Algoritmos e Complexidade
Exame de Recurso
Duração: 2h

25 de Janeiro de 2018

1. Considere a seguinte simplificação da função `bubble` que não altera o array argumento. Apresente um invariante e as condições de verificação necessárias para provar a correcção parcial da função.

```
int dummyBubble (int v[], int N){
  int r, i;
  // N == N0 > 0
  r = 0; i = 0;
  // ... I ...
  while (i<N-1) {
     // ... I ...
     if (v[i] > v[i+1]) r=i+1;
     i=i+1;
  }
  // forall (r<k<N0-1) v[k] <= v[k+1]
  return r;
}
```

2. Relembre o algoritmo de inserção balanceada em árvores de procura (AVL) e apresente a **evolução** de, numa destas árvores de inteiros inicialmente vazia, se inserirem (por esta ordem) os números

   3, 8, 25, 9, 24, 32, 37, 31

   Não se esqueça de apresentar, para cada nodo das árvores, o factor de balanço (<u>E</u>sq, <u>B</u>al ou <u>D</u>ir).

3. Considere as definições ao lado para implementar dicionários usando tabelas de hash com *open addressing* e *linear probing*. Para cada entrada da tabela armazena-se ainda o número de *probes* que foram feitos na inserção dessa entrada (`probeC == 0` significa que a chave foi inserida no índice correspondente ao seu `hash`).
   Defina a função `int update (HTable t, int key, int value)` que adiciona um novo par à tabela, e no caso de a chave existir, actualiza a informação correspondente.
   A função deverá preencher o campo `porbeC` e retornar `0` em caso de sucesso (tabela não cheia).

```
#define HSIZE 23
#define FREE -1

typedef struct entry {
  int probeC; // -1: free
  int key;
  int value;
} HTable [HSIZE];

int hash (int key, int size);
```

4. Admita que existe definida uma função `int dijkstraSP (Grafo g, int v, int pesos[], int pais[])` que calcula os caminhos mais curtos a partir de um dado vértice. Defina uma função `int aproxMeio (Grafo g, int o, int d)` que calcula, caso exista, um vértice intermédio no caminho mais curto entre `o` e `d` e que seja o mais próximo possível do ponto médio entre esses vértices.

   A função deverá retornar o vértice origem caso o caminho mais curto contenha apenas uma aresta.

   A função deverá retornar `-1` caso não exista caminho entre os vértices dados.

5. Considere a definição ao lado que calcula o k-ésimo menor elemento de um array, e que usa a função `partition` (que executa em tempo linear (N)) usada no algoritmo *quicksort*,

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

   (a) Apresente e resolva uma relação de recorrência que traduza o tempo de execução da função `kesimo` assumindo que a invocação `partition (N,v)` retorna sempre `N/2`.

   (b) Apresente uma relação de recorrência que traduza o **tempo médio** de execução da função `kesimo` assumindo que a invocação `partition (N,v)` retorna com igual probabilidade um valor entre `0` e `N-1`.

6. Dado um grafo G, um sub-conjunto X dos vértices diz-se dominante se e só se todos os vértices do grafo G que não pertencem a X são adjacentes a algum elemento de X.

   O problema (de decisão) de determinar se dado um grafo G e um inteiro k existe um conjunto dominante de G com k elementos é NP-completo.

   (a)
   - Descreva um algoritmo não deterministico polinomial que resolva este problema. Para isso, descreva a sintaxe das soluções geradas pelo oráculo e a função determinística `int test (...)` que valida essas soluções.
   - Usando a parte determinística do algoritmo anterior, defina uma função `int kDominant (Grafo g, int k)` que, usando *força bruta* determina se existe um conjunto dominante de G com k elementos.

   (b) Analise a complexidade da função `kDominant` da alínea anterior, bem como da função `test` de validação das soluções.