2 é um programa que, dada uma solução proposta, verifica se se trata de uma solução válida. O seu tempo de execução (em função da dimensão N do seu input) é dado pela função T_2.

$$T_1(N) = \begin{cases} c_1 & \text{se} \quad N \leq 1 \\ 4T_1(N/3) & \text{se} \quad N > 1 \end{cases}$$
$$T_2(N) = \begin{cases} c_2 & \text{se} \quad N \leq 1 \\ 3T_2(N/3) & \text{se} \quad N > 1 \end{cases}$$

1. Para cada uma das recorrências T_1 e T_2, *adivinhe* uma solução e mostre informalmente a sua validade.

2. Em qual (ou quais) das classes de complexidade estudadas (P, NP e NP-completo) poderá incluir o problema em causa? Justifique a sua resposta.

4.2 Exame da Época de Recurso

Questão 20 Considere o problema de identificar todos os números *grandes* de um array de inteiros. Um número diz-se *grande* se for maior do que a soma dos elementos que o sucedem até ao final do array. Por exemplo, os números grandes do array **4, 30, 7, 12, 5, 2, 3** são os números 30, 12 e 3.

```
void  GRANDES (int A[], int teste[], int N)
{ int i, j, soma;

  for (i=0 ; i<N ; i++) {
      soma = 0;
      for (j=i+1 ; j<N ; j++)
          soma = soma + A[j];
      if (A[i]>soma) teste[i] = 1;
                else teste[i] = 0;
  }
}
```

1. Efectue a análise assimptótica do comportamento da função `GRANDES`.
2. A pós-condição deste procedimento pode ser descrita pelo seguinte predicado

$$\forall\, 0 \leq k < N \,.\, \left((\texttt{teste}[k] = 1) \;\Leftrightarrow\; A[k] > \sum_{p=k+1}^{N-1} A[p] \right)$$

   Escreva predicados que caracterizem o invariante de ambos os ciclos deste procedimento.

3. Para escrever uma versão linear deste mesmo procedimento, usou-se o seguinte invariante.

$$soma = \sum_{p=i+1}^{N-1} A[p] \quad \wedge \quad \forall\, i \leq k < N \,.\, \left((\texttt{teste}[k] = 1) \;\Leftrightarrow\; A[k] > \sum_{p=k+1}^{N-1} A[p] \right)$$

   Complete o código abaixo.

```
void grandes (int A[], int teste[], int N)
{ int i, soma;

  i = ........ ; soma = 0 ;
  while (........) {
      if (A[i]>soma) teste[i] = 1;
                else teste[i] = 0;
      ..........
      ..........
      }
  }
  return conta;
}
```

Questão 21 Suponha que se pretende construir uma árvore de procura balanceada a partir de uma árvore de procura. Uma estratégia consiste em começar por construir uma lista ordenada com os elementos da árvore original. É sobre esta lista que o processo de construção da árvore balanceada vai operar. Começa-se por partir a lista em duas, de igual comprimento. Cada uma destas será usada para construir as sub-árvores, enquanto que o elemento do meio será usado como raiz.

1. Comece por definir uma função que, dada uma árvore de procura, preenche um vector com os elementos dessa árvore ordenados por ordem crescente. Essa função deve ainda produzir o número de elementos da árvore.

```
typedef struct treeNode {
   int elem;
   struct treeNode *left, *right;
   } *Tree;

void treeToArray (Tree t, int v[], int *tamanho);
```

2. Use a função da alínea anterior para definir a função que constroi uma nova árvore de procura balanceada a partir de uma árvore de procura.

```
Tree balancea (Tree t);
```

3. Analise o tempo de execução da função `balancea` descrita na alínea anterior. No caso de não ter respondido à primeira alínea, suponha que a operação de criação da lista é feita em tempo linear.

Questão 22 Pretende-se representar a informação referente a uma rede de metropolitano de uma cidade. As estações serão identificadas por um nome e as várias linhas associadas a cores. Note que duas estações podem ser ligadas por mais do que uma linha, o que revela que a estrutura a considerar deverá ser um *multigrafo não orientado*.

1. Defina as estruturas de dados adequadas para armazenar a informação referente ao multigrafo descrito. Considere para o efeito que as cores são o vermelho; amarelo; azul e verde.
2. Defina funções simples de manipulação das estruturas definidas, como sejam:
   - Inicialização da estrutura;
   - Adicionar uma estação;
   - Adicionar uma ligação entre duas estações por uma dada linha;
3. Defina uma função que determine quais as linhas que passam por uma dada estação.
4. Defina uma função que, dadas duas estações e uma cor, verifique se existe ligação entre as estações pela linha associada à cor dada.

4.3 Exame da Época Especial

Questão 23 Considere o problema de armazenar os dados referentes a um dos jogos da Santa Casa da Misericórdia de Lisboa – EuroMilhões.

Cada aposta consiste em 7 números: cinco entre 1 e 50 e os outros dois entre 1 e 9. Por isso o número de apostas possíveis é dado por

$$50 \times 49 \times 48 \times 47 \times 46 \times 9 \times 8 \;=\; 18\,306\,086\,400$$

No entanto o número de apostas registadas em cada semana (em Portugal) é bastante mais baixo. Por exemplo no concurso 30/2005 o número de apostas recebidas foi 21 913 577 e destas, o número de combinações distintas foi de 18 764 150. A cada aposta está associado o número do bilhete (cada bilhete pode ter mais do que uma aposta).

Aquilo que se pretende aqui é analisar alternativas para armazenar a correspondência entre apostas e número do bilhete.

As operações que serão efectuadas são:

- Acrescentar a informação de um bilhete: número do bilhete e apostas efectuadas
- Dada uma aposta, determinar os números de bilhetes que contêm essa aposta.

1. Uma alternativa consiste em definir uma tabela de Hash em que as chaves são as apostas e a informação a guardar são os números dos bilhetes que contêm essa aposta. Sabendo que uma percentagem significativa dos apostadores usam datas para escolherem os seus números, os números mais frequentes são 1..31 (por causa dos dias do mês), e destes os números 1..12 (por causa dos meses). Apresente uma função de hashing que tenha em conta estas preferências, i.e., que dê menos significado aos números mais frequentes. Admita que a dimensão da tabela já foi fixada na constante `HASHSIZE`.
2. Uma outra alternativa consiste em armazenar as chaves num vector ordenado por ordem crescente da aposta. Para esta solução, diga qual a ordem de grandeza do tempo de execução de cada uma das operações pretendidas.
3. Uma terceira alternativa consiste em guardar a informação sob a forma de uma árvore de decisão: cada nodo corresponde à pergunta *qual o número escolhido?* e terá tantos descendentes quantas as repostas possíveis. Repita a análise feita na alínea anterior para esta solução.

4. Finalmente, e tendo em consideração o tempo de execução e a memória gasta por cada uma destas soluções, escolha a melhor solução.

Questão 24 Considere que se representam polinómios guardando apenas os seus coeficientes por ordem decrescente do grau. Assim, por exemplo, o polinómio $4 + 2\,x^3$ será guardado num vector cujas primeiras 3 posições têm os valores 2, 0, 0 e 4.

Considere a funções abaixo que calcula o valor de um polinómio num ponto.

```
double valor (double p[], int g, double x) {
    double r, px;

    r = 0; px=1;
    while (g >= 0) {
    r = r + p[g] * px;
    px *= x;
    g = g-1;
    }
    return (r);
}
```

1. Escreva predicados apropriados à pré e pós-condição desta função (de acordo com a descrição informal feita acima).
2. Apresente o invariante necessário à prova da correcção da função.
3. Determine o tempo de execução em função do grau do polinómio. Admita que a multiplicação de *doubles* se faz em tempo constante.
4. Relembre o problema de cálculo dos factores primos de um dado número, dado como exemplo de um problema NP-completo. Considere ainda que se define uma redução polinomial do problema de cálculo do valor de um polinómio num ponto a este problema de factorização. O que pode concluir sobre a classe de problemas a que pertence o cálculo do valor de um polinómio? Justifique.

Questão 25 Pretende-se representar a informação referente a uma rede de metropolitano de uma cidade. As estações serão identificadas por um nome e as várias linhas associadas a cores. Note que duas estações podem ser ligadas por mais do que uma linha, o que revela que a estrutura a considerar deverá ser um *multigrafo não orientado*.

1. Defina as estruturas de dados adequadas para armazenar a informação referente ao multigrafo descrito. Considere para o efeito que as cores são o vermelho; amarelo; azul e verde.
2. Defina funções simples de manipulação das estruturas definidas, como sejam:
    - Inicialização da estrutura;
    - Adicionar uma estação;
    - Adicionar uma ligação entre duas estações por uma dada linha;
3. Defina uma função que, dadas duas estações, calcule um caminho entre essas estações que usa apenas ligações da mesma cor.