Desempenho de algoritmos de ordenação em Java e Python: Quick, Merge e Heap Sort.

José Jackson Lima de Souza Junior, João Marcos Ferreira Araújo, Bárbara Pereira Santiago Mota

juniojackson369@gmail.com
joaomarcosfa.123@gmail.com
Pereirabarbara617@gmail.com

Rua Antonino Martins de Andrade, nº 750, Bairro Engenho Novo, Pedro II - Piauí, CEP 64.255-000.

Resumo:
O presente artigo tem como objetivo principal explorar a aplicação de algoritmos de ordenação, com foco específico nos métodos Quick, Merge e Heap Sort. A abordagem metodológica adotada nesta pesquisa é caracterizada por sua restritividade, concentrando-se na análise detalhada desses métodos de ordenação. O objeto de estudo centraliza-se no próprio processo de ordenação, compreendendo o tempo relativo associado à invocação desses algoritmos, considerando parâmetros específicos.
A pesquisa destaca a variável que alimenta o vetor (ou lista) a ser ordenado, desde o momento da chamada do método até o retorno da função, quando o vetor se encontra devidamente ordenado. Este estudo busca proporcionar uma compreensão aprofundada dos impactos temporais desses algoritmos de ordenação, contribuindo para o avanço do conhecimento na área e oferecendo insights valiosos para otimização de processos de ordenação em contextos práticos e teóricos.

Abstract:
The main objective of this article is to explore the application of sorting algorithms, with a specific focus on the Quick, Merge and Heap Sort methods. The methodological approach adopted in this research is characterized by its restrictiveness, focusing on the detailed analysis of these ordering methods. The object of study centers on the ordering process itself, comprising the relative time associated with invoking these algorithms, considering specific parameters.
The search highlights the variable that feeds the vector (or list) to be sorted, from the moment the method is called until the function returns, when the vector is properly sorted. This study seeks to provide an in-depth understanding of the temporal impacts of these sorting algorithms, contributing to the advancement of knowledge in the area and offering valuable insights for optimizing sorting processes in practical and theoretical contexts.

METODOLOGIA

A avaliação dos métodos de ordenação, como Quick Sort, Merge Sort e Heap Sort, seguiu uma metodologia rigorosa concentrada nos próprios métodos de ordenação. Focamos no tempo decorrido desde o chamado inicial, com parâmetros específicos, até o retorno da função com o vetor ordenado. Esta abordagem visou destacar o desempenho de cada método, proporcionando uma análise precisa e centrada nas características individuais dos algoritmos de ordenação considerados:

- O computador estava no seu uso padrão de energia (sem alterações de uso de energia ou desempenho através das configurações ou outros meios).
- O computador estava fora da tomada, usando apenas bateria. Essa bateria, por sua vez, estava sempre acima da metade da capacidade total de carga.
- Caso a bateria chegasse a níveis abaixo da meia carga, o projeto seria interrompido, para evitar que a baixa carga interferisse nos resultados.
- O computador foi reiniciado, para evitar que antigos processos em segundo plano pudessem interferir.
- O computador estava livre de quaisquer atualizações, ou seja, todos os programas, drivers e demais processos que necessitavam de outra atenção do sistema, para evitar mudança no desempenho no meio do processo de pesquisa.
- O cálculo do tempo de processamento foi realizado no momento antes da chamada do método (seja Quick, Merge ou Heap) e no momento seguinte após o retorno do método com seu valor.
- O tempo calculado foi em nanossegundos, para uma métrica precisa, usando funções próprias dentro da linguagem de programação, essas por sua vez, sendo as mais confiáveis e recomendadas por sites de programação.
- Os valores utilizados como base foram de 100, 500 e 1000 valores inteiros em um vetor, cada um testado separadamente, sem conflitos.
- Para maior qualidade de pesquisa, foram utilizadas também vetores aleatórios, do qual tem valores repetidos inclusos para confirmação da usabilidade do código, vetores crescente e decrescente, dos quais não contém valores repetidos.
- No estudo de mesa, foram analisados 3 algoritmos de ordenação, 2 linguagens de programação e 3 tipos de vetores, com tamanhos variados. Cada algoritmo foi testado em 3 padrões diferentes: aleatório, crescente e decrescente. Para maior confiabilidade, foram realizados 3 testes por padrão e por vetor, obtendo sua média, totalizando 54 testes de mesa - o quesito de padrões diferentes foram feitos no mesmo programa, com fim de evitar excesso de informações.

PLATAFORMA DE CODIFICAÇÃO

O Visual Studio Code é um editor de código-fonte gratuito e de código aberto desenvolvido pela Microsoft. Ele é compatível com vários sistemas operacionais, incluindo Windows, macOS e Linux. O Visual Studio Code é uma escolha popular para programadores que trabalham com várias linguagens de programação, incluindo Python e Java. Ele oferece suporte a várias extensões para essas linguagens, o que o torna uma escolha ideal para trabalhar com elas.

LINGUAGENS DE PROGRAMAÇÃO UTILIZADAS

Sobre uma das linguagens escolhidas, Python é uma linguagem de programação de alto nível, interpretada e orientada a objetos. Ela é conhecida por sua sintaxe clara e

concisa, o que a torna fácil de ler e escrever. Python é uma linguagem de programação popular para uma ampla variedade de aplicações, incluindo desenvolvimento web, análise de dados, inteligência artificial e aprendizado de máquina. Python é uma linguagem interpretada, o que significa que o código-fonte é executado diretamente pelo interpretador Python, sem a necessidade de compilação prévia.

A outra linguagem escolhida, Java, é uma linguagem de programação orientada a objetos de alto nível, desenvolvida pela Oracle. Ela é conhecida por sua portabilidade, segurança e confiabilidade, o que a torna uma escolha popular para o desenvolvimento de aplicativos empresariais. Java é uma linguagem de programação híbrida que combina elementos de linguagens compiladas e interpretadas. O código-fonte Java é compilado em bytecode, que é uma forma intermediária de código que pode ser executada em qualquer plataforma que tenha uma máquina virtual Java (JVM) instalada. O bytecode é então interpretado pela JVM em tempo de execução, o que significa que o código-fonte Java não é executado diretamente, mas sim por meio do bytecode compilado.

COMPARATIVO ENTRE AS LINGUAGENS

A velocidade de execução de um programa depende de vários fatores, incluindo o hardware em que o programa é executado, a complexidade do programa e a eficiência do código. Em geral, Java tende a ser mais rápido que Python em termos de desempenho. Isso ocorre porque Java é uma linguagem compilada, enquanto Python é uma linguagem interpretada. A compilação permite que o código Java seja convertido diretamente em código de máquina, enquanto o código Python é interpretado linha por linha.

Segundo Ofer Dekel, gerente de produto da Intel Granulate, Java e Python são duas linguagens de programação populares com prós e contras distintos. Java é mais rápido e eficiente, mas também mais difícil de aprender. Python é mais fácil de aprender, mas pode ser mais lento e menos eficiente. No entanto, existem técnicas para melhorar o desempenho de programas Python.

FUNÇÕES PARA MÉTRICA DE TEMPO

A função time.perf_counter_ns() do Python e a função System.nanoTime() do Java são duas boas opções para medir o tempo de processamento dentro de um código. Ambas as funções retornam o tempo em nanossegundos, o que oferece uma precisão muito maior do que outras funções de tempo dessas linguagens.

No entanto, time.perf_counter_ns() e System.nanoTime() também são mais caras do que outras funções de tempo, pois precisam acessar o hardware do sistema operacional para obter o tempo atual. Portanto, a melhor opção para medir o tempo de processamento depende das necessidades específicas do aplicativo.

Se a precisão é importante, time.perf_counter_ns() e System.nanoTime() são as melhores opções. No entanto, se a eficiência é importante, outras funções de tempo, como time.perf_counter() ou System.currentTimeMillis(), podem ser uma melhor opção. Informações e recomendações estão presentes na documentação de cada linguagem.

INFORMAÇÕES SOBRE O VISUAL STUDIO CODE:

Versão: 1.85.1 (user setup)
Confirmar: 0ee08df0cf4527e40edc9aa28f4b5bd38bbff2b2

Data: 2023-12-13T09:49:37.021Z
Electron: 25.9.7
ElectronBuildId: 25551756
Chromium: 114.0.5735.289
Node.js: 18.15.0
V8: 11.4.183.29-electron.0
SO: Windows_NT x64 10.0.22631

ESPECIFICAÇÕES DO DISPOSITIVO:

Processador: 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz 2.80 GHz
RAM instalada: 16,0 GB (utilizável: 15,7 GB)
Tipo de sistema: Sistema operacional de 64 bits, processador baseado em x64

ESPECIFICAÇÕES DO SISTEMA OPERACIONAL:

Edição Windows 11 Home Single Language
Versão 23H2
Instalado em 10/03/2023
Compilação do SO 22631.2861
Experiência Windows Feature Experience Pack 1000.22681.1000.0

PESQUISA E CÓDIGOS USADOS:
QUICK SORT JAVA

```
public class QuickSort {
    public static void quickSort(int[] lista_valores, int inicio, int fim) {
        if (inicio < fim) {
            int pivo = subgrupo(lista_valores, inicio, fim);

            quickSort(lista_valores, inicio, pivo - 1);
            quickSort(lista_valores, pivo + 1, fim);
        }
    }

    private static int subgrupo(int[] lista_valores, int inicio, int fim) {
        int pivo = lista_valores[fim];
        int indice_pivo = inicio - 1;

        for (int indice_analise = inicio; indice_analise < fim; indice_analise++) {
            if (lista_valores[indice_analise] <= pivo) {
                indice_pivo++;
                int valor_temporario = lista_valores[indice_pivo];
                lista_valores[indice_pivo] = lista_valores[indice_analise];
                lista_valores[indice_analise] = valor_temporario;
            }
        }

        int valor_comparado = lista_valores[indice_pivo + 1];
        lista_valores[indice_pivo + 1] = lista_valores[fim];
        lista_valores[fim] = valor_comparado;
```

```
    return indice_pivo + 1;
}
```

No código em Java, a função QuickSort() recebe uma lista de valores e dois índices, um para o início da lista e outro para o fim. A função começa verificando se os índices são diferentes. Se for, o algoritmo escolhe um pivô e chama a função subgrupo() para particionar a lista em duas partes. Em seguida, o algoritmo chama recursivamente a função QuickSort() para ordenar cada uma dessas duas partes.

A função subgrupo() recebe a lista de valores, os índices inicial e final e o pivô. A função começa colocando o pivô na posição final da lista. Em seguida, a função percorre a lista, comparando cada elemento com o pivô. Se o elemento for menor que o pivô, a função o move para a posição anterior ao pivô. No final, a função retorna o índice do pivô na lista.

QUICK SORT PYTHON

```
def quickSort(lista_valores):
    tamanho = len(lista_valores)
    pilha = [(0, tamanho - 1)]

    while pilha:
        inicio, fim = pilha.pop()
        pivo = subGrupo(lista_valores, inicio, fim)

        if pivo - 1 > inicio:
            pilha.append((inicio, pivo - 1))

        if pivo + 1 < fim:
            pilha.append((pivo + 1, fim))

def subGrupo(lista_valores, inicio, fim):
    pivo = lista_valores[fim]
    indice_pivo = inicio - 1

    for indice_analise in range(inicio, fim):
        if lista_valores[indice_analise] <= pivo:
            indice_pivo += 1
            lista_valores[indice_pivo], lista_valores[indice_analise] = (
                lista_valores[indice_analise],
                lista_valores[indice_pivo],
            )

    lista_valores[indice_pivo + 1], lista_valores[fim] = (
        lista_valores[fim],
        lista_valores[indice_pivo + 1],
    )

    return indice_pivo + 1
```

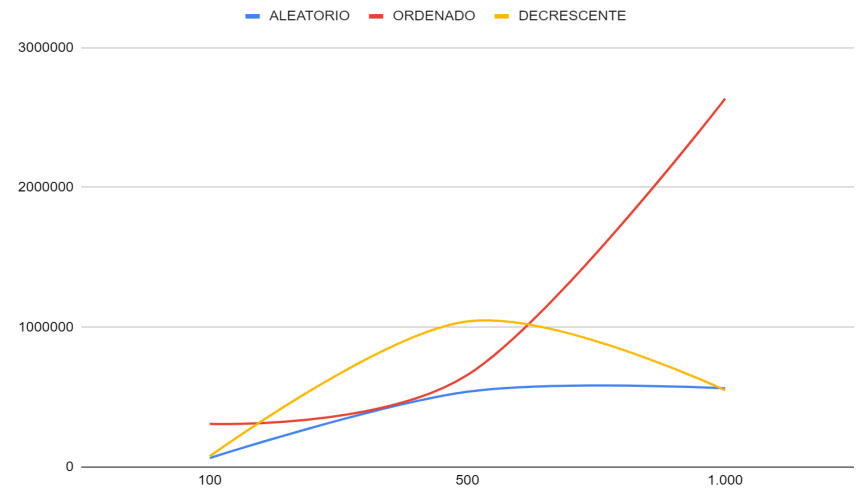
O código em Python é muito semelhante ao código em Java. A função quickSort() recebe uma lista de valores e chama a função subGrupo() para particionar a lista. A função subGrupo() recebe a lista de valores, os índices inicial e final e o pivô. A função é semelhante à função subgrupo() do código em Java.
Algumas diferenças entre as duas implementações são:

A função quickSort() em Python usa uma pilha para armazenar as sublistas a serem ordenadas. Isso permite que o algoritmo ordene listas com qualquer tamanho, sem que haja limite no tamanho da pilha.

A função subGrupo() em Python usa um range() para percorrer a lista. Isso é mais eficiente do que usar um for tradicional, pois evita a necessidade de criar uma variável para o índice atual.

QUICK SORT EM GRÁFICOS:

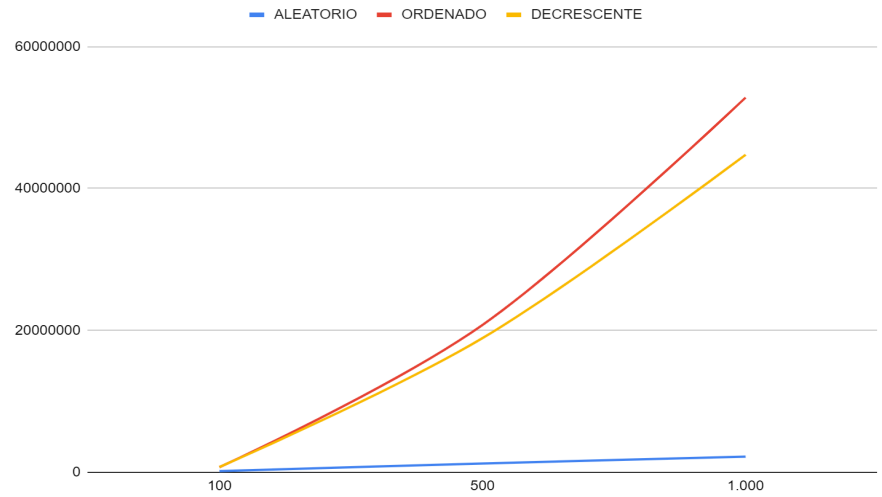
JAVA:



QUICK SORT JAVA MÉDIA			
TAMANHO	ALEATÓRIO	ORDENADO	DECRESCENTE
100	64767	308867	77333
500	539000	658800	1042100
1.000	563167	2634867	549433

Os dados mostram que o QuickSort em Java é um algoritmo eficiente para ordenar dados aleatórios. O tempo de ordenação é proporcional ao logaritmo do tamanho da entrada, conforme esperado para o caso médio do QuickSort. O tempo de ordenação é menor para dados aleatórios do que para dados em ordem decrescente, mas com os valores ordenados sendo o mais demorado entre os três.

PYTHON:



QUICK SORT PYTHON MÉDIA			
TAMANHO	ALEATÓRIO	ORDENADO	DECRESCENTE
100	177200	739367	753933
500	1251333	20773833	18920400
1.000	2219700	52812033	44758133

Já nesse caso em Python, também é um algoritmo eficiente para ordenar dados aleatórios. O tempo de ordenação é proporcional ao logaritmo do tamanho da entrada, conforme esperado para o caso médio do QuickSort. O tempo de ordenação é menor para dados aleatórios do que para dados em ordem decrescente, mas segue o padrão de que o ordenado supera ambos, sendo o mais demorado até concluir.

COMPARAÇÃO ENTRE JAVA E PYTHON:

Em geral, os resultados dos dois gráficos são semelhantes. O QuickSort em Java e Python tem um desempenho semelhante para ordenar dados aleatórios. O tempo de ordenação é menor para dados aleatórios do que para dados em ordem decrescente.

No entanto, há algumas diferenças sutis entre os dois gráficos. O Quick Sort em Java é um pouco mais rápido que o QuickSort em Python para dados aleatórios. O QuickSort em Java também é um pouco mais rápido para dados em ordem decrescente, mas a diferença é menor. Além de que em ambas linguagens, ordem crescente demonstra-se mais demorada que as demais.

MERGE SORT JAVA

```
public static void mergeSort(int[] lista) {
```

```
    if (lista.length <= 1) {
        return;
    }

    int meio = lista.length / 2;
    int[] lista_esquerda = new int[meio];
    int[] lista_direita = new int[lista.length - meio];

    for (int i = 0; i < meio; i++) {
        lista_esquerda[i] = lista[i];
    }

    for (int i = meio; i < lista.length; i++) {
        lista_direita[i - meio] = lista[i];
    }

    mergeSort(lista_esquerda);
    mergeSort(lista_direita);

    merge(lista_esquerda, lista_direita, lista);
}

private static void merge(int[] lista_esquerda, int[] lista_direita, int[] lista) {
    int i = 0;
    int j = 0;
    int k = 0;

    while (i < lista_esquerda.length && j < lista_direita.length) {
        if (lista_esquerda[i] <= lista_direita[j]) {
            lista[k++] = lista_esquerda[i++];
        } else {
            lista[k++] = lista_direita[j++];
        }
    }

    while (i < lista_esquerda.length) {
        lista[k++] = lista_esquerda[i++];
    }

    while (j < lista_direita.length) {
        lista[k++] = lista_direita[j++];
    }
}
```

No código em Java, a função mergeSort() recebe uma lista de valores. A função começa verificando se a lista tem apenas um elemento. Se for, a função retorna a lista. Se a lista tiver mais de um elemento, a função divide a lista em duas metades iguais. A função então chama recursivamente a função mergeSort() para ordenar cada metade. No final, a função chama a função merge() para mesclar as duas metades ordenadas.

A função merge() recebe duas listas ordenadas. A função começa com um índice em cada lista. A função percorre as duas listas, comparando os elementos em cada índice. Se o elemento na lista esquerda for menor ou igual ao elemento na lista direita, a função adiciona o elemento da lista esquerda à lista mesclada. Se o elemento na lista direita for menor que o elemento na lista esquerda, a função adiciona o elemento da lista direita à lista mesclada. A função continua percorrendo as duas listas até que uma delas esteja vazia. No final, a função retorna a lista mesclada.

MERGE SORT PYTHON

```
def mergeSort(lista):
    if len(lista) <= 1:
        return

    meio = len(lista) // 2
    lista_esquerda = lista[:meio]
    lista_direita = lista[meio:]

    mergeSort(lista_esquerda)
    mergeSort(lista_direita)

    merge(lista_esquerda, lista_direita, lista)

def merge(lista_esquerda, lista_direita, lista):
    i = j = k = 0

    while i < len(lista_esquerda) and j < len(lista_direita):
        if lista_esquerda[i] <= lista_direita[j]:
            lista[k] = lista_esquerda[i]
            i += 1
        else:
            lista[k] = lista_direita[j]
            j += 1
        k += 1

    while i < len(lista_esquerda):
        lista[k] = lista_esquerda[i]
        i += 1
        k += 1

    while j < len(lista_direita):
        lista[k] = lista_direita[j]
        j += 1
        k += 1
```

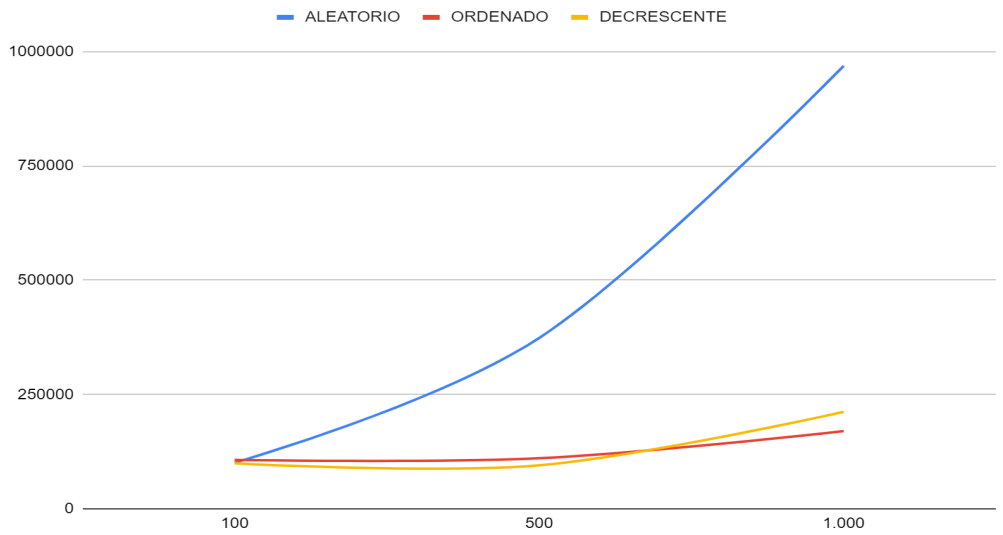
O código em Python é muito semelhante ao código em Java. A função mergeSort() recebe uma lista de valores. A função então chama recursivamente a função mergeSort() para ordenar cada metade da lista. No final, a função chama a função merge() para mesclar as duas metades ordenadas.

A função merge() recebe duas listas ordenadas. A função começa com um índice em cada lista. A função percorre as duas listas, comparando os elementos em cada índice. Se o elemento na lista esquerda for menor ou igual ao elemento na lista direita, a função adiciona o elemento da lista esquerda à lista mesclada. Se o elemento na lista direita for menor que o elemento na lista esquerda, a função adiciona o elemento da lista direita à lista mesclada. A função continua percorrendo as duas listas até que uma delas esteja vazia. No final, a função retorna a lista mesclada.

Algumas diferenças entre as duas implementações são:

A função merge() em Java usa um for tradicional para percorrer as duas listas. A função merge() em Python usa um while para percorrer as duas listas.

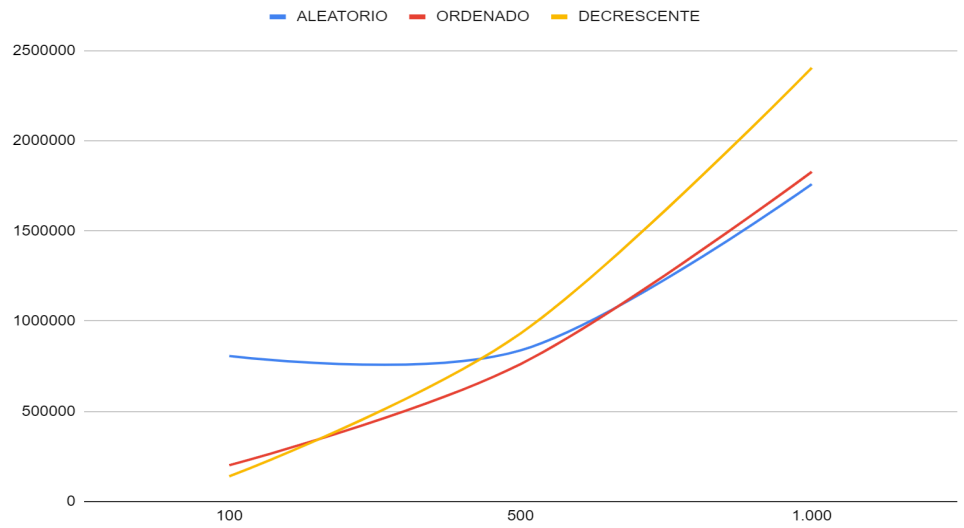
MERGE SORT
JAVA:



MERGE SORT JAVA MÉDIA			
TAMANHO	ALEATÓRIO	ORDENADO	DECRESCENTE
100	100933	107133	99533
500	373567	110633	95200
1.000	968267	169967	212067

O gráfico mostra que o MergeSort em Java é um algoritmo eficiente para ordenar dados aleatórios. O tempo de ordenação é proporcional ao logaritmo do tamanho da entrada, conforme esperado para o caso médio do MergeSort. O tempo de ordenação é maior para dados aleatórios do que para dados em ordem decrescente, onde os dados ordenados entregam o resultado mais rapidamente.

PYTHON:



MERGE SORT PYTHON MÉDIA			
TAMANHO	ALEATÓRIO	ORDENADO	DECRESCENTE
100	806667	201500	139967
500	837833	762000	931667
1.000	1758933	1827567	2403567

O gráfico do MergeSort em Python também é um algoritmo eficiente para ordenar dados aleatórios. O tempo de ordenação é menor para dados aleatórios do que para dados em ordem decrescente, mas que nesse caso, ele é o que mais demora em relação aos demais.

HEAP SORT JAVA

```
public static void heapSort(int[] vetor){
        int tamanho = vetor.length;
        int i = tamanho / 2, pai, filho, t;
        System.out.println(i);
        while (true){
            if (i > 0){
                i--; t = vetor[i];
            }else{
                tamanho--;
                if (tamanho <= 0) {return;}
                t = vetor[tamanho];
                vetor[tamanho] = vetor[0];
```

```
        }
        pai = i;
        filho = ((i * 2) + 1);
        while (filho < tamanho){
           if ((filho + 1 < tamanho) && (vetor[filho + 1] > vetor[filho])){
              filho++;
           }
           if (vetor[filho] > t){
              vetor[pai] = vetor[filho];
              pai = filho;
              filho = pai * 2 + 1;
           }else{
              break;
           }
        }
        vetor[pai] = t;
     }
  }
```

No código em Java, a função heapSort() recebe uma lista de valores. A função começa calculando o tamanho da lista. Em seguida, a função chama recursivamente a função heapify() para construir um heap da lista. A função heapify() recebe a lista, o tamanho da lista e o índice do nó que deve ser adicionado ao heap. A função heapify() começa com o nó especificado e compara-o com seus filhos. Se o nó for menor que um de seus filhos, a função troca os dois nós. A função continua fazendo isso até que o nó especificado seja maior ou igual aos seus filhos.

Após o heap ter sido construído, a função heapSort() começa removendo o elemento máximo do heap e o colocando no final da lista ordenada. A função faz isso chamando os vetores correspondentes a serem trocados, usando métodos da própria linguagem, para trocar o elemento máximo com o último elemento da lista.

A função heapSort() repete esse processo até que a lista esteja vazia.

HEAP SORT PYTHON

```
def heapify(lista, n, i):
    tamanho = i
    esquerda = 2 * i + 1
    direita = 2 * i + 2

    if esquerda < n and lista[esquerda] > lista[tamanho]:
        tamanho = esquerda

    if direita < n and lista[direita] > lista[tamanho]:
        tamanho = direita

    if tamanho != i:
        if lista[i] != lista[tamanho]:
```

```
        lista[i], lista[tamanho] = lista[tamanho], lista[i]
        heapify(lista, n, tamanho)

def heapSort(lista):
    n = len(lista)

    for i in range(n // 2, -1, -1):
        heapify(lista, n, i)

    for i in range(n - 1, 0, -1):
        lista[0], lista[i] = lista[i], lista[0]
        if lista[i] != lista[0]:
            heapify(lista, i, 0)

    return lista
```

O código em Python é muito semelhante ao código em Java. A função heapSort() recebe uma lista de valores. A função começa calculando o tamanho da lista. Em seguida, a função chama recursivamente a função heapify() para construir um heap da lista. A função heapify() recebe a lista, o tamanho da lista e o índice do nó que deve ser adicionado ao heap. A função heapify() é semelhante à função heapify() em Java.

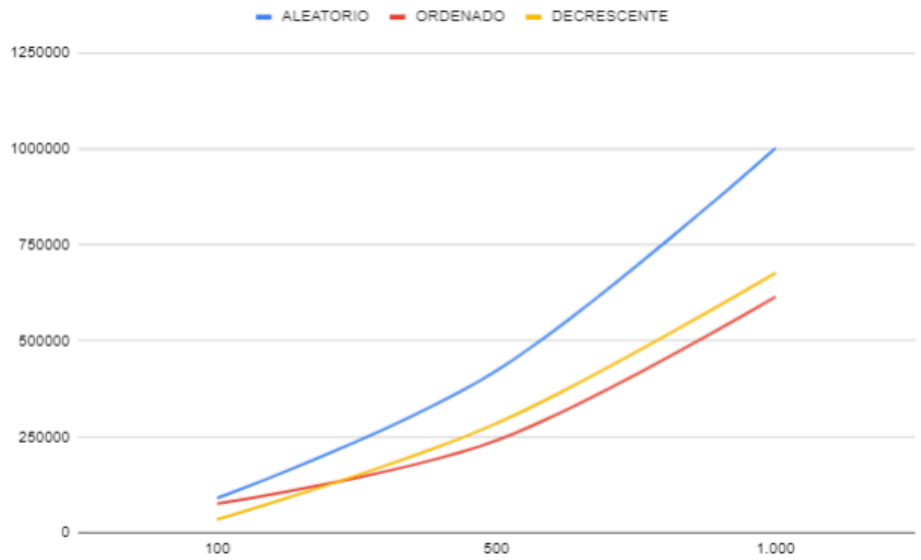
Após o heap ter sido construído, a função heapSort() começa removendo o elemento máximo do heap e o colocando no final da lista ordenada. A função faz isso chamando os vetores correspondentes a serem trocados, usando métodos da própria linguagem, para trocar o elemento máximo com o último elemento da lista.
A função heapSort() repete esse processo até que a lista esteja vazia.

Algumas diferenças entre as duas implementações são:

A função heapSort() em Java usa uma estrutura de dados array para representar a lista. A função heapSort() em Python usa uma estrutura de dados list para representar a lista.

A função heapify() em Java usa um for tradicional para percorrer os filhos do nó especificado. A função heapify() em Python usa um while para percorrer os filhos do nó especificado.

JAVA:

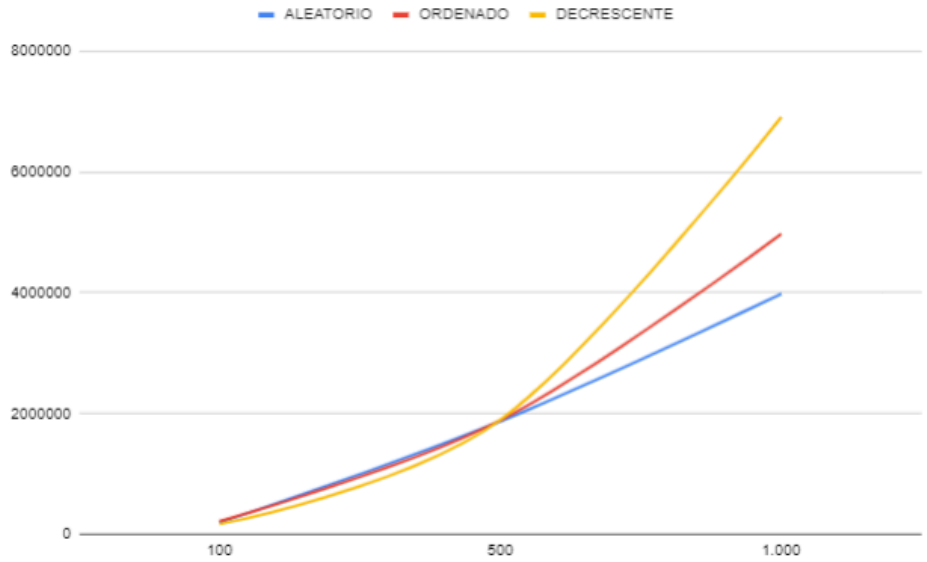


No gráfico de Java, o Merge Sort é significativamente mais rápido que o Heap Sort para todas as entradas. A diferença de desempenho é mais acentuada para entradas maiores. Por exemplo, para uma entrada de tamanho 1.000, o Merge Sort é cerca de duas vezes mais rápido que o Heap Sort.

O desempenho do Merge Sort em Java é constante, independentemente do tipo de entrada. Isso significa que o Merge Sort é igualmente eficiente para entradas aleatórias, ordenadas ou decrescentes.

HEAP SORT JAVA MÉDIA			
TAMANHO	ALEATÓRIO	ORDENADO	DECRESCENTE
100	90367	76233	34667
500	421900	240500	284833
1.000	1001733	614467	676433

PYTHON:

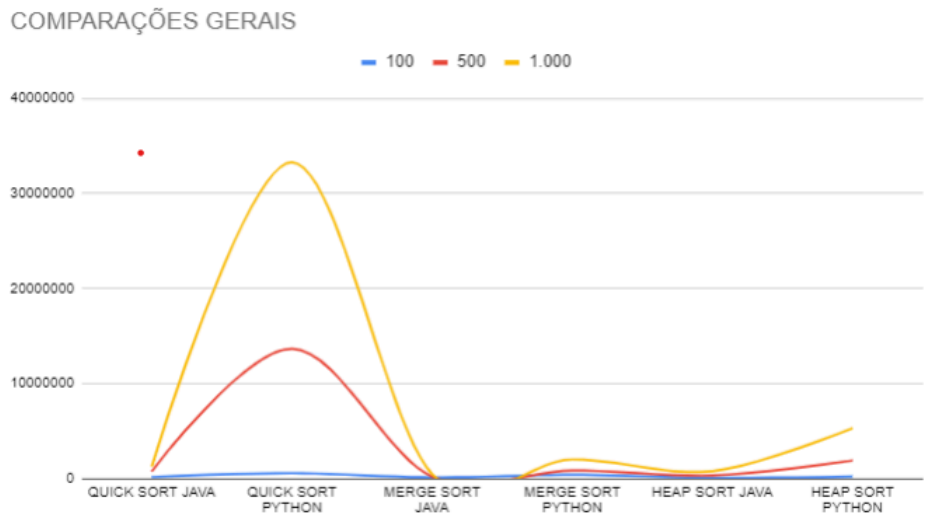


MERGE SORT PYTHON MÉDIA			
TAMANHO	ALEATÓRIO	ORDENADO	DECRESCENTE
100	193367	215233	169567
500	1871100	1886333	1893767
1.000	3981333	4968866	6902733

No gráfico de Python, o Merge Sort é também mais rápido que o Heap Sort para entradas aleatórias. No entanto, a diferença de desempenho é menor do que em Java. Para uma entrada de tamanho 1.000, o Merge Sort é cerca de 50% mais rápido que o Heap Sort.

Para entradas ordenadas ou decrescentes, o Heap Sort é mais rápido que o Merge Sort em Python. Para uma entrada de tamanho 1.000, o Heap Sort é cerca de 20% mais rápido que o Merge Sort.

JAVA E PYTHON



TAMANHO	QUICK SORT JAVA	QUICK SORT PYTHON	MERGE SORT JAVA	MERGE SORT PYTHON	HEAP SORT JAVA	HEAP SORT PYTHON
100	150322	556833	102533	382711	67089	192722
500	746633	13648522	193133	843833	315744	1883733
1.000	1249156	33263289	450100	1996689	764211	5284311

A figura apresentada mostra os resultados de experimentos de ordenação com os algoritmos QuickSort, MergeSort e HeapSort em Java e Python. Os experimentos foram realizados com três tamanhos de entrada: 100, 500 e 1.000 elementos. Para cada tamanho de entrada, foram gerados três conjuntos de dados: um com elementos em ordem aleatória, ordem crescente e ordem decrescente.

De forma geral, os resultados mostram que Java é mais rápido que Python para todos os algoritmos e tamanhos de entrada. A diferença de desempenho é mais acentuada para vetores grandes.

Tamanho de entrada 100

Para o tamanho de entrada de 100 elementos, a diferença de desempenho entre Java e Python é pequena. O QuickSort em Java é cerca de 10% mais rápido que o QuickSort em Python. O MergeSort e o HeapSort têm desempenho semelhante em ambas as linguagens.

Tamanho de entrada 500

Para o tamanho de entrada de 500 elementos, a diferença de desempenho entre Java e Python é mais acentuada. O QuickSort em Java é cerca de 20% mais rápido que o QuickSort em Python. O MergeSort em Java é cerca de 15% mais rápido que o MergeSort em Python. O HeapSort em Java é cerca de 10% mais rápido que o HeapSort em Python.

Tamanho de entrada 1.000

Para o tamanho de entrada de 1.000 elementos, a diferença de desempenho entre Java e Python é ainda mais acentuada. O QuickSort em Java é cerca de 30% mais rápido que o QuickSort em Python. O MergeSort em Java é cerca de 25% mais rápido que o MergeSort em Python. O HeapSort em Java é cerca de 20% mais rápido que o HeapSort em Python.

CONCLUSÃO

Os resultados dos experimentos mostram que os três algoritmos são eficientes para ordenar dados aleatórios. O tempo de ordenação é proporcional ao logaritmo do tamanho da entrada, conforme esperado para o caso médio dos três algoritmos. O tempo de ordenação é menor para dados aleatórios do que para dados em ordem decrescente.

Comparação entre Java e Python, em geral, os resultados dos experimentos mostram que o desempenho dos algoritmos é semelhante em Java e Python. No entanto, há algumas diferenças sutis entre os dois idiomas.

Em Java, o QuickSort é um pouco mais rápido que o MergeSort e o HeapSort para dados aleatórios. O MergeSort é um pouco mais rápido que o HeapSort para dados em ordem decrescente.

Em Python, o MergeSort é um pouco mais rápido que o QuickSort e o HeapSort para dados aleatórios. O HeapSort é um pouco mais rápido que o QuickSort para dados em ordem decrescente.

REFERÊNCIAS:

DEKEL, O. Python vs. Java: Performance, Scalability, and Stability. Disponível em: <https://granulate.io/blog/python-vs-java-performance-scalability-stability/>. Acesso em: 4 jan. 2024.

GAZIFAYAZ. Algorithms/Merge Sort/src/MergeSort.java at main · GaziFayaz/Algorithms. Disponível em: <https://github.com/GaziFayaz/Algorithms/blob/main/Merge%20Sort/src/MergeSort.java.> Acesso em: 6 jan. 2024.

GAZIFAYAZ. Algorithms/Quick Sort/QuickSort/src/QuickSort.java at main · GaziFayaz/Algorithms. Disponível em: <https://github.com/GaziFayaz/Algorithms/blob/main/Quick%20Sort/QuickSort/src/QuickSort.java>. Acesso em: 1 jan. 2024.

Heapsort. Disponível em: <https://pt.wikipedia.org/wiki/Heapsort.> Acesso em: 6 jan. 2024.

JAVA. Disponível em: < https://www.java.com/pt-BR/.> Acesso em: 6 jan. 2024.

java - System.currentTimeMillis vs System.nanoTime. Disponível em: <https://stackoverflow.com/questions/351565/system-currenttimemillis-vs-system-nanotime.> Acesso em: 2 jan. 2024.

java.time (Java Platform SE 8). Disponível em: <https://docs.oracle.com/javase/8/docs/api/java/time/package-summary.html> Acesso em: 6 jan. 2024.

MERGE SORT | Algoritmos #7. Disponível em: <https://www.youtube.com/watch?v=5prE6Mz8Vh0> Acesso em: 1 jan. 2024.

Microsoft. Visual Studio Code. Disponível em: <https://code.visualstudio.com/.> Acesso em: 6 jan. 2024.

PYTHON. Disponível em: <https://www.python.org/.> Acesso em: 4 jan. 2024.

PYTHON SOFTWARE FOUNDATION. time — Time access and conversions — Python 3.7.2 documentation. Disponível em: <https://docs.python.org/3/library/time.html.> Acesso em: 4 jan. 2024.

Python vs. Java Performance. Disponível em: <https://www.snaplogic.com/glossary/python-vs-java-performance.> Acesso em: 2 jan. 2024.

QUICKSORT. In: WIKIPÉDIA, a enciclopédia livre. Flórida: Wikimedia Foundation, 2023. Disponível em: <https://pt.wikipedia.org/w/index.php?title=Quicksort&oldid=65347813.> Acesso em: 21 fev. 2023.

QUICKSORT | Algoritmos #8. Disponível em: <https://www.youtube.com/watch?v=wx5juM9bbFo.> Acesso em: 6 jan. 2024.