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

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

Justifique todas as suas respostas.

Nome: Andri Almeida Oliveira

N. Mec.: 107637

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 **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)`? De 3 a 1000

2.0 **b)** que valores de `i` são impressos? 3 a 1000

3.0 **2:** Ordene as seguintes funções por ordem crescente de ritmo de crescimento.

Número da função	função	termo dominante	ordem
1	$1.7^n + n^{1.5}$	$1,7^n$	4
2	$n^2 + n\log^9 n + \frac{1000}{n}$	n^2	2
3	$\frac{n!}{2.4^n}$	$\frac{n!}{2,4^n}$	5
4	$n^{1.7} + 1.5^n$	$1,5^n$	3
5	$n\log n + n\sqrt{n}$	$n\sqrt{n}$	1

2.5 **3:** No seguinte código,

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

qual é o valor de `b[3]`? b[3] seria um hipotético a[7+4] = a[11] que não existe

3.0 **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" é usada para expressar o limite superior de complexidade de um algoritmo, ou seja, o pior caso do mesmo.
A notação "Big Theta" usa a média da complexidade do algoritmo, ou seja, generalidade do caso.
A primeira é mais usada, pois sabendo o limite superior da complexidade de um algoritmo, é útil para comparar com outros algoritmos escolhendo, assim, o mais adequado a usar.

5.0 **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) $\sum_{i=0}^{m-1}\sum_{j=0}^{4}\sum_{k=0}^{j} 1 = \sum_{i=0}^{m-1}\sum_{j=0}^{4}\sum_{k=1}^{j+1} 1 = \sum_{i=0}^{m-1}\sum_{j=0}^{4}(j+1) =$

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

2.5 **a)** quantas vezes é executada a linha `r1 *= k;`? 15m

2.5 **b)** que valor é devolvido pela função? 0

2.5 **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.)

```
for (int i = 0; i < m; i++)
  for (int j = 0; j < m; j++)
    for (int k = 0; k < m; k++)
      printf ("ola");
```

Irá realizar $m \times m \times m = m^3$ instruções

Segundo teste de **Algoritmos e Estruturas de Dados**

12 de Novembro de 2018 **Duração máxima: 1 hora**

Justifique todas as suas respostas.

3.0 **1:** Explique como pode usar uma lista simplesmente ligada para implementar uma pilha. Que vantagens e desvantagens tem uma implementação deste tipo quando comparada com uma outra que usa um *array*?

3.0 **2:** Um *array* circular pode ser utilizado para implementar uma fila. Explique como. Que vantagens e desvantagens tem uma implementação deste tipo quando comparada com uma outra que usa uma lista ligada?

3.0 **3:** Explique como está organizado um *max-heap* e como se insere informação nesta estrutura de dados. (Para explicar a inserção, pode usar como exemplo inserir os números 1, 2, 3, 4 e 5, num *max-heap* inicialmente vazio.)

2.0 **4:** 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?

4.0 **5:** Numa aplicação que pretende contar o número de ocurrências de palavras num texto extenso usando uma *hash table* (tabela de dispersão, dicionário) um aluno, com pouca experiência nestas coisas, usou a seguinte *hash function*:

```
#define hash_table_size 1000003u

unsigned int hash_function(unsigned char *s)
{
  unsigned int sum;
  for(sum = 0;*s != '\0';s++)
    sum += (unsigned int)(*s);
  return sum % hash_table_size;
}
```

Sabe-se que o número de palavras distintas que existem no texto é inferior mas próximo de um milhão, e que o número de letras médio de uma palavra é inferior a 10.

2.0 **a)** Explique porque é que neste caso deve usar uma implementação da *hash table* que usa *separate chaining*, em vez de usar uma que usa *open addressing*.

2.0 **b)** A *hash function* apresentada acima é muito má. Porquê? Sugira uma outra que seja bem melhor.

Nas duas perguntas seguintes sobre árvores binárias, cada nó da árvore usa a seguinte estrutura de dados:

```
typedef struct tree_node
{
  struct tree_node *left;
  struct tree_node *right;
  long data;
}
tree_node;
```

2.0 **6:** A seguinte função é uma implementação **errada** de uma função recursiva que procura informação numa árvore binária **não ordenada**.

```
tree_node *tree_search(tree_node *n,long data)
{
  if(n == NULL || n->data == data)
    return NULL;
  return tree_search((data < n->data) ? n->left : n->right,data);
}
```

Explique o que está errado na função e corrija-a.

3.0 **7:** Escreva uma função que indique se uma árvore binária está balanceada ou não. Recorda-se que numa árvore binária balanceada as alturas das sub-árvores dos lados direito e esquerdo não podem diferir em mais de 1.

```
int is_balanced(tree_node *n)
{
  // put your code here
}
```

Considere que se o valor devolvido pela função for -1 então a árvore não está balanceada.

1 - Uma fila é uma estrutura de dados do tipo LIFO (last in, first out), o que significa que para retirarmos um elemento teremos de retirar todos os que foram armazenados na fila depois do mesmo. Para adicionar um elemento, precisamos de colocar a head da linked list a apontar para o novo elemento e colocar o novo elemento a apontar para o elemento que antes era o primeiro elemento da linked list. Para remover um elemento, colocamos a head a apontar para o elemento que o elemento que vamos remover apontava

Como vantagens, uma linked list é dinâmica ao contrário de um array que tem tamanho fixo, mas há desperdício de memória como nos arrays e a inserção e remoção de elementos é mais rápido.

Como desvantagens, a memória de um array é alocada faseiramente enquanto que, numa linked list, temos de constantemente atualizá-la, o que torna o processo mais lento e o acesso aleatório a um elemento de uma linked list é uma operação com complexidade $O(n)$.

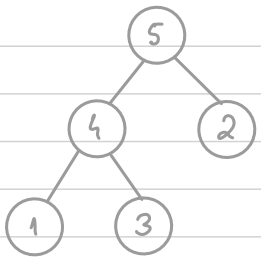
2 - Uma fila é uma estrutura do tipo FIFO (first in, first out). Para realizarmos a sua implementação necessitaremos de dois ponteiros, um de escrita que apontará sempre para o final da fila, de modo que seja possível a adição de elementos e um de leitura que aponte para o início da fila, lendo sempre primeiro os que foram adicionados em primeiro.

Como vantagens, um array circular permite a reutilização de espaço já alocado, evitando a necessidade de realocar memória quando a fila estiver cheia. Além disso, as operações de inserção e remoção são simples e rápidas, pois não envolvem movimentação de elementos na memória.

Como desvantagens, uma linked list permite a inserção e remoção de elementos em qualquer posição, enquanto que num array circular essas operações só podem ser realizadas no começo e no fim da fila. Além disso, a implementação de uma linked list pode ser mais flexível e escalável, enquanto que num array circular tem um tamanho fixo.

3 - Um max-heap é uma árvore binária em que cada nó é maior que a seus ramos.

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



4 - 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 pior dos casos 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.

5 - a) A hash function apresentada, não é capaz de distinguir anagramas e retorna uma gama baixa de valores, logo haverá imensas colisões. O método de separate chaining usa linked lists para os elementos que têm o mesmo retorno da hash function, resolvendo, assim, o problema das colisões.

b) Como já referido, não distingue anagramas e retorna uma gama de valores relativamente baixa, não distribuindo uniformemente as keys pela hash table.

Uma mudança poderá ser `sum += (unsigned int)(*s) * hash_table_size * sum;`

Terceiro teste de **Algoritmos e Estruturas de Dados**

10 de Dezembro de 2018 **Duração: 1 hora**

Justifique todas as suas respostas.

4.0 **1:** Uma procura em profundidade (*depth-first*) é geralmente mais fácil de implementar que uma em largura (*breadth-first*). Explique porquê, e indique as principais vantagens e desvantagens de cada destas duas maneiras de efetuar uma pesquisa exaustiva.

4.0 **2:** Num cubo $n \times n \times n$, em que as coordenadas começam em 0 e acabam em $n-1$, pretende-se contar o número de maneiras de ir da posição $(0,0,0)$ até à posição $(n-1, n-1, n-1)$. Os movimentos válidos são deslocamentos de uma única posição numa das coordenadas, sempre na direção do destino:

$$(\Delta x, \Delta y, \Delta z) = (1,0,0) \text{ ou } (0,1,0) \text{ ou } (0,0,1).$$

Além disso, as coordenadas de todos os pontos do caminho têm de satisfazer as condições

$$0 \leq z \leq y \leq x < n.$$

1.0 **a)** Que técnica algoritmica é recomendada para resolver este problema?

2.0 **b)** Escreva código para resolver este problema (para um valor de n fixo, por exemplo 10).

1.0 **c)** Qual a complexidade computacional da sua solução?

4.0 **3:** Explique como funciona o *merge sort*. Diga também quais são os melhores e piores casos para este algoritmo de ordenação, e mostre que a sua complexidade computacional é $O(n \log n)$. Use o *master theorem*, que 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 $T(n) = O(n^{\log_b a} \log n)$, ou
- 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 **4:** Um grafo tem vértices numerados de 1 a n. Existe uma aresta entre os vértices i e j se i for múltiplo de j ou se j for múltiplo de i (considere que um número não é múltiplo de si próprio).

1.0 **a)** Para $n = 6$, desenhe o grafo.

1.0 **b)** Represente o grafo da alínea a) usando uma matriz de adjacência.

1.0 **c)** Represente o grafo da alínea a) usando listas de adjacência.

1.0 **d)** Para um n genérico, qual das duas representações é melhor?

4.0 **5:** A técnica de programação dinâmica pode ser usada para resolver o problema do caixeiro viajante (*traveling salesman problem*). Explique como, e indique qual a sua complexidade computacional.

1 - A depth first search é mais simples de implementar do que a breadth first search uma vez que, pode ser implementada recursivamente, enquanto que a breadth first search requer uma estrutura de dados adicional, como uma fila, para manter o estado dos nós a serem explorados.
As principais vantagens da depth first search incluem a falta de necessidade de uma estrutura de dados adicional e a capacidade de encontrar caminho mais curto num grafo. As desvantagens incluem a possibilidade de cair em um loop infinito se o grafo contiver ciclos e a possibilidade de exceder a pilha de recursão para grafos muito profundos.
As principais vantagens da breadth first search incluem a garantia de encontrar o caminho mais curto entre dois nós em um grafo não-ponderado e a capacidade de evitar loops infinitos. As desvantagens incluem a necessidade de usar uma estrutura de dados adicional para manter o estado dos nós a serem explorados e a possibilidade de se tornar ineficiente para grafos muito profundos.

2 - b) Programação dinâmica

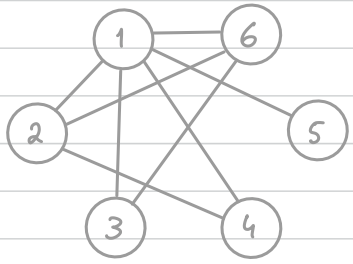
c) $O(m^3)$

3 - O merge sort é um algoritmo de ordenação em que, se o número de elementos de um array for menor que um certo limiar definido, aplicamos outro algoritmo de ordenação como o insertion sort.
Caso contrário, o array é dividido ao meio recursivamente até obtermos arrays com número de elementos abaixo do limiar definido. Os arrays pequenos, já ordenados, vão juntar-se de forma ordenada também usando um array auxiliar.

$a = b$ sempre, logo $T(m) = O(m^{\log_b a} \log m) = O(m \log)$, pois $\log_b a = 1$

Concluímos que a complexidade do melhor e do pior caso são iguais, sendo o pior caso um array na ordem inversa e o melhor caso um array já ordenado

4 - a)



b)

	1	2	3	4	5	6
1	0	1	1	1	1	1
2	1	0	0	1	0	1
3	1	0	0	0	0	1
4	1	1	0	0	0	0
5	1	0	0	0	0	0
6	1	1	1	0	0	0

c)

1	2 -> 3 -> 4 -> 5 -> 6 -> NULL
2	1 -> 4 -> 6 -> NULL
3	1 -> 6 -> NULL
4	1 -> 2 -> NULL
5	1 -> NULL
6	1 -> 2 -> 3 -> NULL

d) A matriz, pois através das posições da matriz é fácil saber quais são multiplos de quais e atualizá-la.