Primeiro teste de **Algoritmos e Estruturas de Dados**

**15 de Outubro de 2018** **Duração máxima: 1 hora**

**Justifique todas as suas respostas.**

**Nome:**

**N. Mec.:**

**Fórmulas:**

- $\sum_{k=1}^{n} 1 = n$
- $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$
- $\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$
- $\sum_{k=1}^{n} k^3 = \left(\frac{n(n+1)}{2}\right)^2$
- $\sum_{k=1}^{n} \frac{1}{k} \approx \log n$
- $n! \approx n^n e^{-n}\sqrt{2\pi n}$

**4.0** $\boxed{\textbf{1:}}$ No seguinte código,

```
#include <stdio.h>

int f(int x) { return x - 2; }
int g(int x) { return x * x; }

int main(void)
{
  for(int i = -1000;i <= 1000;i++)
    if( (f(i) > 0) && (g(i) > 0) )
      printf("%d\n",i);
  return 0;
}
```

2.0 **a)** para que valores da variável `i` é avaliada a função `g(x)`? Para valores de i tais que f(i)>0, ou seja, f(i)>0 <=> i-2>0 <=> i>2

2.0 **b)** que valores de `i` são impressos?

Para qualquer i>2, g(i) = i*i vai ser positivo (>0), logo são impressos todos os inteiros no intervalo ]2, 1000].

**3.0** $\boxed{\textbf{2:}}$ Ordene as seguintes funções por ordem crescente de ritmo de crescimento.

| Número da função | função |
|---|---|
| 1 | $\underline{1.7^n} + n^{1.5}$ |
| 2 | $\underline{n^2} + n\log^9 n + \frac{1000}{n}$ |
| 3 | $\frac{n!}{2.4^n}$ |
| 4 | $n^{1.7} + \underline{1.5^n}$ |
| 5 | $n\log n + \underline{n\sqrt{n}}$ |

growth rate: $1, \log n, \sqrt{n}, n, n\log n, n^2, n^3, 2^n, n!$.

____ = termo dominante

Ordem: 5, 2, 4, 1, 3

**2.5** $\boxed{\textbf{3:}}$ No seguinte código,

```
int a[10],*b = &a[7];
for(int i = 0;i < 10;i++)
  a[i] = -i;
```

no fim de correr o código...

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| a= | 0 | -1 | -2 | -3 | -4 | -5 | -6 | -7 | -8 | -9 | (...) |
| | | | | | | | | b[0] | b[1] | b[2] | b[3] |

qual é o valor de `b[3]`? O valor de b[3] não é conhecido (pois b[3] já não aponta para uma posição de memório cujo valor tenhamos definido como parte do *array* a), mas sabemos que b[3] aponta para o espaço de memória cujo endereço corresponde a &a[9]+4, e portanto o seu valor corresponde ao valor do inteiro que estiver nesse espaço de memória.

**3.0** $\boxed{\textbf{4:}}$ A complexidade computacional de muitos algoritmos é expressa usando a notação "big Oh" ($\boldsymbol{O}$) em vez da notação "Big Theta" ($\boldsymbol{\Theta}$). Porquê? (Nota: dois terços da cotação para uma boa explicação das duas notações, um terço para uma boa explicação do porquê.)

A notação "big Oh" (O) descreve a forma como uma função ou algoritmo cresce explicitando o limite superior deste crescimento (ou seja, o "pior caso"). Dizer que f(n) = O(g(n)) significa que existem n0 e uma constante C tais que, para todo o n>=n0, f(n)<=C*g(n). A notação "big Theta" (Θ) também comenta sobre a taxa de crescimento de uma função/algoritmo, desta vez clarificando tanto um limite inferior como um limite superior, tendo ambos a mesma forma. Dizer que f(n)=Θ(g(n)) significa que existem n0 e duas constantes, C1 e C2, tais que para todo o n>=n0, C1*g(n)<=f(n)<=C2*g(n).

A notação "big Oh" é mais frequentemente usada para expressar a complexidade computacional dos algoritmos exatamente por expressar o pior caso possível para o tempo de execução dos algoritmos em questão. Permite uma comparação fácil da eficiência de diferentes algoritmos que tenham o mesmo objetivo.

**5.0** $\boxed{\textbf{5:}}$ Para a seguinte função,

```
int f(int n)
{
  int i,j,k,r1,r2 = 0;

  for(i = 0;i < n;i++)
  {
    for(j = 0;j <= 4;j++)
    {
      r1 = 1;
      for(k = 0;k <= j;k++)
        r1 *= k;
    }
    r2 += r1;
  }
  return r2;
}
```

(a) $$N = \sum_{i=0}^{n-1}\left(\sum_{j=0}^{4}\left(\sum_{k=0}^{j}(1)\right)\right) = \sum_{i=0}^{n-1}\left(\sum_{j=0}^{4}(j+1)\right)$$

$$= \sum_{i=0}^{n-1}\left(\sum_{j=0}^{4}(j) + \sum_{j=0}^{4}(1)\right) = \sum_{i=0}^{n-1}\left(\sum_{j=1}^{5}(j-1) + \sum_{j=1}^{5}(1)\right)$$

$$= \sum_{i=0}^{n-1}\left(\sum_{j=1}^{5}(j) - \sum_{j=1}^{5}(1) + \sum_{j=1}^{5}(1)\right)$$

$$= \sum_{i=0}^{n-1}\left(\sum_{j=1}^{5}(j)\right) = \sum_{i=0}^{n-1}\left(\frac{5(5+1)}{2}\right) = \sum_{i=0}^{n-1}(15)$$

$$= 15 \cdot \sum_{i=0}^{n-1}(1) = 15 \cdot \sum_{i=1}^{n}(1) = 15n \longrightarrow$$ a linha é executada 15n vezes

**(b)** A linha *r1 *= k;* efetua uma operação de multiplicação. Ora, na primeira iteração do ciclo *for(k=0; k<=j; k++)*, a variável k tem o valor 0, pelo que r1 vai assumir o valor...

r1 = r1*k <=> r1 = r1*0 <=> r1 = 0

Assim sendo, todas as multiplicações consequentes vão resultar em 0. Todas as iterações do ciclo *for(i=0; i<n; i++)* , cuja última instrução é *r2 += r1;*, vão efetuar a operação *r2 += 0;*, que corresponde a *r2=r2;*. Como r2 é inicializado a 0, o valor devolvido pela função é 0.

2.5 **a)** quantas vezes é executada a linha `r1 *= k;`?

2.5 **b)** que valor é devolvido pela função?

**2.5** $\boxed{\textbf{6:}}$ Dê um exemplo de uma função concreta que tenha uma complexidade computacional de $\boldsymbol{\Theta(n^3)}$. (Não se esqueça de justificar a sua resposta.)

**FUNÇÃO:**

```
int main(int argc,char **argv)
{
    int i,j,k,res;
    for (i=0; i<n; i++) {
        for (j=0; j<n; j++) {
            for (k=0; k<n; k++) {
                res += i+j+k;
            }
        }
    }
    return res;
}
```

**EXPLICAÇÃO:**

A função efetua sempre três ciclos *for*, cada um de n iterações, pelo que a linha *res *= i+j+k;* é sempre executada n^3 vezes - logo a função tem complexidade computacional de Θ(n^3).