Universidade Federal de Minas Gerais
Departamento de Ciência da Computação
Algoritmos e Estruturas de Dados II

1ª Lista de Exercícios - GABARITO

- **Esta lista deverá ser entregue para os professores durante a aula do dia 13 de setembro de 2011. Não serão recebidas listas por e-mail.**
- **Todos os programas devem ser feitos em C.**

1. São dados 2*n* números distintos distribuídos em dois vetores com *n* elementos A e B ordenados de maneira tal que:

$$A[1] > A[2] > A[3] > \ldots > A[n] \text{ e } B[1] > B[2] > B[3] > \ldots > B[n]$$

Apresente um algoritmo linear para encontrar o *n*-ésimo maior número dentre estes 2*n* elementos.

Para encontrar o maior elemento, basta comparar o elemento A[1] com B[1], o maior será o maior elemento do conjunto. Para encontrar o segundo maior elemento, basta comparar o elemento A[1] com B[1]. Se o A[1] for o maior, basta compara o A[2] com o B[1], senão serão comparados os elementos A[1] com B[2]. Neste caso, serão necessárias 2 comparações. Para encontrar o *n*-ésimo maior elemento, basta realizar *n* comparações.

2. Considere o problema de encontrar a posição de inserção de um novo elemento em um conjunto ordenado:

$$A[1] > A[2] > A[3] > \ldots > A[n]$$

a) Apresente a situação e/ou entrada de dados em que ocorre o melhor caso e o pior caso.
b) Apresente um algoritmo para resolver o problema acima.

a) Resolução através de busca binária. Existem $n+1$ lugares possíveis para inserir este novo elemento.
Melhor caso: elemento será inserido após o elemento que está na posição $\lceil n/2 \rceil$.
Pior caso: elemento será inserido na posição 1 ou $n+1$. Para inserir um novo elemento serão necessárias, no pior caso $\log(n + 1)$ comparações.

3. Considere a função abaixo:

```
int X(int a)
{
       if (a<=0) then
              return 0;
       else
              return (a + X(a-1));
}
```

a) O que essa função faz?
b) Calcule a sua ordem de complexidade. Mostre como você chegou a esse resultado.
c) Escreva uma função não-recursiva que resolve o mesmo problema. Qual é a ordem de complexidade da sua função? Explique.
d) Qual implementação é mais eficiente? Justifique.

a) Calcula o somatório de 0 até n(a)
b) Equação de recorrência:
$T(n) = c + T(n-1), \quad p/\ n > 0$
$T(n) = d \quad p/\ n = 0$

$T(n) = O(n)$

c) A implementação não recursiva também seria O(n). Nesse caso, apesar das complexidades serem as mesmas, a função não recursiva é preferida por ser simples de implementar e não necessitar de múltiplas chamadas de função (economia de memória e tempo).

4. Considere que a multiplicação de matrizes é $O(n^3)$. Se você tivesse a opção de utilizar um algoritmo exponencial $O(2^n)$ para multiplicar duas matrizes, qual algoritmo você iria preferir? Justifique.

Se você tivesse a opção de utilizar um algoritmo exponencial $O(2^n)$ para multiplicar duas matrizes, ele só valeria a pena para $n < 10$. A partir daí, o algoritmo exponencial torna-se mais caro que o algoritmo $O(n^3)$.

5. O **Casamento de Padrões** é um problema clássico em Ciência da Computação e é aplicado em áreas diversas como pesquisa genética, editoração de textos, buscas na internet, etc. Basicamente, ele consiste em encontrar as ocorrências de um **padrão P** de tamanho **m** em um **texto T** de tamanho **n**. Por exemplo, no texto T = "BELO HORIZONTE" o padrão P = "ORI" é encontrado na posição 6 enquanto o padrão P = "ORA" não é encontrado. O algoritmo mais simples para o casamento de padrões é o algoritmo da "Força Bruta", mostrado abaixo. Analise esse algoritmo e responda: qual é a função de complexidade do número de comparações de caracteres efetuadas no melhor caso e no pior caso. Dê exemplos de entradas que levam a esses dois casos. **Explique sua resposta!**

```
typedef char TipoTexto[MaxTexto];
typedef char TipoPadrao[MaxPadrao];

void ForcaBruta (TipoTexto T, int n, TipoPadrao P, int m)
//-- Pesquisa o padrao P[0..m-1] no texto  T[0..n-1] --
{
     int i, j, k;

     for (i = 0; i < n; i++) {
          k = i;
          j = 0;
          while ((j < m) && (T[k] == P[j])) {
             k++;
             j++;
          }
          if (j == m) {
             printf("Casamento na posicao %d\n", i);
             break;  // sai do for
          }
     }
}
```

- Melhor Caso: padrão encontrado na primeira posição. O for vai ser executado 1 vez e o while m vezes. Portanto, o número de comparações vai ser O(m).
  Exemplo: T : "Prova de AEDSII" P: "Pro"

- Pior caso: uma seqüência de caracteres quase igual ao padrão se repete no texto, com a única diferença sendo o último caractere. Dessa forma, o while será executado m vezes para cada iteração do for, que será executado n-m+1 vezes. Portanto, o número de comparações será (n-m+1).(m) Pior caso: O(m.n)
  Exemplo: T: "aaaaaaaaaaaa" P: "aaab"

6. Vários algoritmos em computação usam a técnica de "Dividir para Conquistar": basicamente eles fazem alguma operação sobre todos os dados, e depois dividem o problema em sub-problemas menores, repetindo a operação. Uma equação de recorrência típica para esse tipo de algoritmo é mostrada abaixo. Resolva essa equação de recorrência.

$$\begin{cases} T(n) = 2T(n/2) + n; \\ T(1) = 1; \end{cases}$$

$$T(n) = O(n.\log_2 n)$$

7. Indique para cada par de expressões (A,B) na tabela abaixo, se A é O, o, Ω, ω ou Θ de B. Assuma que $k \geq 1$ e $0 < \varepsilon < 1 < c$ são constantes. Sua resposta deve ser da forma SIM ou NÃO.

Nota: $\log^k n = \underbrace{\log\log\cdots n}_{k}$ e $n! \approx \left(\frac{n}{e}\right)^n$.

	A	B	O	o	Ω	ω	Θ
(i)	$\log^k n$	n^ε	S	S	N	N	N
(ii)	n^k	c^n	S	S	N	N	N
(iii)	c^n	$c^{n/2}$	N	N	S	S	N
(iv)	$\log(n!)$	$\log(n^n)$	S	N	S	N	S
(v)	$\log^{k+1} n$	$\log^k n$	S	S	N	N	N
(vi)	c^ε	$(c+1)^\varepsilon$	S	N	S	N	S

8. Qual algoritmo você preferiria: um que requer n^5 passos ou um que requer 2^n passos? Justifique sua resposta.

Depende do tamanho do problema. Teremos $n^5 > 2^n$ para $n < 23$, portanto 2^n é melhor ; para $n >= 23$, n^5 é melhor. No limite, uma complexidade polinomial é melhor que uma complexidade exponencial para problemas "grandes".

9. Indique se as afirmativas a seguir são verdadeiras e justifique sua resposta:

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

Verdadeira: existem constantes positivas c e m tais que $2^{n+1} <= 2^n$ para todo n >= m; exemplo: c = 3 e m = 0.

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

Falsa: $2^{2n} = 4^n$, e portanto não existem constantes positivas c e m que atendam à definição neste caso.

c) f(n) = O(u(n)) e g(n) = O(v(n)) → f(n) + g(n) = O(u(n) + v(n))

Verdadeira: na primeira parte, existem constantes positivas c e m tais que f(n) <= c_1.u(n) para n>= m1 e g(n) <= c_2.v(n) para n>= m_2. Portanto, f(n) + g(n) <= c1u(n) + c2 v(n) para c = max(c1, c2) e m = max(m1, m2). Essa constatação equivale a considerar como complexidade o máximo das duas funções.

d) f(n) = O(u(n)) e g(n) = O(v(n)) → f(n) – g(n) = O(u(n) – v(n))

Falsa: se f(n) – g(n) = O(u(n)) – O(v(n)) = O(u(n))+(-1)O(v(n)); como -1 é uma constante, ela pode ser desconsiderada, de acordo com as propriedades da notação O. Portanto, f(n) – g(n) = O(u(n)) + O(v(n)) = O(max(u(n), v(n)). Observe que a notação O corresponde à relação de "menor que ou igual a"(<=), que é alterada pela subtração e pela divisão.

10. Considerando que a operação relevante é o número de vezes que a operação soma é executada, apresente a função de complexidade de tempo para:

a)

```
for i ← 1 to n do
   for j ← 1 to n do
      for k ← 1 to n do
         temp ← temp + i + j + k
```

$$\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n} 3 = \ldots = 3n^3$$

b)

```
for i ← 1 to n do
   for j ← 1 to i do
      for k ← 1 to j do
         temp ← temp + i + j + k
```

$$\sum_{i=1}^{n}\sum_{j=1}^{i}\sum_{k=1}^{j} 3 = \ldots = \frac{n^3}{2} + \frac{3n^2}{2} + n$$

c)

```
for i ← 1 to n do
   for j ← 1 to n do
      for k ← i to n do
         temp ← temp + i + j + k
```

$$\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=i}^{n} 3 = \ldots = \frac{3n^3}{2} + \frac{3n^2}{2}$$

d)

```
for i ← 1 to n do
   for j ← i to n do
      for k ← i to n do
         temp ← temp + i + j + k
```

$$\sum_{i=1}^{n}\sum_{j=i}^{n}\sum_{k=i}^{n} 3 = \ldots = n^3 + \frac{3n^2}{2} + \frac{n}{2}$$

e)

```
for i ← 1 to n do
   for j ← i to n do
      for k ← i to j do
         temp ← temp + i + j + k
```

$$\sum_{i=1}^{n}\sum_{j=i}^{n}\sum_{k=i}^{j} 3 = \ldots = \frac{n^3}{2} + \frac{3n^2}{2} + n$$

11. Resolva as seguintes equações de recorrência:

a) $\begin{cases} T(n) = T(n-1) + c \\ T(1) = 0 \end{cases}$ c constante, $n > 1$

$T(n) = cn - c$

b) $\begin{cases} T(n) = T(n-1) + 2^n \\ T(0) = 1 \end{cases}$ $n \geq 1$

$T(n) = 2^{n+1} - 1$

c) $\begin{cases} T(n) = cT(n-1) \\ T(0) = k \end{cases}$ c, k constantes, $n > 0$

$T(n) = c^n k$

d) $\begin{cases} T(n) = 3T(n/2) + n \\ T(1) = 1 \end{cases}$ $n > 1$

$T(n) = 3n^{\log 3} - 2n$

e) $\begin{cases} T(n) = 3T(n-1) - 2T(n-2) & n > 1 \\ T(0) = 0 \\ T(1) = 1 \end{cases}$

$T(n) = 2^n - 1$

12. Considere o algoritmo a seguir, supondo que a operação crucial é "inspecionar elemento". O algoritmo inspeciona os n elementos de um conjunto e, de alguma forma, consegue descartar 2/5 dos elementos e fazer uma chamada recursiva sobre os 3n/5 elementos restantes.

a) Escreva uma equação de recorrência que descreva esse comportamento
b) Converta a equação de recorrência para um somatório
c) Dê a fórmula fechada para esse somatório

```
void Pesquisa (int n)
{
   if (n < 1)
         'inspecione elemento';
         termine;
   else {
         'para cada um dos elementos, inspecione elemento';
         Pesquisa(3 * n / 5);
   }
}
```

$$\begin{cases} T(n) = T\left(\frac{3}{5}n\right) + n \\ T(1) = 1 \end{cases}$$

$$T(n) = \frac{5}{2}n - \frac{3}{2}$$

13. Torre de Hanói. Em 1883, o matemático francês Edouard Lucas criou um jogo chamado Torre de Hanói. O jogo começa com um conjunto de oito discos empilhados em tamanho decrescente em uma das três varetas, conforme mostrado na Figura 1. O objetivo do jogo é transferir toda a torre para uma das outras varetas, movendo um disco de cada vez, mas nunca movendo um disco maior sobre um menor.

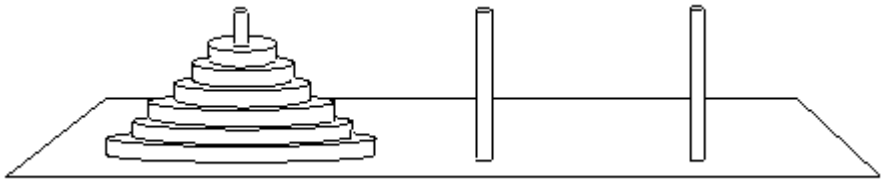
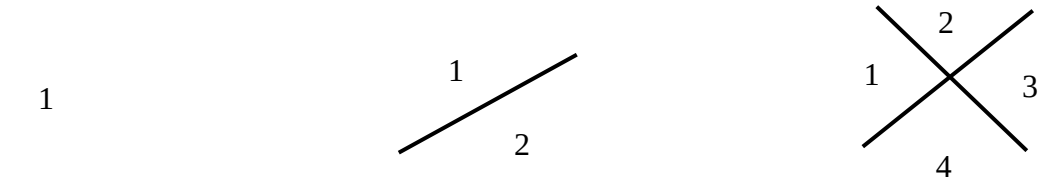


Figura 1: Configuração inicial da Torre de Hanói.

$$\begin{cases} T(n) = 2T(n-1) + 1 \\ T(1) = 1 \end{cases}$$

$$T(n) = 2^n - 1$$

14. Linhas no plano ou Cortando a sua pizza favorita. Quantas fatias de pizza uma pessoa pode obter ao fazer *n* cortes retos com uma faca? Ou, expressando de outra forma, qual é o número máximo de regiões L_n determinado por *n* retas no plano? Lembre-se que um plano sem nenhuma reta tem uma região, com uma reta tem duas regiões e com duas retas tem quatro regiões, conforme mostrado na figura 2.



$L_0=1$ $L_1=2$ $L_2=4$

Figura 2: Regiões no plano.

$$\begin{cases} T(n) = T(n-1) + n \\ T(0) = 1 \end{cases}$$

$$T(n) = \frac{n^2 + n + 2}{2}$$

15. Use o teorema mestre para derivar um limite assintótico Θ para as seguintes recorrências:

a) T(n) = 2T(n/2) + n – 1

$$T(n) = \Theta(n \log n)$$

b) T(n) = 3T(n/2) + n

$$T(n) = \Theta(n^{\log_2^3})$$

c) $T(n) = 4T(n/2) + n^2$

$$T(n) = \Theta(n^2 \log n)$$

d) $T(n) = 4T(n/2) + n^3$

$$T(n) = \Theta(n^3)$$

Fórmulas Úteis: $\sum_{i=1}^{n} k = n.k$ $\sum_{i=1}^{n} i = \frac{n.(n+1)}{2}$ $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$ $\sum_{i=1}^{n} a^i = a\frac{a^n - 1}{a - 1}$