UNIVERSIDADE FEDERAL DE GOIÁS – UFG
CAMPUS SAMAMBAIA
INSTITUTO DE INFORMÁTICA
BACHARELADO EM INTELIGÊNCIA ARTIFICIAL - BIA

Relatório de Desempenho de Métodos de Ordenação

Andriel F. Furtado

Carlos Henrique Rodrigues de Jesus

Marcelo Henrique Alves Pereira Sobrinho

Victor Lucas Sousa Arantes

GOIÂNIA

2023

Sumário

Introdução

Este trabalho tem como objetivo explorar a importância das tarefas de ordenação e busca em diversas áreas e atividades, desde negócios até solução de problemas cotidianos. Embora não sejam as tarefas principais em muitos contextos, a ordenação de dados e a busca por registros são constantes e podem ser cruciais para o sucesso em determinados projetos.

Neste sentido, este trabalho irá abordar a importância da ordenação de dados em dois momentos distintos - quando a base está totalmente desordenada e quando ela já está ordenada, mas recebe atualizações - bem como as diferentes formas de busca em conjuntos de dados ordenados ou não ordenados.

Serão apresentados exemplos e casos práticos que ilustram a relevância dessas tarefas em diversas áreas e como elas podem ser aprimoradas para obter melhores resultados. Além disso, serão discutidos alguns algoritmos e técnicas utilizados para a ordenação e busca de dados, bem como as vantagens e desvantagens de cada um deles.

Métodos de Ordenação

Brevemente vamos explicar um pouco sobre cada um dos métodos de ordenação, juntamente com as suas características, para assim entendermos melhor do que se trata cada um dos modelos. Vamos trabalhar com os seguintes modelos principais de ordenação:

1. **InsertionSort:** É um algoritmo simples e eficiente que organiza um conjunto de dados, um por um, comparando cada elemento com os demais e inserindo-os na posição correta, até que todos estejam em ordem. É como se você tivesse um monte de bolas com números bagunçadas em uma mesa e fosse pegando cada uma delas, comparando com as outras e inseri-las na posição correta de acordo com a ordem numérica. Esse processo é repetido até que todas as bolas com números estejam em ordem.
   O InsertionSort é considerado uma das formas mais básicas de ordenação.

2. **SelectionSort:** É um algoritmo que organiza um conjunto de dados encontrando o menor valor e o colocando na primeira posição, depois o segundo menor valor e colocando na segunda posição, e assim por diante, até que todo o conjunto esteja em ordem. É como se você estivesse escolhendo os menores valores em um conjunto de bolas com números e colocando-os em uma pilha separada. Esse processo é repetido até que todos os valores tenham sido adicionados à pilha e estejam em ordem.
   O SelectionSort é uma forma simples e eficiente de ordenação, mas pode ser menos eficiente do que outros algoritmos em conjuntos de dados muito grandes.

3. **ShellSort:** É um algoritmo que organiza um conjunto de dados em várias etapas, dividindo o conjunto em subgrupos menores e aplicando o método de ordenação por inserção em cada um desses subgrupos. Essa divisão em subgrupos permite a ordenação de elementos distantes entre si e melhora a eficiência do algoritmo em relação ao Insertion Sort. É como se você tivesse um conjunto de bolas e fosse separando-as em grupos menores. Em seguida, ordenaria cada grupo usando o método de ordenação por inserção e, por fim, juntaria todos os grupos em um único conjunto ordenado.
   O ShellSort é considerado um algoritmo de ordenação eficiente em muitas situações, especialmente em conjuntos de dados de tamanho relativamente moderado.
4. **MergeSort:** É um algoritmo que divide um conjunto de dados em partes menores, ordena cada parte e, em seguida, mescla as partes ordenadas para produzir o conjunto final ordenado. É como se você tivesse um monte de cartas, dividisse-as em pilhas menores, ordenasse cada pilha e, em seguida, mesclasse essas pilhas menores em uma única pilha ordenada.
   O MergeSort é considerado um dos algoritmos de ordenação mais eficientes, especialmente em conjuntos de dados muito grandes. Ele é capaz de lidar com conjuntos de dados que não cabem na memória do computador.

5. **QuickSort:** É um algoritmo que divide um conjunto de dados em duas partes menores, com base em um elemento principal, e em seguida ordena essas partes recursivamente. É como se você tivesse um monte de cartas, escolhesse uma carta

como principal, e separasse as outras cartas em duas pilhas menores, uma para cartas menores que a principal e outra para cartas maiores. Em seguida, ordenar essas duas pilhas menores, usando o mesmo processo de dividir e escolher uma nova principal, e juntar as pilhas ordenadas em uma única pilha.
O QuickSort é considerado um dos algoritmos de ordenação mais eficientes, especialmente em conjuntos de dados maiores. Contudo, ele pode ser menos eficiente em casos em que o conjunto de dados está quase ordenado, pois a principal pode acabar sendo escolhida de forma ineficiente.

Métodos de busca

1. **Busca binária**

   1.1. **Funcionamento geral:**

   - A busca binária é aplicável apenas a conjuntos de dados ordenados.
   - Divide repetidamente o conjunto de dados ao meio, comparando o valor procurado com o valor no meio do conjunto de dados.
   - Continua a dividir o conjunto de dados pela metade até encontrar o valor procurado ou determinar que o valor não está presente.

   1.2. **Quando usar:**

   - Conjuntos de dados grandes e ordenados.
   - Quando a eficiência é uma prioridade, ou seja, quando o tempo de execução é crítico.
   - Quando a posição do elemento procurado é conhecida.

   1.2. **Quando não usar:**

   - Conjuntos de dados pequenos ou desordenados.
   - Quando a posição do elemento procurado é desconhecida.

   É muito eficiente para encontrar um elemento em um conjunto de dados grande e ordenado. No entanto, ela só pode ser usada em conjuntos de dados ordenados, o que pode limitar sua utilidade em algumas situações.

   Além disso, é importante lembrar que a busca binária só é útil se a posição do elemento procurado for conhecida. Se não soubermos em qual parte do conjunto de dados o elemento está, ainda pode ser necessário usar métodos de busca mais genéricos, como a busca sequencial.

2. **Busca sequencial**

2.1. **Funcionamento geral:**

- Percorre os elementos do conjunto de dados sequencialmente, comparando cada elemento com o elemento procurado.
- Para quando encontra a correspondência ou chega ao final do conjunto de dados.

2.2. **Quando usar:**

- Conjuntos de dados pequenos.
- Quando a posição do elemento procurado é desconhecida.
- Quando a eficiência não é uma prioridade, ou seja, quando o tempo de execução não é crítico.

2.3. **Quando não usar:**

- Conjuntos de dados grandes.
- Quando a posição do elemento procurado é conhecida.
- Quando a eficiência é uma prioridade, ou seja, quando o tempo de execução é crítico.

É eficiente em certas situações, especialmente quando se trabalha com conjuntos de dados menores ou quando a posição do elemento procurado é desconhecida.

Ferramenta Desenvolvida

Inicialmente a ferramenta desenvolvida vai exibir um menu inicial, na qual mostra todos os bancos de dados que estão sendo utilizados, assim você pode fazer a seleção com qual deles você deseja trabalhar, após a seleção do banco de dados, irá exibir uma seleção na qual você pode optar por fazer uma busca ou uma ordenação.

Caso 1: Em caso da seleção da busca, outro menu irá aparecer na qual você pode optar pela busca sequencial ou pela busca binária.

Caso 1.1: Se optar pela busca sequencial, vai ser printado a mensagem de qual ID você deseja buscar, após inserir ele vai exibir o número de comparações e em qual posição o ID se encontra.

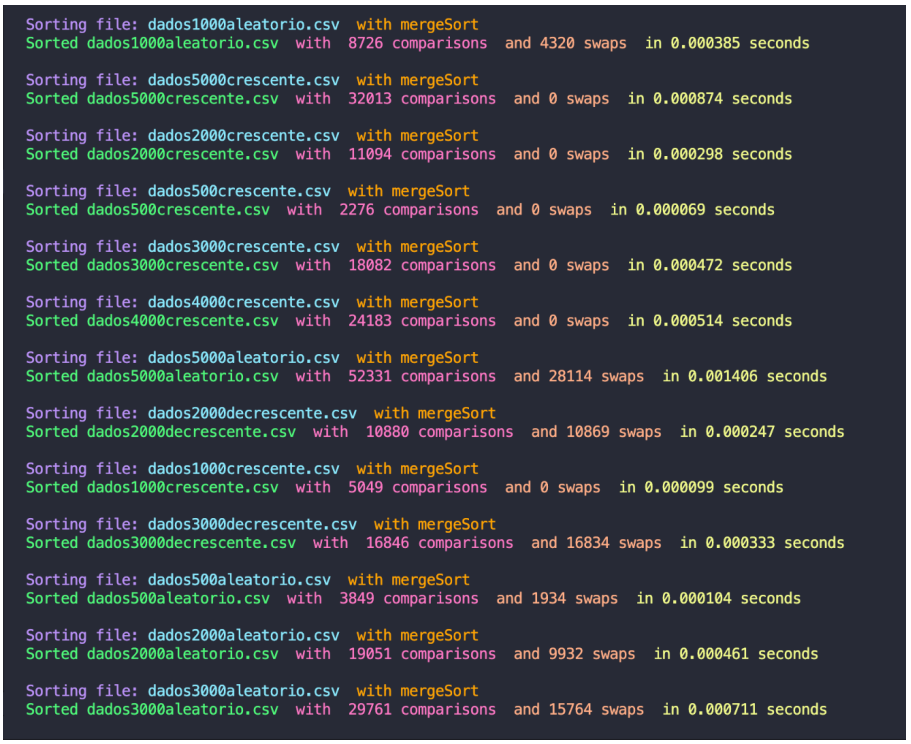
Caso 1.2: Caso opte pela busca binária, o algoritmo vai printar um menu que pergunta qual ordenação você deseja utilizar, após a seleção, ele pergunta o ID que você deseja buscar e depois de inserir a informação, é printado a posição do ID desejado e o número de comparações feito.

Caso 2: Já no caso da seleção da ordenação, o algoritmo vai printar outro menu, na qual pede a informação de qual algoritmo de ordenação você deseja usar, após a seleção do algoritmo, a ordenação é feita e ao final ele exibe uma lista ordenada, com o número de comparações feita, o número de trocas e o tempo gasto para o total da execução.

Os principais problemas que encontramos para a idealização e execução do algoritmo foram mais por parte da linguagem em C, que dificultou um pouco mais esse processo, ainda mais pela adaptação que tivemos que ter do python para C, que nesse caso em específico foi um pouco mais delicado e confuso, mas conseguimos suprir grandes necessidades em si, que foram desenvolvidas.

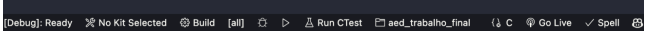
Resultados obtidos e conclusões

Ao analisar os resultados obtidos sobre os algoritmos de ordenação, podemos notar que cada um deles apresenta um desempenho diferente, dependendo do tipo de arquivo a ser ordenado. O mergeSort, por exemplo, mostrou-se bastante eficiente para arquivos aleatórios e crescentes, sendo capaz de ordená-los com um número baixo de comparações e swaps em um curto espaço de tempo. Já o insertionSort apresentou um desempenho inferior em relação ao mergeSort, sendo mais lento e necessitando de um número maior de comparações e swaps para ordenar os arquivos. O selectionSort, por sua vez, apresentou um desempenho razoável, porém inferior ao mergeSort, sendo bastante ineficiente para arquivos grandes e em ordem crescente. Por fim, o quickSort e o shellSort também apresentaram resultados interessantes, com o quickSort mostrando-se bastante eficiente para arquivos aleatórios e o shellSort apresentando bons resultados para arquivos pequenos. Em suma, a escolha do algoritmo de ordenação deve ser feita com base no tipo de arquivo a ser ordenado, levando em consideração o desempenho de cada algoritmo em relação àquele arquivo específico.



```
Sorting file: dados1000aleatorio.csv  with insertionSort
Sorted dados1000aleatorio.csv  with  251371 comparisons  and 991 swaps  in 0.002414 seconds

Sorting file: dados5000crescente.csv  with insertionSort
Sorted dados5000crescente.csv  with  5000 comparisons  and 0 swaps  in 0.000048 seconds

Sorting file: dados2000crescente.csv  with insertionSort
Sorted dados2000crescente.csv  with  2000 comparisons  and 0 swaps  in 0.000008 seconds

Sorting file: dados500crescente.csv  with insertionSort
Sorted dados500crescente.csv  with  500 comparisons  and 0 swaps  in 0.000002 seconds

Sorting file: dados3000crescente.csv  with insertionSort
Sorted dados3000crescente.csv  with  3000 comparisons  and 0 swaps  in 0.000010 seconds

Sorting file: dados4000crescente.csv  with insertionSort
Sorted dados4000crescente.csv  with  4000 comparisons  and 0 swaps  in 0.000013 seconds

Sorting file: dados5000aleatorio.csv  with insertionSort
Sorted dados5000aleatorio.csv  with  6283078 comparisons  and 4994 swaps  in 0.013533 seconds

Sorting file: dados2000decrescente.csv  with insertionSort
Sorted dados2000decrescente.csv  with  2001000 comparisons  and 1999 swaps  in 0.004410 seconds

Sorting file: dados1000crescente.csv  with insertionSort
Sorted dados1000crescente.csv  with  1000 comparisons  and 0 swaps  in 0.000003 seconds

Sorting file: dados3000decrescente.csv  with insertionSort
Sorted dados3000decrescente.csv  with  4501500 comparisons  and 2999 swaps  in 0.009838 seconds

Sorting file: dados500aleatorio.csv  with insertionSort
Sorted dados500aleatorio.csv  with  64747 comparisons  and 496 swaps  in 0.000146 seconds

Sorting file: dados2000aleatorio.csv  with insertionSort
Sorted dados2000aleatorio.csv  with  991393 comparisons  and 1994 swaps  in 0.002189 seconds

Sorting file: dados3000aleatorio.csv  with insertionSort
Sorted dados3000aleatorio.csv  with  2279019 comparisons  and 2996 swaps  in 0.004979 seconds
```

```
Sorting file: dados1000aleatorio.csv  with selectionSort
Sorted dados1000aleatorio.csv  with  500500 comparisons  and 991 swaps  in 0.002477 seconds

Sorting file: dados5000crescente.csv  with selectionSort
Sorted dados5000crescente.csv  with  12502500 comparisons  and 0 swaps  in 0.053723 seconds

Sorting file: dados2000crescente.csv  with selectionSort
Sorted dados2000crescente.csv  with  2001000 comparisons  and 0 swaps  in 0.005856 seconds

Sorting file: dados500crescente.csv  with selectionSort
Sorted dados500crescente.csv  with  125250 comparisons  and 0 swaps  in 0.000343 seconds

Sorting file: dados3000crescente.csv  with selectionSort
Sorted dados3000crescente.csv  with  4501500 comparisons  and 0 swaps  in 0.011749 seconds

Sorting file: dados4000crescente.csv  with selectionSort
Sorted dados4000crescente.csv  with  8002000 comparisons  and 0 swaps  in 0.018182 seconds

Sorting file: dados5000aleatorio.csv  with selectionSort
Sorted dados5000aleatorio.csv  with  12502500 comparisons  and 4994 swaps  in 0.026435 seconds

Sorting file: dados2000decrescente.csv  with selectionSort
Sorted dados2000decrescente.csv  with  2001000 comparisons  and 1000 swaps  in 0.004080 seconds

Sorting file: dados1000crescente.csv  with selectionSort
Sorted dados1000crescente.csv  with  500500 comparisons  and 0 swaps  in 0.001060 seconds

Sorting file: dados3000decrescente.csv  with selectionSort
Sorted dados3000decrescente.csv  with  4501500 comparisons  and 1500 swaps  in 0.009620 seconds

Sorting file: dados500aleatorio.csv  with selectionSort
Sorted dados500aleatorio.csv  with  125250 comparisons  and 495 swaps  in 0.000279 seconds

Sorting file: dados2000aleatorio.csv  with selectionSort
Sorted dados2000aleatorio.csv  with  2001000 comparisons  and 1991 swaps  in 0.004272 seconds

Sorting file: dados3000aleatorio.csv  with selectionSort
Sorted dados3000aleatorio.csv  with  4501500 comparisons  and 2993 swaps  in 0.009660 seconds
```




```
Sorting file: dados1000aleatorio.csv  with quickSort
Sorted dados1000aleatorio.csv  with  10685 comparisons  and 5654 swaps  in 0.000142 seconds

Sorting file: dados5000crescente.csv  with quickSort
Sorted dados5000crescente.csv  with  12502500 comparisons  and 12507500 swaps  in 0.043370 seconds

Sorting file: dados2000crescente.csv  with quickSort
Sorted dados2000crescente.csv  with  2001000 comparisons  and 2003000 swaps  in 0.005922 seconds

Sorting file: dados500crescente.csv  with quickSort
Sorted dados500crescente.csv  with  125250 comparisons  and 125750 swaps  in 0.000394 seconds

Sorting file: dados3000crescente.csv  with quickSort
Sorted dados3000crescente.csv  with  4501500 comparisons  and 4504500 swaps  in 0.013168 seconds

Sorting file: dados4000crescente.csv  with quickSort
Sorted dados4000crescente.csv  with  8002000 comparisons  and 8006000 swaps  in 0.022828 seconds

Sorting file: dados5000aleatorio.csv  with quickSort
Sorted dados5000aleatorio.csv  with  66105 comparisons  and 36570 swaps  in 0.000361 seconds

Sorting file: dados2000decrescente.csv  with quickSort
Sorted dados2000decrescente.csv  with  1999001 comparisons  and 1001000 swaps  in 0.005111 seconds

Sorting file: dados1000crescente.csv  with quickSort
Sorted dados1000crescente.csv  with  500500 comparisons  and 501500 swaps  in 0.001504 seconds

Sorting file: dados3000decrescente.csv  with quickSort
Sorted dados3000decrescente.csv  with  4498501 comparisons  and 2251500 swaps  in 0.011158 seconds

Sorting file: dados500aleatorio.csv  with quickSort
Sorted dados500aleatorio.csv  with  5187 comparisons  and 2848 swaps  in 0.000031 seconds

Sorting file: dados2000aleatorio.csv  with quickSort
Sorted dados2000aleatorio.csv  with  23653 comparisons  and 12172 swaps  in 0.000137 seconds

Sorting file: dados3000aleatorio.csv  with quickSort
Sorted dados3000aleatorio.csv  with  42006 comparisons  and 22056 swaps  in 0.000214 seconds
```

```
Sorting file: dados1000aleatorio.csv  with shellSort
Sorted dados1000aleatorio.csv  with  8091 comparisons  and 8091 swaps  in 0.000446 seconds

Sorting file: dados5000crescente.csv  with shellSort
Sorted dados5000crescente.csv  with  0 comparisons  and 0 swaps  in 0.000461 seconds

Sorting file: dados2000crescente.csv  with shellSort
Sorted dados2000crescente.csv  with  0 comparisons  and 0 swaps  in 0.000047 seconds

Sorting file: dados500crescente.csv  with shellSort
Sorted dados500crescente.csv  with  0 comparisons  and 0 swaps  in 0.000009 seconds

Sorting file: dados3000crescente.csv  with shellSort
Sorted dados3000crescente.csv  with  0 comparisons  and 0 swaps  in 0.000074 seconds

Sorting file: dados4000crescente.csv  with shellSort
Sorted dados4000crescente.csv  with  0 comparisons  and 0 swaps  in 0.000092 seconds

Sorting file: dados5000aleatorio.csv  with shellSort
Sorted dados5000aleatorio.csv  with  75352 comparisons  and 75352 swaps  in 0.000687 seconds

Sorting file: dados2000decrescente.csv  with shellSort
Sorted dados2000decrescente.csv  with  10400 comparisons  and 10400 swaps  in 0.000084 seconds

Sorting file: dados1000crescente.csv  with shellSort
Sorted dados1000crescente.csv  with  0 comparisons  and 0 swaps  in 0.000021 seconds

Sorting file: dados3000decrescente.csv  with shellSort
Sorted dados3000decrescente.csv  with  15692 comparisons  and 15692 swaps  in 0.000130 seconds

Sorting file: dados500aleatorio.csv  with shellSort
Sorted dados500aleatorio.csv  with  3113 comparisons  and 3113 swaps  in 0.000051 seconds

Sorting file: dados2000aleatorio.csv  with shellSort
Sorted dados2000aleatorio.csv  with  21271 comparisons  and 21271 swaps  in 0.000260 seconds

Sorting file: dados3000aleatorio.csv  with shellSort
Sorted dados3000aleatorio.csv  with  32189 comparisons  and 32189 swaps  in 0.000413 seconds
```