PONTIFÍCIA UNIVERSIDADE CATÓLICA DO PARANÁ
ESCOLA POLITÉCNICA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO
TDE3 - RESOLUÇÃO DE PROBLEMAS ESTRUTURADOS EM COMPUTAÇÃO
Estudantes: Enzo Curcio Stival, Hiann Wonsowicz Padilha, Marcos Paulo Ruppel

RELATÓRIO DA ATIVIDADE – ALGORITMOS DE ORDENAÇÃO

Esta atividade constitui o terceiro Trabalho Discente Efetivo (TDE) da disciplina de Resolução de Problemas Estruturados em Computação, visando avaliar o conhecimento sobre operações em estruturas de dados com o contexto de algoritmos de ordenação de dados.

O código-fonte do programa desenvolvido para esse projeto encontra-se disponível em < https://github.com/MarcosRuppel/TDE03-EstruturaDeDados >

1. OBJETIVOS:

Para esta atividade foi proposta a implementação em *Java* de um total de 6 (seis) algoritmos distintos de ordenação de vetores, e a aplicação de tais algoritmos para ordenar diferentes vetores de numerais inteiros positivos. Os algoritmos escolhidos para esta atividade foram:

- *Insertion Sort;*
- *Selection Sort;*
- *Heap Sort;*
- *Radix Sort;*
- *Counting Sort.*

Após a implementação destes, os mesmos devem ser executados para ordenar vetores de 1000, 10.000, 100.000, 500.000 e 1.000.000 elementos, randomizados utilizando uma seed. Cada algoritmo deve ser executado em 5 conjuntos de dados, para obter-se uma média de tempo, trocas (swaps) e iterações de cada algoritmo.

2. ALGORITMOS

Nesta seção serão detalhados os métodos de operação dos algoritmos escolhidos, bem como o modo como foram implementados no programa em Java.

2.1. INSERTION SORT

O *Insertion Sort* organiza um *array* da seguinte forma:

- **Divisão Implícita do Array**: Divide o *array* em uma parte ordenada e outra não ordenada. Inicialmente, a parte ordenada contém apenas o primeiro elemento do *array*, e a parte não ordenada contém o restante.
- **Inserção do Próximo Elemento**: Para cada elemento da parte não ordenada, o algoritmo o compara com os elementos da parte ordenada (da direita para a esquerda) até encontrar a posição correta para inseri-lo.
- **Deslocamento e Inserção**: Move os elementos da parte ordenada para abrir espaço até encontrar a posição correta e, então, insere o elemento.
- **Repetição**: Repete esse processo para cada elemento do *array* até que todo o *array* esteja ordenado.

2.1.1. IMPLEMENTAÇÃO DO INSERTION SORT

- O loop `for (int i = 1; i < array.length; i++)` trata cada elemento a partir do índice 1 como o próximo a ser inserido na parte ordenada, que fica à esquerda.
- `int key = array[i];` armazena o próximo elemento da parte não ordenada que será inserido na parte ordenada.
- O `while (j >= 0 && array[j] > key)` percorre a parte ordenada, deslocando elementos maiores que key para a direita, criando espaço para o `key`.
- Após o *while*, `array[j + 1] = key;` insere o key na posição correta.

2.2. SELECTION SORT

No *Selection Sort*, o algoritmo realiza as seguintes etapas para cada posição do *array*:

- **Busca o menor elemento** no *subarray* não ordenado (da posição atual até o final do *array*).
- **Troca o menor elemento encontrado** com o elemento na posição atual, se o menor elemento não for o próprio elemento da posição inicial.
- **Repete o processo** para a próxima posição do *array*, até que todo o *array* esteja ordenado.

2.2.1. IMPLEMENTAÇÃO DO SELECTION SORT

O algoritmo foi implementado da seguinte forma:

- O loop externo `for (int i = 0; i < array.length - 1; i++)` percorre cada posição do array para onde um elemento ordenado será posicionado.
- O loop interno `for (int j = i + 1; j < array.length; j++)` encontra o menor elemento no subarray não ordenado.
- Após encontrar o índice `minIndex` do menor elemento, o código realiza a troca (swap) apenas se `minIndex` for diferente de `i`.

2.3. HEAP SORT

Heap Sort é um algoritmo de ordenação baseado na estrutura de dados *Heap* e funciona da seguinte forma:

1. **Construção do Heap**:

- A primeira etapa é construir um *Max-Heap* (ou *Min-Heap* para ordenação crescente) a partir do *array*. No *Max-Heap*, cada nó pai é maior ou igual a seus filhos.
- Para transformar o *array* em um *Max-Heap*, o *Heap Sort* começa do meio do *array* e aplica a operação `heapify` para cada elemento, do último nó não-folha até o início.

2. **Extração e Ordenação**:

- Após construir o *Max-Heap*, o maior elemento (raiz do *heap*) está na primeira posição do *array*.
- O *Heap Sort* então troca o primeiro elemento com o último elemento do *heap* (reduzindo o tamanho do *heap* em uma unidade), e aplica `heapify` novamente para restaurar a estrutura do *Max-Heap*.
- Esse processo é repetido até que todos os elementos estejam ordenados.

2.3.1. IMPLEMENTAÇÃO DO HEAP SORT

1. **Construção do Heap**:

- O loop `for (int i = n / 2 - 1; i >= 0; i--)` inicia do último nó não-folha e vai até o início do *array*, aplicando `heapify` para garantir que cada subárvore obedeça à propriedade do *Max-Heap*.
- A função `heapify` reestrutura o *heap* e retorna um array contendo o número de *iterations* (comparações) e *swaps* (trocas), que são acumulados nas variáveis principais do algoritmo.

2. **Extração e Ordenação**:

- O loop `for (int i = n - 1; i >= 0; i--)` realiza a ordenação do *array*. Em cada iteração, o maior elemento (na raiz do *heap*, `array[0]`)

é trocado com o último elemento do *heap* (`array[i]`), e `heapify` é chamado novamente para ajustar o *heap* reduzido.

- A cada troca, o número de swaps aumenta, e o `heapify` é chamado para restaurar a propriedade de *Max-Heap* na parte restante do *array*.

2.3.1.1. FUNÇÃO HEAPIFY

A função `heapify` garante que o subárvore com raiz em `i` mantém a propriedade de *Max-Heap*.

1. **Comparações com os Filhos**:

   - O código define inicialmente `largest = i` (presumindo que o nó raiz seja o maior) e calcula os índices `left` e `right`.
   - Se `array[left] > array[largest]`, o índice `largest` é atualizado para o `left`, e `iterations++` é incrementado para contar a comparação.
   - Da mesma forma, `right` é comparado, e `iterations++` é contado.

2. **Troca e Recursão**:

   - Se `largest` foi atualizado para `left` ou `right`, uma troca é feita entre `array[i]` e `array[largest]` para mover o maior valor para a raiz do subárvore, e `swaps++` é incrementado.
   - `heapify` é chamado recursivamente no índice `largest` para garantir que a subárvore resultante também obedeça à propriedade de *Max-Heap*, acumulando *iterations* e *swaps* na chamada recursiva.

2.4. BUCKET SORT

O *Bucket Sort* é um algoritmo de ordenação que distribui os elementos de um *array* em várias "baldes" (*buckets*). Sua execução ocorre da seguinte forma:

1. **Encontra os Valores Mínimo e Máximo:**
   - Primeiro, determina o valor mínimo e máximo no *array* para facilitar a distribuição dos elementos nos *buckets*.

2. **Cria os Buckets:**
   - Cria uma quantidade fixa de *buckets*, que, em geral, é determinada como a raiz quadrada do número de elementos no *array* (ou outra heurística).

3. **Distribui os Elementos nos Buckets:**
   - A cada elemento do *array*, calcula-se em qual *bucket* ele deve ir, com base em sua posição relativa entre o mínimo e o máximo.

4. **Ordena os Buckets:**
   - Para cada *bucket* que contém elementos, aplica-se um algoritmo de ordenação (como *Insertion Sort*).

5. **Combina os Buckets:**
   - Finalmente, combinam-se todos os *buckets* de volta em um único *array* ordenado.

2.4.1. IMPLEMENTAÇÃO DO BUCKET SORT

- **Cálculo do Mínimo e Máximo**:
  O uso das funções `getMax` e `getMin` permite determinar os limites do *array*. Isso é essencial para a distribuição correta dos elementos.

- **Criação de Buckets**:
  A estrutura:
  ```
  int[][] buckets = new int[bucketCount][array.length];
  ```
  cria um *array* bidimensional, onde cada *bucket* pode armazenar até o tamanho total do *array*.

- **Distribuição de Elementos nos Buckets**:
  O cálculo do `bucketIndex` determina aonde cada elemento vai, e `bucketSizes[bucketIndex]++` rastreia quantos elementos estão em cada *bucket*. O incremento de *iterations* conta as operações de distribuição.

- **Ordenação de Cada Bucket**:
  A função `bucketInsSort` aplica *Insertion Sort* em cada *bucket*. Aqui, swaps é contabilizado para cada troca realizada durante a ordenação de um *bucket*.

- **Combinação dos Buckets**:
  O loop final combina todos os elementos de cada *bucket* de volta no *array* original, com a contagem de *iterations* para cada elemento transferido.

2.4.2. EXCEÇÃO "*OUTOFMEMORY*" NA EXECUÇÃO DO BUCKET SORT

Quando definimos a seguinte estrutura no código:

```
int[][] buckets = new int[bucketCount][array.length];
```

Estamos criando uma estrutura onde cada *bucket* pode ter até `array.length` elementos. Isso significa que se estivermos ordenando um array de 1 milhão de elementos e `bucketCount` for, por exemplo, 1000, estamos gerando um *array* de (1000 x 1 milhão) = 1 bilhão de inteiros. Isso pode exceder a memória disponível no *heap* do Java, resultando na exceção "*OutOfMemory*".

2.5. RADIX SORT

O *Radix Sort* é um algoritmo de ordenação não-comparativo usado para ordenar números inteiros, que ordena os elementos com base nos seus dígitos individuais (ou "dígitos significativos"), processando um dígito por vez, do menos ao mais significativo. Ele utiliza o *Counting Sort* como sub-rotina para ordenar os números em cada dígito. Este algoritmo opera da seguinte forma:

1. **Identifica o Maior Elemento:**
    - Primeiro, determina o maior valor no array para saber quantos dígitos ele possui e assim quantas passagens de ordenação por dígitos são necessárias.]

2. **Ordenação por Dígito (Counting Sort):**
    - A cada iteração, aplica o Counting Sort aos números, baseando-se em um dígito específico:
        - Na primeira passagem, ordena pelos dígitos da unidade (posição 1).
        - Na segunda, ordena pelos dígitos da dezena (posição 10), e assim por diante.

3. **Distribuição Estável:**
    - O *Counting Sort* é usado de maneira estável para cada dígito, garantindo que a ordem relativa de elementos com o mesmo dígito permaneça a mesma entre as passagens. Essa estabilidade é crucial para o *Radix Sort*.

2.5.1. IMPLEMENTAÇÃO DO RADIX SORT

Identificação do Maior Elemento:

- `int max = getMax(array);` encontra o maior valor no *array* para definir quantos dígitos serão usados na ordenação.

Loop sobre os Dígitos:

- `for (int exp = 1; max / exp > 0; exp *= 10)` itera pelos dígitos de cada posição (unidade, dezena, centena etc.). Em cada iteração, `countSort` é chamado para ordenar o *array* baseado no dígito correspondente (`exp`).

Função `countSort`:

- Contagem dos Elementos: O primeiro loop `for` conta quantos elementos têm cada dígito na posição atual.
- Acumulação de Contagens: O segundo loop transforma as contagens em posições cumulativas, permitindo que o algoritmo posicione os elementos no *array* de saída de maneira estável.
- Preenchimento do Array de Saída: O terceiro loop coloca os elementos do *array* de entrada em output com base nos dígitos da posição atual e no *array* de contagem acumulada.
- Cópia do Array de Saída para o Original: Por fim, o último loop copia output de volta para *array*.

Contabilização de Trocas e Iterações:

- `swaps` permanece zero, o que é apropriado, pois o *Radix Sort* não realiza trocas diretas entre elementos no *array* original.
- `iterations` é incrementado para cada operação relevante dentro de `countSort` para refletir o número total de operações executadas.

2.6. COUNTING SORT

Counting Sort é um algoritmo de ordenação não-comparativo que é eficiente para ordenar *arrays* com números inteiros em um intervalo limitado. Ele conta a frequência de cada elemento e usa essas contagens para ordenar os elementos no *array* original.

1. **Encontrar o Intervalo de Valores**:
- Determina o valor mínimo e máximo para saber o tamanho necessário do array de contagem.

2. **Contar Frequências**:
- Conta a frequência de cada elemento do array, ajustando o índice do array de contagem de acordo com o valor mínimo do array de entrada.

3. **Acumular Contagens**:
- Modifica o array de contagem para acumular as frequências, transformando as contagens em índices.

4. **Preencher o Array de Saída**:
- Percorre o array de entrada de trás para frente, colocando cada elemento em sua posição correta no array de saída com base no array de contagem e decrementando a contagem.

5. **Copiar o Array de Saída**:
- Copia o array de saída para o array original.

2.6.1. IMPLEMENTAÇÃO DO COUNTING SORT

1. **Encontrar o Máximo e o Mínimo:** `getMax(array)` e `getMin(array)` encontram o valor máximo e mínimo, respectivamente, para calcular o intervalo de valores `(range = max - min + 1)`. Isso permite que o *Counting Sort* lide com *arrays* contendo valores negativos.

2. **Contagem dos Elementos:** O loop `for (int j : array)` percorre cada elemento de *array* e incrementa o valor em `count[j - min]`, que ajusta o índice da contagem para permitir valores negativos. Cada iteração é contada em `iterations++`.
3. **Acumulação das Contagens:** O loop `for (int i = 1; i < range; i++)` acumula os valores em `count` para transformar as frequências em índices. Esse *array* acumulado permite que cada elemento seja posicionado corretamente no *array* de saída.
4. **Ordenação com Base na Contagem:** O loop `for (int i = array.length - 1; i >= 0; i--)` usa o *array* `count` para posicionar cada elemento de *array* em output com base em suas contagens acumuladas. O `count[array[i] - min]` é decrementado após cada uso para que elementos iguais sejam colocados nas posições corretas.

   As iterações desse loop também são contadas em `iterations++`.
5. **Cópia para o Array Original:** `System.arraycopy(output, 0, array, 0, array.length);` copia o *array* de saída ordenado (output) para `array`, tornando `array` a versão ordenada final.

2.6.1.1. Considerações

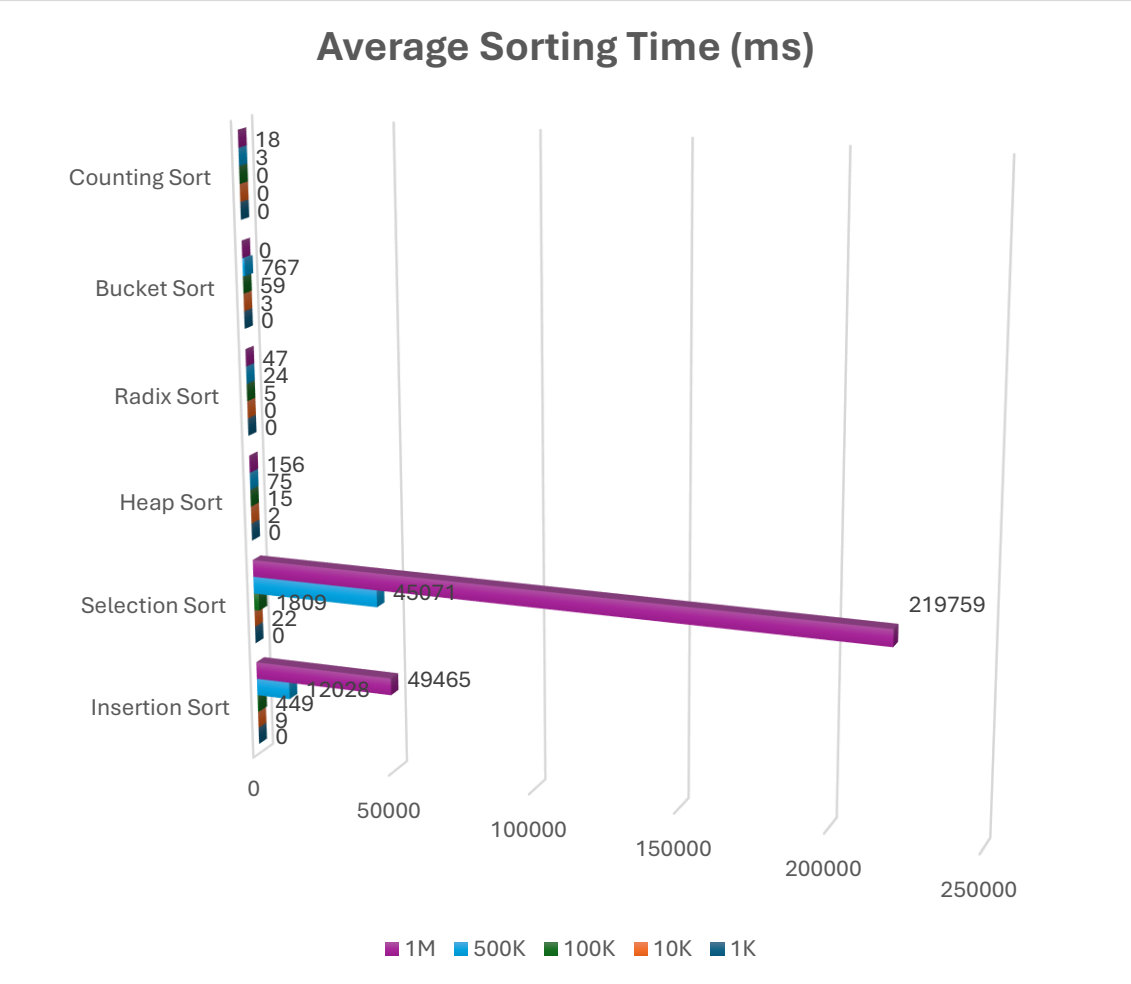
Em *Counting Sort e Radix Sort*, não há trocas (*swaps*) no sentido tradicional, pois eles não reordenam elementos diretamente no *array* original por troca de posições. Em vez disso, eles posicionam diretamente os elementos no *array* de saída. A variável `swaps` permanece `0`.

3. TABELAS E GRÁFICOS DA EXECUÇÃO DOS ALGORITMOS

Esta seção contém as tabelas contendo as amostras dos dados dos tempos de execução, quantidade de iterações e swaps de cada um dos algoritmos especificados anteriormente.

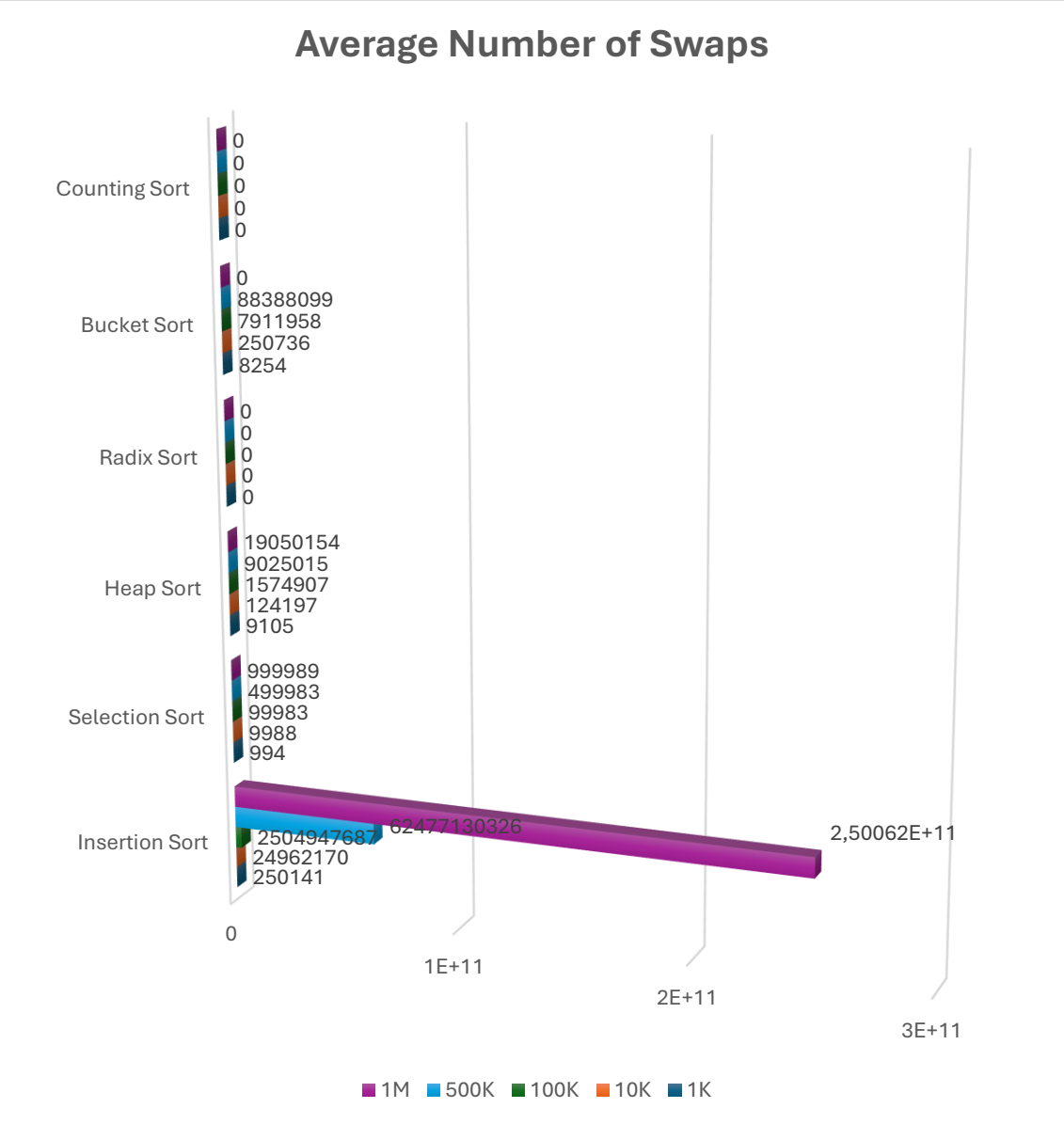
3.1. TEMPO MÉDIO DE EXECUÇÃO (ms)

Avg. Time (ms)					
Array size/Sorting Alg	**1K**	**10K**	**100K**	**500K**	**1M**
Insertion Sort	0	9	449	12028	49465
Selection Sort	0	22	1809	45071	219759
Heap Sort	0	2	15	75	156
Radix Sort	0	0	5	24	47
Bucket Sort	0	3	59	767	ERROR
Counting Sort	0	0	0	3	18



3.2. QUANTIDADE MÉDIA DE TROCAS (*SWAPS*):

Avg. Swaps					
Array size/Sorting Alg	**1K**	**10K**	**100K**	**500K**	**1M**
Insertion Sort	250141	24962170	2504947687	62477130326	2,50062E+11
Selection Sort	994	9988	99983	499983	999989
Heap Sort	9105	124197	1574907	9025015	19050154
Radix Sort	0	0	0	0	0
Bucket Sort	8254	250736	7911958	88388099	ERROR
Counting Sort	0	0	0	0	0



3.3. QUANTIDADE MÉDIA DE ITERAÇÕES

Avg. Iterations					
Array size / Sorting Alg	**1K**	**10K**	**100K**	**500K**	**1M**
Insertion Sort	251140	24972169	2505047686	62477630325	2,50063E+11
Selection Sort	499500	49995000	4999950000	1,25E+11	5E+11
Heap Sort	19210	258394	3249814	18550030	39100308
Radix Sort	9027	120036	1500045	9000054	18000054
Bucket Sort	11121	280637	8211643	89887393	ERROR
Counting Sort	2997	29996	299998	1499998	2999998