Universidade Federal de São Paulo – UNIFESP

Campus: Parque Tecnológico – São José dos Campos



Instituto de Ciência e Tecnologia – ICT

Bacharelado em Ciência e Tecnologia – BCT

**Algoritmos e Estruturas de Dados II**

**Algoritmos de Ordenação**

Prof. Reginaldo Massanobu Kuroshu

**João Pedro da Silva Zampoli** **RA: 168880**

**Luiza de Souza Ferreira** **RA: 170453**

**Viviane Flor Park** **RA: 169259**

São José dos Campos, 2024

[**1 INTRODUÇÃO 4**](#_gjdgxs)

[1.1 Algoritmos de Ordenação 4](#_jwx88eqmn7xk)

[1.1.1 Insertion Sort 5](#_jibfdo7h1aya)

[1.1.2 Heapsort 5](#_iemsr6ohz8um)

[1.1.3 Counting Sort 6](#_fn84fni2qemc)

[**2 OBJETIVOS 7**](#_30j0zll)

[2.1 Objetivo geral 7](#_1wd24ok1cvk3)

[2.2 Objetivos específicos 7](#_74zyc21p3tqj)

[**3 MÉTODOS 8**](#_1fob9te)

[3.1 Criação das tabelas utilizadas 9](#_lwbk742juj9f)

[3.2 Códigos implementados 12](#_hz9wprmgyuyl)

[3.2.1 Implementações do Insertion Sort 12](#_qa18tp76znmp)

[3.2.2 Implementações do Heapsort 16](#_iumxjb45kmsi)

[3.2.3 Implementações do Counting Sort 21](#_5h5q2favhxd6)

[**4 RESULTADOS E DISCUSSÕES 27**](#_3znysh7)

[4.1 Insertion Sort 27](#_14teze4ggubz)

[4.2 Heapsort 32](#_pv21dpavub5)

[4.3 Counting Sort 37](#_pvpmph5o09l3)

[4.4 Discussão 41](#_fxxplo2ym9ym)

[**5 CONCLUSÃO 44**](#_2et92p0)

**RESUMO**

Neste relatório, foram observados 3 algoritmos de ordenação apresentados na UC “Algoritmos e Estruturas de Dados II”, sendo eles: Insertion Sort, Heapsort e Counting Sort, tendo como objetivo compará-los em relação aos seus tempos de execução para três diferentes cenários em três linguagens de programação. Após a obtenção e comparação dos resultados, foram apresentados os códigos e linguagens mais eficientes.

# 1 INTRODUÇÃO

Um algoritmo é uma sequência finita de ações executáveis que visam obter uma solução para um determinado tipo de problema. Algoritmos podem ser, menos intuitivamente, o processo de lavar roupa, já que há uma sequência de passos que têm de ser seguidos a fim da roupa ficar finalmente limpa, ou, mais intuitivamente, o processo de ordenar um conjunto de números em ordem crescente.

Existem problemas para os quais existem diversos algoritmos, sendo uns mais eficientes que outros. Por exemplo, encher uma piscina com água: é possível 1) usar um copo de 200mL para enchê-la, realizando diversas viagens da torneira mais próxima até a piscina; ou 2) usar um balde de 10L em vez do copo; ou 3) usar uma mangueira de vazão suficiente para transportar a água da torneira até a piscina. Intuitivamente, é possível dizer que o terceiro método é o mais eficiente em questão de esforço físico.

Revisitando a ideia de ordenação de números, existem diversos algoritmos já criados para a resolução desse problema. Na área da computação tais algoritmos são denominados de algoritmos de ordenação.

## 1.1 Algoritmos de Ordenação

Os algoritmos de ordenação não são exclusivos para a ordenação de números, mas a ordenação desse tipo de dado é muitas vezes a causa de seu estudo.

Quando aplicados à computação, os algoritmos de ordenação podem ser analisados de algumas perspectivas diferentes, tais como complexidade computacional, uso de memória, estabilidade, trocas, comparações, etc. Em relação principalmente à complexidade, existem diferentes classes de algoritmos, como , , , entre outras, que impactam diretamente no tempo de execução de um dado conjunto de dados.

Os algoritmos de ordenação Insertion Sort, Heapsort e Counting Sort são exemplos de algoritmos das classes , e , respectivamente.

### 1.1.1 Insertion Sort

O Insertion Sort é um método bastante intuitivo de ordenação. A ideia é criar uma lista ordenada de dados pela comparação individual de cada elemento da lista com aqueles ao seu lado, fazendo uma inserção por vez. Ele pertence à classe de algoritmos de ordenação quadráticos, ou seja, , e é considerado estável.

### 1.1.2 Heapsort

O Heapsort foi desenvolvido em 1964 por Robert W. Floyd e J.W.J. Williams, e é considerado um algoritmo de ordenação por seleção. Sua implementação requer a interpretação dos dados em forma de um *heap*, estrutura que pode ser visualizada como uma árvore binária.

A partir de um conjunto de dados, é construído um *max heap*, no caso de ordenação em ordem crescente, onde o valor contido no nó pai é sempre maior que o contido no nó filho. Desse ponto em diante, o nó raiz é removido e inserido numa partição ordenada do conjunto, e a árvore restante é reordenada em forma de *max heap*.

Esse algoritmo de ordenação pertence à classe de complexidade , e não é primariamente reconhecido como estável.

### 1.1.3 Counting Sort

O Counting Sort é um algoritmo de ordenação criado em 1954 por Harold H. Seward. Ele funciona para números inteiros positivos, idealmente pequenos, e pertence à classe de algoritmos (ou ).

Quando ordenando um dado vetor, sua ideia é criar um vetor auxiliar de tamanho , sendo o maior número pertencente ao vetor original. Nesse vetor auxiliar, na posição (sendo que pode assumir valores inteiros de 0 até ), é guardado o índice onde o número deve estar no vetor ordenado, o que é obtido a partir de uma pequena sequência de cálculos e operações. Então, a partir da análise do vetor original e do vetor auxiliar, é possível criar um terceiro vetor, dessa vez ordenado.

Trata-se de um algoritmo eficiente e estável, porém de uso muito específico e que faz uso de memória auxiliar.

# 2 OBJETIVOS

## 2.1 Objetivo geral

Comparar os tempos de execução de três algoritmos de ordenação em diferentes cenários e linguagens de programação.

## 2.2 Objetivos específicos

* Implementar códigos em três linguagens de programação.
* Elaborar tabelas comparando tempos de execução.
* Observar pontos positivos e negativos de cada algoritmo, linguagem e cenário.

# 3 MÉTODOS

Em primeiro lugar, foi feita a escolha de três algoritmos de ordenação diferentes, cada um pertencente a um grupo de complexidade diferente. Foram escolhidos os algoritmos: Insertion Sort, de complexidade , o Heapsort, de complexidade , e o Counting Sort, de complexidade .

Em seguida, foram decididos os três cenários a serem analisados: números inteiros em ordem aleatória, números inteiros em ordem decrescente, e números inteiros quase completamente ordenados em ordem crescente. Os algoritmos foram implementados para organizar a situação dos dados em cada cenário em ordem crescente, portanto o primeiro caso representa o “caso médio”, o segundo o “pior caso” e o terceiro o “melhor caso”.

Para cada cenário foi criado um mínimo de 6 tabelas, armazenadas em um arquivo csv, contendo os valores a serem ordenados. As tabelas criadas também diferem no número de valores, podendo variar de a dados. As tabelas foram geradas usando três scripts diferentes em Python, cada um cobrindo um dos casos a serem analisados. A seção 3.1 mostra com mais detalhes o processo de criação das tabelas.

Os algoritmos escolhidos foram aplicados em forma de código em três linguagens de programação diferentes: C, Java e Python. Os códigos implementados de cada algoritmo em cada uma das linguagens de programação escolhidas podem ser encontrados na seção 3.2. Cada tabela foi testada três vezes para cada versão do código, com exceção da versão do código do Insertion Sort em Python que, devido a problemas de tempo, foi testado somente para três tabelas de cada categoria. Foram essas as tabelas A6, A7 e A8; B2, B5 e B6; e C1, C3 e C6.

Os testes com os códigos em C e Java foram realizados por meio do terminal da plataforma Replit, enquanto os testes com os códigos em Python foram realizados num terminal de macOS. Para a obtenção dos tempos de execução, foi utilizado o comando .

Os tempos de execução (real, de usuário e de sistema) foram anotados em uma planilha e, posteriormente, serviram como base para a criação de gráficos na plataforma RStudio, para que fosse possível o seu estudo.

## 3.1 Criação das tabelas utilizadas

Três códigos foram implementados em Python com o objetivo de gerar tabelas que atendessem a 3 cenários diferentes, sendo eles: números inteiros aleatórios, números inteiros em ordem decrescente e números inteiros quase ordenados em ordem crescente. Cada tabela gerada tem entre e linhas e apenas uma coluna, sendo que cada linha contém um número inteiro aleatório entre e .

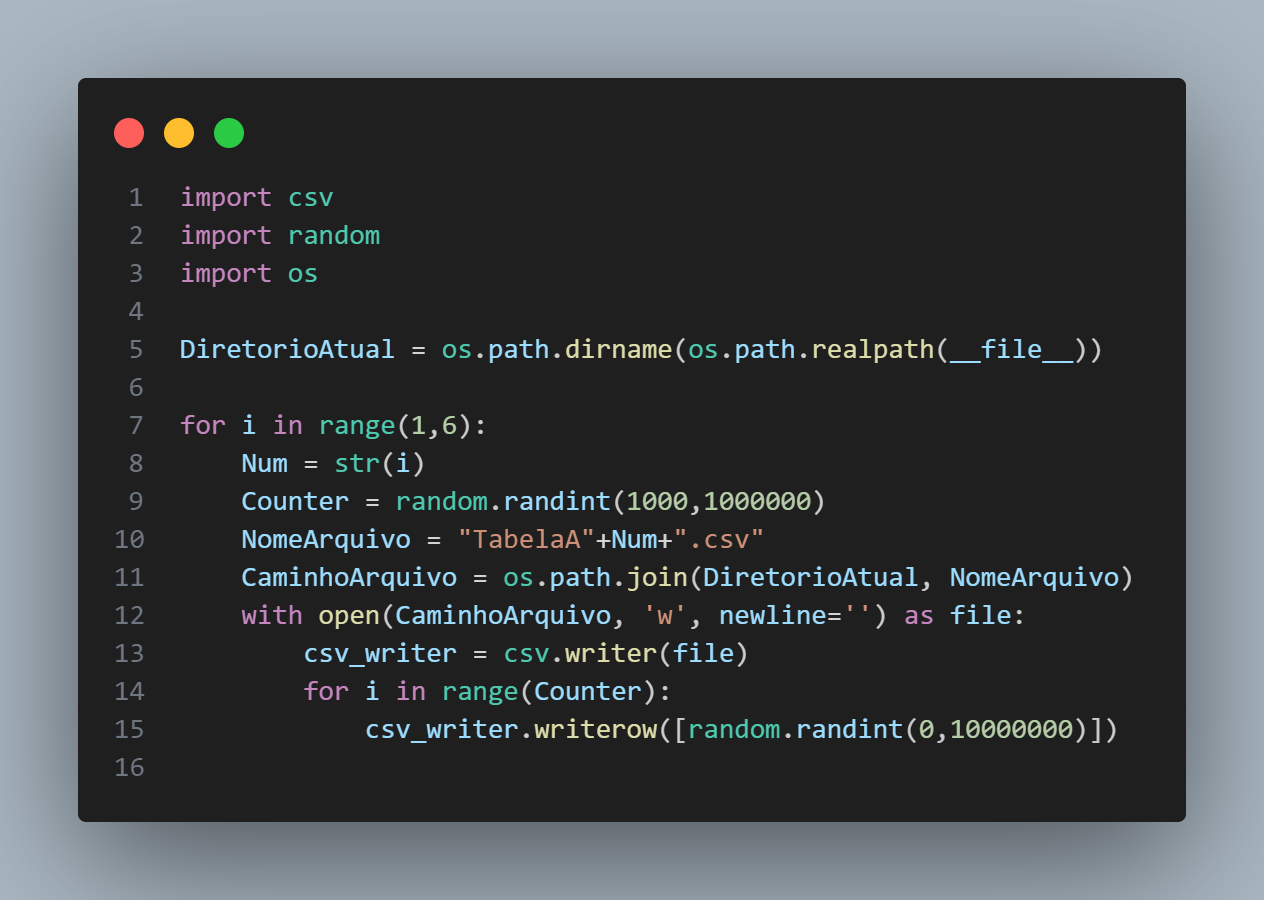
****

Figura 3.1.1. Código usado para gerar tabelas com números inteiros aleatórios em ordem aleatória (A).

****

Figura 3.1.2. Código usado para gerar tabelas com números inteiros aleatórios em ordem decrescente (B).

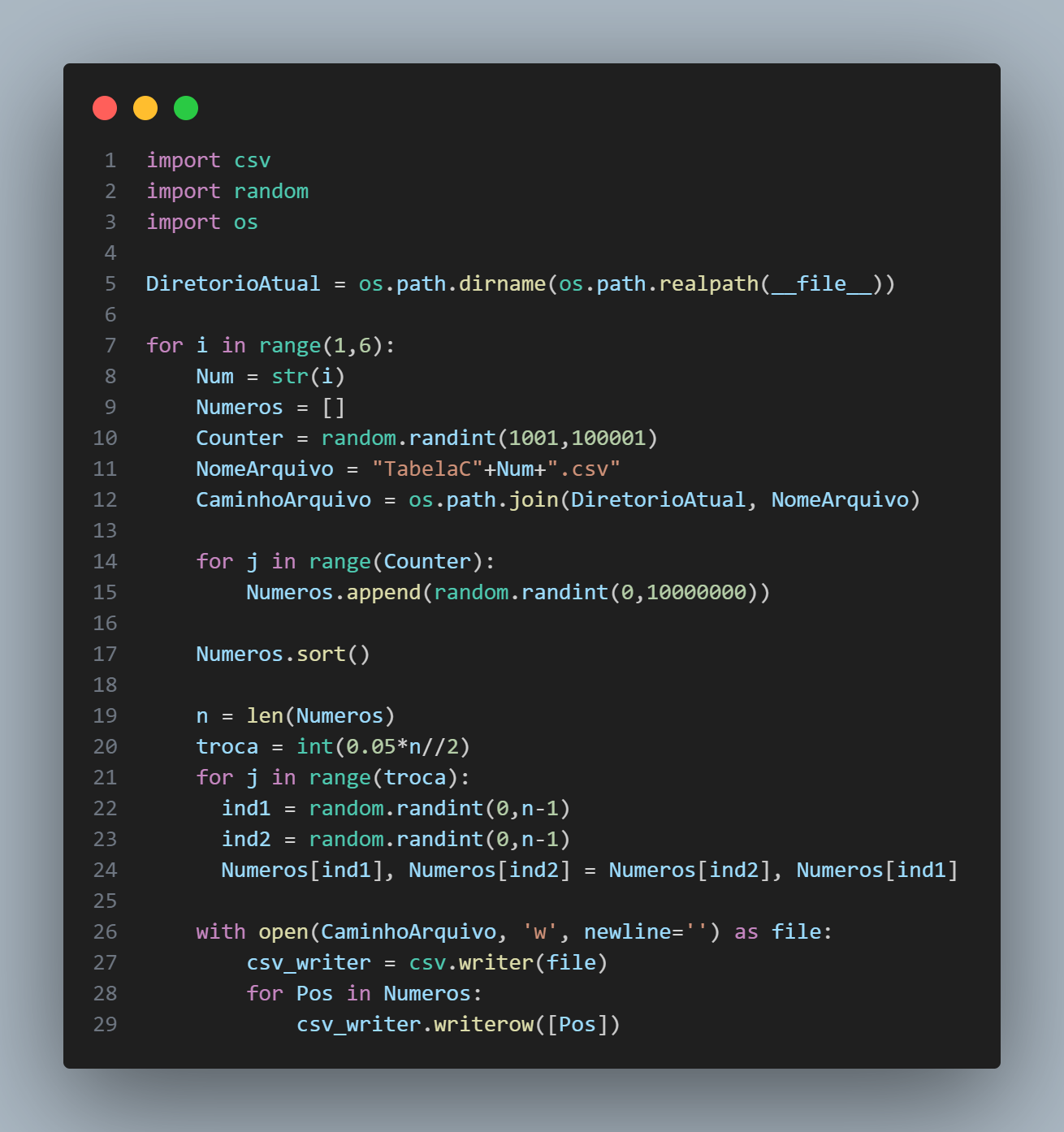
****

Figura 3.1.3. Código usado para gerar tabelas com números inteiros aleatórios quase completamente ordenados em ordem crescente (C).

As tabelas criadas para o primeiro cenário foram identificadas com a letra “A”, assim como as criadas para o segundo cenário foram identificadas com a letra “B”, e para o terceiro com a letra “C”. A Tabela 3.1.1 mostra a quantidade de dados contida em cada tabela.

Tabela 3.1.1. Quantidade de dados contidos em cada tabela gerada.

| **Primeiro Cenário** | | **Segundo Cenário** | | **Terceiro Cenário** | |
| --- | --- | --- | --- | --- | --- |
| A1 | 295950 | B1 | 931534 | C1 | 4325 |
| A2 | 505889 | B2 | 312724 | C2 | 59254 |
| A3 | 821442 | B3 | 555726 | C3 | 101114 |
| A4 | 837738 | B4 | 563162 | C4 | 385729 |
| A5 | 360333 | B5 | 6346 | C5 | 917140 |
| A6 | 7800 | B6 | 102784 | C6 | 298144 |
| A7 | 110388 |  |  |  |  |
| A8 | 302543 |  |  |  |  |

Note que as tabelas de cada cenário escolhidas para testar o código do Insertion Sort em Python se encaixam em três faixas de quantidade de dados: 1) menos de (A6, B5 e C1), 2) aproximadamente (A7, B6 e C3); e 3) aproximadamente (A8, B2 e C6).

## **3.2 Códigos implementados**

Foram implementados versões dos algoritmos de ordenação do Insertion Sort, Heapsort e Counting Sort nas linguagens de programação C, Java (sendo duas linguagens compiladas) e Python (uma linguagem interpretada).

### 3.2.1 Implementações do Insertion Sort

As Figuras 3.2.1.1, 3.2.1.2 e 3.2.1.3 mostram os códigos implementados do algoritmo Insertion Sort em C, Java e Python, respectivamente.



Figura 3.2.1.1. Código em C que realiza ordenação de um vetor por Insertion Sort.



Figura 3.2.1.2. Código em Java que realiza ordenação de um vetor por Insertion Sort.

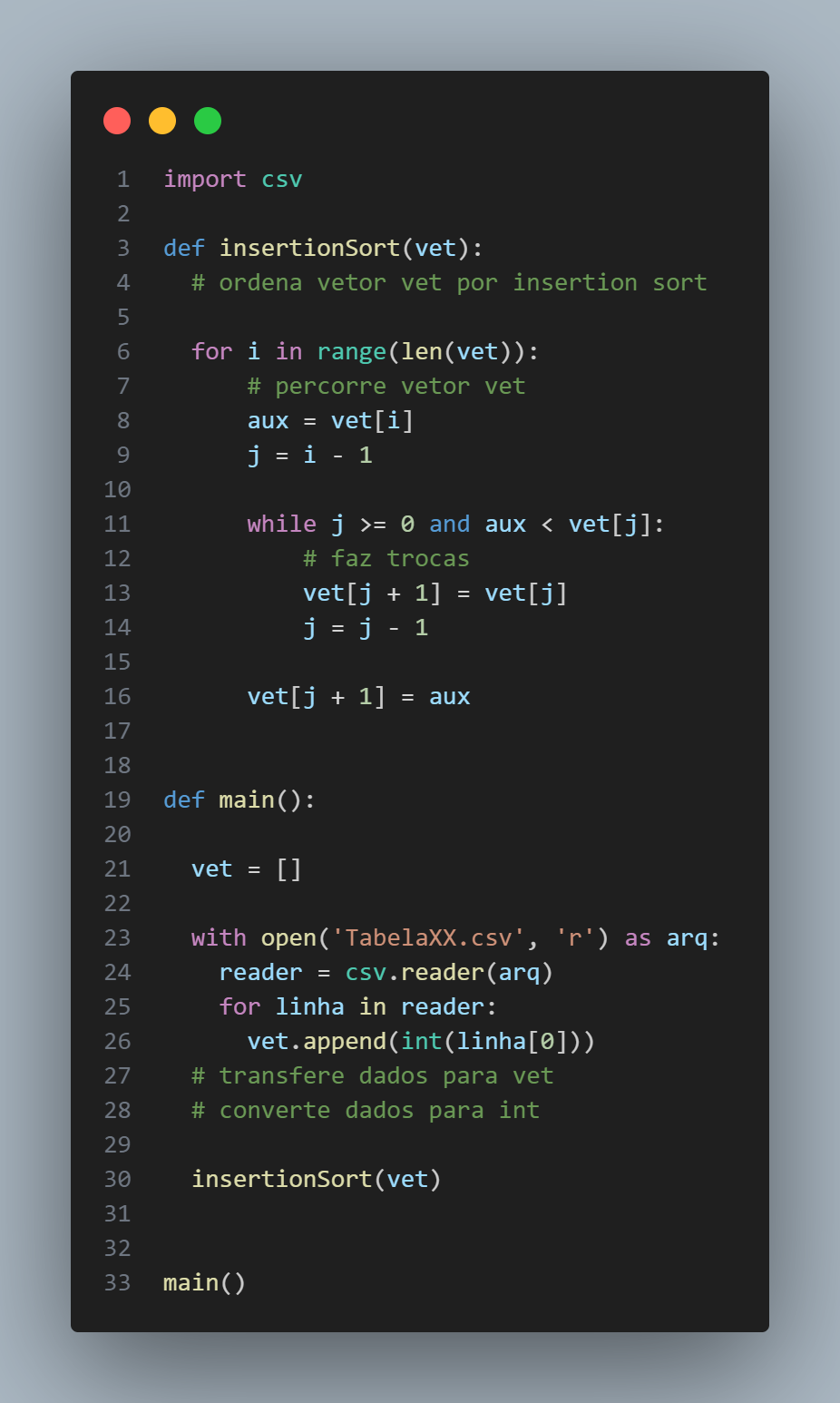
****

Figura 3.2.1.3. Código em Python que realiza ordenação de um vetor por Insertion Sort.

### 3.2.2 Implementações do Heapsort

As Figuras 3.2.2.1 e 3.2.2.2 mostram os códigos implementados em C, enquanto que as figuras 3.2.2.3 e 3.2.2.4 mostram a implementação em Java, já as figuras 3.2.2.5 e 3.2.2.6 representam a implementação do Heapsort em Python.



Figura 3.2.2.1. Trecho 01 do código em C que realiza ordenação de um vetor por Heapsort.



Figura 3.2.2.2. Trecho 02 do código em C que realiza ordenação de um vetor por Heapsort.



Figura 3.2.2.3. Trecho 01 do código em Java que realiza ordenação de um vetor por Heapsort.



Figura 3.2.2.4. Trecho 02 do código em Java que realiza ordenação de um vetor por Heapsort.



Figura 3.2.2.5. Trecho 01 do código em Python que realiza ordenação de um vetor por Heapsort.



Figura 3.2.2.6. Trecho 02 do código em Python que realiza ordenação de um vetor por Heapsort.

### 3.2.3 Implementações do Counting Sort

As Figuras 3.2.3.1, 3.2.3.2 mostram os códigos implementados do algoritmo Counting Sort em C, a figura 3.2.3.3 em Java enquanto que as figuras 3.2.3.4 e 3.2.3.5 representam a implementação do algoritmo em Python.

****

Figura 3.2.3.1. Trecho 01 do código em C que realiza ordenação de um vetor por Counting Sort.

****

Figura 3.2.3.2. Trecho 02 do código em C que realiza ordenação de um vetor por Counting Sort.

****

Figura 3.2.3.3. Código em Java que realiza ordenação de um vetor por Counting Sort.

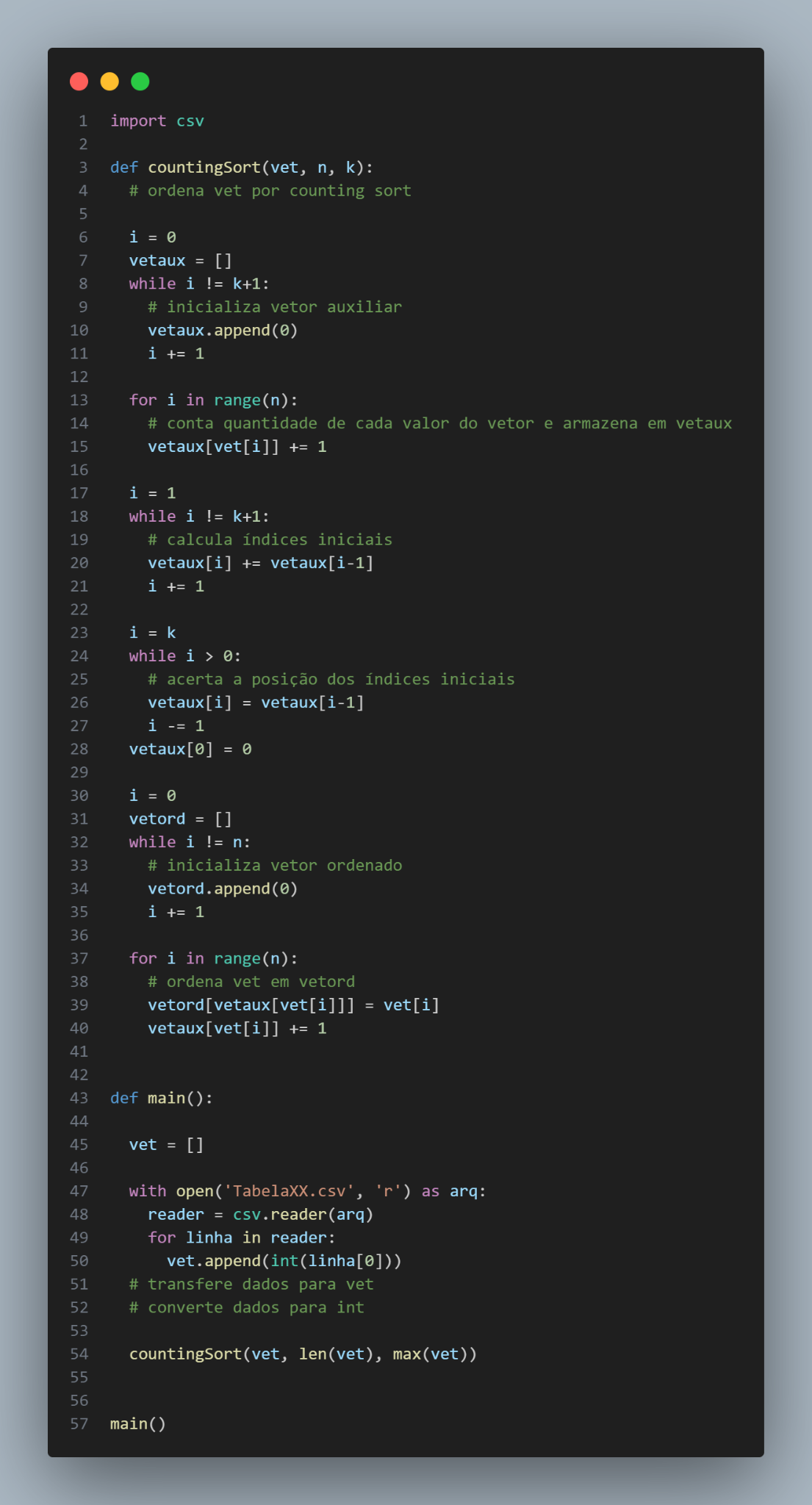


Figura 3.2.3.4. Trecho 01 do código em Python que realiza ordenação de um vetor por Counting Sort.

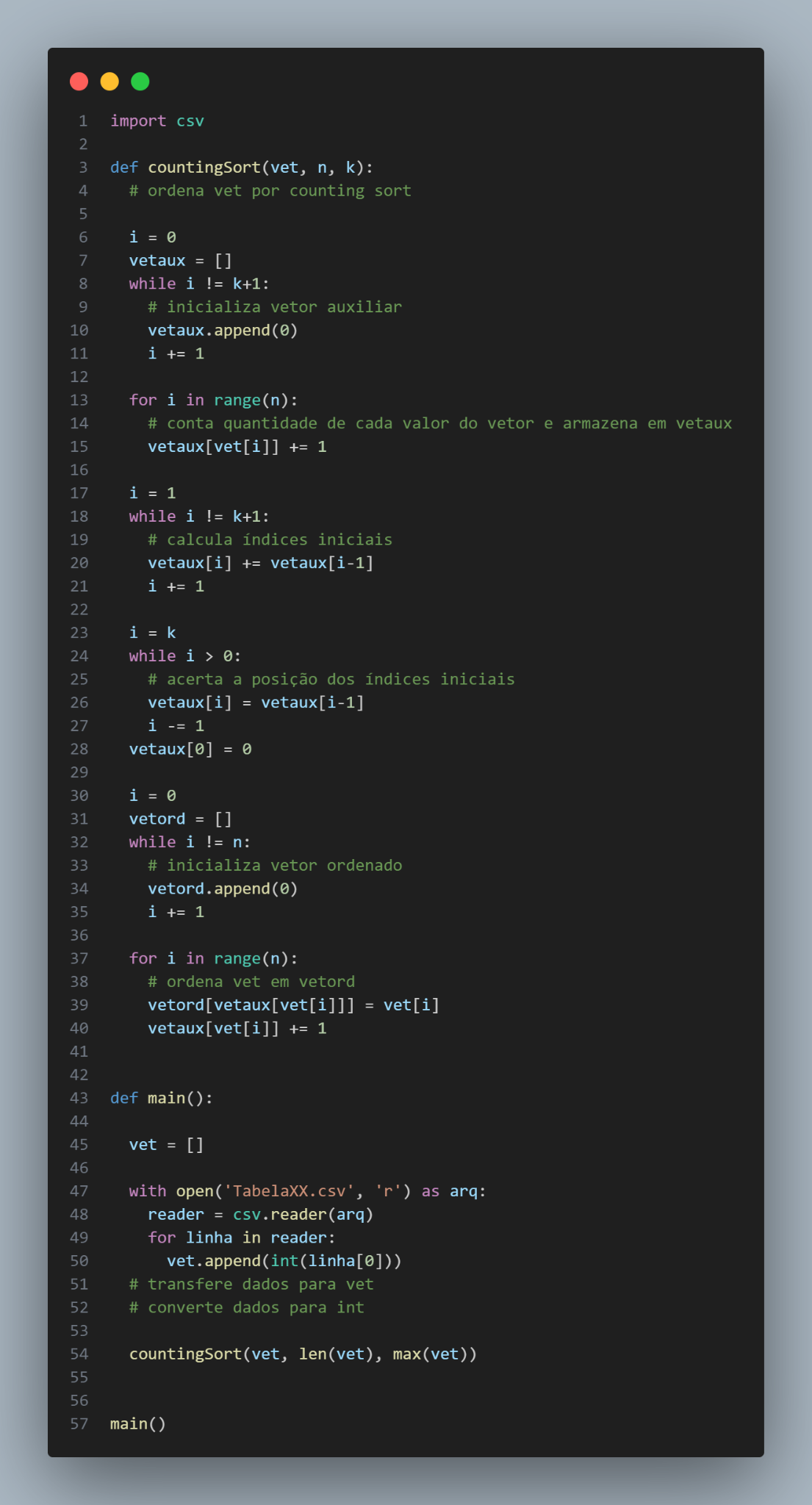


Figura 3.2.3.5. Trecho 02 do código em Python que realiza ordenação de um vetor por Counting Sort.

# 4 RESULTADOS E DISCUSSÕES

## 4.1 Insertion Sort

As Tabelas 4.1.1, 4.1.2 e 4.1.3 mostram os dados de tempo coletados referentes à execução do Insertion Sort em C, Java e Python, respectivamente.

Tabela 4.1.1. Tempo real, de usuário e de sistema para três execuções do Insertion Sort em C, para as tabelas A, B e C.

| C | | A1 | A2 | A3 | A4 | A5 | A6 | A7 | A8 | B1 | B2 | B3 | B4 | B5 | B6 | C1 | C2 | C3 | C4 | C5 | C6 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | real | 38,696 | 110,057 | 284,024 | 311,515 | 55,071 | 0,041 | 5,170 | 41,255 | 826,517 | 82,051 | 262,739 | 235,793 | 0,065 | 9,330 | 0,035 | 0,078 | 0,269 | 3,893 | 28,073 | 2,637 |
| user | 18,293 | 51,962 | 134,911 | 145,473 | 26,246 | 0,027 | 2,501 | 19,353 | 373,932 | 39,470 | 126,223 | 113,725 | 0,051 | 4,456 | 0,012 | 0,054 | 0,148 | 1,909 | 13,007 | 1,258 |
| sys | 0,240 | 0,565 | 1,239 | 1,743 | 0,256 | 0,013 | 0,034 | 0,255 | 4,925 | 0,388 | 1,050 | 0,438 | 0,016 | 0,055 | 0,013 | 0,017 | 0,014 | 0,014 | 0,163 | 0,027 |
| 2 | real | 36,346 | 109,832 | 297,868 | 360,860 | 74,314 | 0,040 | 5,577 | 37,282 | 708,108 | 81,686 | 234,939 | 244,379 | 0,114 | 9,891 | 0,027 | 0,130 | 0,276 | 3,733 | 22,718 | 2,757 |
| user | 17,326 | 51,924 | 140,530 | 161,533 | 26,684 | 0,020 | 2,638 | 17,927 | 341,906 | 39,352 | 113,878 | 117,854 | 0,052 | 4,710 | 0,011 | 0,060 | 0,141 | 1,836 | 10,966 | 1,340 |
| sys | 0,196 | 0,594 | 1,798 | 2,400 | 0,324 | 0,019 | 0,038 | 0,124 | 2,619 | 0,331 | 0,519 | 0,643 | 0,012 | 0,074 | 0,013 | 0,014 | 0,024 | 0,015 | 0,121 | 0,022 |
| 3 | real | 35,835 | 104,336 | 326,673 | 294,965 | 53,274 | 0,082 | 4,909 | 38,256 | 751,766 | 82,658 | 236,024 | 264,303 | 0,093 | 9,083 | 0,022 | 0,135 | 0,352 | 3,903 | 22,498 | 2,534 |
| user | 17,135 | 49,636 | 153,493 | 140,123 | 25,433 | 0,036 | 2,392 | 18,411 | 361,170 | 39,672 | 113,582 | 127,970 | 0,045 | 4,322 | 0,014 | 0,067 | 0,169 | 1,907 | 10,863 | 1,225 |
| sys | 0,119 | 0,542 | 1,878 | 1,374 | 0,263 | 0,032 | 0,036 | 0,148 | 2,899 | 0,451 | 0,540 | 1,017 | 0,013 | 0,070 | 0,010 | 0,013 | 0,017 | 0,032 | 0,135 | 0,020 |

Tabela 4.1.2. Tempo real, de usuário e de sistema para três execuções do Insertion Sort em Java, para as tabelas A, B e C.

| Java | | A1 | A2 | A3 | A4 | A5 | A6 | A7 | A8 | B1 | B2 | B3 | B4 | B5 | B6 | C1 | C2 | C3 | C4 | C5 | C6 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | real | 40,858 | 117,408 | 309,218 | 317,687 | 60,506 | 1,275 | 7,071 | 41,138 | 830,692 | 84,725 | 284,278 | 304,887 | 0,756 | 12,573 | 0,729 | 1,069 | 1,569 | 5,791 | 31,524 | 4,830 |
| user | 19,671 | 56,290 | 147,972 | 152,098 | 28,781 | 0,436 | 3,192 | 19,379 | 392,449 | 40,206 | 134,799 | 143,865 | 0,252 | 5,761 | 0,253 | 0,424 | 0,683 | 2,684 | 14,691 | 2,013 |
| sys | 0,226 | 0,609 | 1,387 | 1,143 | 0,373 | 0,213 | 0,223 | 0,329 | 4,358 | 0,468 | 1,640 | 1,970 | 0,130 | 0,183 | 0,101 | 0,115 | 0,101 | 0,139 | 0,386 | 0,304 |
| 2 | real | 41,529 | 114,896 | 312,765 | 367,227 | 57,621 | 1,399 | 8,243 | 39,843 | 798,978 | 92,133 | 271,328 | 290,018 | 0,851 | 10,894 | 0,766 | 1,050 | 1,354 | 6,017 | 32,213 | 4,699 |
| user | 19,914 | 55,260 | 150,044 | 172,173 | 27,440 | 0,427 | 3,582 | 18,863 | 380,390 | 42,742 | 128,722 | 143,993 | 0,303 | 5,075 | 0,254 | 0,422 | 0,552 | 2,713 | 15,126 | 2,100 |
| sys | 0,300 | 0,638 | 1,386 | 1,924 | 0,326 | 0,225 | 0,229 | 0,265 | 3,941 | 0,468 | 1,497 | 1,408 | 0,117 | 0,174 | 0,113 | 0,112 | 0,133 | 0,176 | 0,296 | 0,159 |
| 3 | real | 40,937 | 112,997 | 313,613 | 325,412 | 57,575 | 1,320 | 11,771 | 40,507 | 773,650 | 84,782 | 276,381 | 292,808 | 0,819 | 9,856 | 0,542 | 1,058 | 1,227 | 6,972 | 36,093 | 4,925 |
| user | 19,555 | 54,314 | 150,525 | 154,556 | 27,356 | 0,400 | 3,603 | 19,115 | 368,533 | 40,214 | 130,976 | 144,945 | 0,309 | 4,589 | 0,181 | 0,393 | 0,407 | 2,913 | 16,969 | 2,192 |
| sys | 0,278 | 0,150 | 1,339 | 1,666 | 0,367 | 0,240 | 0,308 | 0,382 | 3,563 | 0,518 | 1,206 | 1,563 | 0,103 | 0,157 | 0,090 | 0,125 | 0,157 | 0,390 | 0,302 | 0,177 |

Tabela 4.1.3. Tempo real, de usuário e de sistema para três execuções do Insertion Sort em Python, para as tabelas A, B e C.

| Py | | A1 | A2 | A3 | A4 | A5 | A6 | A7 | A8 | B1 | B2 | B3 | B4 | B5 | B6 | C1 | C2 | C3 | C4 | C5 | C6 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | real |  |  |  |  |  | 3,152 | 602,935 | 5.370,962 |  | 8.631,352 |  |  | 3,854 | 961,196 | 0,189 |  | 35,891 |  |  | 283,504 |
| user |  |  |  |  |  | 2,998 | 594,323 | 5.297,609 |  | 8.522,116 |  |  | 3,772 | 932,485 | 0,154 |  | 35,795 |  |  | 275,508 |
| sys |  |  |  |  |  | 0,058 | 2,790 | 19,650 |  | 31,438 |  |  | 0,047 | 4,962 | 0,021 |  | 0,040 |  |  | 2,217 |
| 2 | real |  |  |  |  |  | 3,069 | 599,231 | 5.355,602 |  | 10.123,665 |  |  | 3,850 | 935,269 | 0,203 |  | 35,607 |  |  | 340,584 |
| user |  |  |  |  |  | 2,992 | 592,214 | 5.273,866 |  | 9.880,089 |  |  | 3,772 | 922,857 | 0,119 |  | 35,499 |  |  | 333,943 |
| sys |  |  |  |  |  | 0,044 | 2,699 | 20,289 |  | 48,522 |  |  | 0,044 | 3,890 | 0,029 |  | 0,041 |  |  | 2,228 |
| 3 | real |  |  |  |  |  | 3,386 | 596,918 | 5.727,752 |  | 8.664,327 |  |  | 3,855 | 933,824 | 0,139 |  | 35,648 |  |  | 280,445 |
| user |  |  |  |  |  | 3,153 | 589,336 | 5.417,857 |  | 8.578,331 |  |  | 3,773 | 923,240 | 0,116 |  | 35,535 |  |  | 273,896 |
| sys |  |  |  |  |  | 0,081 | 2,713 | 36,166 |  | 30,776 |  |  | 0,046 | 3,552 | 0,015 |  | 0,038 |  |  | 2,108 |

Os dados das tabelas acima foram utilizados para criar os gráficos das Figuras 4.1.1, 4.1.2 e 4.1.3.

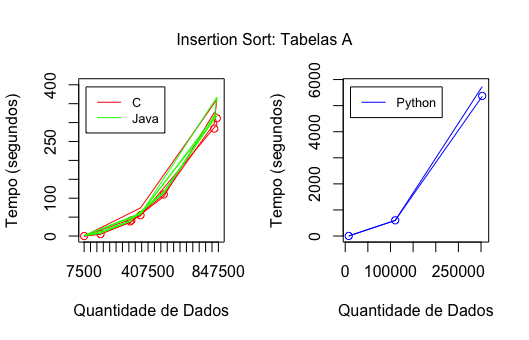


Figura 4.1.1. Gráficos de tempo (em segundos) pela quantidade de dados das tabelas A nas linguagens C e Java (à esquerda) e em Python (à direita).

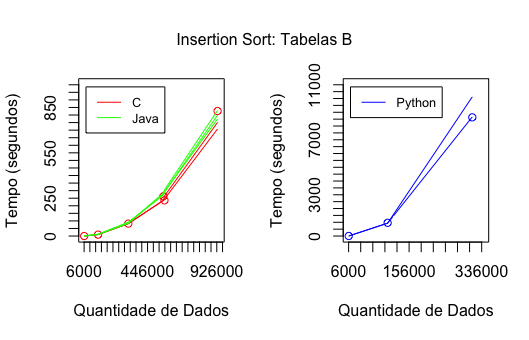


Figura 4.1.2. Gráficos de tempo (em segundos) pela quantidade de dados das tabelas B nas linguagens C e Java (à esquerda) e em Python (à direita), referentes ao Insertion Sort.

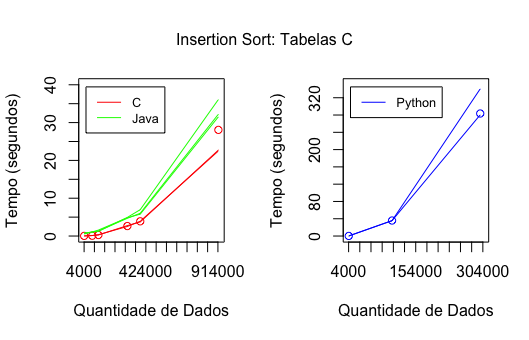


Figura 4.1.3. Gráficos de tempo (em segundos) pela quantidade de dados das tabelas C nas linguagens C e Java (à esquerda) e em Python (à direita), referentes ao Insertion Sort.

Os gráficos das figuras acima mostram, de fato, o que se espera do Insertion Sort. Por se tratar de um algoritmo de complexidade , espera-se que conforme o número de dados cresça, o tempo de execução cresça também de maneira acelerada. Para quantidades de dados primeiramente relativamente pequenas, o Insertion Sort se prova um algoritmo com tempo de execução relativamente eficiente, o que não se mantém verdade conforme essa quantidade aumenta, já que o número de comparações e inserções também aumenta proporcionalmente.

Também percebe-se que no seu melhor caso, no caso do gráfico da Figura 4.1.3, o tempo geral de execução foi significativamente melhor que aquele do pior caso, ou seja, do cenário apresentado na Figura 4.1.2. Já no caso médio, o tempo tende a ser significativamente maior que em seu melhor caso, porém ainda menor do que o tempo no seu pior caso.

Percebe-se, também, que o tempo de execução em Python foi significativamente maior que em Java ou C. Para a tabela B2, ou seja, no pior caso em que foi testado o Insertion Sort em Python, podemos a partir dos dados de tempo real da Tabela 4.1.3 calcular o tempo médio de execução . Uma possível explicação para a discrepância no tempo médio de execução pode ser o fato de Python ser uma linguagem interpretada, enquanto C e Java são linguagens compiladas.

## 4.2 Heapsort

As Tabelas 4.2.1, 4.2.2 e 4.2.3 mostram os dados de tempo coletados referentes à execução do Heapsort em C, Java e Python, respectivamente.

Tabela 4.2.1. Tempo real, de usuário e de sistema para três execuções do Heapsort em C, para as tabelas A, B e C.

| C | | A1 | A2 | A3 | A4 | A5 | A6 | A7 | A8 | B1 | B2 | B3 | B4 | B5 | B6 | C1 | C2 | C3 | C4 | C5 | C6 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | real | 1,141 | 0,452 | 0,656 | 0,896 | 0,354 | 0,031 | 0,087 | 0,390 | 0,479 | 0,156 | 0,302 | 0,671 | 0,157 | 0,097 | 0,027 | 0,045 | 0,098 | 0,255 | 0,613 | 0,107 |
| user | 0,203 | 0,170 | 0,319 | 0,389 | 0,175 | 0,022 | 0,040 | 0,136 | 0,236 | 0,079 | 0,184 | 0,249 | 0,025 | 0,045 | 0,019 | 0,020 | 0,038 | 0,116 | 0,327 | 0,073 |
| sys | 0,059 | 0,053 | 0,024 | 0,051 | 0,017 | 0,007 | 0,025 | 0,030 | 0,035 | 0,024 | 0,011 | 0,031 | 0,016 | 0,013 | 0,007 | 0,019 | 0,011 | 0,020 | 0,015 | 0,022 |
| 2 | real | 0,507 | 0,439 | 0,700 | 0,598 | 0,240 | 0,035 | 0,056 | 0,220 | 0,635 | 0,231 | 0,510 | 0,668 | 0,060 | 0,103 | 0,025 | 0,044 | 0,076 | 0,244 | 0,548 | 0,176 |
| user | 0,194 | 0,194 | 0,352 | 0,310 | 0,127 | 0,014 | 0,040 | 0,127 | 0,276 | 0,113 | 0,225 | 0,342 | 0,011 | 0,048 | 0,014 | 0,029 | 0,045 | 0,121 | 0,289 | 0,086 |
| sys | 0,054 | 0,021 | 0,026 | 0,029 | 0,017 | 0,014 | 0,015 | 0,020 | 0,041 | 0,019 | 0,041 | 0,015 | 0,029 | 0,007 | 0,010 | 0,015 | 0,016 | 0,019 | 0,009 | 0,013 |
| 3 | real | 0,587 | 0,407 | 0,647 | 0,676 | 0,386 | 0,028 | 0,054 | 0,260 | 0,652 | 0,218 | 0,344 | 0,693 | 0,031 | 0,058 | 0,026 | 0,044 | 0,045 | 0,195 | 0,719 | 0,149 |
| user | 0,195 | 0,208 | 0,307 | 0,333 | 0,167 | 0,010 | 0,041 | 0,115 | 0,314 | 0,105 | 0,157 | 0,314 | 0,017 | 0,040 | 0,012 | 0,031 | 0,029 | 0,087 | 0,363 | 0,084 |
| sys | 0,048 | 0,023 | 0,037 | 0,037 | 0,041 | 0,016 | 0,010 | 0,023 | 0,023 | 0,022 | 0,015 | 0,027 | 0,014 | 0,016 | 0,013 | 0,011 | 0,016 | 0,027 | 0,019 | 0,015 |

Tabela 4.2.2. Tempo real, de usuário e de sistema para três execuções do Heapsort em Java, para as tabelas A, B e C.

| Java | | A1 | A2 | A3 | A4 | A5 | A6 | A7 | A8 | B1 | B2 | B3 | B4 | B5 | B6 | C1 | C2 | C3 | C4 | C5 | C6 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | real | 1,188 | 1,534 | 1,616 | 2,028 | 1,692 | 0,518 | 1,101 | 1,432 | 1,987 | 1,682 | 1,550 | 1,901 | 0,578 | 1,000 | 0,472 | 0,797 | 1,115 | 3,228 | 2,052 | 1,451 |
| user | 0,451 | 0,609 | 0,658 | 0,803 | 0,642 | 0,170 | 0,391 | 0,550 | 0,814 | 0,585 | 0,645 | 0,680 | 0,184 | 0,419 | 0,191 | 0,294 | 0,394 | 0,923 | 0,798 | 0,505 |
| sys | 0,142 | 0,150 | 0,139 | 0,193 | 0,185 | 0,106 | 0,138 | 0,141 | 0,191 | 0,167 | 0,124 | 0,234 | 0,088 | 0,084 | 0,068 | 0,111 | 0,142 | 0,508 | 0,190 | 0,222 |
| 2 | real | 1,306 | 1,478 | 1,759 | 1,913 | 1,673 | 0,503 | 1,011 | 1,254 | 1,859 | 1,320 | 1,661 | 1,825 | 0,557 | 0,962 | 0,533 | 0,828 | 1,041 | 3,235 | 2,064 | 1,841 |
| user | 0,524 | 0,582 | 0,728 | 0,733 | 0,701 | 0,154 | 0,343 | 0,491 | 0,782 | 0,502 | 0,658 | 0,666 | 0,147 | 0,373 | 0,151 | 0,302 | 0,369 | 0,930 | 0,789 | 0,610 |
| sys | 0,114 | 0,153 | 0,141 | 0,215 | 0,120 | 0,119 | 0,150 | 0,140 | 0,152 | 0,161 | 0,170 | 0,194 | 0,131 | 0,140 | 0,116 | 0,110 | 0,129 | 0,551 | 0,213 | 0,288 |
| 3 | real | 1,257 | 1,474 | 1,818 | 1,971 | 1,711 | 0,602 | 1,073 | 1,441 | 1,792 | 1,636 | 1,595 | 1,970 | 0,468 | 0,920 | 0,431 | 0,780 | 1,247 | 2,774 | 1,940 | 1,534 |
| user | 0,487 | 0,568 | 0,750 | 0,818 | 0,639 | 0,170 | 0,407 | 0,579 | 0,676 | 0,638 | 0,655 | 0,747 | 0,163 | 0,337 | 0,143 | 0,310 | 0,370 | 1,039 | 0,732 | 0,595 |
| sys | 0,142 | 0,158 | 0,133 | 0,169 | 0,185 | 0,117 | 0,122 | 0,145 | 0,209 | 0,187 | 0,138 | 0,189 | 0,087 | 0,126 | 0,088 | 0,093 | 0,121 | 0,281 | 0,206 | 0,157 |

Tabela 4.2.3. Tempo real, de usuário e de sistema para três execuções do Heapsort em Python, para as tabelas A, B e C.

|  | | A1 | A2 | A3 | A4 | A5 | A6 | A7 | A8 | B1 | B2 | B3 | B4 | B5 | B6 | C1 | C2 | C3 | C4 | C5 | C6 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | real | 3,484 | 7,348 | 12,178 | 12,402 | 5,000 | 0,108 | 1,048 | 3,120 | 11,746 | 3,636 | 6,885 | 6,863 | 0,100 | 0,911 | 0,084 | 0,732 | 1,192 | 5,159 | 12,346 | 3,327 |
| user | 3,332 | 7,170 | 12,090 | 12,309 | 4,895 | 0,087 | 1,023 | 3,079 | 11,622 | 3,593 | 6,768 | 6,805 | 0,083 | 0,886 | 0,068 | 0,669 | 1,165 | 4,992 | 12,245 | 3,151 |
| sys | 0,061 | 0,081 | 0,057 | 0,062 | 0,052 | 0,013 | 0,017 | 0,029 | 0,073 | 0,032 | 0,064 | 0,041 | 0,014 | 0,016 | 0,012 | 0,022 | 0,018 | 0,070 | 0,057 | 0,065 |
| 2 | real | 3,385 | 7,097 | 12,596 | 12,325 | 5,475 | 0,100 | 1,072 | 3,564 | 11,816 | 3,834 | 6,786 | 6,803 | 0,105 | 0,910 | 0,086 | 0,694 | 1,191 | 4,907 | 12,602 | 3,002 |
| user | 3,227 | 7,013 | 12,319 | 12,237 | 5,181 | 0,083 | 1,047 | 3,364 | 11,654 | 3,772 | 6,723 | 6,742 | 0,085 | 0,885 | 0,068 | 0,670 | 1,193 | 4,855 | 12,490 | 2,964 |
| sys | 0,066 | 0,052 | 0,111 | 0,058 | 0,110 | 0,012 | 0,017 | 0,075 | 0,086 | 0,038 | 0,042 | 0,045 | 0,015 | 0,016 | 0,014 | 0,017 | 0,019 | 0,037 | 0,060 | 0,028 |
| 3 | real | 3,221 | 7,143 | 12,813 | 12,341 | 5,300 | 0,101 | 1,071 | 3,557 | 12,290 | 3,664 | 6,736 | 6,833 | 0,098 | 0,897 | 0,090 | 0,690 | 1,280 | 5,124 | 12,402 | 3,377 |
| user | 3,140 | 7,071 | 12,349 | 12,248 | 4,911 | 0,082 | 1,037 | 3,334 | 11,839 | 3,606 | 6,675 | 6,773 | 0,081 | 0,874 | 0,070 | 0,666 | 1,225 | 5,006 | 12,312 | 3,204 |
| sys | 0,045 | 0,045 | 0,131 | 0,057 | 0,086 | 0,012 | 0,017 | 0,071 | 0,102 | 0,037 | 0,044 | 0,043 | 0,014 | 0,016 | 0,014 | 0,016 | 0,029 | 0,061 | 0,061 | 0,069 |

Os dados das tabelas acima foram utilizados para criar os gráficos das Figuras 4.2.1, 4.2.2 e 4.2.3.

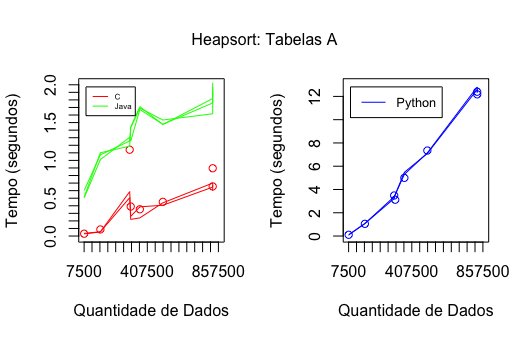


Figura 4.2.1. Gráficos de tempo (em segundos) pela quantidade de dados das tabelas A nas linguagens C e Java (à esquerda) e em Python (à direita), referentes ao Heapsort.

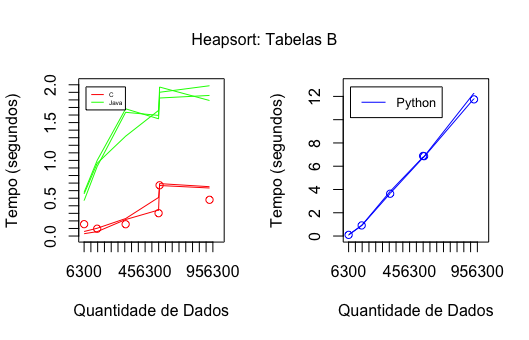


Figura 4.2.2. Gráficos de tempo (em segundos) pela quantidade de dados das tabelas B nas linguagens C e Java (à esquerda) e em Python (à direita), referentes ao Heapsort.

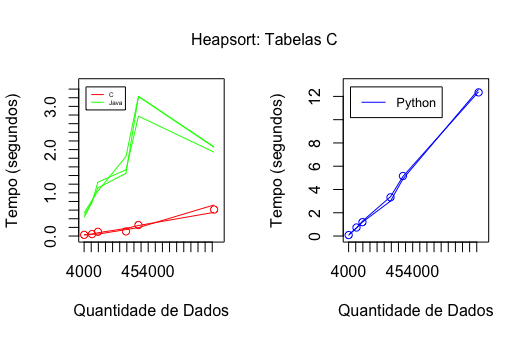


Figura 4.2.3. Gráficos de tempo (em segundos) pela quantidade de dados das tabelas C nas linguagens C e Java (à esquerda) e em Python (à direita), referentes ao Heapsort.

Nos gráficos de tempo para a linguagem Python, pode-se perceber uma tendência aproximadamente linear do aumento do tempo conforme há o aumento do número de dados. Nos gráficos das linguagens C e Java já não se pode afirmar com certeza sobre a natureza seu comportamento, mas como o Heapsort é um algoritmo de complexidade , poderia se esperar um comportamento parecido com, uma função , para .

Pode-se perceber que os tempos de execução do Heapsort, nas três linguagens, permaneceram relativamente constantes independente do cenário investigado. Isso diz respeito à constância do algoritmo para seu melhor, pior e médio caso.

Para o Heapsort é possível observar, novamente, a discrepância entre os tempos em Python e os em C e Java.

## 4.3 Counting Sort

As Tabelas 4.3.1, 4.3.2 e 4.3.3 mostram os dados de tempo coletados referentes à execução do Counting Sort em C, Java e Python, respectivamente.

Tabela 4.3.1. Tempo real, de usuário e de sistema para três execuções do Counting Sort em C, para as tabelas A, B e C.

| C | | A1 | A2 | A3 | A4 | A5 | A6 | A7 | A8 | B1 | B2 | B3 | B4 | B5 | B6 | C1 | C2 | C3 | C4 | C5 | C6 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | real | 0,405 | 0,425 | 0,566 | 0,543 | 0,265 | 0,099 | 0,123 | 0,209 | 1,095 | 0,208 | 0,337 | 0,910 | 0,189 | 0,377 | 0,108 | 0,136 | 0,133 | 0,406 | 0,615 | 0,246 |
| user | 0,083 | 0,152 | 0,249 | 0,209 | 0,093 | 0,026 | 0,065 | 0,077 | 0,370 | 0,086 | 0,123 | 0,225 | 0,031 | 0,065 | 0,023 | 0,050 | 0,043 | 0,097 | 0,221 | 0,089 |
| sys | 0,133 | 0,051 | 0,053 | 0,076 | 0,051 | 0,043 | 0,031 | 0,049 | 0,183 | 0,043 | 0,053 | 0,207 | 0,097 | 0,141 | 0,044 | 0,037 | 0,042 | 0,134 | 0,114 | 0,052 |
| 2 | real | 0,207 | 0,404 | 0,542 | 0,581 | 0,368 | 0,235 | 0,121 | 0,264 | 1,045 | 0,183 | 0,351 | 0,715 | 0,092 | 0,517 | 0,236 | 0,165 | 0,770 | 0,265 | 0,598 | 0,232 |
| user | 0,091 | 0,179 | 0,214 | 0,205 | 0,133 | 0,048 | 0,054 | 0,092 | 0,377 | 0,085 | 0,146 | 0,181 | 0,030 | 0,085 | 0,013 | 0,035 | 0,079 | 0,117 | 0,261 | 0,083 |
| sys | 0,048 | 0,049 | 0,075 | 0,118 | 0,057 | 0,084 | 0,033 | 0,047 | 0,086 | 0,044 | 0,036 | 0,166 | 0,042 | 0,132 | 0,113 | 0,050 | 0,085 | 0,047 | 0,062 | 0,048 |
| 3 | real | 0,336 | 0,419 | 0,613 | 0,573 | 0,270 | 0,120 | 0,131 | 0,222 | 0,706 | 0,274 | 0,301 | 1,014 | 0,188 | 0,375 | 0,088 | 0,125 | 0,202 | 0,243 | 0,456 | 0,204 |
| user | 0,092 | 0,157 | 0,219 | 0,226 | 0,106 | 0,044 | 0,046 | 0,072 | 0,230 | 0,108 | 0,132 | 0,261 | 0,024 | 0,075 | 0,034 | 0,037 | 0,036 | 0,089 | 0,181 | 0,090 |
| sys | 0,100 | 0,040 | 0,070 | 0,088 | 0,046 | 0,027 | 0,037 | 0,048 | 0,109 | 0,042 | 0,060 | 0,199 | 0,105 | 0,135 | 0,045 | 0,048 | 0,072 | 0,049 | 0,076 | 0,057 |

Tabela 4.3.2. Tempo real, de usuário e de sistema para três execuções do Counting Sort em Java, para as tabelas A, B e C.

| Java | | A1 | A2 | A3 | A4 | A5 | A6 | A7 | A8 | B1 | B2 | B3 | B4 | B5 | B6 | C1 | C2 | C3 | C4 | C5 | C6 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | real | 1,556 | 1,607 | 2,211 | 1,890 | 3,359 | 1,120 | 1,073 | 1,762 | 1,814 | 1,460 | 2,058 | 1,811 | 0,733 | 1,083 | 0,660 | 1,065 | 1,310 | 1,667 | 2,162 | 1,557 |
| user | 0,575 | 0,621 | 0,743 | 0,711 | 1,006 | 0,284 | 0,423 | 0,657 | 0,706 | 0,564 | 0,685 | 0,624 | 0,241 | 0,443 | 0,232 | 0,371 | 0,456 | 0,615 | 0,810 | 0,590 |
| sys | 0,187 | 0,206 | 0,156 | 0,218 | 0,613 | 0,267 | 0,130 | 0,230 | 0,202 | 0,157 | 0,318 | 0,239 | 0,123 | 0,121 | 0,119 | 0,166 | 0,186 | 0,195 | 0,237 | 0,190 |
| 2 | real | 1,396 | 1,471 | 2,638 | 2,012 | 3,484 | 0,852 | 1,198 | 1,734 | 1,814 | 1,524 | 1,739 | 1,790 | 0,665 | 1,123 | 0,677 | 1,081 | 1,542 | 1,754 | 2,257 | 1,576 |
| user | 0,547 | 0,599 | 0,654 | 0,738 | 1,014 | 0,251 | 0,438 | 0,627 | 0,706 | 0,581 | 0,581 | 0,670 | 0,206 | 0,427 | 0,207 | 0,414 | 0,446 | 0,576 | 0,827 | 0,605 |
| sys | 0,158 | 0,132 | 0,156 | 0,238 | 0,636 | 0,182 | 0,144 | 0,219 | 0,202 | 0,157 | 0,234 | 0,220 | 0,135 | 0,137 | 0,146 | 0,118 | 0,285 | 0,283 | 0,239 | 0,191 |
| 3 | real | 1,354 | 1,574 | 2,206 | 1,762 | 3,769 | 0,816 | 1,267 | 1,859 | 1,890 | 1,426 | 1,737 | 1,676 | 0,792 | 1,210 | 0,635 | 1,029 | 1,239 | 1,816 | 2,494 | 1,668 |
| user | 0,561 | 0,612 | 0,660 | 0,676 | 1,153 | 0,269 | 0,484 | 0,648 | 0,719 | 0,550 | 0,631 | 0,610 | 0,281 | 0,440 | 0,217 | 0,366 | 0,411 | 0,638 | 0,858 | 0,595 |
| sys | 0,135 | 0,156 | 0,212 | 0,207 | 0,638 | 0,139 | 0,152 | 0,244 | 0,202 | 0,166 | 0,203 | 0,225 | 0,118 | 0,150 | 0,104 | 0,143 | 0,173 | 0,246 | 0,238 | 0,230 |

Tabela 4.3.3. Tempo real, de usuário e de sistema para três execuções do Counting Sort em Python, para as tabelas A, B e C.

| Py | | A1 | A2 | A3 | A4 | A5 | A6 | A7 | A8 | B1 | B2 | B3 | B4 | B5 | B6 | C1 | C2 | C3 | C4 | C5 | C6 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | real | 5,587 | 7,135 | 7,881 | 7,924 | 6,894 | 4,965 | 5,384 | 5,513 | 7,464 | 6,677 | 6,925 | 6,911 | 6,079 | 5,100 | 6,138 | 6,294 | 6,665 | 6,671 | 7,299 | 6,199 |
| user | 5,287 | 6,901 | 7,643 | 7,675 | 6,663 | 4,787 | 4,990 | 5,313 | 7,121 | 6,437 | 6,667 | 6,670 | 5,901 | 4,913 | 5,951 | 6,082 | 6,294 | 6,455 | 7,068 | 5,719 |
| sys | 0,188 | 0,210 | 0,221 | 0,222 | 0,208 | 0,158 | 0,166 | 0,181 | 0,256 | 0,210 | 0,219 | 0,217 | 0,163 | 0,176 | 0,167 | 0,181 | 0,248 | 0,201 | 0,215 | 0,250 |
| 2 | real | 5,553 | 7,104 | 7,901 | 7,798 | 6,845 | 4,933 | 5,741 | 5,504 | 7,314 | 6,892 | 6,944 | 6,925 | 6,065 | 5,283 | 6,351 | 6,206 | 6,536 | 6,823 | 7,650 | 5,607 |
| user | 5,284 | 6,877 | 7,659 | 7,558 | 6,622 | 4,745 | 5,241 | 5,308 | 7,066 | 6,531 | 6,706 | 6,694 | 5,891 | 4,995 | 6,029 | 6,006 | 6,176 | 6,572 | 7,421 | 5,304 |
| sys | 0,193 | 0,210 | 0,222 | 0,222 | 0,207 | 0,165 | 0,268 | 0,182 | 0,230 | 0,251 | 0,220 | 0,216 | 0,161 | 0,204 | 0,223 | 0,178 | 0,236 | 0,207 | 0,214 | 0,217 |
| 3 | real | 5,482 | 7,897 | 8,603 | 7,905 | 7,100 | 4,900 | 5,136 | 6,027 | 7,383 | 6,743 | 6,903 | 6,907 | 6,144 | 5,115 | 6,717 | 6,709 | 6,336 | 6,729 | 7,493 | 5,556 |
| user | 5,278 | 7,272 | 8,015 | 7,655 | 6,792 | 4,724 | 4,959 | 5,832 | 7,131 | 6,488 | 6,670 | 6,679 | 5,954 | 4,932 | 6,175 | 6,266 | 6,126 | 6,510 | 7,166 | 5,296 |
| sys | 0,187 | 0,316 | 0,353 | 0,227 | 0,258 | 0,160 | 0,166 | 0,180 | 0,230 | 0,219 | 0,217 | 0,214 | 0,167 | 0,173 | 0,289 | 0,256 | 0,187 | 0,202 | 0,256 | 0,200 |

Os dados das tabelas acima foram utilizados para criar os gráficos das Figuras 4.3.1, 4.3.2 e 4.3.3.

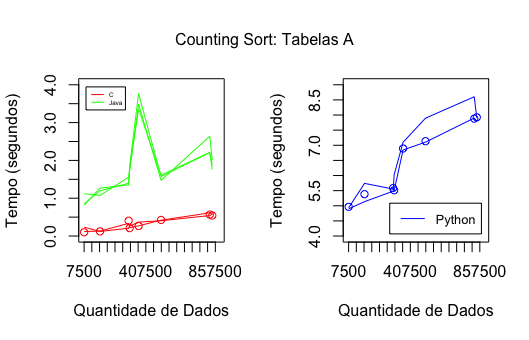
****

Figura 4.3.1. Gráficos de tempo (em segundos) pela quantidade de dados das tabelas A nas linguagens C e Java (à esquerda) e em Python (à direita), referentes ao Counting Sort.

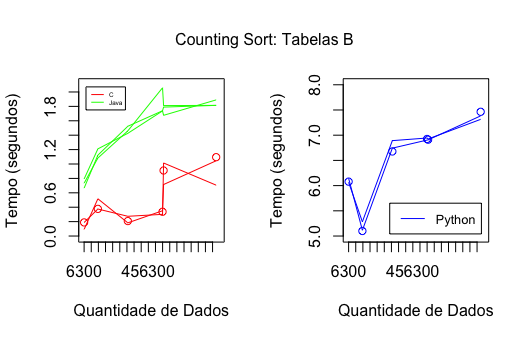


Figura 4.3.2. Gráficos de tempo (em segundos) pela quantidade de dados das tabelas B nas linguagens C e Java (à esquerda) e em Python (à direita), referentes ao Counting Sort.

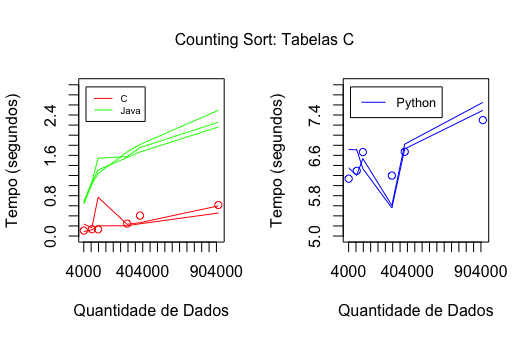


Figura 4.3.3. Gráficos de tempo (em segundos) pela quantidade de dados das tabelas B nas linguagens C e Java (à esquerda) e em Python (à direita), referentes ao Counting Sort.

Nos gráficos das figuras acima é possível identificar um padrão inicial de crescimento muito rápido de tempo, especialmente em Java e Python, o que se torna um crescimento mais constante (ou menos acelerado) conforme aumenta a quantidade de dados.

O Counting Sort é um algoritmo de complexidade , então espera-se um comportamento relativamente linear do tempo em função da quantidade de dados. A visualização desse padrão pode ter sido prejudicada por vários motivos, como pela quantidade de dados muito pequena, pela escala do gráfico, por fatores que possam ter interferido no processamento dos dados enquanto eram coletados etc.

## 4.4 Discussão

Os gráficos 4.4.1, 4.4.2 e 4.4.3 foram criados a partir das médias dos tempos reais de execução dos dados das tabelas A8, B6 e C6 para cada linguagem de programação utilizada.

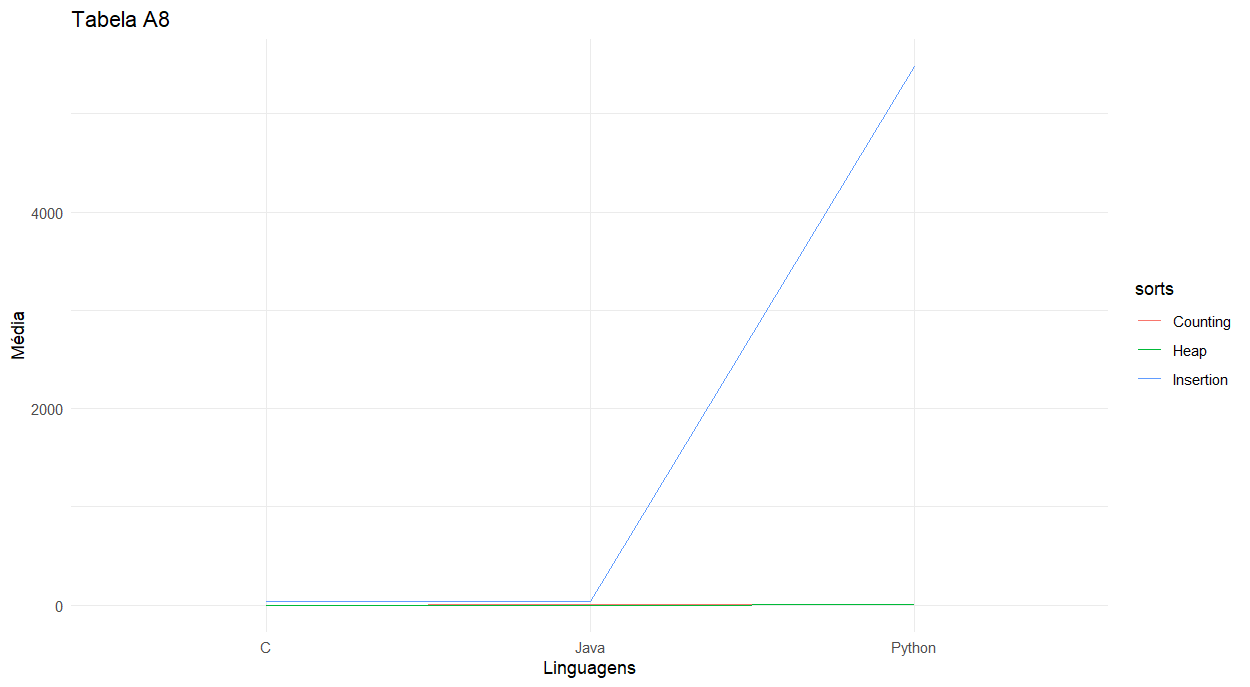
****

Figura 4.4.1. Gráfico de linhas da Tabela A8, comparando as médias dos tempos reais de execução.

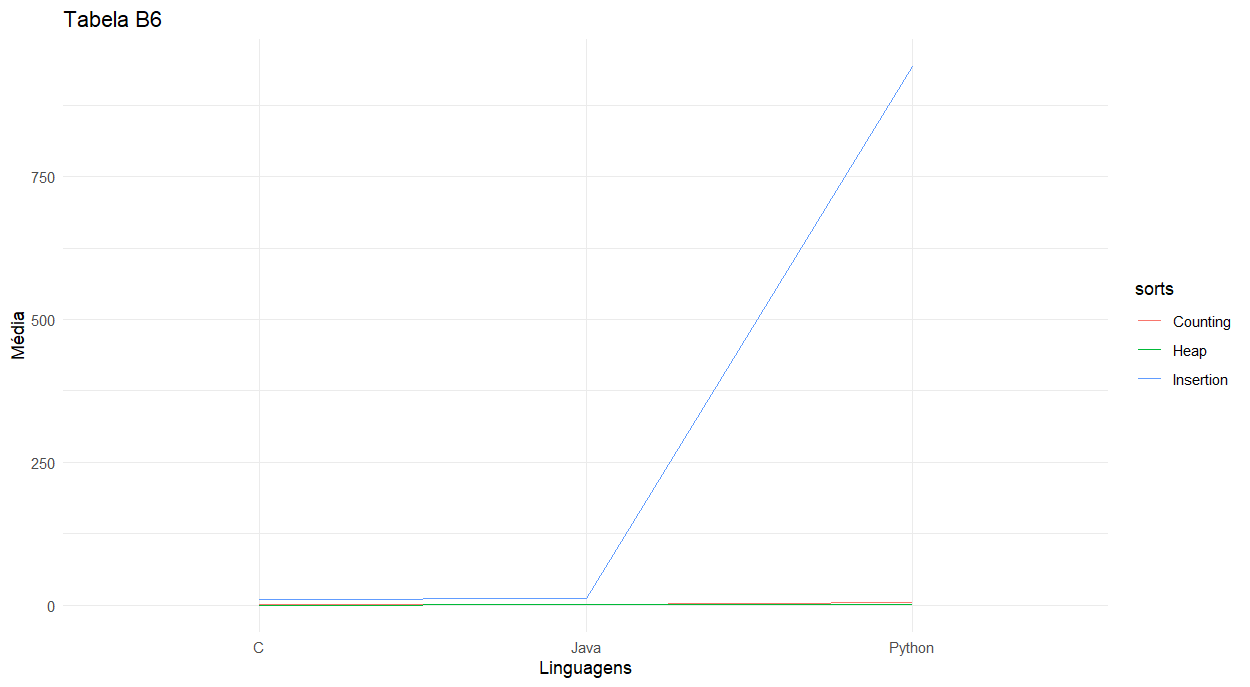


Figura 4.4.2. Gráfico de linhas da Tabela B6, comparando as médias dos tempos reais de execução.

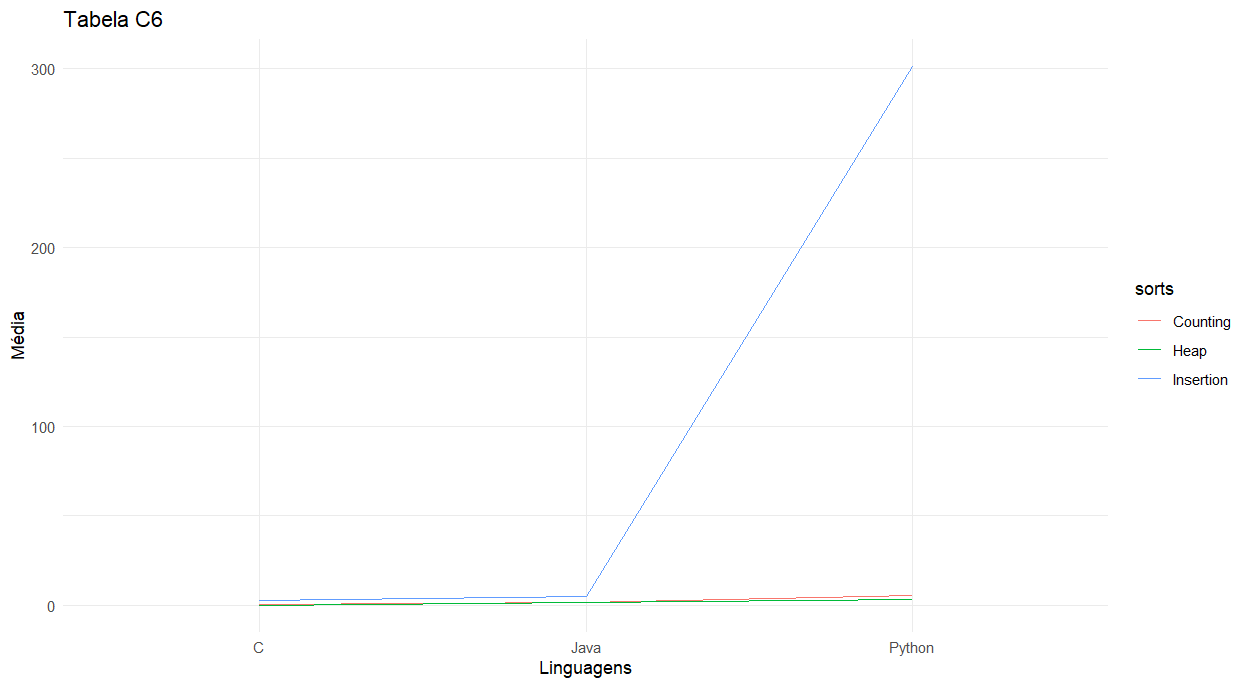


Figura 4.4.3 Gráfico de linhas da Tabela C6, comparando as médias dos tempos reais de execução.

Sejam as tabelas A8, B6 e C6 representantes de cada um dos cenários estudados, foi observado que o Insertion Sort é o algoritmo de ordenação com maior tempo de execução em comparação com o Heapsort e o Counting Sort, que apresentam baixos tempos de execução. Essa diferença é resultado da complexidade de cada algoritmo.

Uma linguagem de programação de tipagem estática não permite alterar o tipo da variável depois de declarada, e essa verificação de tipo ocorre em tempo de compilação, já em uma linguagem de programação de tipagem dinâmica, ela permite que a mesma variável possa receber tipos diferentes ao longo do código, fazendo com que a verificação de tipo ocorra em tempo de execução.

Sendo assim, o fato de Python ter o maior tempo de execução é coerente, pois é uma linguagem de programação de tipagem dinâmica, e como C e Java são de tipagem estática é coerente os seus tempos de execução serem menores.

# 5 CONCLUSÃO

De acordo com os resultados obtidos, os tempos de execução variaram bastante entre os algoritmos, e essa diferença foi ainda mais discrepante em relação ao Insertion Sort por este apresentar tempo de execução muito alto. Sendo assim, pode-se concluir que o Insertion Sort é o algoritmo de ordenação menos eficiente entre os três escolhidos, baseando-se nos seus altos tempos de execução.

Além disso, entre as linguagens de programação escolhidas, os algoritmos em C foram os que obtiveram menor tempo de execução, apresentando pouca diferença com o tempo de execução em Java, e muita diferença com o em Python.

Portanto, a opção mais eficiente para ordenação dentre os 3 algoritmos e linguagens seria o Heapsort programado em linguagem C ou o Counting Sort programado em linguagem C, pois estes algoritmos apresentaram menores tempos de execução nos diferentes cenários analisados.