Ministério da Educação
UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
Campus Medianeira
Curso de Bacharelado em Ciência da Computação

Comparação dos métodos de ordenação e pesquisa

Augusto Azevedo

1. RESUMO

Este artigo avalia o desempenho de diversos métodos de ordenação e pesquisa de dados em listas de números inteiros. Os métodos de ordenação avaliados incluem Bubblesort, Insertsort, Selectionsort, Shellsort, Quicksort (Hoare e Lomuto), Mergesort, Radixsort e Heapsort.

As avaliações foram realizadas em listas de 500.000, 750.000 e 1.000.000 de registros, considerando três cenários distintos: melhor caso, pior caso e caso aleatório. A avaliação dos métodos de pesquisa também considerou os mesmos três cenários. Os resultados mostraram que os métodos de ordenação e pesquisa apresentam desempenhos diferentes dependendo do tamanho do arquivo.

Meu código completo GitHub: https://github.com/azevedontc/datasearchandSorting.

2. INTRODUÇÃO

A ordenação e pesquisa de dados são processos fundamentais na área de Ciência da Computação. A ordenação de dados envolve a reorganização de um conjunto de dados em uma ordem específica, enquanto a pesquisa de dados envolve a localização de um determinado registro dentro de um conjunto de dados. A eficiência desses processos é fundamental para a melhoria do desempenho de sistemas de computação, especialmente em aplicações que envolvem grande quantidade de dados.

O objetivo deste estudo é avaliar o desempenho de diversos métodos de ordenação e pesquisa de dados em listas de números inteiros. Os métodos de ordenação avaliados incluem Bubblesort, Insertsort, Selectionsort, Shellsort, Quicksort (Hoare e Lomuto), Mergesort, Radixsort e Heapsort. Os métodos de pesquisa avaliados são sequenciais e binário.

3. MATERIAIS E MÉTODOS

Para a implementação do meu código utilizei o **REPLIT**, e está hospedado no meu GitHub: https://github.com/azevedontc/datasearchandSorting

Os algoritmos de Ordenação utilizados foram:

1. BubbleSort
2. InsertSort
3. SelectionSort
4. ShellSort
5. QuickSort Hoare
6. QuickSort Lomuto
7. MergeSort
8. RadixSort
9. HeapSort

E de Pesquisa, os utilizados foram:

1. Sequencial
2. Binária

Meu computador utiliza um I5 de 4° geração e 8GB de memória RAM e um SSD de 120GB, mas é pouco relevante pois o **REPLIT** limita meu hardware, mantendo assim, o tempo relativamente igual em qualquer máquina.

O Procedimento para medir o tempo de execução do algoritmo consiste basicamente em iniciar a contagem quando chamar a função, e finalizar quando entregar o arquivo ordenado:

```
case 1:
  inicio = clock();
  bubbleSort(numeros, tam);
  fim = clock();
  sprintf(nomeAlgoritimo, "BublleSort");
  break;
```

“inicio = clock();” Inicia a contagem do tempo

“fim = clock();” Finaliza a contagem do tempo

```
double times(clock_t inicio) {
  clock_t fim = clock();
  double tempo;
  tempo = (double)(fim - inicio) / CLOCKS_PER_SEC;
  printf("Tempo de execução: %.4f segundos.\n", tempo);
  return tempo;
}
```

Meu algoritmo que está implementado a função do tempo e realiza o cálculo.

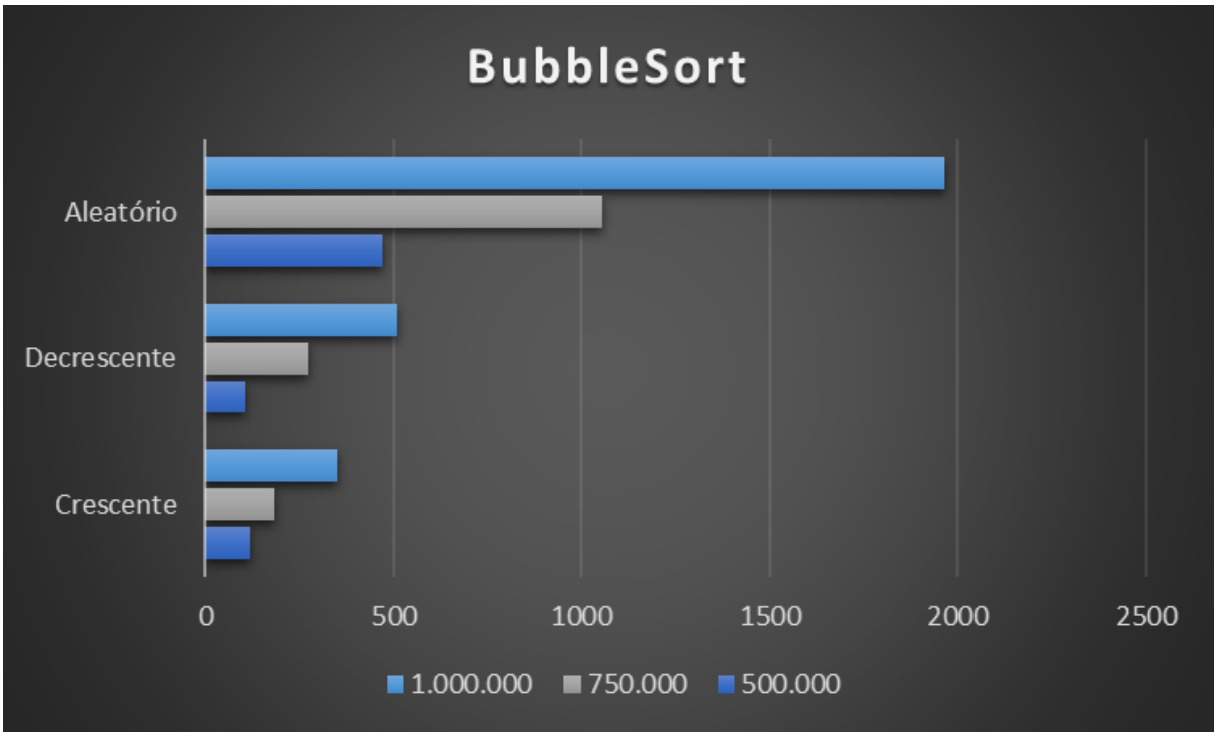
4. RESULTADOS E DISCUSSÃO

Tempos de Execução BubbleSort

Formatação dos dados em forma de Tabela:

Algoritmo	Organização dos Dados	Quantidade de Dados	Tempo de Execução
BubbleSort	Crescente	500.000	119 seg.
		750.000	182 seg.
		1.000.000	350 seg.
	Decrescente	500.000	104 seg.
		750.000	272 seg.
		1.000.000	511 seg.
	Aleatório	500.000	469 seg.
		750.000	1056 seg.
		1.000.000	1966 seg.

Formatação dos dados em forma de Gráfico:



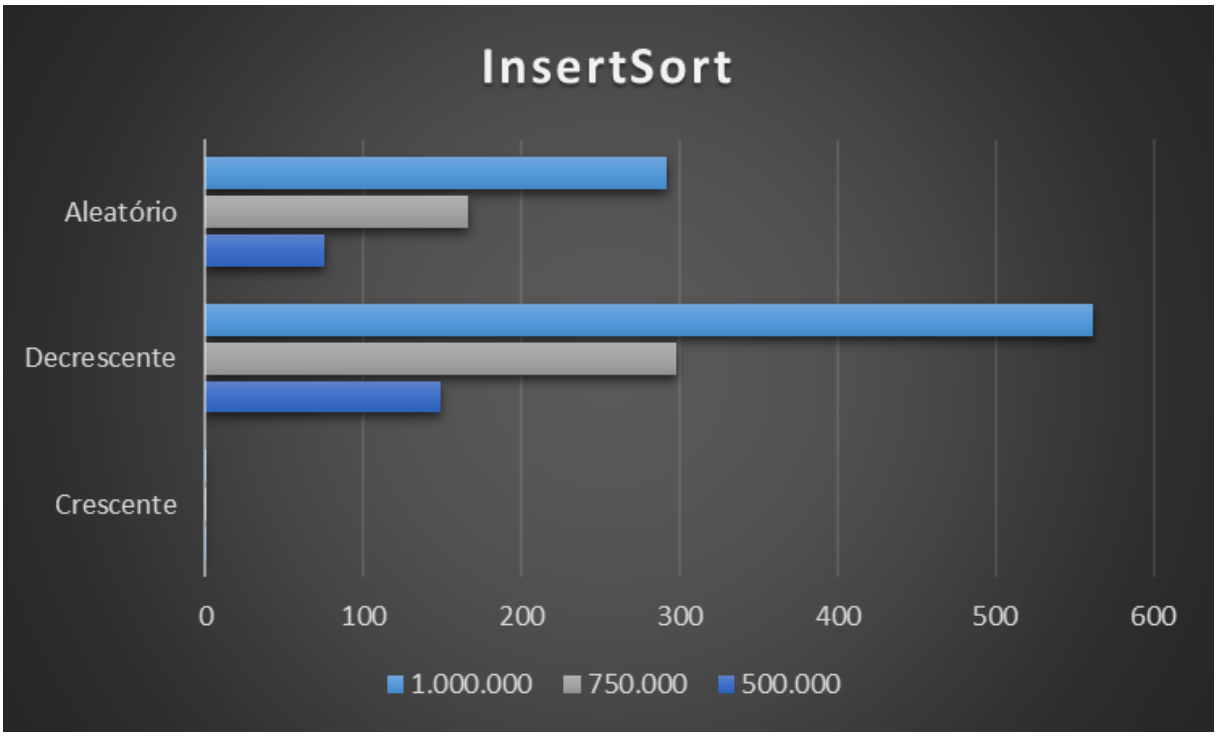
O BubbleSort é um algoritmo de ordenação simples e ineficiente que tem complexidade O(n^2). Ele é útil apenas para pequenos conjuntos de dados, pois seu tempo de execução aumenta exponencialmente à medida que o número de elementos a serem ordenados aumenta.

Tempos de Execução InsertSort

Formatação dos dados em forma de Tabela:

Algoritmo	Organização dos Dados	Quantidade de Dados	Tempo de Execução
InsertSort	Crescente	500.000	0.008 seg.
		750.000	0.013 seg.
		1.000.000	0.016 seg.
	Decrescente	500.000	149 seg.
		750.000	298 seg.
		1.000.000	562 seg.
	Aleatório	500.000	75 seg.
		750.000	166 seg.
		1.000.000	292 seg.

Formatação dos dados em forma de Gráfico:



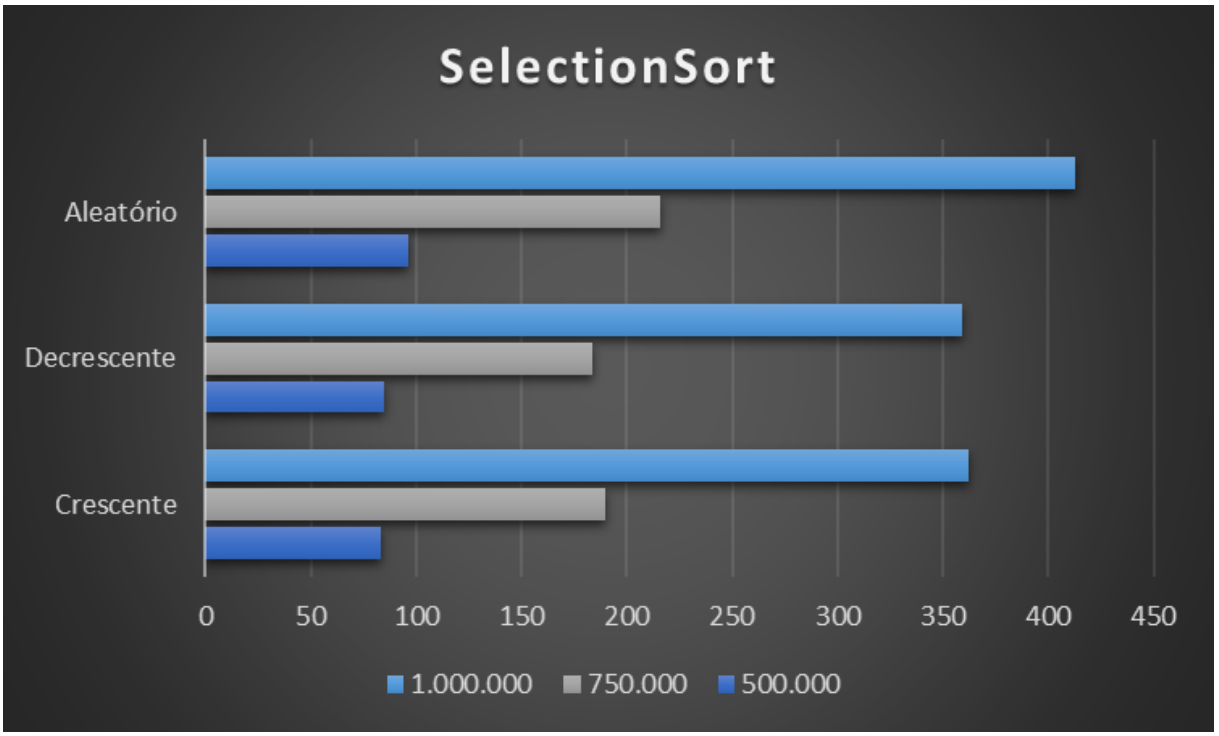
O InsertSort também tem complexidade O(n^2), mas é mais eficiente que o BubbleSort. Ele é adequado para conjuntos de dados pequenos e pode ser uma boa opção para conjuntos de dados parcialmente ordenados, como vimos ele foi muito rápido com arquivo já ordenado.

Tempos de Execução SelectionSort

Formatação dos dados em forma de Tabela:

Algoritmo	Organização dos Dados	Quantidade de Dados	Tempo de Execução
SelectionSort	Crescente	500.000	83 seg.
		750.000	190 seg.
		1.000.000	362 seg.
	Decrescente	500.000	85 seg.
		750.000	184 seg.
		1.000.000	359 seg.
	Aleatório	500.000	96 seg.
		750.000	216 seg.
		1.000.000	413 seg.

Formatação dos dados em forma de Gráfico:



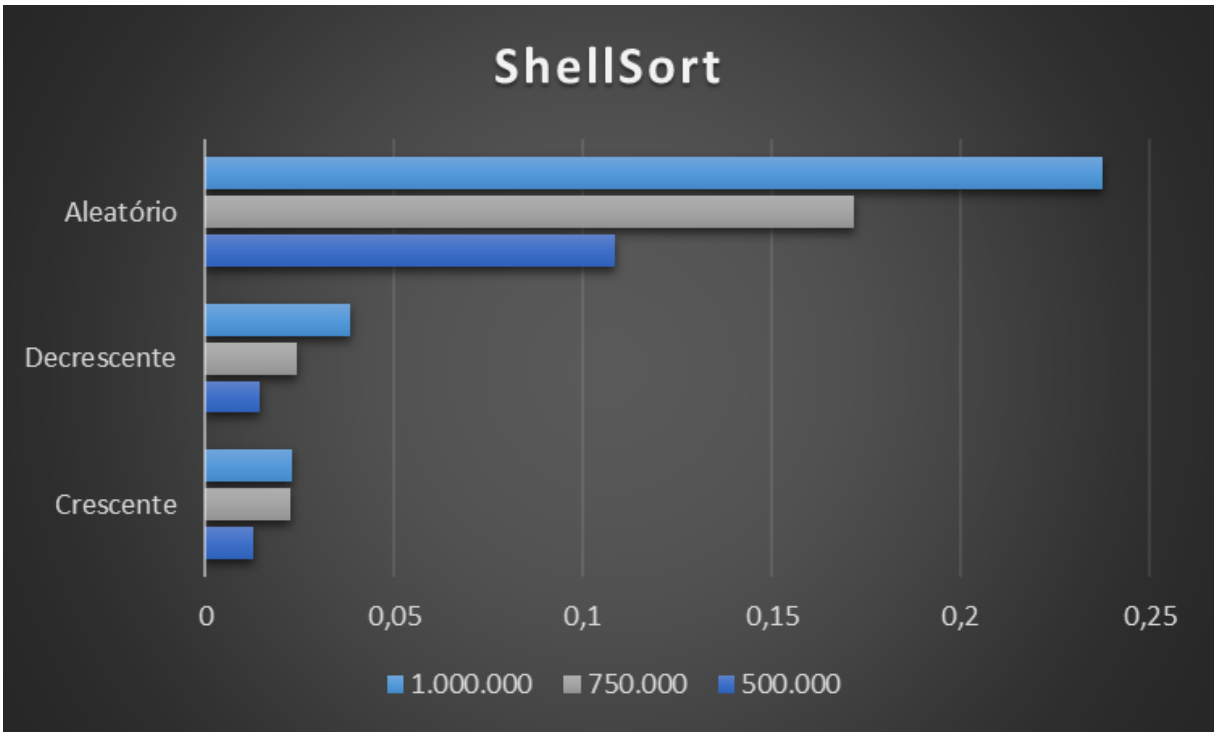
O SelectionSort tem complexidade O(n^2) e é um dos algoritmos de ordenação mais simples. Ele é adequado para conjuntos de dados pequenos, mas é ineficiente para conjuntos de dados maiores.

Tempos de Execução ShellSort

Formatação dos dados em forma de Tabela:

Algoritmo	Organização dos Dados	Quantidade de Dados	Tempo de Execução
ShellSort	Crescente	500.000	0.0128 seg.
		750.000	0.0223 seg.
		1.000.000	0.0230 seg.
	Decrescente	500.000	0.0142 seg.
		750.000	0.0243 seg.
		1.000.000	0.385 seg.
	Aleatório	500.000	0.1085 seg.
		750.000	0.1718 seg.
		1.000.000	0.2375 seg.

Formatação dos dados em forma de Gráfico:



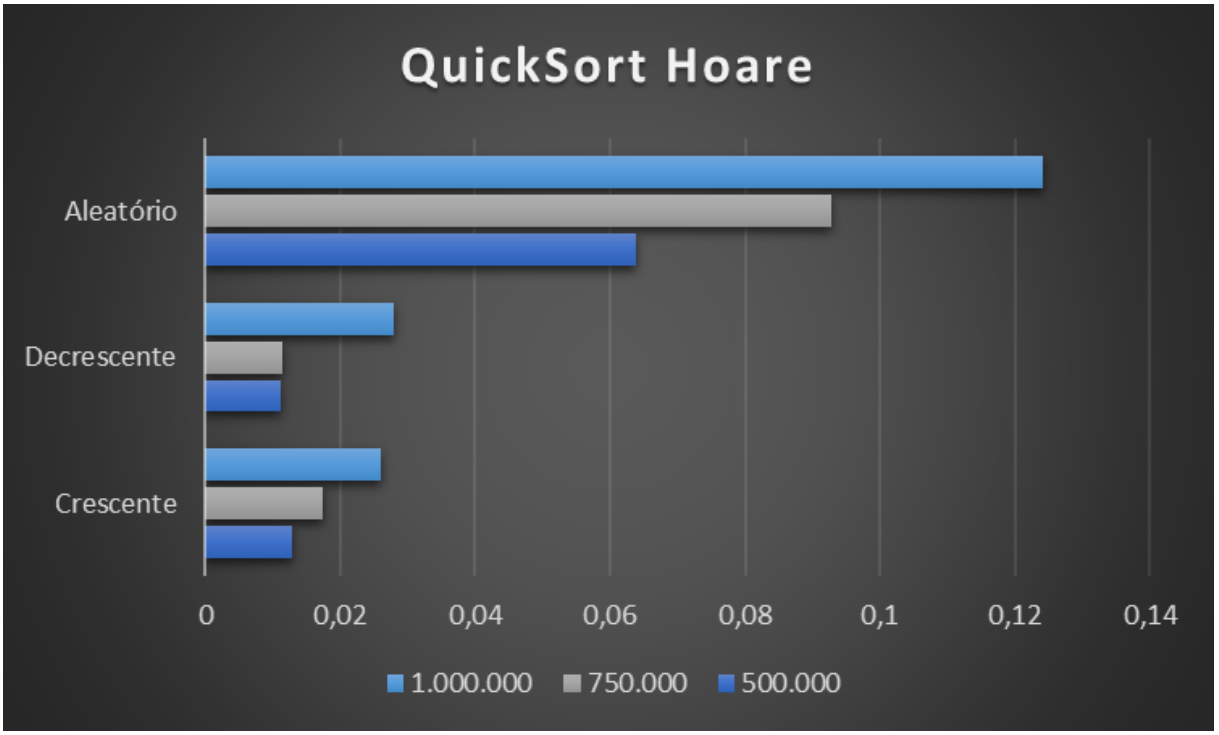
O ShellSort é um algoritmo de ordenação mais eficiente do que os três anteriores, com complexidade O(n log n) ou melhor. Ele é especialmente útil para conjuntos de dados grandes.

Tempos de Execução QuickSort Hoare

Formatação dos dados em forma de Tabela:

Algoritmo	Organização dos Dados	Quantidade de Dados	Tempo de Execução
QuickSortHoare	Crescente	500.000	0.0128 seg.
		750.000	0.0174 seg.
		1.000.000	0.0261 seg.
	Decrescente	500.000	0.0112 seg.
		750.000	0.0115 seg.
		1.000.000	0.0279 seg.
	Aleatório	500.000	0.0638 seg.
		750.000	0.0929 seg.
		1.000.000	0.1241 seg.

Formatação dos dados em forma de Gráfico:



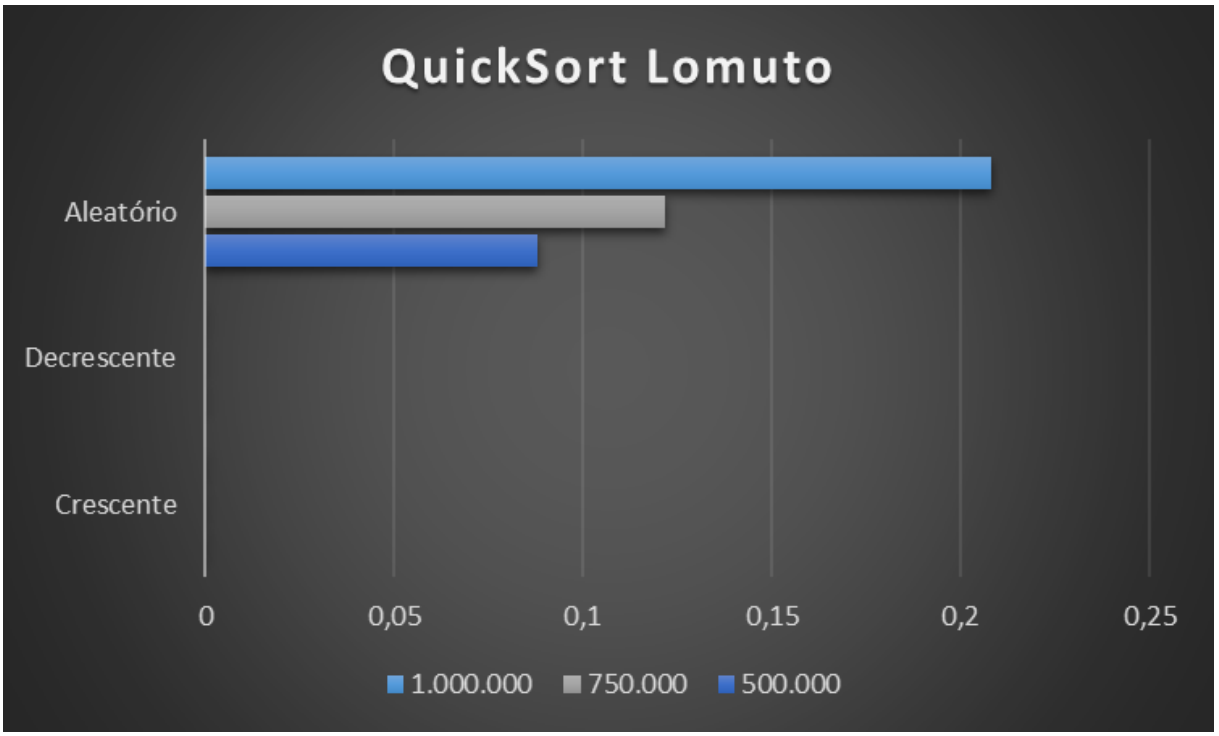
O QuickSort Hoare é um algoritmo de ordenação rápido e eficiente que tem complexidade O(n log n) em média, mas pode chegar a O(n^2) no pior caso. Ele é amplamente utilizado em aplicações devido à sua velocidade e eficiência.

Tempos de Execução QuickSort Lomuto

Formatação dos dados em forma de Tabela:

Algoritmo	Organização dos Dados	Quantidade de Dados	Tempo de Execução
QuickSortLomuto	Crescente	500.000	0 seg.
		750.000	0 seg.
		1.000.000	0 seg.
	Decrescente	500.000	0 seg.
		750.000	0 seg.
		1.000.000	0 seg.
	Aleatório	500.000	0.0879 seg.
		750.000	0.1215 seg.
		1.000.000	0.2079 seg.

Formatação dos dados em forma de Gráfico:



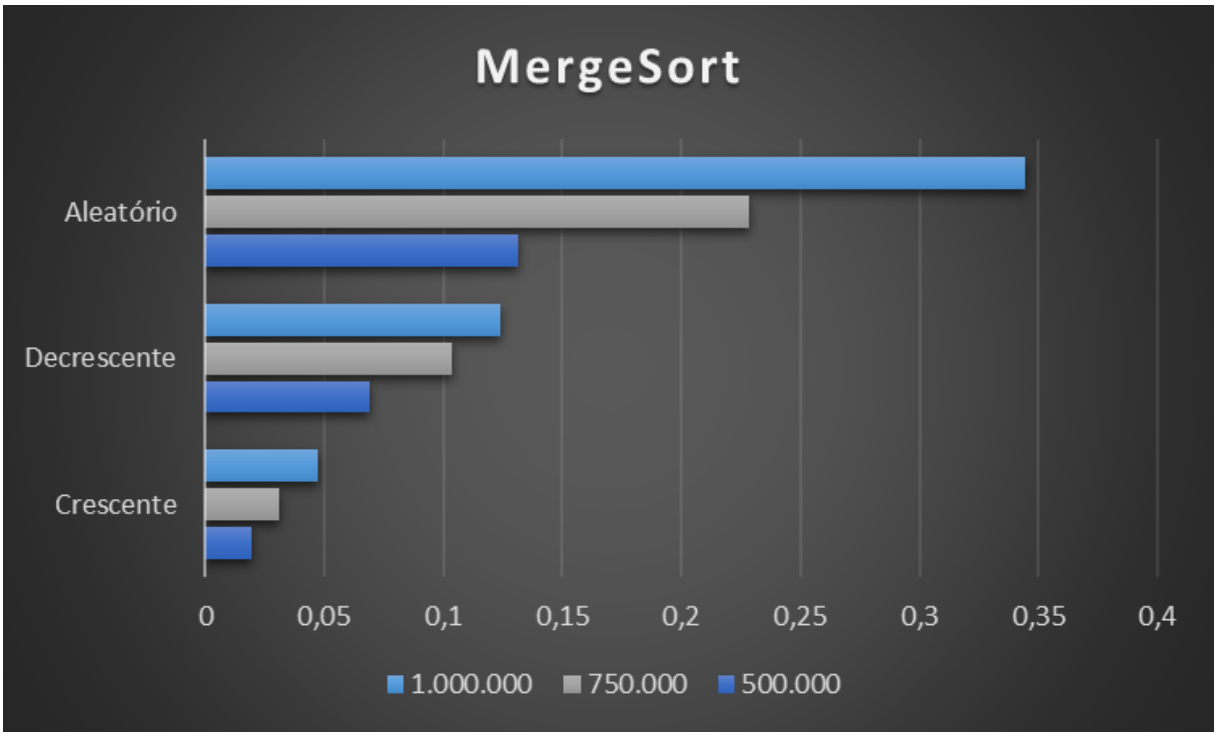
O QuickSort Lomuto é semelhante ao QuickSort Hoare, mas tem uma implementação mais simples. Ele tem complexidade O(n log n) em média e O(n^2) no pior caso.
(Meu código não está funcionando com arquivo Crescente e Decrescente).

Tempos de Execução MergeSort

Formatação dos dados em forma de Tabela:

Algoritmo	Organização dos Dados	Quantidade de Dados	Tempo de Execução
MergeSort	Crescente	500.000	0.0196 seg.
		750.000	0.0310 seg.
		1.000.000	0.0475 seg.
	Decrescente	500.000	0.0691 seg.
		750.000	0.1035 seg.
		1.000.000	0.1242 seg.
	Aleatório	500.000	0.1314 seg.
		750.000	0.2285 seg.
		1.000.000	0.3444 seg.

Formatação dos dados em forma de Gráfico:



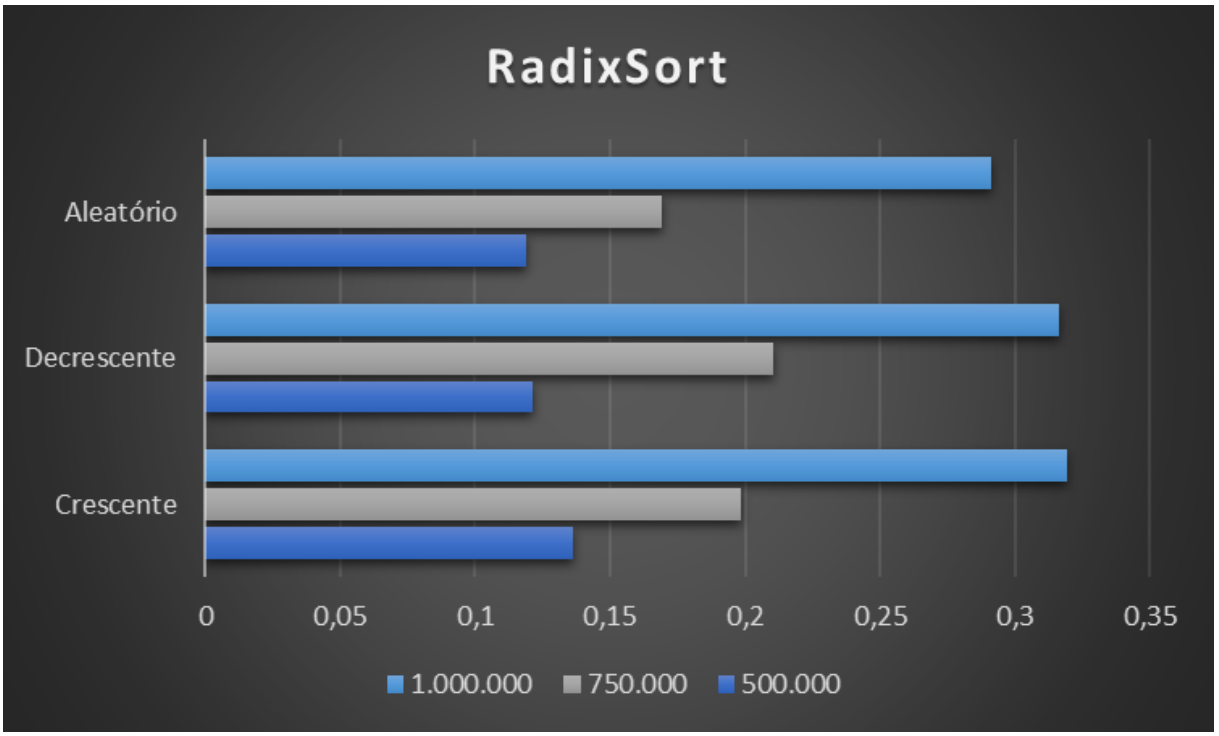
O MergeSort é um algoritmo de ordenação eficiente com complexidade O(n log n). Ele é útil para conjuntos de dados grandes, mas requer espaço adicional para alocar uma matriz temporária durante a execução.

Tempos de Execução RadixSort

Formatação dos dados em forma de Tabela:

Algoritmo	Organização dos Dados	Quantidade de Dados	Tempo de Execução
RadixSort	Crescente	500.000	0.1363 seg.
		750.000	0.1984 seg.
		1.000.000	0.3196 seg.
	Decrescente	500.000	0.1210 seg.
		750.000	0.2103 seg.
		1.000.000	0.3162 seg.
	Aleatório	500.000	0.1191 seg.
		750.000	0.1691 seg.
		1.000.000	0.2915 seg.

Formatação dos dados em forma de Gráfico:



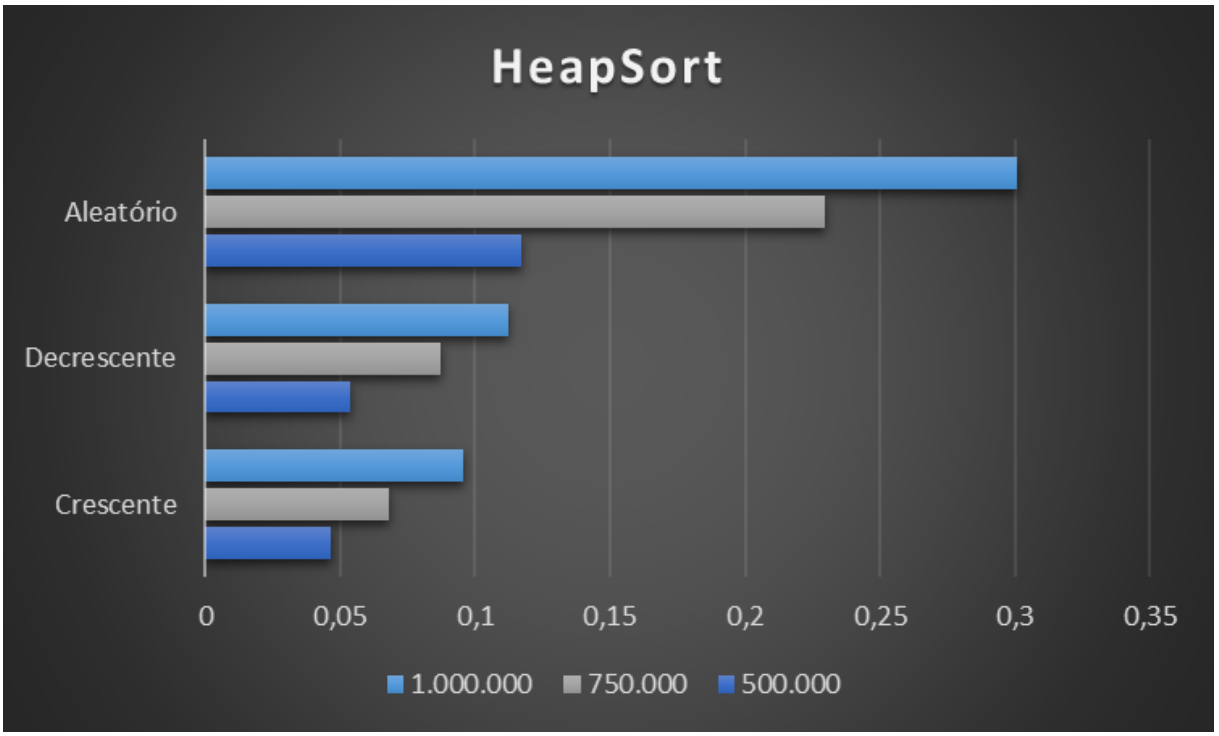
O RadixSort é um algoritmo de ordenação que tem complexidade O(nk), onde n é o número de elementos a serem ordenados e k é o número de dígitos. Ele é especialmente útil para ordenar números inteiros de tamanho fixo.

Tempos de Execução HeapSort

Formatação dos dados em forma de Tabela:

Algoritmo	Organização dos Dados	Quantidade de Dados	Tempo de Execução
HeapSort	Crescente	500.000	0.0466 seg.
		750.000	0.0679 seg.
		1.000.000	0.0954 seg.
	Decrescente	500.000	0.0538 seg.
		750.000	0.0870 seg.
		1.000.000	0.1122 seg.
	Aleatório	500.000	0.1172 seg.
		750.000	0.2297 seg.
		1.000.000	0.3008 seg.

Formatação dos dados em forma de Gráfico:



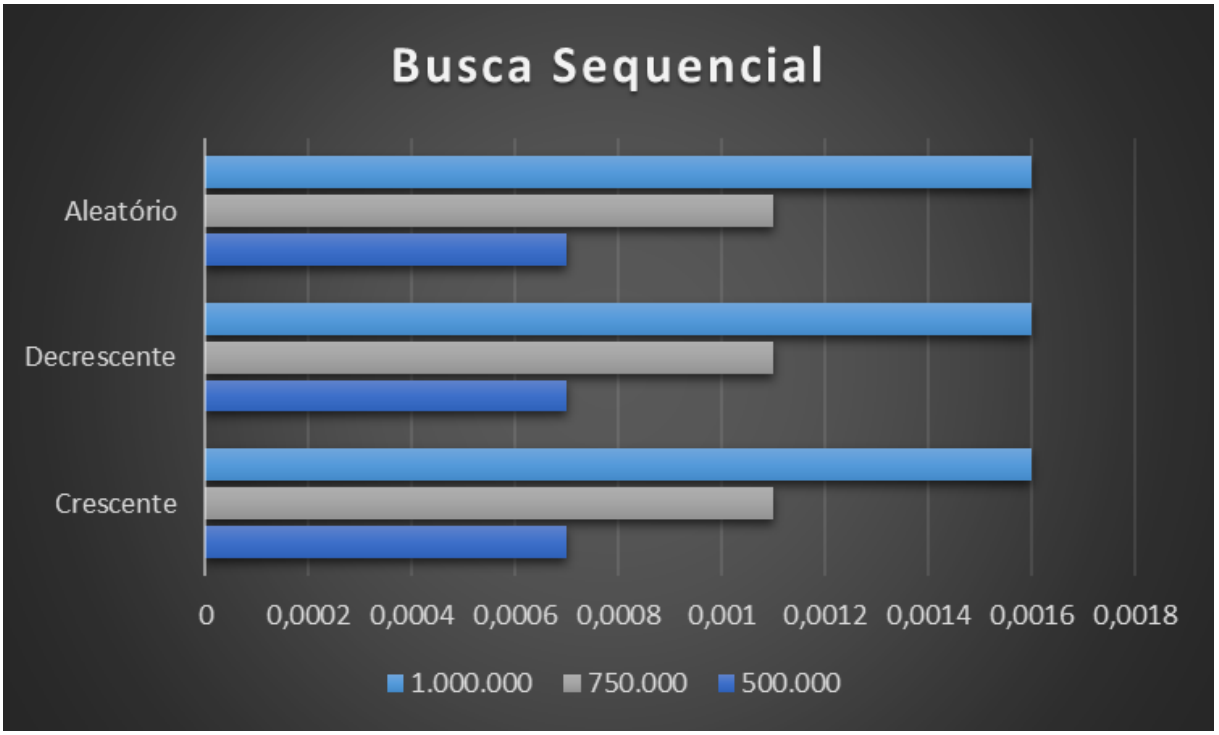
O HeapSort é um algoritmo de ordenação com complexidade O(n log n). Ele usa uma estrutura de dados chamada heap para realizar a ordenação, o que o torna eficiente para conjuntos de dados grandes.

Tempos de Execução Busca Sequencial

Formatação dos dados em forma de Tabela:

Algoritmos	Organização dos Dados	Quantidade de Dados	Tempo de Execução
Ordenado pelo HeapSort Busca Sequencial	Crescente	500.000	0.0007 seg.
		750.000	0.0011 seg.
		1.000.000	0.0016 seg.
	Decrescente	500.000	0.0007 seg.
		750.000	0.0011 seg.
		1.000.000	0.0001 seg.
	Aleatório	500.000	0.0016 seg.
		750.000	0.0011 seg.
		1.000.000	0.0016 seg.

Formatação dos dados em forma de Gráfico:



A busca sequencial é um algoritmo de busca simples que percorre todos os elementos de um vetor até encontrar o valor desejado. Sua complexidade de tempo é linear, ou seja, O(n), onde n é o número de elementos do vetor. Em média, a busca sequencial precisa verificar metade dos elementos do vetor antes de encontrar o valor desejado.

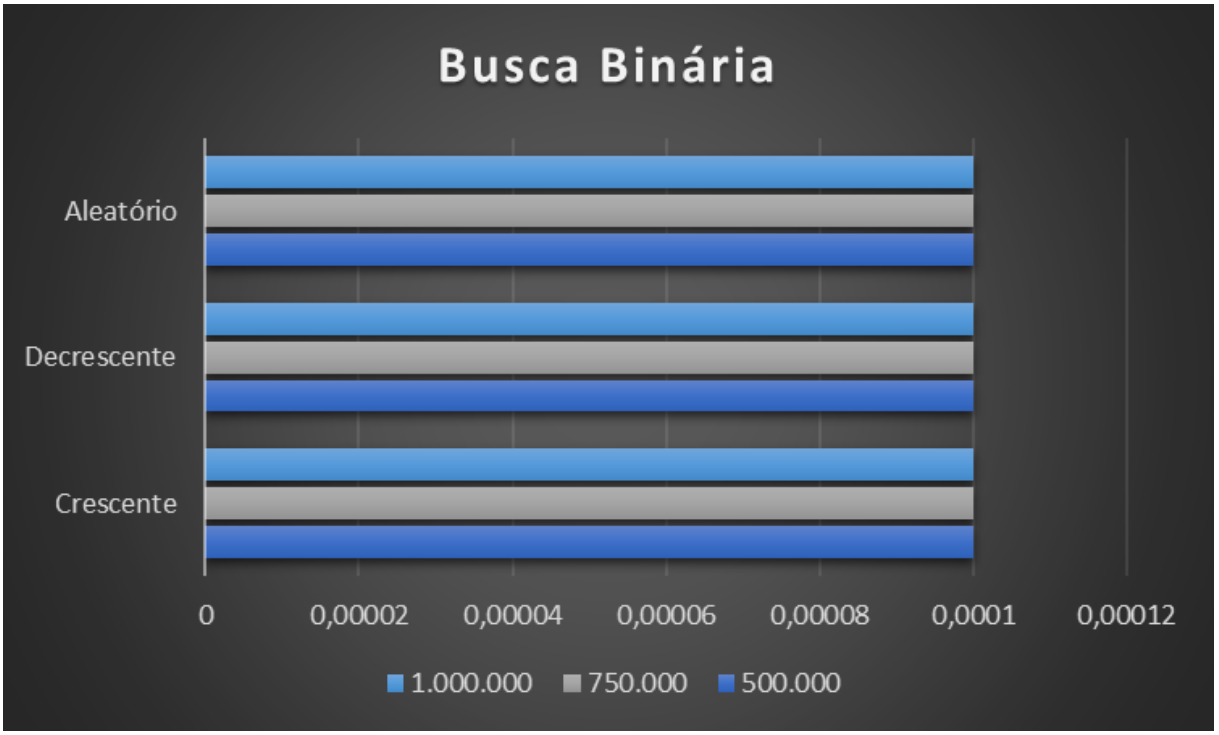
Neste caso eu utilizei como busca o último elemento, 500.000, 750.000 e 1.000.000.

Tempos de Execução Busca Binária

Formatação dos dados em forma de Tabela:

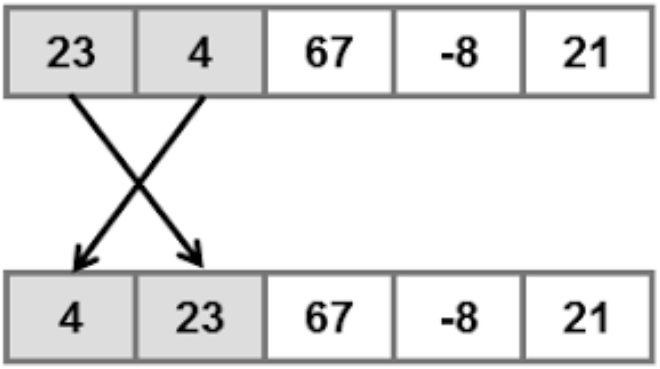
Algoritmos	Organização dos Dados	Quantidade de Dados	Tempo de Execução
Ordenado pelo HeapSort Busca Binária	Crescente	500.000	0.0001 seg.
		750.000	0.0001 seg.
		1.000.000	0.0001 seg.
	Decrescente	500.000	0.0001 seg.
		750.000	0.0001 seg.
		1.000.000	0.0001 seg.
	Aleatório	500.000	0.0001 seg.
		750.000	0.0001 seg.
		1.000.000	0.0001 seg.

Formatação dos dados em forma de Gráfico:



A busca binária é um algoritmo de busca mais eficiente para vetores ordenados, que consiste em dividir repetidamente o vetor pela metade até encontrar o valor desejado. Sua complexidade de tempo é logarítmica, ou seja, O(log n), onde n é o número de elementos do vetor. Isso faz com que a busca binária seja muito mais rápida do que a busca sequencial em vetores grandes.

No entanto, a busca binária exige que o vetor esteja ordenado.



Resumindo, o ShellSort, o QuickSort e o HeapSort são os algoritmos de ordenação mais eficientes para conjuntos de dados maiores, enquanto o BubbleSort, o InsertSort e o SelectionSort são mais adequados para conjuntos de dados menores ou parcialmente ordenados. O MergeSort e o RadixSort são úteis em situações específicas em que a ordenação precisa ser feita de uma determinada maneira.

Além das diferenças nos seus tempos de execução, esses algoritmos de ordenação também têm outras características que podem influenciar na escolha do algoritmo a ser utilizado. Por exemplo, o QuickSort é conhecido por ser um algoritmo eficiente em termos de tempo de execução, mas pode ter problemas de desempenho em casos extremos de distribuição de elementos (pior caso). Já o MergeSort é mais estável em relação ao desempenho, mas pode ter um alto consumo de memória em comparação com outros algoritmos.

O HeapSort é um algoritmo de ordenação por seleção que é baseado em uma estrutura de dados conhecida como Heap, que permite que o algoritmo seja implementado com uma complexidade de tempo O(n log n) no pior caso. Já o RadixSort é um algoritmo de ordenação que trabalha com os dígitos individuais de um número e é particularmente útil quando se tem uma grande quantidade de números a serem ordenados com tamanhos iguais ou similares.

Em geral, a escolha do algoritmo de ordenação dependerá das características do conjunto de dados que se deseja ordenar e das necessidades específicas do contexto em que ele será utilizado. É importante entender as propriedades e desempenhos de cada algoritmo para fazer a escolha mais adequada para cada caso.

E sobre os algoritmos de busca, a escolha entre a busca sequencial e a busca binária depende do tamanho do vetor, se ele está ordenado ou não e do número de vezes que a busca será realizada. Para vetores pequenos ou não ordenados, a busca sequencial pode ser mais adequada, enquanto que para vetores grandes e ordenados, a busca binária é mais eficiente.

5. CONSIDERAÇÕES FINAIS

Ao analisar os algoritmos de ordenação apresentados, podemos notar que cada um possui vantagens e desvantagens em relação aos demais. Alguns se destacam por sua simplicidade e facilidade de implementação, enquanto outros são mais complexos e exigem mais recursos computacionais, mas são mais eficientes em termos de tempo de execução.

Em resumo, a escolha do algoritmo de ordenação a ser utilizado dependerá do tamanho do conjunto de dados a ser ordenado, da natureza dos dados (se são inteiros, floats, strings, etc.) e dos recursos computacionais disponíveis. É importante conhecer as vantagens e desvantagens de cada algoritmo para escolher o mais apropriado para cada situação.

Meus testes poderiam ser mais amplos e eficazes, se os algoritmos rodassem em máquinas diferentes e paralelas, com IDE 's diferentes para realizar vários cálculos de tempo de execução e ambientes melhores e piores para cada algoritmo específico.

6. REFERÊNCIAS BIBLIOGRÁFICAS

GitHub, 14 Maio 2023, https://github.com/.

Wikipédia, 14 Maio 2023, https://pt.wikipedia.org/wiki/Wiki.

Lamb, Juliano Rodrigo Lamb, Slides apresentados em sala de aula, Moodle.