Algoritmos e Complexidade

Exame de Recurso

7 de Fevereiro de 2013

Parte I

1. Apresente as condições de verificação necessárias à prova da correcção parcial do seguinte programa anotado para calcular a representação binária de um número inteiro positivo.

```
// n == n0 >= 0
i = 0;
// n == n0 >= 0 && i == 0
while (n > 0) {
    // n0 = (n*2^i) + sum{k=0}{k=i-1} b[k] * (2^k) && n >= 0
    if (n%2 == 0) { b[i] = 0; n = n/2;}
    else { b[i] = 1; n = (n-1)/2;}
    i = i+1;
}
// n0 = sum{k=0}{k=i-1} b[k] * (2^k)
```

2. Considere a função que calcula a potência inteira de um número.
   Assumindo que as operaçõe elementares sobre inteiros (multiplicação, divisão e resto da divisão) de dois números executam em tempo linear no número de bits da representação binária dos números, faça a análise da complexidade (usando uma recorrência) desta função para o pior caso (não se esqueça de caracterizar esse pior caso), em função do número de bits usados na representação dos números inteiros.

```
int pot (int x, int n) {
    int r = 1;

    if (n>0) {
        r = pot (x*x,n/2);
        if (m%2 == 1) r = r * x;
    }
    return r;
}
```

3. Uma definição alternativa de árvores AVL consiste em guardar, para cada nodo da árvore, a altura da árvore que aí se inicia em vez de guardar o factor de balanço desse nodo.

   Considere então a seguinte definição de uma dessas árvores (de inteiros). Defina uma função `AVL rotateLeft (AVL a)` que faz uma rotação (simples) à esquerda na raíz de uma destas árvores.
   Assuma que a rotação é possível e não se esqueça de actualizar o campo `altura` dos nodos envolvidos.

```
typedef struct nodo{
      int valor;
      int altura;
      struct nodo *esq, *dir;
    } Node, *AVL;
```

4. Considere que se usa um grafo pesado e não orientado para representar uma rede de distribuição de água. Os vértices correspondem a bifurcações enquanto que os pesos das arestas correspondem à secção do tubo.

   Defina uma função `int ligacao (Grafo g, int v1, int v2, int seccao)` que, dado um grafo e dois vértices, determina se existe uma ligação entre esses dois vértices envolvendo apenas tubos com uma secção superior a um dado valor.

5. Dado um vector `v` de `N` números inteiros, a mediana do vector define-se como o elemento do vector em que existem no máximo `N/2` elementos (estritamente) menores do que ele, e existem no máximo `N/2` elementos (estritamente) maiores do que ele. Se o vector estiver ordenado, a mediana corresponde ao valor que está na posição `N/2`.

   Considere a seguinte definição de uma função que calcula a mediana de um vector.

   Assumindo que a função `quantos` executa em tempo linear no comprimento do vector de input, identifique o melhor e pior caso de execução da função `mediana`. Para cada um desses casos determine a complexidade assimptótica da função `mediana`.

```
int mediana (int v[], int N) {
    int i, m, M;
    for (i=0; i<N; i++) {
        quantos (v,N, v[i], &m, &M);
        if (m <= N/2) && (M <= N/2) break;
    }
    return v[i];
}
```

Parte II

1. Calcule a complexidade média da função apresentada na alínea anterior. Para isso, assuma que os valores do vector são perfeitamente aleatórios e, por isso, que a probabilidade de o elemento numa qualquer posição do vector ser a mediana é uniforme ($= \frac{1}{N}$).

2. Considere as definições ao lado para implementar tabelas de Hash dinâmicas com tratamento de colisões por open addressing e linear probing.
   Note a existência de uma função `int hash (Key, int)` que recebe como argumentos uma chave e o tamanho da tabela.
   Defina uma função `void remApagados (THash h)` que remove os elementos apagados de uma tabela, i.e., que efectua as operações necessárias para que deixem de existir células marcadas como apagadas.

```
#define Livre 0
#define Ocupado 1
#define Apagado 2

typedef struct key *Key;
struct celula {
    Key k;
    void *info;
    int estado; //Livre/Ocupado/Apagado
}
typedef shash {
   int tamanho, ocupados, apagados;
   struct celula *Tabela;
} *THash;
int hash (Key, int);
```

3. No contexto da alínea anterior considere que, na operação de remoção de um elemento da tabela, sempre que o número de chaves apagadas é igual ao número de chaves efectivas (`ocupados = apagados`), a função da alínea anterior é invocada.

   Assumindo que a função da alínea anterior executa em tempo linear no número de chaves efectivas e que a inserção e remoção *normal* (sem invocação da função `remApagados`) executam em tempo constante, mostre, usando um dos métodos estudados, que o custo amortizado da remoção é constante.