UNIVERSIDADE FEDERAL DE SÃO PAULO
CAMPUS SÃO JOSÉ DOS CAMPOS

DEPARTAMENTO DE CIÊNCIA E TECNOLOGIA (DCT)

Exercícios de Projeto e Análise de Algoritmos
Lista 1

UC: **Projeto e Análise de Algoritmos**

Aluno: **Thauany Moedano**

RA: **92486**

Professor: **Dr. Reginaldo Massanobu Kuroshu.**

Entrega: **24/09/2015**

Resumo

Resolução da lista 1 de exercícios da aula de Projeto e Análise de Algoritmos no 2º semestre de 2015. **Aviso**: Por convenção da calculadora todos os logs foram tomados como base 10

Exercício 1. Verdadeiro ou falso? Justifique.

a) $2^{n+1} = O(2^n)$?

Resolução: $f(n) \leq c.g(n) \forall n \geq n_0$?

Ou seja,

$2^{n+1} \leq c.2^n$? Abrindo a expressão, temos:

$2.2^n \leq c.2^n$ Portanto, basta atribuir $c = 2$ e um $n_0 = 1$, é possível satisfazer a desigualdade

Logo, a afirmação é **verdadeira**

b) $2^{2n} = O(2^n)$?

Resolução: $f(n) \leq c.g(n) \forall n \geq n_0$?

Ou seja,

$2^{2n} \leq c.2^n$? Abrindo a expressao, temos:

$2^{2^n} \leq c.2^n$ Dividindo ambos os lados por 2^n, teremos:

$2^n \leq c$, como n cresce conforme o tempo, não há C suficientemente grande que satisfaça a desigualdade, logo a afirmação é **falsa**

c) $\sqrt{n} = O(\log n)$?

Resolução: $f(n) \leq c.g(n) \forall n \geq n_0$?

$\sqrt{n} \leq c.\log n$, Passando o log para o outro lado da expressão, temos:

$\frac{\sqrt{n}}{\log n} \leq c$, Como não é possível eliminar n da expressão e c é constante e também não cresce conforme o tempo, a expressão é **falsa**.

d) $\sum_{i=1}^{n} 3^i = \Theta(3^n)$?

Resolução: $g(n).c_1 \leq f(n) \leq c_2.g(n) \forall n \geq n_0$?

$3^n.c_1 \leq \sum_{i=1}^{n} 3^i \leq 3^n.c_2$?

Temos que $\sum_{i=1}^{n} 3^i$ é o somatório de uma PG, portanto, possui fórmula fechada e podemos substituir por:

$3^n.c_1 \leq \sum_{i=1}^{n} 3^i \leq 3^n.c_2$

$3^n.c_1 \leq \frac{(3^{n+1}-3)}{2} \leq 3^n.c_2$

$3^n.c_1 \leq \frac{3^{n+1}}{2} - \frac{3}{2} \leq 3^n.c_2$

Se dividirmos todos os lados por 3^n, teremos:

$c_1 \leq \frac{\frac{3^{n+1}}{2} - \frac{3}{2}}{3^n} \leq c_2$, considerando um n_0 como 1, basta definir as constantes $c_1 = 1$ e $c_2 = 2$ e a afirmação sera **verdadeira**.

Exercício 2. Para cada dos itens seguintes, escolha uma das seguintes relações:
$f(n) = O(g(n))$,
$f(n) = \Omega(g(n))$ ou
$f(n) = \Theta(g(n))$:

a) $f(n) = \log n^2$; $g(n) = \log(n + 5)$
Resolução:

$\log n^2 \geq \log n + 5.c_1$, Dividindo todas as parcelas por $\log n + 5$

$\frac{2.\log n}{\log n + 5} \geq c_1$

Definindo n_0 como 2 e c_1 como 0.05 provamos que a relação é verdade uma vez que o lado esquerdo tende a aumentar. Portanto podemos definir a relação como $\log n^2 = \Omega(\log n + 5)$

b) $f(n) = n$; $g(n) = \log^2 n$
Resolução: Vamos testar a relação:

$n \geq c.\log^2 n$, Dividindo ambos os lados por $\log^2 n$

$\frac{n}{\log^2 n} \geq c$

Agora basta definir n_0 e c que satisfaçam a desigualdade. Escolhendo $n_0 = 2$ e $c = 1$ a desigualdade sempre é verdadeira.

Portanto, podemos concluir que $n = \Omega(\log^2 n)$

c) $f(n) = 2^n$; $g(n) = 3^n$
Resolução: Como potências de base 3 sempre resultam em valores maiores que potências de base 2, podemos definir a relação como:

$2^n \leq 3^n.c_1$, escolhendo $n_0 = 0$ e $c_1 = 1$, a afirmação é sempre verdade e portanto $2^n = O(3^n)$

Exercício 3. Mostre que:

a) $f(n) = 5n^2 + 10n = \Theta(n^2)$
Resolução: Queremos mostrar que:

$n^2.c_1 \leq 5n^2 + 10n \leq n^2.c_2$, Dividindo todas as parcelas por n^2

$c_1 \leq 5 + \frac{10}{n} \leq c_2$ Analisando esta parcela podemos perceber que para um n suficientemente grande, teríamos que algo próximo a 5 e para um n suficientemente pequeno teríamos algo próximo a 15.

Então definindo n_0 como 1, poderíamos definir constantes $c_1 = 5$ e $c_2 = 15$ satisfazendo a relação.

b) $f(n) = 100n^2 = O(n^2)$
Resolução:

Queremos mostrar que:

$100n^2 \leq c.n^2$ Dividindo os dois lados por n^2, temos:

$100 \leq c$ Portanto, basta definir uma constante maior ou igual a 100. Tomando $n_0 = 1$ e $c = 100$ a relação é satisfeita

c) $f(n) = 100n^2 = \Omega(n^2)$

Resolução: Queremos mostrar que:

$100.n^2 \geq c.n^2$ Dividindo os dois lados por n^2

$100 \geq c$, Portanto basta definir uma constante menor ou igual a 100. Definindo $n_0 = 1$ e $c = 100$, a relação é satisfeita.

Exercício 4. Qual valor a seguinte função retorna? Expresse sua resposta em função de n. Forneça a complexidade do tempo de execução do pior caso usando a notação O.

```
1    int loops(n){
2       int i, j, k, r=0;
3         for(i=1; i<=n-1; i++)
4          for(j=i+1; j<=n; j++)
5                for(k=1; k<=j; k++)
6                   r+=1;
7       return r;
8     }
```

Resolução: Podemos analisar linha a linha como o algoritmo se comporta:

linha 1: 1 iteração para chamar a função.

linha 2: 1 iteração para atribuir os códigos.

linha 3: Este *for* segue uma quantidade linear de chamadas e é atribuído n vezes.

linha 4: Neste *for*, o j está em função de i e sua chamada corresponde a um somatório dado por:

$\sum_{i=1}^{n} n - i$

linha 5: Agora o *for* está em função de j que depende de i. Portanto, podemos expressá-lo em função do seguinte somatório:

$\sum_{i=1}^{n} \sum_{j=i+1}^{n} j$

Portanto para descobrir a complexidade do algoritmo para somar os somatórios que representam os valores de maior ordem:

$\sum_{i=1}^{n} n - i + \sum_{i=1}^{n} \sum_{j=i+1}^{n} j$

$\sum_{i=1}^{n} n - \sum_{i=1}^{n} i + \sum_{i=1}^{n} \sum_{j=i+1}^{n} j$

Resultando em um polinômio cujo complexidade é de grau $O(n^3)$

Exercício 5. Considere o código abaixo para cálculo de fatorial. Mostre por invariante de laço que o seguinte algoritmo para calcular o fatorial de um número natural está correto.

```
int fatorial(int a){
   int x = 1;
     int i = 1;
     while (i <= a){
      x = x * i;
         i = i + 1;
     }
     return x;
 }
```

Resolução: Temos que enunciar o invariante de laço e prová-lo para três situações: inicialização, manuntenção e terminação.

Invariante de laço: O fatorial de um número n sempre é a mulltiplicação de i pelo fatorial de (i-1).

Inicialização: Antes do laço, i=1. Como o fatorial de zero é 1, o fatorial se mantém um.

Manutenção: Durante o laço, i sempre é incrementado de 1. Durante o processo, o fatorial atual é (i-1) e sempre é multiplicado por i mantendo o invariante.

Terminação: O algoritmo termina quando i é maior que n. Assim o último laço é calculado quando i = n. Ou seja, o fatorial será o fatorial de $(n-1).n$ o que resulta no fatorial de n.

Exercício 6. Considere o algoritmo de ordenação *Selection sort* apresentado no código abaixo.

```
1  Selecao(A){
2     for i=1 to n-1 do
3        min=i
4           for j=i+1 to n do
5              if (A[j] < A[min])
6                 min = j
7                troca (A[min], A[i]);
8   }
```

a) Quantas comparações entre dois elementos do vetor A o algoritmo realiza? Qual a complexidade no pior caso? E no melhor caso?
Resolução: O algoritmo de seleção faz o mesmo número de comparações tanto no pior caso quanto o melhor caso. O número de comparações sai de uma análise linha a linha dos processamentos de maior ordem:

A linha 4 executa um processamento que surge um somatório, ou seja:

$\sum_{i=1}^{n} n - i$

$\sum_{i=1}^{n} n - \sum_{i=1}^{n} i$

$n(n-1) - \frac{n(n-1)}{2}$

$\frac{n^2}{2} - \frac{n}{2}$

Portanto, a complexidade do algoritmo é sempre $O(n^2)$

b) Demonstre a corretude do algoritmo por invariantes de laço do laço externo.
Resolução: Devemos enunciar o invariante de laço e prová-lo para três ocasiões: Inicialização, manutenção, terminação.

Invariante de laço: Suponha o arranjo A[1....i....n] que deve ser ordenado. O sub arranjo com i-1 elementos estão ordenados em sua posição final.

Inicialização: Antes da primeira iteração, i = 1 e o arranjo B[1...i-1] não possui elementos para ordenar.

Manutenção: Durante o processo, o elemento da posição i é trocado com o menor elemento do arranjo C[i+1...n]. Assim o i é acrescido de 1 e novamente o arranjo B[1....i-1] está ordenado.

Terminação: Quando o laço termina existe n-1 menores elementos ordenados. Isso significa que o elemento A[n] é o maior dentre todos os elementos e por está em sua posição final.

c) Escreva uma versão recursiva da função Selecao.
Resolução:

```c
int SelecaoRec(TItem *A, int indice, int n, int menor){
    if(indice >= n)
      return(1);

   int i, troca;
   for(i = indice+1; i < n; i++) {
      if(A[menor].Chave > A[i].Chave)
         menor = i;
   }

   troca = A[indice].Chave;
   A[indice].Chave = A[menor].Chave
   A[menor].Chave = troca;

   return(SelecaoRec(A, indice+1, n, indice+1));

 }
```

d) Demonstre a corretude do Selection sort recursivo através de indução.
Resolução:

Objetivo: Mostrar que para um arranjo A[1....n] de elementos em qualquer ordem, podemos ordená-lo em ordem crescente utilizando o método *Selection Sort*

Caso base: O caso base é para $n = 1$. Como todo conjunto de apenas um elemento já está ordenado, tem-se que é o caso base é verdadeiro.

Hipótese Indutiva: Suponha um arranjo com k elementos do qual podemos ordená-los pelo método *Selection Sort*. Queremos mostrar que podemos ordenar $k + 1$ elementos.

Passo indutivo: Sabemos pela hipótese indutiva que sabemos ordenar uma sequência de k elementos. Então seja S_{k+1} a sequência de $k + 1$ elementos a ser ordenada, encontre inicialmente o elemento de menor valor da sequência (MIN) e troque de lugar com o elemento na primeira posição do arranjo. Isto garante que MIN estará em sua posição final. Desta forma resta uma sequência de S_k de k elementos a ser ordenada que pela hipótese de indução nos garante que é possivel.