

Nome: André Almeida Oliveira

N. Mec.: 107637

4.0 **1:** No seguinte código,

```
#include <stdio.h>
```

```
int f(int x) { return x * x - 1; }
int g(int x) { return x / 3; }
```

```
int main(void)
{
    int c = 0;
    for(int i = 0; i <= 10; i++)
        if( f(i) && g(i) )
        {
            printf("i = %d\n", i);
            c += g(i);
        }
    printf("c = %d\n", c);
}
```

Fórmulas:

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

1.0 a) para que valores da variável i é avaliada a função $g(x)$? 0, 2, 3, 4, 5, 6, 7, 8, 9, 10

1.0 b) que valores de i são impressos? 3, 4, 5, 6, 7, 8, 9, 10

1.0 c) que valor de c é impresso? $\left\lfloor \frac{3}{3} \right\rfloor + \left\lfloor \frac{4}{3} \right\rfloor + \left\lfloor \frac{5}{3} \right\rfloor + \left\lfloor \frac{6}{3} \right\rfloor + \left\lfloor \frac{7}{3} \right\rfloor + \left\lfloor \frac{8}{3} \right\rfloor + \left\lfloor \frac{9}{3} \right\rfloor + \left\lfloor \frac{10}{3} \right\rfloor = 1+1+1+2+2+2+3+3 = 15$

1.0 d) neste caso concreto, considerando que um `int` tem 32 *bits*, existe a possibilidade de *overflow* aritmético ao correr o programa?

Não, pois um `int` suporta valores inteiros de 0 a $2^{31}-1$, logo com a valores dados i impossível ultrapassar este valor.

- 4.0 **2:** 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 `dst`.

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

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[4], &c[5], 4)`; ter sido executado?

Resposta:

0	1	2	3	4	4	4	4	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[4], 4)`; ter sido executado?

Resposta:

0	1	2	3	5	6	7	8	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:

```
void mem_copy(char *src, char *dst, size_t size)
{
    for(size_t i = size - 1; i >= 0; i--)
        dst[i] = src[i];
}
```

- 4.0 **3:** 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). Na coluna do termo dominante indique, usando a notação *Big Oh*, qual é o termo dominante; por exemplo, se na primeira coluna estivesse $3n + 7$, na segunda coluna deveria colocar $\mathcal{O}(n)$.

função	termo dominante	ordem
$42 \frac{n^n}{n!}$	$\frac{n^n}{n!}$	5
$\sum_{k=1}^n \left(k^3 + \frac{1}{k} \right)$	$\sum_{k=1}^n k^3 = \left(\frac{n(n+1)}{2} \right)^2$	2
$4n^4 \log n^4 + 2022$	$n^4 \log n$	3
$1000n^3 + 1.001^n$	$1,001^n$	4
$\frac{700}{n} + 300$	$\frac{1}{n}$	1

- 4.0 **4:** A notação *big Oh* é usualmente usada para descrever a complexidade computacional de um algoritmo. Porquê?

Resposta (tente não exceder as 100 palavras):

A notação *big Oh* é usada para indicar o número máximo de instruções que o computador vai ter que executar para completar o código.

Entendendo este valor, pôs queramos saber uma ideia de tempo máximo que o programa vai demorar, pois o tempo é diretamente proporcional à quantidade de instruções.

4.0 **5:** Para a seguinte função,

```
int f(int n)
{
    int r = -20220204;

    for(int i = 0; i <= n; i++)
        for(int j = 2 * i; j <= 2 * n; j++)
            r += j / (i + 1);
    return r;
}
```

- 3.0 a) quantas vezes é executada a linha `r += j / (i + 1);`? $(n+1)^2$
 1.0 b) qual é a complexidade computacional da função? $O(n^2)$

$$\begin{aligned}
 \text{a)} \quad & \sum_{i=0}^n \sum_{j=2i}^{2n} 1 = \sum_{i=0}^n \sum_{j=2i-2i+1}^{2n-2i+1} 1 = \sum_{i=0}^n \sum_{j=1}^{2n-2i+1} 1 = \sum_{i=0}^n (2n-2i+1) = \left(2n \sum_{i=0}^n 1 - 2 \sum_{i=0}^n i + \sum_{i=0}^n 1 \right) = \left(2n \sum_{i=1}^{n+1} 1 - 2 \sum_{i=1}^{n+1} (i-1) + \sum_{i=1}^{n+1} 1 \right) = \\
 & = \left(2n(n+1) - 2 \left(\sum_{i=1}^{n+1} i - \sum_{i=1}^{n+1} 1 \right) + n+1 \right) = \left(2n(n+1) - 2 \left(\frac{(n+1)(n+2)}{2} - n+1 \right) + n+1 \right) = \\
 & = (2n^2 + 2n - n^2 - 3n - 2 + 2n + 2 + n + 1) = (n^2 + 2n + 1) = (n+1)^2
 \end{aligned}$$

Segunda parte do teste final de Algoritmos e Estruturas de Dados

4 de fevereiro de 2022

17h00m – 17h55m

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

Nome: André Almeida Vilarinho

N. Mec.: 107637

- 4.0 **1:** Explique como está organizado um *min-heap*. Para o *min-heap* apresentado a seguir, insira o número 3. Não apresente apenas o resultado final; mostre, passo a passo, o que acontece ao *array* durante a inserção. Em cada linha, basta escrever as entradas do *array* que foram alteradas.

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

- 4.0 **2:** Explique como funciona a rotina de ordenação *insertion sort*. Indique a sua complexidade computacional e quais os seus melhores e piores casos.

Insertion sort é um algoritmo de ordenação que divide um conjunto de dados em dois subconjuntos, ou seja, um ordenado e outro desordenado. A cada iteração, o algoritmo pega o próximo elemento do subconjunto desordenado e insere-o na sua posição correta do subconjunto ordenado. Isto é feito comparando o elemento com os já ordenados, deslocando-os para a direita ou esquerda (depende do caso) se necessário.

Bom caso : $O(n^2)$ → array por ordem inversa

Pior caso : $O(n)$ → array já ordenado

- 4.0 **3:** Explique como pode procurar informação numa lista biligada não ordenada, e indique qual a complexidade computacional do algoritmo que descreveu. O que é que pode fazer para tornar a procura mais eficiente quando alguns itens de informação são mais procurados que outros? Tende não usar mais de 100 palavras.

Sendo a lista não ordenada teremos de percorrer todos os elementos da lista e, para cada um, avaliar se é igual ao pretendido ou não. Este algoritmo tem complexidade $O(n)$, pois no fim da lista terá que percorrer a lista toda.
Uma maneira de tornar esta procura mais eficiente é, colocar a informação com maior procura no início da lista.

- 4.0 **4:** Um programador pretende utilizar uma *hash table* (tabela de dispersão, dicionário) para contar o número de ocorrências de palavras num ficheiro de texto. O programador está à espera que o ficheiro tenha cerca de 6000 palavras distintas, pelo que usou uma *hash table* do tipo *separate chaining* com 10007 entradas, e usou a seguinte *hash function*:

```
unsigned int hash_function(unsigned char *s,unsigned int hash_table_size)
{
    unsigned int sum = 0u;

    for(int i = 0;s[i] != '\0';s++)
        sum += (unsigned int)(i + 1) * (unsigned int)s[i];
    return sum % hash_table_size;
}
```

Infelizmente, as expetativas do programador estavam erradas, e o ficheiro de texto era muito maior que o esperado, tendo cerca de 1000000 palavras distintas. Responda às seguintes perguntas:

- 1.0 a) A *hash function* apresentada não é das piores. Porquê?
- 3.0 b) Com *separate chaining* a *hash table* pode acomodar o milhão de palavras distintas mesmo tendo a *array* apenas 10007 entradas. Explique porquê, e explique o que é que acontece ao desempenho desta estrutura de dados.

- a) Não é das piores, pois embora seja simples e não duvida uma gama muito alta de valores, consegue distinguir anagramas. Por exemplo, "amor" e "roma" não ter hash codes diferentes.
- b) Sim, poderia acomodar o milhão de palavras devido ao uso de listas ligadas quando ocorrem colisões na hash code. Um problema que este caso possui, é que cada head da lista ligada corresponderá em média a 100 palavras, o que levará imenso tempo a percorrer a mesma.

4.0 **5:** Apresentam-se a seguir várias funções (f1 a f5) que visitam todos os nós de uma árvore binária, e mostram-se várias ordens pelas quais a função `visit` foi chamada para cada um dos nós (1 significa que o nó correspondente foi o primeiro a chamar a função `visit`, 2 que foi o segundo, e assim por diante). Para cada uma das ordens apresentadas, indique que função, ou funções, deram origem a essa ordem.

```
void f1(tree_node *n)
{
    queue *q = new_queue();
    enqueue(q,n);
    while(is_empty(q) == 0)
    {
        n = dequeue(q);
        if(n != NULL)
        {
            enqueue(q,n->right);
            enqueue(q,n->left);
            visit(n);
        }
    }
    free_queue(q);
}
```

```
void f2(tree_node *n)
{
    stack *s = new_stack();
    push(s,n);
    while(is_empty(s) == 0)
    {
        n = pop(s);
        if(n != NULL)
        {
            push(s,n->right);
            visit(n);
            push(s,n->left);
        }
    }
    free_stack(s);
}
```

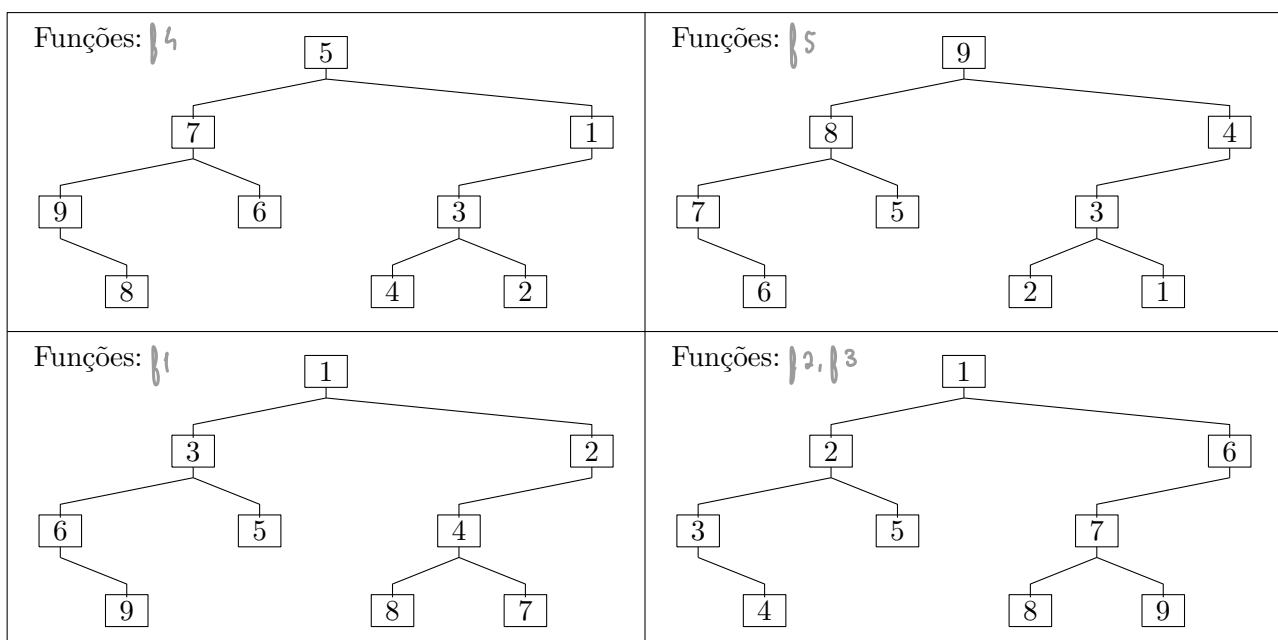
```
int cnt = 0;

void visit(tree_node *n)
{
    printf("%d\n",++cnt);
}
```

```
void f3(tree_node *n)
{
    if(n != NULL)
    {
        visit(n);
        f3(n->left);
        f3(n->right);
    }
}
```

```
void f4(tree_node *n)
{
    if(n != NULL)
    {
        f4(n->right);
        visit(n);
        f4(n->left);
    }
}
```

```
void f5(tree_node *n)
{
    if(n != NULL)
    {
        f5(n->right);
        f5(n->left);
        visit(n);
    }
}
```



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

Nome: André Almeida Oliveira

N. Mec.: 107637

4.0 **1:** O algoritmo *merge sort* divide o *array* a ser ordenado ao meio, ordena (recursivamente) cada uma das duas partes, e depois junta-as. A sua complexidade computacional é $\Theta(n \log n)$. Um aluno está convencido que se em vez de se dividir o *array* em duas partes se se dividir em cinco partes (todas mais ou menos do mesmo tamanho), então a complexidade computacional desta variante do *merge sort* será ainda mais baixa. Responda às seguintes perguntas:

1.0 a) Que estratégia algorítmica usa o *merge sort*?

2.0 b) O aluno tem razão? Justifique.

1.0 c) Nesta variante do *merge sort*, a fase de *merge* é mais fácil ou mais complicada que a do algoritmo original?

a) A estratégia usada é *Divide and Conquer*, pois o *array* inicial é dividido em vários subarrays em que cada um é ordenado individualmente e, só no fim, juntamos todos.

b) O aluno não tem razão, pois a complexidade é baseada no número de operações para dividir, ordenar e juntar a quantidade de listas que a original foi dividida. Embora seja mais rápido ordenar listas mais pequenas, fazer a junção de todas será um processo mais demorado.

$a = 5 \rightarrow$ ordenar as 5 sublistas

$b = 5 \rightarrow$ dividir a lista em 5 sublistas

$$T(n) = O(n^{\log_b a} \log n) = O(n^{\log_5 5} \log n)$$

c) Não é complicado, pois haverá um maior número de sublistas a ordenar e juntar no final.

O *master theorem* afirma que se $T(n) = aT(n/b) + f(n)$ então

- se $f(n) = O(n^{\log_b a - \epsilon})$ para um $\epsilon > 0$, então $T(n) = \Theta(n^{\log_b a})$,
- se $f(n) = \Theta(n^{\log_b a})$, então $T(n) = O(n^{\log_b a} \log n)$,
- se $f(n) = \Omega(n^{\log_b a + \epsilon})$ para um $\epsilon > 0$ e se $af(\frac{n}{b}) \leq cf(n)$ para $c < 1$ e n suficientemente grande, então $T(n) = \Theta(f(n))$.

- 4.0 **2:** Num tabuleiro de xadrez, pretende-se ir do canto inferior esquerdo **(0,0)** para o canto superior direito **(7,7)** fazendo movimentos apenas para a direita e para cima. O seguinte código apresenta uma maneira de calcular o número de maneiras de fazer isso.

```
1  long eval(int x,int y)
   {
2    if(x < 0 || x > 7 || y < 0 || y > 7)
2      return 0L;
3    else if(x == 7 && y == 7)
3      return 1L;
4    else
4      return eval(x + 1,y) + eval(x,y + 1);
   }
5  long count_paths(void)
   {
6    return eval(0,0);
   }
```

Responda às seguintes perguntas:

- 1.0 a) Que estratégia algorítmica é usada por este código? *Recursividade*
- 3.0 b) Explique por palavras qual é o objetivo de cada uma das partes numeradas do código.

- 1 → "Cabeçalho" da função que retorna o número de caminhos pretendidos, função eval
2 → "Inicializa" a função de chamada da função eval está fora do tabuleiro
3 → Retorna o valor 1 quando chegamos à peça desejada
4 → Avançar caso através dos movimentos possíveis
5 → "Cabeçalho" da função count_paths
6 → Retorna o número de caminhos possíveis desde a peça (0,0) até a peça (7,7) através da função eval

- 4.0 **3:** Tal como no problema anterior, num tabuleiro de xadrez, pretende-se ir do canto inferior esquerdo **(0,0)** para o canto superior direito **(7,7)** fazendo movimentos apenas para a direita e para cima. O seguinte código apresenta uma maneira de calcular o número de maneiras de fazer isso.

```
1  long count_data[8][8];
2  long eval(int x,int y)
   {
3    if(x < 0 || x > 7 || y < 0 || y > 7)
3      return 0L;
4    if(count_data[x][y] < 0L)
4      count_data[x][y] = eval(x - 1,y) + eval(x,y - 1);
5    return count_data[x][y];
   }
6  long count_paths(void)
   {
7    for(int x = 0;x < 8;x++)
7      for(int y = 0;y < 8;y++)
7        count_data[x][y] = -1L;
8    count_data[0][0] = 1L;
9    return eval(7,7);
   }
```

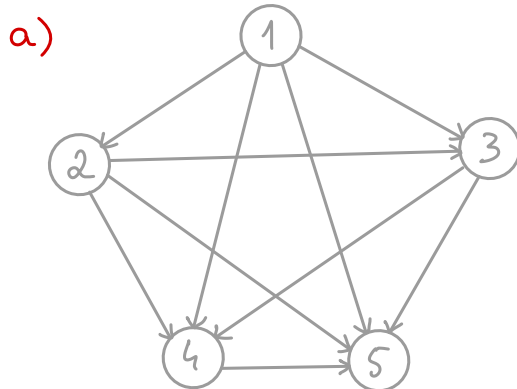
Responda às seguintes perguntas:

- 1.0 a) Que estratégia algorítmica é usada por este código? *Programação dinâmica*
3.0 b) Explique por palavras qual é o objetivo de cada uma das partes numeradas do código.

- 1 → Array bidimensional que guarda os caminhos à medida que são calculados
2 → "Cabeçalho" da função que retorna o número de caminhos pretendidos, função eval
3 → Verifica se a posição de chamada da função eval está fora do tabuleiro
4 → Se um caminho ainda não tiver sido calculado, procederemos ao cálculo e ao respetivo salvamento no array count_data
5 → Retorna o valor pretendido da caminhada, através da valores guardados na lista count_data
6 → "Cabeçalho" da função count_paths
7 → Inicialização da valores da lista count_data com um valor fora da parâmetros
8 → Indicação de chegada à posição final
9 → Retorna o número de caminhos possíveis desde a posição (7,7) até a posição (0,0) através da função eval

4.0 **4:** Um grafo com 5 vértices, numerados de 1 a 5, tem uma aresta entre o vértice número i e o número j se e só se $i < j$. Responda às seguintes perguntas:

- 1.0 a) Desenhe o grafo.
 1.0 b) Represente o grafo usando uma matriz de adjacência.
 1.0 c) Represente o grafo usando listas de adjacência.
 1.0 d) É possível representar este grafo usando um único inteiro de **32 bits**? Se sim, como? Se não, por que não?



b)

1	→ 2 → 3 → 4 → 5 → NULL
2	→ 3 → 4 → 5 → NULL
3	→ 4 → 5 → NULL
4	→ 5 → NULL
5	→ NULL

c)

	1	2	3	4	5
1	0	0	0	0	0
2	1	0	0	0	0
3	1	1	0	0	0
4	1	1	1	0	0
5	1	1	1	1	0

d) Toma 5 vértices, logo precisa de $5 \times 5 = 25$ bits para representar a matriz de adjacência

4.0 **5:** Você foi capturado pelos Borg e levado para um cubo Borg para ser assimilado.¹ Você conseguiu fugir mas está perdido dentro do cubo Borg, que pode ser considerado um labirinto tridimensional. Responda às seguintes perguntas:

- 2.0 a) Para encontrar uma saída do cubo, usaria *depth search* ou *breadth search*?
 2.0 b) Que material levaria consigo para o ajudar implementar a estratégia algorítmica que escolheu na alínea anterior?

a) Usaria *depth search*, pois eu teria algum método de teletransporte ou iria ter que andar imenso.

b) Levaria alguma coisa para marcar os lugares por onde já tiveres passado

¹Os Borg são uma raça alienígena do universo *Star Trek*. As suas maiores naves são os cubos Borg, que têm o volume de **27** quilómetros cúbicos. A resistência é fútil!