Primeiro teste de Algoritmos e Estruturas de Dados

21 de Outubro de 2019 **14h10m – 15h00m**

Responda a todas as perguntas no enunciado do teste. Justifique todas as suas respostas.

Nome: ______________________________

N. Mec.: __________

Fórmulas:

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

3.0 **1:** No seguinte código,

```c
#include <stdio.h>

int f(int x) { return x % 3; }
int g(int x) { return x % 2; }

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

$i \in [-1000, 1000]$
- 1000 valores +
- 1000 valores −
- zero

→ c só é incrementado quando i>0 (porque && = and)

1.0 **a)** para que valores da variável `i` é avaliada a função `g(x)`?

1.0 **b)** que valores de i são impressos?

1.0 **c)** que valor de c é impresso?

Respostas:

(a) A função g só é avaliada se a condição f(i) for verdadeira. Ora, f(i) só é verdadeira para valores de i que são múltiplos de 3, logo g(x) é avaliada para os múltiplos de 3 (−999, −996, …, 0, …, 996, 999).

(b) São impressos todos os valores de i que são múltiplos de 3 e pares (−996, −990, −984, …, 0, …, 984, 990, 996).

(c) No intervalo de i ($i \in [-1000, 1000]$), existem 1000 valores positivos (1 … 1000). Desses, apenas 333 são múltiplos de 3 (3, 6, 9, …, 999) e, destes, apenas 166 são pares (6, 12, 18, …, 996). Assim sendo, o valor impresso de c é 166.

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

3.0 **2:** Ordene as seguintes funções por ordem crescente de ritmo de crescimento. Responda nas duas colunas da direita da tabela. Na coluna da ordem, coloque o número 1 na função com o ritmo de crescimento menor (e, obviamente, coloque o número 5 na com o ritmo de crescimento maior).

Número da função	função	termo dominante	ordem	
1	$n^{99} + 1.1^n$	1.1^n	3	k^n, $k = 1.1$
2	$\dfrac{n!}{42^n}$	$\dfrac{n!}{42^n}$	5	$n!$
3	$n^2 \log n^{99} + n^2 \sqrt[3]{n}$ (*)	$n^2 \sqrt[3]{n}$	1	n^k, $k = \frac{7}{3}$
4	$1.2^n + n^2$	1.2^n	4	k^n, $k = 1.2$
5	$\sum_{k=1}^{n}(k^2 + k)$	$\sum_{k=1}^{n}(k^2)$	2	n^k, $k = 3$

(*) $= n^2 . \log(n^{99}) + n^{7/3}$, $\frac{7}{3} > 2$ logo $n^{7/3}$ cresce mais rápido

4.0 **3:** Um programador inexperiente escreveu a seguinte função para copiar uma zona de memória com `size` bytes que começa no endereço `src` para uma outra zona de memória que começa no endereço `dest`.

```
void mem_copy(char *src,char *dest,size_t size)
{
  for(size_t i = 0;i < size;i++)
    dest[i] = src[i];
}
```

raciocínio (a)

0 1 2 3 4 5 6 7 8 9 (estado inicial)
0 1 2 3 4 3 6 7 8 9 (i=0)
0 1 2 3 4 3 4 7 8 9 (i=1)
0 1 2 3 4 3 4 3 8 9 (i=2)
0 1 2 3 4 3 4 3 4 9 (i=3

raciocínio (b)

(estado inicial) 0 1 2 3 4 5 6 7 8 9
(i=0) 0 1 2 5 4 5 6 7 8 9
(i=1) 0 1 2 5 6 5 6 7 8 9
(i=2) 0 1 2 5 6 7 6 7 8 9
(i=3) 0 1 2 5 6 7 8 7 8 9

Responda às seguintes perguntas, considerando que para cada uma das duas primeiras o conteúdo **inicial** do array `c` é `char c[10] = { 0,1,2,3,4,5,6,7,8,9 };`

1.3 **a)** qual o conteúdo do array `c` depois de `mem_copy(&c[3],&c[5],4);` ter sido executado?

Resposta:

0	1	2	3	4	3	4	3	4	9
c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]	c[8]	c[9]

1.3 **b)** qual o conteúdo do array `c` depois de `mem_copy(&c[5],&c[3],4);` ter sido executado?

Resposta:

0	1	2	5	6	7	8	7	8	9
c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]	c[8]	c[9]

1.4 **c)** num dos casos anteriores a cópia do conteúdo de parte do array não foi feita corretamente; sugira uma maneira de corrigir este problema (não é obrigatório escrever código).

Resposta:

O caso que está errado é o da alínea (a). Uma maneira de corrigir o problema seria tornar a função recursiva em vez de iterativa, e guardar os valores originais de c numa variável temporária...

```
void mem_copy(char *src, char *dest, size_t size, size_t counter)
{
        if (size==counter) { return; }

        int temp;                                   // variável temp
        temp = dest[counter];                       // guardar valor original na variável temp
        dest[counter] = src[counter];
        dest[counter] = temp;                       // recuperar valor original

        counter++; // incrementar counter
        mem_copy(dest[counter], src[counter], size, counter);      // chamada recursiva
}
```

3.0 **4:** A notação "big Oh" é usualmente usada para descrever a complexidade computacional do pior caso de um algoritmo. Porquê?

Resposta (tente não exceder as 100 palavas):

A notação "big Oh" é um tipo de notação matemática que examina a taxa de crescimento de uma função/algoritmo através do limite superior deste mesmo crescimento. Ora, em geral, quanto maior for a taxa de crescimento de um algoritmo, maior é o seu tempo de execução (e pior é o caso em questão). Assim sendo, o limite superior dado pela notação "big Oh" é usualmente representativo do pior caso de um algoritmo.

2.0 **5:** Escreva o código de uma função que tenha uma complexidade computacional de $\Theta(\sqrt{n})$. Como alternativa, pode optar por escrever o código de uma função de tenha uma complexidade computacional de $\Theta(\log n)$. (Pode usar pseudo-código, se bem que uma função em `C` será mais valorizada.)

Resposta:

Θ(sqrt(n))

```
void main(int n)
{
      int i = 0;
      int s = 0;
      while (s<=n)
      {
            s += i;
            i += 1;
      }
}
```

Explicação:

Se tivermos um k tal que s = 1+2+3+...+k = n, então o número de iterações é dado por k.
Como 1+2+3+...+k = (k*(k+1))/2, então a equação (k*(k+1))/2 = n dá-nos k.

(k*(k+1))/2 = n <=> k*(k+1) = 2*n
<=> k^2 + k = 2*n
<=> k^2 + k - 2*n = 0
<=> k = (-1 + sqrt(1+8n))/2, pela fórmula resolvente

logo temos Θ(sqrt(n))

Θ(log(n))

```
void main(int n)
{
      int i = 0;
      int s = 0;
      while (s<=n)
      {
            s = e^i;
            i += 1;
      }
}
```

Explicação:

Se tivermos um k tal que s = e^k = n, então o número de iterações é dado por k.
Então, a equação e^k = n dá-nos k.

e^k = n <=> k = log(n)

logo temos Θ(log(n))

5.0 **6:** Para a seguinte função,

```
int f(int n)
{
  int r = -1;

  for(int i = -2;i <= n;i++)        → n+3 iterações
    for(int j = i;j >= -3;j--)      → n+4 iterações
      r += i - j;
  return r;
}
```

2.0 **a)** quantas vezes é executada a linha `r += i - j;`?

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

1.0 **c)** qual é a complexidade computacional da função?

Respostas:

(a) $(n+3)(n+4)$ vezes

(b) valor devolvido:

$$r = -1 + \sum_{i=-2}^{n} \left(\sum_{j=-3}^{i} (i-j) \right)$$

→ somar de i a −3 ou −3 a i é igual

(c) $O(n^2)$