**Universidade Tecnológica Federal do Paraná (UTFPR)**
**Departamento Acadêmico de Informática (DAINF)**
**Estrutura de Dados I**
**Professor: Rodrigo Minetto**
**Lista de exercícios (Merge-Sort)**

---

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

---

---

**Exercício 1)** Implemente em linguagem C o algoritmo Merge-Sort conforme visto em aula e o utilize para ordenar sequências com 10, 100, 1.000, 10.000, 100.000, 1 milhão e 10 milhões de números inteiros em ordem:

- aleatória ($V = \{9, 3, 7, 0, \dots\}$)
- crescente ($V = \{1, 2, 3, 4, \dots\}$)
- decrescente ($V = \{9, 8, 7, 6, \dots\}$)
- todos elementos iguais ($V = \{2, 2, 2, 2, \dots\}$)

Existe alguma configuração que o algoritmo funciona mais rápido? Utilize o programa **merge-sort.c** (junto ao material em **arquivos.zip**).

---

**Exercício 2)** A análise de execução de alguns algoritmos recursivos as vezes é complexa. Uma estratégia interessante para facilitar a visualização da pilha de recursão é imprimir uma quantidade fixa de deslocamentos (vazios) para cada nível em que o programa está na pilha de recursão. Por exemplo, considere as chamadas recursivas que o algoritmo Merge-Sort realiza para um array $A = \{5, 2, 7, 4, 8, 1, 9, 8\}$ (o mesmo array utilizado como exemplo nos slides da aula)

```
Merge-Sort (0,3,7)
    Merge-Sort (0,1,3)
        Merge-Sort (0,0,1)
            Merge-Sort (0,0,0)
            Merge-Sort (1,1,1)
            Intercalando = {2, 5}
        Merge-Sort (2,2,3)
            Merge-Sort (2,2,2)
            Merge-Sort (3,3,3)
            Intercalando = {4, 7}
        Intercalando = {2, 4, 5, 7}
    Merge-Sort (4,5,7)
        Merge-Sort (4,4,5)
            Merge-Sort (4,4,4)
            Merge-Sort (5,5,5)
            Intercalando = {1, 8}
        Merge-Sort (6,6,7)
            Merge-Sort (6,6,6)
```

```
        Merge-Sort (7,7,7)
        Intercalando = {8, 9}
    Intercalando = {1, 8, 8, 9}
Intercalando = {1, 2, 4, 5, 7, 8, 8, 9}
```

Considere acima que cada chamada ao algoritmo Merge-Sort exibe os índices (esquerda, meio, direita). Note que os deslocamentos para a direita mostram os "filhos" de cada chamada recursiva. Para este exercício, codifique o programa **merge-debug.c** (junto ao material em **arquivos.zip**), para explorar as chamadas recursivas conforme mostrado no exemplo acima.

---

**Exercício 3)** O algoritmo Merge-Sort que estudamos em aula utiliza um array auxiliar (vetor $O$, da palavra *output*, conforme visto em aula) durante a ordenação (versão não in-place, ou seja, necessita de outro vetor para ordenar os elementos, não conseguindo assim ordenar os valores dentro do próprio vetor desordenado). No entanto, o uso de memória adicional é indesejável devido ao custo de armazenamento associado, e tentativas de otimizações são bem vindas. Avalie o funcionamento de duas otimizações conforme descrito nos itens abaixo:

```
     Ae         Ad
A = {1, 3, ..., 2, 5, ...}
     i          j
```

- ao intercalar dois segmentos ordenados $A_e$ (segmento da esquerda) e $A_d$ (segmento da direita) de um array $A$, se o menor elemento está na participação $A_e$ então avance uma posição em $A_e$ (incremento da variável $i$ no esquema acima); se o menor elemento está em $A_d$ então realize uma troca de elementos entre $A_e$ e $A_d$ e avance uma posição tanto em $A_e$ quanto $A_d$ (incremento das variáveis $i$ e $j$).
- ao intercalar dois segmentos ordenados $A_e$ (segmento da esquerda) e $A_d$ (segmento da direita) de um array $A$, se o menor elemento está na participação $A_e$ então avance uma posição em $A_e$ (incremento da variável $i$ no esquema acima); se o menor elemento está em $A_d$ então realize uma troca de elementos entre $A_e$ e $A_d$ e avance uma posição apenas em $A_e$ (incremento da variável $i$).

Discuta se cada uma das otimizações acima propostas acima funcionam adequadamente. Considere que a função principal do algoritmo Merge-Sort não tem qualquer modificações (é exatamente conforme vista em aula).

---

**Exercício 4)** Explique se a função alternativa abaixo — para intercalar dois segmentos ordenados de um array — funciona. O que ela difere da que vimos em sala? Ela utiliza o vetor auxiliar $O$? A complexidade de tempo dela é pior ou melhor que a vista em sala? Considere que a função principal do algoritmo Merge-Sort não tem qualquer modificações (é exatamente conforme vista em aula).

```
void intercala (int A[], int l, int m, int r) {
  while (m <= r) {
    int c = A[m], i;
    for (i = m-1; (i >= l) && (A[i] > c); i--)
```

```
      A[i+1] = A[i];
    A[i+1] = c;
    m++;
  }
}
```

---

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

---

**Exercício 5)** Vimos em aula que o algoritmo Merge-Sort considera as seguintes partições para o array $A = \{5, 2, 7, 4, 8, 1, 9, 8\}$ durante a ordenação:

```
           {5,2,7,4,8,1,9,8}
     {5,2,7,4}             {8,1,9,8}
  {5,2}     {7,4}     {8,1}     {9,8}
{5}   {2} {7}   {4} {8}   {1} {9}   {8}
  {2,5}     {4,7}     {1,8}     {8,9}
     {2,5,4,7}             {1,8,8,9}
           {1,2,4,5,7,8,8,9}
```

O que acontece quando o número de elementos em $A$ é ímpar?

- Mostre as partições do Merge-Sort para $A = \{5, 2, 7, 4, 8, 1, 9, 8, 3\}$.
- Mostre as partições do Merge-Sort para $A = \{5, 2, 4, 8, 1, 9, 8\}$.

---

**Exercício 6)** O que acontece se a comparação "$A[i] \leq A[j]$" for trocada por "$A[i] < A[j]$" na linha 5 do algoritmo intercala?

```
void Intercalar (int A[], int l, int m, int r, int O[]) {
 1.  int i = l;
 2.  int j = m + 1;
 3.  int k = l;
 4.  while ( (i <= m) && (j <= r) ) {
 5.    if (A[i] <= A[j])
         ...
```

---

**Exercício 7)** Modifique o algoritmo Merge-Sort para determinar o número de inversões em um array com tempo proporcional a $O(n \log n)$ no pior caso. O número de inversões de um array $A[0, \ldots, n-1]$ consiste no número de pares ordenados $(i, j)$, onde $0 \leq i < j < n$, tal que $A[i] > A[j]$. Por exemplo, o array $A = \{2, 4, 1, 3, 5\}$ tem três inversões $(2, 1), (4, 1), (4, 3)$.

---

**Exercício 8)** É dado um número inteiro $s$ e um array $A[0,\ldots,n-1]$ de números inteiros. Desenvolva um algoritmo que determina se há dois números em $A$ cuja soma seja exatamente $s$, ou seja, devolve **true** (1) caso existam o par de números e **false** (0) caso contrário. A complexidade do algoritmo deve ser $O(n \log n)$ no pior caso. Dica. o exercício requer apenas um par de números caso exista mais de um.

---

**Exercício 9)** Na fase de divisão, o algoritmo Merge-Sort divide o array de entrada em segmentos de comprimento unitário (critério de parada para a recursão). Um algoritmo alternativo seria a interrupção desta divisão quando os segmentos do array fossem crescentes maximais. Por exemplo, seja $A = \{1,2,3,4,2,4,6,4,5,6,7,8,9\}$, então os segmentos crescentes maximais de $A$ são $\{1,2,3,4\}$, $\{2,4,6\}$ e $\{4,5,6,7,8,9\}$. Explore essa ideia em sua implementação.

---

**Exercício 10)** A distância de classificação tau de Kendall é uma métrica que conta o número de desacordos entre pares entre duas listas de classificação. A distância é igual a 0 se as duas listas forem idênticas, e quanto maior a distância, mais diferentes são as duas listas. Suponha duas permutações, digamos $X[0,\ldots,n-1]$ e $Y[0,\ldots,n-1]$, de um mesmo conjunto de números (em um intervalo de 0 até $n-1$ sem repetições). A distância tau entre $X$ e $Y$ é o número de pares de elementos do conjunto que estão em ordem diferente em $X$ e $Y$, por exemplo

```
X = {0,3,1,6,2,5,4}
Y = {1,0,3,6,4,2,5}
Distância tau = 4
```

pois os pares $(0,1)$, $(3,1)$, $(2,4)$ e $(5,4)$ estão em ordens diferentes nos rankings em $X$ e $Y$, ou seja, o par de números $(0,1)$ aparece com o 0 antes do 1 em $X$, enquanto que em Y o 1 aparece antes do 0. Note que pares como o $(3,6)$ aparecem tanto em $X$ quanto em $Y$ com o elemento 3 antes do 6 nas sequências. A distância tau é importante para determinar o coeficiente de correlação de dois rankings por exemplo. Suponha que dois observadores fazem um ranking de 12 jogadores do pior para o melhor:

| Jogador | Obs. 1 | Obs. 2 |
|---|---|---|
| Pelé | 0 | 0 |
| Romário | 1 | 1 |
| Ronaldo | 2 | 2 |
| Bebeto | 3 | 4 |
| Ronaldinho | 4 | 3 |
| Alex | 5 | 6 |
| Messi | 6 | 5 |
| Maradona | 7 | 7 |
| Rivelino | 8 | 9 |
| Garrincha | 9 | 8 |
| Rivaldo | 10 | 10 |
| Neymar | 11 | 11 |

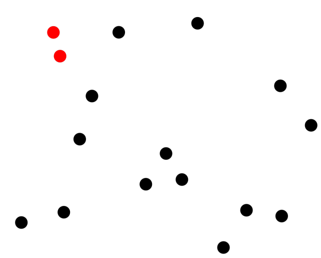
Neste caso a distância de Kendall é apropriada para computar a correlação entre os dois rankings. O número de pares em desacordo para o exemplo acima é três. A distância Kendall usa esse número. Para maiores detalhes sobre a distância Kendall consulte:

https://www.statology.org/kendalls-tau/.

Neste exercício, modifique o algoritmo Merge-Sort para calcular a **distância tau**.

---

**Exercício 11)** O problema do par de pontos mais próximo consiste em, dado um conjunto de $n$ pontos em um plano, encontrar os dois pontos do conjunto que possuem a menor distância um do outro. Este problema é muito estudado devido a aplicações em processameto gráfico, visão computacional, sistemas de processamento geográfico, modelagem molecular, controle de tráfego áreo, etc. Como exemplo de entrada considere os pontos escuros abaixo, e como solução os dois pontos em vermelho, conforme a figura abaixo.



- Descreva um algoritmo por força bruta (complexidade de tempo $O(n^2)$) para resolver o problema acima (não é necessário codificar).
- Descreva um algoritmo por divisão e conquista (semelhante ao Merge-Sort) para resolver o problema acima (não é necessário codificar e nem que ele seja melhor que o algoritmo por força bruta).