Universidade Tecnológica Federal do Paraná (UTFPR)
Departamento Acadêmico de Informática (DAINF)
Estrutura de Dados I
Professor: Rodrigo Minetto
Lista de exercícios (pesquisa de elementos)

Exercícios (seleção): necessário entregar TODOS (moodle)!

Exercício 1) Implemente três formas de pesquisa de elementos, especificamente, escreva uma função para **busca linear**, e uma versão **iterativa** e outra **recursiva** do algoritmo **busca binária**. Para avaliar a corretude dos seus códigos, pesquise por palavras em um dicionário (em anexo ao material da aula), teste com as seguintes palavras:

```
Pesquisa:          Saída desejada:
ACADEMICO          índice: 202
FORNECEDOR         índice: 13398
ZOOLOGIA           índice: 29840
RODRIGO            índice: -1
```

Para este exercício, codifique a funcionalidade que falta no programa **word-search.c** (em **arquivos.zip**). Para comparar uma string, utilize a função **strcmp** da biblioteca **string.h**.

Exercício 2) Suponha uma lista ordenada A com n números inteiros distintos, que estão no intervalo de 0 a $m-1$, tal que $m > n$. Escreva uma função $\mathcal{O}(\log n)$ para determinar o menor número inteiro que não está em A. Por exemplo:

```
Entrada                  Saída

A: {0,1,2,6,9}             3
n: 5, m: 10

A: {4,5,10,11}             0
n: 4, m: 12

A: {0,1,2,3}               4
n: 4, m: 5

A: {0,1,2,3,4,5,6,7,10}    8
n: 9, m: 11
```

Para este exercício, codifique a funcionalidade que falta no programa **missing.c** (em **arquivos.zip**).

Exercício 3) Escreva uma função para contar o número de ocorrências de um dado elemento k em um vetor ordenado A. Por exemplo, suponha que $A = \{2, 5, 5, 5, 6, 6, 8, 9, 9, 9\}$ e que $k = 5$, então sua função deve retornar como resultado o valor 3. **Importante**: sua função deve ter complexidade $\mathcal{O}(\log n)$. Portanto, percorrer o vetor do início ao fim em busca de k tem complexidade $\mathcal{O}(n)$ e não atende os requisitos. Utilizar uma busca binária simples também não, pois suponha que $k = 5$ e $A = \{5, 5, 5, 5, 5, 5, 5, 5, 5, 5\}$, nesse caso a busca retorna qualquer índice e é necessário percorrer os elementos à esquerda e direita de k o que também incorre em um custo $\mathcal{O}(n)$, que também não atende os requisitos. Utilize o seguinte protótipo para a sua função:

```
int find_occurrences (int *A, int n, int k);
```

Para este exercício, codifique a funcionalidade que falta no programa **occurrences.c** (em **arquivos.zip**).

Exercícios (aprofundamento): escolha UM para entregar!

Exercício 4) Qual a complexidade, utilizando a notação assintótica $\mathcal{O}(?)$, para as seguintes questões:

	Melhor caso	Pior caso
Busca linear		
Busca binária		

Exercício 5) Faz sentido realizar uma busca binária em uma **lista encadeada** (simples ou dupla) com elementos em ordem crescente?

Exercício 6) Dados dois arrays de inteiros desordenados A e B de tamanho m e n, respectivamente, encontre a interseção entre eles. A interseção de dois arrays é uma lista de números distintos que estão presentes em ambos os arrays. Os números na interseção podem estar em qualquer ordem. Por exemplo, se $A = \{1, 4, 3, 2, 5, 8, 9\}$ e $B = \{6, 3, 2, 7, 5\}$, então os elementos de intersecção são 2, 3 e 5 (basta imprimir na tela não precisa salvar). O seu código deve ter complexidade $\mathcal{O}(m \log m + n \log n)$. Utilize o seguinte protótipo para a sua função:

```
void intersection (int *A, int m, int *B, int n);
```

Observação: se você utilizar cada elemento em A para procurar em B o seu código tem complexidade $\mathcal{O}(m \times n)$.

Exercício 7) Uma questão clássica em entrevistas é a seguinte: suponha que você é um gerente de produto e atualmente lidera uma equipe para desenvolver um novo produto, enquanto alguns membros da equipe trabalham en um branch, outros trabalham em outros branches do produto, e no final todos fazem um merge no branch master. O merge é feito várias vezes por semana. Infelizmente, é observado que uma aplicação que antes funcionava perfeitamente pára de funcionar, mas ninguém sabe dizer quem foi o responsável nem quando aconteceu. Como cada versão é desenvolvida com base na versão anterior, então todas as versões posteriores também contém a falha. As n versões $\{1, 2, \ldots, n\}$ estão armazenadas no git e desejamos descobrir a primeira versão defeituosa. Escreva um algoritmo eficiente ($\mathcal{O}(\log n)$) para resolver este problema. Lembre-se que no git (ou em banco de dados) é possível recuperar qualquer uma das n versões salvas antes de um commit.

Para simplificar o problema acima, podemos rescrevê-lo da seguinte forma: suponha um array binário $A = \{0, 0, 0, 0, 1, 1, 1\}$, tal que 0 no nosso contexto é uma versão sem o bug e 1 com o defeito. Localize de forma eficiente a primeira ocorrência com valor 1. Suponha que o defeito existe senão você não tentaria localizar.

Utilize o seguinte protótipo para a sua função:

```
int bad_commit (int *A, int n);
```

Algumas entradas com respectivas saídas:

Entrada	Saída
A: {0,0,0,0,0,0,1}	6
A: {1,1,1,1,1,1,1}	0
A: {0,1,1,1,1,1,1}	1

Exercício 8) Escreva uma função que usa busca binária para determinar se um número inteiro n é ou não um **quadrado perfeito**. Você **não** pode utilizar a função raiz para resolver este exercício. A definição para um número quadrado perfeito é dada por: um número natural inteiro positivo cuja raiz quadrada é, também, um número natural inteiro positivo. Utilize o seguinte protótipo para a sua função:

```
int perfect_square (int n);
```

A saída do seu código pode ser um loop que imprime todos os números que são quadrado perfeito de 1 a 100, por exemplo:

```
Número 1 é um quadrado perfeito.
Número 4 é um quadrado perfeito.
Número 9 é um quadrado perfeito.
Número 16 é um quadrado perfeito.
Número 25 é um quadrado perfeito.
Número 36 é um quadrado perfeito.
Número 49 é um quadrado perfeito.
Número 64 é um quadrado perfeito.
Número 81 é um quadrado perfeito.
```