Algoritmos e Complexidade – Exame de Recurso

28 de Janeiro de 2016 – Duração: 2h30m

Parte A

1. Considere o seguinte programa (parcialmente) anotado.

```
// n == n0 > 1 && f == 2
while (n%f != 0)
   // I
   f = f+1;
// n0 % f == 0 && forall(1<k<f) n0%f != 0
```

   (a) Descreva por palavras a especificação apresentada.

   (b) Apresente um variante e um invariante que lhe permitam provar a **correcção total** do programa face à especificação apresentada.

   (c) Gere as condições de verificação correspondentes apenas à **correcção parcial** do programa.

2. Considere a definição ao lado que calcula quantas vezes ocorre o elemento mais frequente de um array.

   - Identifique o melhor e pior casos da execução desta função.
   - Para o pior caso identificado acima, determine o número de comparações efectuadas entre elementos do vector.

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

3. Considere a seguinte definição de um tipo para representar árvores AVL de inteiros (**balanceadas**). Considere ainda a definição da função `deepest` que determina o nodo mais profundo de uma árvore binária, retornando o nível em que ele se encontra.

   (a) Mostre que a função apresentada tem uma complexidade linear no número de elementos da árvore argumento.

   (b) Apresente uma definição alternativa, substancialmente mais eficiente, tirando partido da informação presente no factor de balanço de cada nodo. Diga qual a complexidade da função que definiu.

```
typedef struct avlnode {
  int value;
  int bal; // Left/Bal/Right
  struct avlnode *left, *right;
} *AVLTree;

int deepest (AVLTree *a) {
  AVLTree l, r;
  int hl, hr, h;
  if (*a == NULL) h = 0;
  else {
    l = a->left; r = a->right;
    hl = deepest (&l); hr = deepest (&r);
    if (hl > hr) {*a = l; h = hl+1;}
    else         {*a = r; h = hr+1;}
  }
  return r;
}
```

4. Considere as seguintes definições para implementar tabelas de hash de números inteiros (em open-address e com chaining).

```
#define HSIZE 1000
int hash (int chave, int size);
typedef struct lista {
  int valor;
  struct lista *prox;
} *HashChain[HSIZE];
typedef struct celula {
  int estado; //Livre/Ocupado/Removido
  int valor;
} HashOpen[HSIZE];
```

   (a) Defina uma função `int fromChain (HashChain h1, HashOpen h2)` que preenche a tabela `h2` com as chaves presentes na tabela `h1`. Note que ambas as tabelas têm a mesma dimensão.

   (b) Que alterações deveria efectuar na definição anterior se as tabelas tivessem dimensão diferente?

5. Relembre a definição recursiva da função *merge-sort* de ordenação de um vector e que usa uma função `merge` de fusão de dois sub-arrays. Admita por hipótese que a função `merge` executa no melhor caso em tempo constante. Calcule a complexidade da função apresentada no melhor caso.

```
void merge_sort (int N, int v[N]) {
  int m;
  if (N>1) {
    m = N/2;
    merge_sort (m,v);
    merge_sort (N-m,v+m);
    merge (v,N,m);
  }
}
```

6. Considere que existe disponível uma função `int dijkstraSP (Graph g, int o, int pais[] int pesos[])` que calcula o caminho mais curto do vértice `o` para todos os vértices do grafo `g`. Essa função preenche os dois vectores: `pais` com a árvore dos caminhos (antecessores)e `pesos` com as respectivas distâncias. Considere que se `peso[x] == -1` após a invocação da função então o vértice `x` não é alcançável a partir de `o`.

   (a) Usando esta função, defina a função `int maisArestas (Graph g, int o)` que calcula o número de arestas do caminho mais curto que liga o vértice `o` ao que lhe é mais distante.

   (b) Admitindo que a função `dijkstraSP` executa em tempo $\Theta(V^2 + V.E)$ qual a complexidade assimptótica da função que definiu? Justifique.

Parte B

1. Uma forma de implementar uma queue é usando duas stacks: stack A e stack B. As operações de inserção (enqueue) e remoção (dequeue) são realizadas usando as operações de inserção (push) e remoção (pop) de stacks da seguinte forma:

   enqueue a inserção faz-se sempre com um pop na stack A

   dequeue se existirem elementos na stack B, a remoção faz-se através de um pop na stack B; caso contrário começa-se por passar todos os elementos da stack A para a stack B (fazendo pop em A e push em B). de seguida remove-se (pop) o elemento da stack B.

   Assumindo que todas as operações sobre stacks executam em tempo constante, o pior caso da operação de dequeue é linear no tamanho da queue.

   Mostre que ambas as operações descritas sobre queues executam em tempo amortizado constante. Se decidir usar o método do potencial, use como função de potencial $\Phi(Q) = 2 * \text{size}\ (Q_A)$ (em que Q_A corresponde à stack A da queue Q).

2. Considere o problema de determinar se num grafo pesado G existe um caminho cujo custo (soma dos pesos das suas arestas) é igual a um dado valor.

   - Mostre que este problema é da classe NP descrevendo um algoritmo não determinístico polinomial que o resolva. Esta descrição deve indicar (1) como são codificadas as soluções propostas (resultado da fase não determinística) (2) incluir uma definição em C da fase determinística e (3) a argumentação de que cada uma destas componentes é de facto polinomial.
   - Como procederia para demonstrar que se trata de um problema NP-completo?