Algoritmos e Complexidade

MIEI, LCC 2º ano

3 de Novembro de 2018 – Duração: 90 min

1. Considere a função que determina se dois *arrays* de inteiros são iguais.

   (a) Determine um invariante `I` e um variante `V` apropriados para provar a **correcção total** do algoritmo.

   (b) Mostre que o programo é correcto apresentando **apenas** as condições de verificação do programa.

```
// 0 <= n
int eq (int u[], int v[], int n)
{
    i = 0;
    result = 1;
    while (i<n && result == 1)
       // I, V
    {
      if (u[i] != v[i]) { result = 0; }
      i = i+1;
    }
}
// (result == 0 || result == 1) &&
// (result == 1 <==>
//  forall x:: 0 <= x < n ==> u[x] == v[x])
```

   (c) De forma a analisar o tempo de execução desta função:

      i. Identifique o melhor e pior casos do tempo de execução.

      ii. Assumindo (i) que ambos os arrays se encontram preenchidos de forma aleatória, e (ii) que o tipo `int` corresponde aos inteiros de K bits (e que por isso a probabilidade de dois inteiros aleatórios serem iguais é de $\frac{1}{2^k}$), apresente, justificando, um somatório que permita estimar o **caso médio** do número de comparações entre elementos dos dois *arrays*.

2. Considere a seguinte definição de uma função que calcula o quadrado de um número natural (inteiro não negativo). Analise a complexidade desta função em função do número de bits necessários para representar o argumento. Note que se são necessários N bits para representar um número, então esse número pertence ao intervalo $[2^{N-1}..2^N - 1]$.
   Faça a sua análise contando o número de operações aritméticas (adições, divisões e multiplicações) efectuadas. Para isso escreva uma recorrência que traduza o comportamento da função **no pior caso** e apresente uma solução dessa recorrência.

```
int square (int x) {
  int r, m=x/2;
  if (x==0)
    r = 0;
  else if (x%2 == 0)
    r = 4* square (m);
  else
    r = 4*(square (m) + m) + 1;

  return r;
}
```

Algoritmos e Complexidade

MIEI, LCC 2º ano

12 de Janeiro de 2019 – Duração: 90 min

1. Considere o problema de, dado um array `v` com `N` elementos determinar o `k`-ésimo maior elemento (com $k < N$), isto é, o elemento que estaria na posição `N-k` caso o array estivesse ordenado por ordem crescente.

   Uma forma de resolver este problema consiste em percorrer o array guardando numa estrutura auxiliar apenas os k maiores elementos lidos até ao momento. No final o elemento pretendido é o menor dos elementos armazenados.

   Defina uma função `int kmaior (int v[], int N, int k)` que implementa o algoritmo descrito, usando como estrutura auxiliar uma *min-heap* (com `k` elementos). Admita que existem disponíveis as funções `void bubbleDown (int h[], int N)` (que faz o *bubble down* do elemento `h[0]` na heap `h` com `N` elementos) e `void bubbleUp (int h[], int N)` (que faz o *bubble up* do elemento `h[N-1]` na heap `h`).

   Identifique o pior caso da solução apresentada e diga, justificando, qual a complexidade dessa solução nesse caso.

2. Apresente a evolução de uma tabela de hash de inteiros (com tamanho `HSize=17`, função de hash `hash (x) = x%HSize` e tratamento de colisões por linear probing), inicialmente vazia e onde foram inseridos (por esta ordem) os elementos:

   30, 18, 43, 25, 60, 35, 40, 20 e 41

   Use a simulação feita para calcular o número médio de consultas ao array que foram feitos por cada inserção, neste caso.

3. Considere uma estrutura de dados sobre a qual é executada uma sequência de operações. É sabido que o tempo de execução da n-ésima operação da sequência é dado por $T_{op}(n) = n^2$ quando n é uma potência de 2 (i.e., $n = 2^k$ para algum $k \in \mathbf{N}$), e $T_{op}(n) = 1$ para todos os outros valores de n.

   Efectue a análise amortizada do tempo de execução de uma operação arbitrária desta sequência, utilizando para isso **duas técnicas** à sua escolha.

4. Dado um grafo G um caminho $\langle v_0, v_1, \ldots, v_n \rangle$ diz-se um caminho simples sse não contiver elementos repetidos. Defina uma função `int simplePath (Graph g, int v[], int k)` que testa se a sequência ⟨`v[0],v[1],...,v[k-1]`⟩ é um caminho simples. A função deve verificar que se trata realmente de um caminho e que não tem vértices repetidos.
   Indique, justificando, a complexidade da solução apresentada.

```
#define N ...
typedef struct edge {
    int dest, cost;
    struct edge *next;
} * Graph [N];
```

5. Considere o grafo pesado e não orientado representado na matriz de adjacência à direita (`(-1)` é usado para marcar que **não há** aresta).
   Relembre o algoritmo de Dijkstra para calcular o caminho mais curto entre um par de vértices.

```
int dijkstraSP (GMat g, int o, int d,
                int pesos[], int pais[]);
```

   Apresente a evolução do algoritmo de Dijkstra para calcular o caminho mais curto entre os vértices 2 e 3, i.e., a invocação `dijkstraSP (g1, 2, 3, pesos, pais);`
   Nomeadamente, apresente a evolução dos arrays `pais` e `pesos`.

```
#define N 6
typedef int GMat [N][N];

GMat g1 =
  // 0  1  2  3  4  5
  {{-1, 6, 7,-1, 3,-1}, // 0
   { 6,-1,-1, 5, 4, 2}, // 1
   { 7,-1,-1,-1, 8,-1}, // 2
   {-1, 5,-1,-1,-1, 2}, // 3
   { 3, 4, 8,-1,-1, 3}, // 4
   {-1, 2,-1, 2, 3,-1}} // 5
```