1-

Tabelas:

Número de comparações – 1000 elementos do tipo int

	Seleção	**Bolha**	**Inserção**	**Quicksort**	**Mergesort**	**Heapsort**
Vetor Crescente	499500	499500	999	7987	9976	15302
Vetor Decrescente	499500	499500	499500	6996	9976	7485
Vetor Aleatório	499500	499500	246238	6623	9976	8708

Número de movimentações – 1000 elementos do tipo int

	Seleção	**Bolha**	**Inserção**	**Quicksort**	**Mergesort**	**Heapsort**
Vetor Crescente	999	0	999	511	9976	15107
Vetor Decrescente	999	499500	500499	1010	9976	7315
Vetor Aleatório	999	245246	246245	2695	9976	8531

Tempo de Execução – 1000 elementos do tipo int

	Seleção	**Bolha**	**Inserção**	**Quicksort**	**Mergesort**	**Heapsort**
Vetor Crescente	00:00:00.00	00:00:00.00	00:00:00.00	00:00:00.00	00:00:00.00	00:00:00.00
Vetor Decrescente	00:00:00.00	00:00:00.00	00:00:00.00	00:00:00.00	00:00:00.00	00:00:00.00
Vetor Aleatório	00:00:00.00	00:00:00.00	00:00:00.00	00:00:00.00	00:00:00.00	00:00:00.00

Número de comparações – 1000 elementos do tipo decimal

	Seleção	**Bolha**	**Inserção**	**Quicksort**	**Mergesort**	**Heapsort**
Vetor Crescente	499500	499500	999	7987	9976	15302
Vetor Decrescente	499500	499500	499500	6996	9976	7485
Vetor Aleatório	499500	499500	254067	8635	9976	8709

Número de movimentações – 1000 elementos do tipo decimal

	Seleção	Bolha	Inserção	Quicksort	Mergesort	Heapsort
Vetor Crescente	999	0	999	511	9976	15107
Vetor Decrescente	999	499500	500499	1010	9976	7315
Vetor Aleatório	999	253073	254072	2546	9976	8558

Tempo de Execução – 1000 elementos do tipo decimal

	Seleção	Bolha	Inserção	Quicksort	Mergesort	Heapsort
Vetor Crescente	00:00:00.01	00:00:00.01	00:00:00.00	00:00:00.00	00:00:00.00	00:00:00.00
Vetor Decrescente	00:00:00.00	00:00:00.01	00:00:00.01	00:00:00.00	00:00:00.00	00:00:00.00
Vetor Aleatório	00:00:00.01	00:00:00.01	00:00:00.00	00:00:00.00	00:00:00.00	00:00:00.00

Número de comparações – 500000 elementos do tipo int

	Seleção	Bolha	Inserção	Quicksort	Mergesort	Heapsort
Vetor Crescente	445698416	445698416	499999	8475732	9475712	16502620
Vetor Decrescente	445698416	445698416	445698416	7975750	9475712	8233169
Vetor Aleatório	45698416	445698416	1974443782	7467539	9475712	8842231

Número de movimentações – 500000 elementos do tipo int

	Seleção	Bolha	Inserção	Quicksort	Mergesort	Heapsort
Vetor Crescente	499999	0	499999	262143	9475712	16396051
Vetor Decrescente	499999	445698416	446198415	512142	9475712	8150965

Vetor Aleatório	499999	1974943771	1974443772	2385586	9475712	8757359

Tempo de execução – 500000 elementos do tipo int

	Seleção	**Bolha**	**Inserção**	**Quicksort**	**Mergesort**	**Heapsort**
Vetor Crescente	00:04:34.30	00:04:48.35	00:00:00.00	00:00:00.02	00:00:00.08	00:00:00.41
Vetor Decrescente	00:04:48.26	00:09:12.45	00:06:50.59	00:00:00.02	00:00:00.08	00:00:00.31
Vetor Aleatório	00:04:37.13	00:11:21.23	00:03:29.23	00:00:00.09	00:00:00.12	00:00:00.36

Código para obtenção dos dados utilizando números inteiros:

```
{
    static int compCount, movCount;

    static void Main(string[] args)
    {
        int[] tamanhos = { 1000, 500000 };
        Random random = new Random();

        foreach (int tamanho in tamanhos)
        {
            Console.WriteLine($"Gerando vetores de tamanho {tamanho}...");

            int[] vetorCrescente = new int[tamanho];
            for (int i = 0; i < tamanho; i++) vetorCrescente[i] = i + 1;

            int[] vetorDecrescente = new int[tamanho];
            for (int i = 0; i < tamanho; i++) vetorDecrescente[i] = tamanho - i;

            int[] vetorAleatorio = new int[tamanho];
            for (int i = 0; i < tamanho; i++) vetorAleatorio[i] = random.Next(1, tamanho +
1);

            TestarAlgoritmos(vetorCrescente, tamanho, "Crescente");
            TestarAlgoritmos(vetorDecrescente, tamanho, "Decrescente");
            TestarAlgoritmos(vetorAleatorio, tamanho, "Aleatório");
```

```
            Console.ReadLine();
        }
    }

    static void TestarAlgoritmos(int[] vetor, int tamanho, string tipoVetor)
    {
        Console.WriteLine($"\nTestando algoritmos com vetor {tipoVetor}, tamanho
{tamanho}:");

        int[] copia;

        copia = (int[])vetor.Clone();
        TestarOrdenacao(Selecao, copia, tamanho, "Seleção");

        copia = (int[])vetor.Clone();
        TestarOrdenacao(Bolha, copia, tamanho, "Bolha");

        copia = (int[])vetor.Clone();
        TestarOrdenacao(Insercao, copia, tamanho, "Inserção");

        copia = (int[])vetor.Clone();
        TestarOrdenacao((arr, n) => Quicksort(arr, 0, n - 1), copia, tamanho,
"Quicksort");

        copia = (int[])vetor.Clone();
        TestarOrdenacao((arr, n) => Mergesort(arr, 0, n - 1), copia, tamanho,
"Mergesort");

        copia = (int[])vetor.Clone();
        TestarOrdenacao(Heapsort, copia, tamanho, "Heapsort");
    }

    static void TestarOrdenacao(Action<int[], int> metodoOrdenacao, int[] array, int n,
string nomeMetodo)
    {
        compCount = 0;
        movCount = 0;
        Stopwatch stopwatch = Stopwatch.StartNew();

        metodoOrdenacao(array, n);

        stopwatch.Stop();
        TimeSpan ts = stopwatch.Elapsed;
```

```
        string elapsedTime = String.Format("{0:00}:{1:00}:{2:00}.{3:00}",
            ts.Hours, ts.Minutes, ts.Seconds, ts.Milliseconds / 10);
        Console.WriteLine($"{nomeMetodo}: Tempo = {elapsedTime}, Comparações =
{compCount}, Movimentações = {movCount}");
    }

    static void Selecao(int[] array, int n)
    {
        for (int i = 0; i < (n - 1); i++)
        {
            int menor = i;
            for (int j = (i + 1); j < n; j++)
            {
                compCount++;
                if (array[menor] > array[j])
                {
                    menor = j;
                }
            }
            int temp = array[menor];
            array[menor] = array[i];
            array[i] = temp;
            movCount++;
        }
    }

    static void Bolha(int[] array, int n)
    {
        int temp;
        for (int i = 0; i < n - 1; i++)
        {
            for (int j = n - 1; j > i; j--)
            {
                compCount++;
                if (array[j] < array[j - 1])
                {
                    temp = array[j];
                    array[j] = array[j - 1];
                    array[j - 1] = temp;
                    movCount++;
                }
            }
        }
    }
```

```
static void Insercao(int[] array, int n)
{
    for (int i = 1; i < n; i++)
    {
        int tmp = array[i];
        int j = i - 1;

        compCount++;

        while (j >= 0 && array[j] > tmp)
        {
            array[j + 1] = array[j];
            j--;
            movCount++;

            if (j >= 0) compCount++;
        }

        array[j + 1] = tmp;
        movCount++;
    }
}

static void Quicksort(int[] array, int esq, int dir)
{
    int i = esq, j = dir;
    int pivo = array[(esq + dir) / 2];
    while (i <= j)
    {
        while (array[i] < pivo) { i++; compCount++; }
        while (array[j] > pivo) { j--; compCount++; }
        if (i <= j)
        {
            Trocar(array, i, j);
            i++; j--;
            movCount++;
        }
    }
    if (esq < j) Quicksort(array, esq, j);
    if (i < dir) Quicksort(array, i, dir);
}

static void Mergesort(int[] array, int esq, int dir)
```

```
{
    if (esq < dir)
    {
        int meio = (esq + dir) / 2;
        Mergesort(array, esq, meio);
        Mergesort(array, meio + 1, dir);
        Intercalar(array, esq, meio, dir);
    }
}

static void Intercalar(int[] array, int esq, int meio, int dir)
{
    int nEsq = meio - esq + 1;
    int nDir = dir - meio;
    int[] arrayEsq = new int[nEsq + 1];
    int[] arrayDir = new int[nDir + 1];
    Array.Copy(array, esq, arrayEsq, 0, nEsq);
    Array.Copy(array, meio + 1, arrayDir, 0, nDir);
    arrayEsq[nEsq] = int.MaxValue;
    arrayDir[nDir] = int.MaxValue;

    for (int iEsq = 0, iDir = 0, k = esq; k <= dir; k++)
    {
        compCount++;
        if (arrayEsq[iEsq] <= arrayDir[iDir])
        {
            array[k] = arrayEsq[iEsq++];
        }
        else
        {
            array[k] = arrayDir[iDir++];
        }
        movCount++;
    }
}

static void Heapsort(int[] array, int n)
{
    for (int tam = 2; tam <= n; tam++)
    {
        Construir(array, tam);
    }

    for (int tam = n; tam > 1; tam--)
```

```
    {
        Trocar(array, 1, tam - 1);
        Reconstruir(array, tam - 1);
    }
}

static void Construir(int[] array, int tam)
{
    for (int i = tam - 1; i > 0 && array[i] > array[(i - 1) / 2]; i = (i - 1) / 2)
    {
        Trocar(array, i, (i - 1) / 2);
        movCount++;
        compCount++;
    }
}

static void Reconstruir(int[] array, int tam)
{
    int i = 1;
    while (HasFilho(i, tam))
    {
        int filho = GetMaiorFilho(array, i, tam);
        compCount++;
        if (array[i] < array[filho])
        {
            Trocar(array, i, filho);
            i = filho;
            movCount++;
        }
        else
        {
            break;
        }
    }
}

static bool HasFilho(int i, int tam) => 2 * i < tam;

static int GetMaiorFilho(int[] array, int i, int tam)
{
    int filhoEsq = 2 * i;
    int filhoDir = filhoEsq + 1;
    return (filhoDir < tam && array[filhoDir] > array[filhoEsq]) ? filhoDir : filhoEsq;
}
```

```
    static void Trocar(int[] array, int i, int j)
    {
        int temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }
}
```

Código para obtenção dos dados utilizando números decimais:

```
{
    static int compCount, movCount;

    static void Main(string[] args)
    {
        int[] tamanhos = { 1000, 500000 };
        Random random = new Random();

        foreach (int tamanho in tamanhos)
        {
            Console.WriteLine($"Gerando vetores de tamanho {tamanho} para
decimal...");

            // Vetores de decimais
            decimal[] vetorCrescente = new decimal[tamanho];
            for (int i = 0; i < tamanho; i++) vetorCrescente[i] = i + 1;

            decimal[] vetorDecrescente = new decimal[tamanho];
            for (int i = 0; i < tamanho; i++) vetorDecrescente[i] = tamanho - i;

            decimal[] vetorAleatorio = new decimal[tamanho];
            for (int i = 0; i < tamanho; i++) vetorAleatorio[i] =
(decimal)(random.NextDouble() * tamanho);

            TestarAlgoritmos(vetorCrescente, tamanho, "Crescente");
            TestarAlgoritmos(vetorDecrescente, tamanho, "Decrescente");
            TestarAlgoritmos(vetorAleatorio, tamanho, "Aleatório");
        }
    }

    static void TestarAlgoritmos(decimal[] vetor, int tamanho, string tipoVetor)
    {
```

```
        Console.WriteLine($"\nTestando algoritmos com vetor {tipoVetor}, tamanho
{tamanho}:");

        decimal[] copia;

        copia = (decimal[])vetor.Clone();
        TestarOrdenacao(Selecao, copia, tamanho, "Seleção");

        copia = (decimal[])vetor.Clone();
        TestarOrdenacao(Bolha, copia, tamanho, "Bolha");

        copia = (decimal[])vetor.Clone();
        TestarOrdenacao(Insercao, copia, tamanho, "Inserção");

        copia = (decimal[])vetor.Clone();
        TestarOrdenacao((arr, n) => Quicksort(arr, 0, n - 1), copia, tamanho,
"Quicksort");

        copia = (decimal[])vetor.Clone();
        TestarOrdenacao((arr, n) => Mergesort(arr, 0, n - 1), copia, tamanho,
"Mergesort");

        copia = (decimal[])vetor.Clone();
        TestarOrdenacao(Heapsort, copia, tamanho, "Heapsort");
    }

    static void TestarOrdenacao(Action<decimal[], int> metodoOrdenacao, decimal[]
array, int n, string nomeMetodo)
    {
        compCount = 0;
        movCount = 0;
        Stopwatch stopwatch = Stopwatch.StartNew();

        metodoOrdenacao(array, n);

        stopwatch.Stop();
        TimeSpan ts = stopwatch.Elapsed;

        string elapsedTime = String.Format("{0:00}:{1:00}:{2:00}.{3:00}",
            ts.Hours, ts.Minutes, ts.Seconds, ts.Milliseconds / 10);
        Console.WriteLine($"{nomeMetodo}: Tempo = {elapsedTime}, Comparações =
{compCount}, Movimentações = {movCount}");
    }
```

```
// Algoritmos de ordenação adaptados para decimal[]
static void Selecao(decimal[] array, int n)
{
    for (int i = 0; i < (n - 1); i++)
    {
        int menor = i;
        for (int j = (i + 1); j < n; j++)
        {
            compCount++;
            if (array[menor] > array[j])
            {
                menor = j;
            }
        }
        decimal temp = array[menor];
        array[menor] = array[i];
        array[i] = temp;
        movCount++;
    }
}

static void Bolha(decimal[] array, int n)
{
    decimal temp;
    for (int i = 0; i < n - 1; i++)
    {
        for (int j = n - 1; j > i; j--)
        {
            compCount++;
            if (array[j] < array[j - 1])
            {
                temp = array[j];
                array[j] = array[j - 1];
                array[j - 1] = temp;
                movCount++;
            }
        }
    }
}

static void Insercao(decimal[] array, int n)
{
    for (int i = 1; i < n; i++)
    {
```

```
        decimal tmp = array[i];
        int j = i - 1;

        compCount++;

        while (j >= 0 && array[j] > tmp)
        {
            array[j + 1] = array[j];
            j--;
            movCount++;

            if (j >= 0) compCount++;
        }

        array[j + 1] = tmp;
        movCount++;
    }
}

static void Quicksort(decimal[] array, int esq, int dir)
{
    int i = esq, j = dir;
    decimal pivo = array[(esq + dir) / 2];
    while (i <= j)
    {
        while (array[i] < pivo) { i++; compCount++; }
        while (array[j] > pivo) { j--; compCount++; }
        if (i <= j)
        {
            Trocar(array, i, j);
            i++; j--;
            movCount++;
        }
    }
    if (esq < j) Quicksort(array, esq, j);
    if (i < dir) Quicksort(array, i, dir);
}

static void Mergesort(decimal[] array, int esq, int dir)
{
    if (esq < dir)
    {
        int meio = (esq + dir) / 2;
        Mergesort(array, esq, meio);
```

```
        Mergesort(array, meio + 1, dir);
        Intercalar(array, esq, meio, dir);
    }
}

static void Intercalar(decimal[] array, int esq, int meio, int dir)
{
    int nEsq = meio - esq + 1;
    int nDir = dir - meio;
    decimal[] arrayEsq = new decimal[nEsq + 1];
    decimal[] arrayDir = new decimal[nDir + 1];
    Array.Copy(array, esq, arrayEsq, 0, nEsq);
    Array.Copy(array, meio + 1, arrayDir, 0, nDir);
    arrayEsq[nEsq] = decimal.MaxValue;
    arrayDir[nDir] = decimal.MaxValue;

    for (int iEsq = 0, iDir = 0, k = esq; k <= dir; k++)
    {
        compCount++;
        if (arrayEsq[iEsq] <= arrayDir[iDir])
        {
            array[k] = arrayEsq[iEsq++];
        }
        else
        {
            array[k] = arrayDir[iDir++];
        }
        movCount++;
    }
}

static void Heapsort(decimal[] array, int n)
{
    for (int tam = 2; tam <= n; tam++)
    {
        Construir(array, tam);
    }

    for (int tam = n; tam > 1; tam--)
    {
        Trocar(array, 1, tam - 1);
        Reconstruir(array, tam - 1);
    }
}
```

```
static void Construir(decimal[] array, int tam)
{
    for (int i = tam - 1; i > 0 && array[i] > array[(i - 1) / 2]; i = (i - 1) / 2)
    {
        Trocar(array, i, (i - 1) / 2);
        movCount++;
        compCount++;
    }
}

static void Reconstruir(decimal[] array, int tam)
{
    int i = 1;
    while (HasFilho(i, tam))
    {
        int filho = GetMaiorFilho(array, i, tam);
        compCount++;
        if (array[i] < array[filho])
        {
            Trocar(array, i, filho);
            i = filho;
            movCount++;
        }
        else
        {
            break;
        }
    }
}

static bool HasFilho(int i, int tam) => 2 * i < tam;

static int GetMaiorFilho(decimal[] array, int i, int tam)
{
    int filhoEsq = 2 * i;
    int filhoDir = filhoEsq + 1;
    return (filhoDir < tam && array[filhoDir] > array[filhoEsq]) ? filhoDir : filhoEsq;
}

static void Trocar(decimal[] array, int i, int j)
{
    decimal temp = array[i];
    array[i] = array[j];
```

```
        array[j] = temp;
    }
}
```

2-

1. **Seleção (Selection Sort)**:
    - **Vantagens**: É fácil de entender e implementar. É eficiente em vetores pequenos devido à sua simplicidade.
    - **Desvantagens**: Tem uma complexidade de tempo O(n2)O(n^2)O(n2), o que faz com que seja extremamente lento para vetores grandes, como o de 500000 elementos. Esse método sempre realiza um número fixo de comparações, independentemente da ordenação inicial do vetor.
2. **Bolha (Bubble Sort)**:
    - **Vantagens**: Também é fácil de implementar e funciona bem em casos onde o vetor já está quase ordenado.
    - **Desvantagens**: É altamente ineficiente para vetores grandes devido ao tempo O(n2)O(n^2)O(n2) no pior e médio caso. Em vetores maiores, ele tem um desempenho muito inferior, especialmente para dados em ordem decrescente.
3. **Inserção (Insertion Sort)**:
    - **Vantagens**: Funciona bem com vetores pequenos ou parcialmente ordenados, pois a complexidade do pior caso O(n2)O(n^2)O(n2) raramente é atingida.
    - **Desvantagens**: É lento em vetores grandes, como o de 500000 elementos, e seu desempenho é muito influenciado pela disposição inicial dos elementos.
4. **Quicksort**:
    - **Vantagens**: Um dos algoritmos mais rápidos na prática para ordenação em vetores grandes. Tem complexidade O(nlogn)O(n \log n)O(nlogn) na média e, geralmente, é mais rápido que a maioria dos algoritmos.
    - **Desvantagens**: No pior caso (quando o vetor está quase ordenado e o pivô é mal escolhido), a complexidade pode se tornar O(n2)O(n^2)O(n2). Ele requer técnicas como a escolha aleatória do pivô para evitar o pior caso.
5. **Mergesort**:
    - **Vantagens**: É estável e possui uma complexidade de O(nlogn)O(n \log n)O(nlogn) garantida, independente da disposição inicial dos elementos. É ideal para vetores grandes, como o de 500000

elementos, pois mantém o desempenho mesmo em casos desfavoráveis.
    - **Desvantagens**: Requer espaço adicional proporcional ao tamanho do vetor, o que pode ser uma limitação em ambientes de memória restrita.
6. **Heapsort**:
    - **Vantagens**: Tem complexidade O(nlogn)O(n \log n)O(nlogn) no pior caso, sem a necessidade de espaço adicional significativo como o Mergesort. É eficiente para vetores grandes.
    - **Desvantagens**: Embora seja eficiente, na prática, pode ser um pouco mais lento que o Quicksort devido à estrutura de dados da heap, e não é estável (ou seja, não preserva a ordem dos elementos iguais).

Observação Geral

Para vetores pequenos (1000 elementos), os algoritmos de complexidade O(n2)O(n^2)O(n2) podem ter desempenho aceitável, mas para vetores grandes (500000 elementos), os algoritmos com complexidade O(nlogn)O(n \log n)O(nlogn), como Quicksort, Mergesort e Heapsort, são mais vantajosos em termos de tempo de execução.

3-

```
static void Main()
{
    string caminhoArquivo = "players.csv";
    List<Jogador> jogadores = new List<Jogador>();

    using (var leitor = new StreamReader(caminhoArquivo))
    {
        leitor.ReadLine();

        while (!leitor.EndOfStream)
        {
            var linha = leitor.ReadLine();
```

```
        var campos = linha.Split(',');

        Jogador jogador = new Jogador(
            int.Parse(campos[0]),
            campos[1],
            float.Parse(campos[2]),
            float.Parse(campos[3]),
            campos[4],
            int.Parse(campos[5]),
            campos[6],
            campos[7]
        );

        jogadores.Add(jogador);
    }
}

jogadores = MergeSort(jogadores);

foreach (var jogador in jogadores)
{
    Console.WriteLine(jogador);
}
Console.ReadLine();
```

```
}

public static List<Jogador> MergeSort(List<Jogador> lista)
{
    if (lista.Count <= 1)
        return lista;

    int meio = lista.Count / 2;
    List<Jogador> esquerda = lista.GetRange(0, meio);
    List<Jogador> direita = lista.GetRange(meio, lista.Count - meio);

    esquerda = MergeSort(esquerda);
    direita = MergeSort(direita);

    return Merge(esquerda, direita);
}

public static List<Jogador> Merge(List<Jogador> esquerda, List<Jogador> direita)
{
    List<Jogador> resultado = new List<Jogador>();
    int i = 0, j = 0;

    while (i < esquerda.Count && j < direita.Count)
    {
        if (esquerda[i].AnoNasc < direita[j].AnoNasc ||
```

```
            (esquerda[i].AnoNasc == direita[j].AnoNasc &&
string.Compare(esquerda[i].Nome, direita[j].Nome) <= 0))
        {
            resultado.Add(esquerda[i]);
            i++;
        }
        else
        {
            resultado.Add(direita[j]);
            j++;
        }
    }

    while (i < esquerda.Count)
    {
        resultado.Add(esquerda[i]);
        i++;
    }

    while (j < direita.Count)
    {
        resultado.Add(direita[j]);
        j++;
    }
```

```
        return resultado;
    }
}

class Jogador
{
    private int id;
    private string nome;
    private float altura;
    private float peso;
    private string universidade;
    private int anoNasc;
    private string cidadeNasc;
    private string estadoNasc;

    public Jogador(int id, string nome, float altura, float peso, string universidade, int
anoNasc, string cidadeNasc, string estadoNasc)
    {
        this.id = id;
        this.nome = nome;
        this.altura = altura;
        this.peso = peso;
        this.universidade = universidade;
        this.anoNasc = anoNasc;
        this.cidadeNasc = cidadeNasc;
```

```
        this.estadoNasc = estadoNasc;
    }

    public int AnoNasc => anoNasc;
    public string Nome => nome;

    public override string ToString()
    {
        return $"ID: {id}, Nome: {nome}, Altura: {altura}, Peso: {peso}, Universidade: {universidade}, Ano Nasc: {anoNasc}, Cidade Nasc: {cidadeNasc}, Estado Nasc: {estadoNasc}";
    }
```

4-

```
using System;
using System.Collections.Generic;
using System.IO;

class Program
{
    class Pais
    {
        public string Nome;
        public int Ouro;
        public int Prata;
        public int Bronze;
```

```
}

static void Main()
{
    List<Pais> paises = new List<Pais>();
    string[] linhas = File.ReadAllLines("olimpiadas.txt");

    for (int i = 0; i < linhas.Length; i++)
    {
        if (string.IsNullOrWhiteSpace(linhas[i]))
            continue;

        try
        {
            string nome = linhas[i].Trim();
            int ouro = int.Parse(linhas[++i].Trim());
            int prata = int.Parse(linhas[++i].Trim());
            int bronze = int.Parse(linhas[++i].Trim());

            paises.Add(new Pais
            {
                Nome = nome,
                Ouro = ouro,
                Prata = prata,
                Bronze = bronze
```

```
            });
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Erro ao processar os dados na linha {i + 1}. Verifique
o formato dos dados.");
            Console.WriteLine($"Detalhes do erro: {ex.Message}");
        }
    }

    paises = MergeSort(paises);

    Console.WriteLine("Quadro de Medalhas:");
    Console.WriteLine("País\t\tOuro\tPrata\tBronze");

    foreach (Pais pais in paises)
    {
Console.WriteLine($"{pais.Nome,-15}\t{pais.Ouro}\t{pais.Prata}\t{pais.Bronze}");
    }

    Console.WriteLine("Pressione qualquer tecla para sair.");
    Console.ReadKey();
  }

  static List<Pais> MergeSort(List<Pais> paises)
```

```
{
    if (paises.Count <= 1)
        return paises;

    int meio = paises.Count / 2;
    List<Pais> esquerda = MergeSort(paises.GetRange(0, meio));
    List<Pais> direita = MergeSort(paises.GetRange(meio, paises.Count - meio));

    return Merge(esquerda, direita);
}

static List<Pais> Merge(List<Pais> esquerda, List<Pais> direita)
{
    List<Pais> resultado = new List<Pais>();
    int i = 0, j = 0;

    while (i < esquerda.Count && j < direita.Count)
    {
        if (ComparaPaises(esquerda[i], direita[j]) <= 0)
        {
            resultado.Add(esquerda[i]);
            i++;
        }
        else
        {
```

```
            resultado.Add(direita[j]);
            j++;
        }
    }

    while (i < esquerda.Count)
    {
        resultado.Add(esquerda[i]);
        i++;
    }

    while (j < direita.Count)
    {
        resultado.Add(direita[j]);
        j++;
    }

    return resultado;
}

static int ComparaPaises(Pais a, Pais b)
{
    if (a.Ouro != b.Ouro)
        return b.Ouro.CompareTo(a.Ouro);
    if (a.Prata != b.Prata)
```

```
            return b.Prata.CompareTo(a.Prata);

        if (a.Bronze != b.Bronze)

            return b.Bronze.CompareTo(a.Bronze);


        return a.Nome.CompareTo(b.Nome);

    }

}
```

5-

1. Algoritmo de Seleção (Selection Sort)

No Selection Sort, em cada iteração, o algoritmo encontra o menor elemento da parte não ordenada do vetor e o coloca na posição correta.

Passos:

- **Passo 1**: O menor elemento da lista completa é 1, e ele será trocado com o elemento da posição 0.
    - Vetor: **[1, 1, 3, 20, 5, 6, 10, 4, 9, 2]**
- **Passo 2**: O menor elemento da sublista **[1, 3, 20, 5, 6, 10, 4, 9, 2]** é 1 (já está na posição correta).
    - Vetor: **[1, 1, 3, 20, 5, 6, 10, 4, 9, 2]** (nenhuma troca).
- **Passo 3**: O menor elemento da sublista **[3, 20, 5, 6, 10, 4, 9, 2]** é 2. Ele será trocado com 3.
    - Vetor: **[1, 1, 2, 20, 5, 6, 10, 4, 9, 3]**
- **Passo 4**: O menor elemento da sublista **[20, 5, 6, 10, 4, 9, 3]** é 3. Ele será trocado com 20.
    - Vetor: **[1, 1, 2, 3, 5, 6, 10, 4, 9, 20]**
- **Passo 5**: O menor elemento da sublista **[5, 6, 10, 4, 9, 20]** é 4. Ele será trocado com 5.
    - Vetor: **[1, 1, 2, 3, 4, 6, 10, 5, 9, 20]**
- **Passo 6**: O menor elemento da sublista **[6, 10, 5, 9, 20]** é 5. Ele será trocado com 6.
    - Vetor: **[1, 1, 2, 3, 4, 5, 10, 6, 9, 20]**
- **Passo 7**: O menor elemento da sublista **[10, 6, 9, 20]** é 6. Ele será trocado com 10.
    - Vetor: **[1, 1, 2, 3, 4, 5, 6, 10, 9, 20]**
- **Passo 8**: O menor elemento da sublista **[10, 9, 20]** é 9. Ele será trocado com 10.
    - Vetor: **[1, 1, 2, 3, 4, 5, 6, 9, 10, 20]**
- **Passo 9**: O menor elemento da sublista **[10, 20]** é 10 (já está na posição correta).
    - Vetor: **[1, 1, 2, 3, 4, 5, 6, 9, 10, 20]**
- **Passo 10**: O vetor está completamente ordenado.

2. Algoritmo de Bolha (Bubble Sort)

O Bubble Sort compara elementos adjacentes e os troca se estiverem na ordem errada. Esse processo é repetido até que o vetor esteja ordenado.

Passos:

- **Passo 1**: Comparar **10** com **1** → troca.
  - Vetor: **[1, 10, 3, 20, 5, 6, 1, 4, 9, 2]**
- **Passo 2**: Comparar **10** com **3** → troca.
  - Vetor: **[1, 3, 10, 20, 5, 6, 1, 4, 9, 2]**
- **Passo 3**: Comparar **10** com **20** → sem troca.
  - Vetor: **[1, 3, 10, 20, 5, 6, 1, 4, 9, 2]**
- **Passo 4**: Comparar **20** com **5** → troca.
  - Vetor: **[1, 3, 10, 5, 20, 6, 1, 4, 9, 2]**
- **Passo 5**: Comparar **20** com **6** → troca.
  - Vetor: **[1, 3, 10, 5, 6, 20, 1, 4, 9, 2]**
- **Passo 6**: Comparar **20** com **1** → troca.
  - Vetor: **[1, 3, 10, 5, 6, 1, 20, 4, 9, 2]**
- **Passo 7**: Comparar **20** com **4** → troca.
  - Vetor: **[1, 3, 10, 5, 6, 1, 4, 20, 9, 2]**
- **Passo 8**: Comparar **20** com **9** → troca.
  - Vetor: **[1, 3, 10, 5, 6, 1, 4, 9, 20, 2]**
- **Passo 9**: Comparar **20** com **2** → troca.
  - Vetor: **[1, 3, 10, 5, 6, 1, 4, 9, 2, 20]**

Agora o maior elemento 20 está na posição final.

Repetimos o processo até que o vetor esteja ordenado. Após vários passos, o vetor será: [1, 1, 2, 3, 4, 5, 6, 9, 10, 20].

3. Algoritmo de Inserção (Insertion Sort)

O Insertion Sort pega cada elemento e o insere na posição correta em uma sublista ordenada.

Passos:

- **Passo 1:** Inserir 1 na posição correta.
  - Vetor: **[1, 10, 3, 20, 5, 6, 1, 4, 9, 2]**
- **Passo 2:** Inserir 3 na posição correta.
  - Vetor: **[1, 3, 10, 20, 5, 6, 1, 4, 9, 2]**
- **Passo 3:** Inserir 20 na posição correta.
  - Vetor: **[1, 3, 10, 20, 5, 6, 1, 4, 9, 2]**
- **Passo 4:** Inserir 5 na posição correta.
  - Vetor: **[1, 3, 5, 10, 20, 6, 1, 4, 9, 2]**
- **Passo 5:** Inserir 6 na posição correta.

    - Vetor: **[1, 3, 5, 6, 10, 20, 1, 4, 9, 2]**
- **Passo 6:** Inserir 1 na posição correta.
    - Vetor: **[1, 1, 3, 5, 6, 10, 20, 4, 9, 2]**
- **Passo 7:** Inserir 4 na posição correta**.**
    - Vetor: **[1, 1, 3, 4, 5, 6, 10, 20, 9, 2]**
- **Passo 8:** Inserir 9 na posição correta.
    - Vetor: **[1, 1, 3, 4, 5, 6, 9, 10, 20, 2]**
- **Passo 9:** Inserir 2 na posição correta.
    - Vetor: **[1, 1, 2, 3, 4, 5, 6, 9, 10, 20]**
- **Passo 10**: O vetor está completamente ordenado.

4. Algoritmo Quicksort

No Quicksort, escolhemos um pivô (neste caso, o elemento do meio) e particionamos o vetor em duas sublistas, uma com elementos menores que o pivô e outra com elementos maiores. O processo é repetido recursivamente.

Passos:

- **Passo 1**: O pivô é **20** (meio do vetor). Particiona em **[10, 1, 3, 5, 6, 1, 4, 9, 2]** (menores ou iguais ao pivô) e **[20]** (maiores).
    - Vetor após particionar: **[10, 1, 3, 5, 6, 1, 4, 9, 2]** | **[20]**
- Repetimos o processo para as duas sublistas até o vetor estar completamente ordenado.

5. Heapsort

O Heapsort começa criando um heap (uma árvore binária que respeita a propriedade de heap) e depois remove o maior elemento, ajustando o heap a cada remoção.

Passo 1: Construção do Max-Heap

Começamos com o vetor [10, 1, 3, 20, 5, 6, 1, 4, 9, 2] e construímos o max-heap. Para isso, reorganizamos o vetor de forma que o maior valor fique na raiz e as subárvores também sejam heaps.

Vetor inicial:

```
[10, 1, 3, 20, 5, 6, 1, 4, 9, 2]
```

Após organizar o max-heap, o vetor fica assim:

```
[20, 10, 9, 6, 5, 3, 1, 4, 2, 1]
```

Passo 2: Remover o maior elemento (20) e reestruturar o heap

Agora, removemos o 20, colocando-o na última posição e ajustando o heap.

Vetor após remover o 20:

```
[1, 10, 9, 6, 5, 3, 1, 4, 2, 20]
```

Reajustamos o heap:

1. O maior valor é 10, então trocamos 1 com 10.
2. Reajustamos as subárvores.

Vetor após o ajuste:

```
[10, 6, 9, 4, 5, 3, 1, 2, 1, 20]
```

Passo 3: Remover o próximo maior elemento (10) e reestruturar

Agora, removemos o 10, colocando-o na penúltima posição e ajustamos o heap novamente.

Vetor após remover o 10:

```
[1, 6, 9, 4, 5, 3, 1, 2, 10, 20]
```

Reajustamos o heap:

1. O maior valor é 9, então trocamos 1 com 9.
2. Depois, o maior valor da subárvore é 6, então trocamos 1 com 6.

Vetor após o ajuste:

```
[9, 6, 3, 4, 5, 1, 1, 2, 10, 20]
```

Passo 4: Continuar removendo o maior elemento e ajustando

Repetimos o processo de remoção e reestruturação até que todos os elementos sejam removidos e o vetor esteja completamente ordenado.

1. Remover o 9, colocar na posição final e reestruturar.
2. Remover o 6, colocar na posição final e reestruturar.
3. Remover o 5, colocar na posição final e reestruturar.
4. E assim por diante...

Vetor final após Heapsort:

```
[1, 1, 2, 3, 4, 5, 6, 9, 10, 20]
```

6. Mergesort

O **Mergesort** é um algoritmo de **divisão e conquista**. Ele divide recursivamente o vetor em duas metades até que cada sublista tenha um único elemento. Depois, ele faz a **mesclagem** dessas sublistas de forma ordenada.

Passos do Mergesort:

- **Vetor inicial**: [10, 1, 3, 20, 5, 6, 1, 4, 9, 2]

Passo 1: Dividir o vetor

Dividimos o vetor ao meio repetidamente até que tenhamos sublistas de tamanho 1.

1. Dividindo o vetor: [10, 1, 3, 20, 5] e [6, 1, 4, 9, 2]
2. Dividindo cada metade:
    - [10, 1, 3] e [20, 5]
    - [6, 1] e [4, 9, 2]
3. Continuando a divisão até termos sublistas de tamanho 1:
    - [10], [1], [3], [20], [5], [6], [1], [4], [9], [2]

Passo 2: Mesclar as sublistas de forma ordenada

Agora, começamos a mesclar as sublistas de forma ordenada.

1. Mesclar [10] e [1]: [1, 10]
2. Mesclar [1, 10] e [3]: [1, 3, 10]
3. Mesclar [20] e [5]: [5, 20]
4. Mesclar [6] e [1]: [1, 6]
5. Mesclar [4], [9] e [2]: [2, 4, 9]

Passo 3: Mesclar as metades

Agora, vamos mesclar as listas maiores:

1. Mesclar [1, 3, 10] e [5, 20]: [1, 3, 5, 10, 20]
2. Mesclar [1, 6] e [2, 4, 9]: [1, 2, 4, 6, 9]

Passo 4: Mesclar as duas metades finais

Por fim, mesclamos as duas metades finais para obter o vetor ordenado.

- Mesclar [1, 3, 5, 10, 20] e [1, 2, 4, 6, 9]:
  [1, 1, 2, 3, 4, 5, 6, 9, 10, 20]

7. Counting Sort

Passo 1: Contar a frequência de cada elemento

Começamos com o vetor **[10, 1, 3, 20, 5, 6, 1, 4, 9, 2]** e contamos a frequência de cada elemento.

Contagem:

```
[0, 2, 1, 1, 1, 1, 1, 0, 1, 1] (contagem dos elementos de 0 a 20)
```

Passo 2: Construir o vetor ordenado com base na contagem

Agora, usamos o array de contagem para preencher o vetor ordenado. O array de contagem indica quantas vezes cada número aparece no vetor.

1. O número **1** aparece 2 vezes.
2. O número **2** aparece 1 vez.
3. O número **3** aparece 1 vez.
4. O número **4** aparece 1 vez.
5. O número **5** aparece 1 vez.
6. E assim por diante...

Vetor final após Counting Sort:

```
[1, 1, 2, 3, 4, 5, 6, 9, 10, 20]
```