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

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

Exercício 1) Codifique e execute os algoritmos bubble-sort, selection-sort e insertion-sort para ordenar um quantidade de 10, 100, 1.000, 10.000, 100.000 e 200.000 elementos em ordem **aleatória**. Indique na tabela abaixo o tempo para cada uma das entradas acima. **Não inclua** no tempo de execução a impressão dos elementos na tela (comente a função **print** para estes testes de performance).

Algoritmo	10	100	1.000	10.000	100.000	200.000
Bubble-Sort						
Selection-Sort						
Insertion-Sort						

Para auxiliar implemente a funcionalidade que falta conforme indicado nos programas **bubble.c, selection.c** e **insertion.c** (em **arquivos.zip**).

Exercício 2) Implemente uma versão recursiva do algoritmo **insertion-sort**. Não é necessário eliminar todos os iteradores, basta escolher **um loop** para transformar em recursão — para isso, observe qual deles é menos acoplado ao código. Utilize o seguinte protótipo para a sua função:

```
void insertion_sort_recursive (int *A, int n);
```

Não altere o protótipo acima, ou seja, não adicione outras variáveis como argumentos de entrada na função para facilitar o projeto da recursão.

Exercício 3) A filtragem de imagens visando a redução do ruído é uma tarefa muito importante em processamento de imagens, e encontra diversas aplicações nas áreas de sensoriamento remoto, inspeção industrial e na medicina. O filtro da mediana é uma transformação bastante comum para suavizar ruídos do tipo impulsivo (sal-e-pimenta) em sinais e imagens digitais.



Antes de definir o filtro da mediana para imagens vamos definir o conceito de vizinhança: seja M uma matriz de inteiros positivos com m linhas e n colunas, e seja p um inteiro positivo ímpar. Dada uma coordenada (i, j) em M, a vizinhança de tamanho $p \times p$ em torno de (i, j) é a submatriz $M_{i,j}$ de M com p linhas e p colunas e centro em (i, j). Por exemplo, dada a seguinte matriz 5×5

$$M = \begin{pmatrix} 10 & 18 & 21 & 0 & 8 \\ 1 & 7 & 2 & 7 & 1 \\ 0 & 9 & 34 & 23 & 12 \\ 16 & 18 & 6 & 15 & 19 \\ 1 & 18 & 3 & 23 & 4 \end{pmatrix} \tag{1}$$

a vizinhança 3×3 em torno da coordenada $(1, 1)$ é a submatriz

$$M_{1,1} = \begin{pmatrix} 10 & 18 & 21 \\ 1 & 7 & 2 \\ 0 & 9 & 34 \end{pmatrix} \tag{2}$$

Note que a vizinhança não é bem definida em algumas coordenadas (por exemplo, em $(0, 0)$).

Uma imagem digital pode ser representada por uma matriz. Dada uma matriz M de inteiros positivos com m linhas e n colunas, e um inteiro positivo ímpar p, o filtro da mediana calcula uma matriz F com o mesmo tamanho de M, de forma que $F(i, j)$ contém a mediana dos números em $M(i, j)$ (a vizinhança $p \times p$ em torno de (i, j)).

No caso do exemplo anterior, os números em torno de (1, 1) são 10, 18, 21, 1, 7, 2, 0, 9, 34. Logo, $F(1, 1) = 9$ (valor da mediana em 0, 1, 2, 7, 9, 10, 18, 21, 34). Quando a vizinhança de uma coordenada (i, j) não estiver bem definida, usaremos a convenção $F(i, j) = 0$.

No caso da matriz-exemplo da seção anterior, o resultado do filtro da mediana com uma vizinhança de 3×3 é a seguinte matriz:

$$F = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 9 & 9 & 12 & 0 \\ 0 & 7 & 9 & 12 & 0 \\ 0 & 9 & 18 & 15 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{3}$$

Neste exercício utilizaremos o formato PGM (portable graymap) para armazenar imagens em arquivos. Este formato tem duas variações, uma binária (o PGM "normal" ou raw) e outra textual (o PGM ASCII ou plain). Em ambos os casos, o arquivo deve conter um cabeçalho e a matriz correspondente à imagem. O exemplo a seguir mostra um arquivo PGM textual:

```
P2
5 4
34
10  18  21   0   8
 1   7   2   7   1
 0   9  34  23  12
16  18   6  15  19
```

A primeira linha do arquivo contém obrigatoriamente uma palavra-chave, no caso desse exercício ela deve ser P2 (arquivo PGM textual). A segunda linha contém dois números inteiros que indicam o número de colunas e o número de linhas da matriz, respectivamente. A terceira linha contém um número inteiro positivo maxval, que deve ser igual ao maior elemento da matriz. Na definição do formato PGM, maxval não pode ser maior que 65535. Para fins deste exercício, entretanto, maxval é no máximo 255. Os demais números do arquivo são os elementos de uma matriz de inteiros com os tons de cinza de cada ponto da imagem. Cada tom de cinza é um número entre 0 e maxval, com 0 indicando "preto" e maxval indicando "branco". Para quem usa o sistema operacional Windows use o programa IrfawView para visualizar as imagens de exemplo.

Neste exercício, uma parte da lógica para implementar a filtragem pelo ruído sal-e-pimenta em uma imagem já está codificada. No entanto, falta terminar o deslocamento da máscara sobre a imagem e inserir os pixels de cada um destes deslocamentos em um array para ordenar. Também é necessário descrever qual a ideia por trás do uso de ordenação para resolver esse problema. Avalie o efeito ao utilizar diversos tamanhos de vizinhança para filtragem, 5×5, 9×9, 17×17, ... Essa avaliação pode ser descrita dentro do próprio código.

Utilize o material de apoio em **ruido.c** (dentro de **arquivos.zip**) para auxiliá-lo. O diretório "arquivos" também contém 3 exemplos de imagens com ruído sal-e-pimenta. Teste seu algoritmo com todas elas.

Exercícios (aprofundamento): não é necessário entregar mas é importante estudar!

Exercício 4) Implemente uma versão recursiva do algoritmo **selection-sort**. Não é necessário eliminar todos os iteradores, basta escolher **um loop** para transformar em recursão — para isso, observe qual deles é menos acoplado ao código. Utilize o seguinte protótipo para a sua função:

```
void selection_sort_recursive (int *A, int n);
```

Não altere o protótipo acima, ou seja, não adicione outras variáveis como argumentos de entrada na função para facilitar o projeto da recursão.

Exercício 5) Implemente uma versão recursiva do algoritmo **bubble-sort**. Não é necessário eliminar todos os iteradores, basta escolher **um loop** para transformar em recursão — para isso, observe qual deles é menos acoplado ao código. Utilize o seguinte protótipo para a sua função:

```
void bubble_sort_recursive (int *A, int n);
```

Não altere o protótipo acima, ou seja, não adicione outras variáveis como argumentos de entrada na função para facilitar o projeto da recursão.

Exercício 6) Para cada um dos algoritmos abaixo, indique a complexidade utilizando a notação $\mathcal{O}(?)$ para o pior e melhor caso; indique a organização (**org**) dos elementos para quando o pior e melhor acontecem (por exemplo, valores em ordem crescente, ou decrescente, ou aleatório; e comprove as informações fornecidas ao executar o algoritmo e medir o tempo t (em minutos) para ordenar 100 mil elementos tanto para o pior quanto para o melhor caso).

Algoritmo	Pior $\mathcal{O}(?)$	Pior (org.)	Pior (t)	Melhor $\mathcal{O}(?)$	Melhor (org.)	Melhor (t)
Bubble-Sort						
Selection-Sort						
Insertion-Sort						

Considere as versões dos algoritmos acima conforme vistas em aula, ou seja, não otimizadas para um melhor ou pior caso.

Exercício 7) Implemente uma versão alternativa do algoritmo **bubble-sort** cuja ideia é deslocar o maior elemento da iteração atual até o final do array e no retorno, deslocar o menor elemento até o início do array. Suponha por exemplo o seguinte array $A = \{5, 4, 3, 2, 1\}$, então o processo de ordenação ocorre da seguinte forma:

```
I)  5 4 3 2 1    entrada/input
1)  4 3 2 1 5    deslocando maior
2)  1 4 3 2 5    deslocando menor
3)  1 3 2 4 5    deslocando maior
4)  1 2 3 4 5    deslocando menor
```

Utilize o seguinte protótipo para a sua função:

```
void bubble_sort_bidirecional (int *A, int n);
```

Exercício 8) Implemente uma versão do algoritmo **selection-sort** para ordenação decrescente, Por exemplo, se o array de entrada for $A = \{2, 9, 0, 6, 5\}$, então o array após ordenação é dado por $A = \{9, 6, 5, 2, 0\}$.

Utilize o seguinte protótipo para a sua função:

```
void selection_sort_descending_order (int *A, int n);
```

Exercício 9) Seja $A[0...n-1]$ um array de números inteiros, todos diferentes entre si. A **mediana** de A é um elemento que é maior que metade dos elementos e menor que (a outra) metade dos elementos. Quando o número elementos de um conjunto é par, a mediana é encontrada pela média dos dois valores centrais. Não confunda mediana com média. Por exemplo, a média de $\{99, 1, 2\}$ é 51, enquanto a mediana é 2. Escreva um algoritmo que encontre a mediana em A.

Utilize o seguinte protótipo para a sua função:

```
double median (int *A, int n);
```

Exercício 10) Um algoritmo de ordenação é estável (**stable**) se não altera a posição relativa de elementos com a mesma chave. Digamos, por exemplo, que um array é formado através de uma struct com dois campos: o primeiro contém o nome de uma pessoa e o segundo contém o ano de nascimento desta pessoa. Suponha que o array original tem dois "João da Silva", primeiro o que nasceu em 1980 e depois o que nasceu em 2020. Se o array for ordenado por um algoritmo estável com base no primeiro campo, os dois "João da Silva" continuarão na mesma ordem relativa: primeiro o de 1980 e depois o de 2020, já um algoritmo não estável pode alterar a ordem dos dois. Classifique os algoritmos bubble, selection e insertion em estáveis ou não.

Se facilitar, utilize o programa "**stable.c**' para realizar os experimentos.

Exercício 11) Considere o seguinte jogo: dado um número inteiro n, realize o sorteio de uma sequência de números aleatórios com tamanho n. O jogo acontece através de disputas alternadas entre dois competidores A e B, tal que o objetivo a cada rodada é escolher um par de elementos consecutivos que estão fora de ordem na sequência e trocar ambos os elementos. Por exemplo, dada a sequência $\{1, 5, 3, 4, 2\}$, um jogador pode trocar 3 e 5 ou 4 e 2, mas não pode trocar 3 e 4, nem 5 e 2. Continuando com o exemplo, se o jogador A decidir trocar 5 e 3, a nova sequência será $\{1, 3, 5, 4, 2\}$, e o jogo continua com o jogador B. Mais cedo ou mais tarde a sequência será ordenada. O jogador que não puder fazer uma nova jogada perde. Considere que o jogador A sempre inicia o jogo.

Escreva um programa tal que dado A determina qual jogador vence e o número total de movimentos (**rounds**) feitos pelos dois jogadores. Por exemplo:

```
Sequência (entrada)      Vencedor       Tot. rounds
{1,5,3,4,2}                 A               5
{5,1,3,4,2}                 B               6
```

```
{1,2,3,4,5}                B              0
{5,4,3,2,1}                B             10
{3,5,2,1,4,6}              B              6
{6,5,4,3,2,1}              A             15
```

Utilize o seguinte protótipo para a sua função:

```
char game (int *A, int n);
```

Exercício 12) A ordem lexicográfica entre strings é análoga à ordem das palavras em um dicionário. Para comparar duas strings s e t, procura-se a primeira posição, digamos k, em que as duas strings diferem. Se $s[k]$ vem antes de $t[k]$ na tabela ISO então s é lexicograficamente menor que t. Se k não está definido então s e t são idênticas ou uma é prefixo próprio da outra; nesse caso, a string mais curta é lexicograficamente menor que a mais longa. Organize o dicionário "**palavras.txt**" em ordem lexicográfica, e use, se facilitar, o programa "**dictionary.c**", que está em "**arquivos.zip**" — anexo ao material da aula. Suponha que nenhuma palavra do dicionário tem mais de 256 caracteres. Você pode utilizar funções da biblioteca string.h se julgar necessário. Você pode utilizar qualquer um dos algoritmos de ordenação vistos em aula.