



Universidade Federal do Rio Grande do Norte – UFRN
Centro de Ensino Superior do Seridó – CERES
Departamento de Computação e Tecnologia – DCT
Bacharelado em Sistemas de Informação – BSI

Relatório de Desempenho em Tempo de Execução de Algoritmos

Alec Can Yalcin

Orientador: Prof. Dr. Nome Completo do Professor

Relatório Técnico feito como avaliação ins-
titucional na matéria de Estrutura de Dados,
DCT0008..

$\mathcal{L}\alpha\beta e\pi$

Laboratório de Elementos do Processamento da Informação – LabEPI
Caicó, RN, 15 de julho de 2024

Resumo

Este trabalho apresenta uma análise de tempo de execução de diferentes algoritmos de ordenação utilizando-se da linguagem de programação python. Incluindo a análise analítica do tempo de execução como teoria base para comparar a tendência de crescimento e evolução de algoritmos de ordenação.

Palavras-chave: Python; Algoritmos; Ordenação; Análise; Tempo; Execução.

Abstract

This document presents a run time analysis of different types of sort algorithms using the programming language python. Including analytical run time analysis as the basis theory to compare the evolution and tendency of growth from sort algorithms.

Keywords: Algorithms; Sort; Analysis; Run time.

Sumário

1	Introdução	4
1.1	Motivação	4
1.2	Objetivos	4
1.3	Organização do trabalho	4
2	Metodologia	5
2.1	Coleta de Dados	5
2.2	Limitações	6
3	Desenvolvimento	7
3.1	Apresentação dos Resultados	7
3.1.1	Algoritmos	7
3.1.2	Tempo de Execução	14
3.2	Discussão dos Resultados	27
4	Conclusões	29

1. Introdução

Relatório feito para a atividade avaliativa de Estrutura de Dados(DCT0008). Tem como foco a análise de algoritmos de ordenação estudados em sala apartir de sua implementação e comparação de valores de execução e tempo. Esse capítulo está organizado da seguinte forma: Motivação, Objetivos e Organização do trabalho.

1.1 Motivação

A grande justificativa desse trabalho é analisar e compreender o funcionamento de algoritmos de ordenação e como sua análise de tempo de execução consegue prever falhas e dificuldades que poderão ser encontradas na formulação desses códigos. Sendo uma importante experiência e oportunidade de conhecimento adquirido através de testes.

1.2 Objetivos

Este trabalho tem como objetivo analisar algoritmos associados ao problema de ordenação. Considerando diferentes códigos, suas vantagens e desvantagens, tempo de execução, custo operacional e comparação.

1.3 Organização do trabalho

Os capítulos abaixo estão organizados em Metodologia, onde é abordado a maneira como os experimentos desse relatório foram organizados, Desenvolvimento, abordando especificamente o que os testes e a elaboração a partir da metodologia geraram, e Conclusão, onde é discutivo os resultados obtidos a partir de uma perspectiva de análise de código.

2. Metodologia

Para entender o desenvolvimento desse documento, é preciso analisar como o experimento científico foi realizado. Para isso, esse Capítulo está organizado da seguinte forma: Desenho do Estudo; onde é abordado o tipo de estudo e a justificativa de sua escolha; Objeto de Estudo; onde se define o que exatamente vai ser estudado sendo esta a população e amostras do trabalho; Coleta de Dados; onde é abordado os instrumentos de coleta dos dados, procedimentos e análise dos mesmos; Limitações; onde é explicado toda a parte que o trabalho não abrange e suas dificuldade, assim como suas falhas.

2.1 Coleta de Dados

Para coletar os dados sobre a eficácia dos algoritmos de ordenação, a linguagem escolhida foi Python pela sua facilidade de manipulação. Todos os códigos foram escritos na linguagem de maneira individual em cada arquivo. Organizados em funções em seus respectivos arquivos, cada um deles possui uma outra função denominada `__main__` que foi reescrita para receber valores durante a chamada do arquivo no terminal, dessa forma ao realizar a chamada do script seria colocado o tamanho do vetor. Por fim, cada código de ordenação possui variações conforme sua análise analítica de tempo de execução, em algoritmos que possuem melhores casos e piores casos de execução foram criados três variações de arquivo: best, mean e worse.

Para avaliar a coleta de dados, o instrumento utilizado foi o script `iterate.sh` para gerar de forma automática um arquivo de texto com os tempos de execução. tendo o script, o algoritmo formatado e um local; gera uma série de arquivos de acordo com o tamanho inicial, incremento e tamanho final estabelecido. O script funciona com os seguintes critérios:

```
$ sh iterate.sh

SYNTAX:
iterate.sh <executions> <size-start> <size-pass> <size-end> <program> <name>

<executions> - number of program execution iterations
<size-start> - initial value of 'n' (size of the problem instance)
<size-pass> - increment value of 'n' (size of the problem instance)
<size-end> - final value of 'n' (size of the problem instance)
<program> - program string (name plus additional arguments)
<name> - base string used to create filenames

$ sh iterate.sh 1000 100 900 "python3 <algoritmo>.py"
↪ <caminho>/<algoritmo>_<mean-best-worse>
```

Os arquivos gerados estão na pasta destinada com os nomes formatados de acordo com as condições estabelecidas no critério `jname`. Esses arquivos mostram somente números, sendo estes o tempo de execução de cada uso do algoritmo repetidamente. O programa gera um arquivo final que contem as informações do tamanho do vetor, do tempo de execução médio de ordenação baseado em todos os outros arquivos, e outras informações que não são necessárias para essa explicação. Após gerar esses arquivos, o próximo passo é gerar as imagens de gráficos que serão usadas de comparativo entre as informações. Para isso, foi utilizada a ferramenta gnuplot, também executada no terminal. Tendo o arquivo final e com o uso de algumas linhas de comando, podemos gerar um .png do gráfico que o programa produz:

```
$ gnuplot
gnuplot> set title "Gráfico <Nome do Algoritmo>"
gnuplot> set xlabel "Tamanho do Vetor"
gnuplot> set ylabel "Tempo em milisegundos"
gnuplot> set autoscale
gnuplot> plot "<arquivo_iterate>" using 1:2 with linespoints title <algoritmo>
gnuplot> set terminal png
gnuplot> set output "grafico_<algoritmo>.png"
gnuplot> reprint
```

Para gerar arquivos de algoritmos com multiplos casos, na linha do comando "plot", ao final, coloca-se uma "," e repete o comando mas mudando para o caso desejado.

```
gnuplot> plot "<arquivo_iterate_best>" using 1:2 with linespoints title "best", \
"<arquivo_iterate_mean>" using 1:2 with linespoints title "mean", \
"<arquivo_iterate_worse>" using 1:2 with linespoints title "worse"
```

Com os gráficos gerados e transformado em .png, a análise comparativa está quase completa. Por fim, a análise analítica de tempo foi feita e colocada logo acima dos gráficos, mostrando a diferença teórica e estatística dos algoritmos.

2.2 Limitações

Devido a escolha de Python pela facilidade, o resultado da pesquisa pode não ser satisfatório como a teoria propõe. Embora fácil, Python certamente possui muitas ineficiências que acabam matando certos processos rápidos em outras linguagens. A quantidade de interações por tamanho de array pode não ser suficiente para mostrar a média geral, e valores maiores tendem a dificultar o processamento do python devido a recursividade de certos códigos.

3. Desenvolvimento

Neste capítulo as seções estão organizadas na análise de cada algoritmo. Sendo primeiro a apresentação dos resultados, mostrando os algoritmos construídos e o seu tempo de execução analítico e logo em seguida os resultados de interação com o script iterate, com gráficos e o texto gerado pelas execuções.

3.1 Apresentação dos Resultados

3.1.1 Algoritmos

Todos os elementos dessa seção contam com os algoritmos em python escritos e seu cálculo de tempo de execução analítico em suas diversas formas (best, mean, worse). Além de uma explicação do funcionamento do algoritmo antes de qualquer desenvolvimento.

Selection

Da lista de algoritmos analisados, o Selection Sort é o primeiro e mais simples. Ao mesmo tempo que é o código que possui maior custo computacional. Ele se baseia na lógica de comparação e substituição, onde percorre o vetor escolhido e compara o índice atual com todo o restante da lista, pegando o menor valor e substituindo-o no final pelo índice atual, assim ele passa para o próximo índice e repete o mesmo processo para os itens restantes da lista.

```
1 import sys
2 import time
3 import random as rm
4
5 def selection_sort(A, n):
6     for i in range(n):
7         min_index = i
8         for j in range(i+1, n):
9             if (A[j] < A[min_index]):
10                 min_index = j
11         A[i], A[min_index] = A[min_index], A[i]
12     return A
13
14 # médio caso: vetor aleatório
15 if __name__ == "__main__":
16     n = int(sys.argv[1])
17     v = [rm.randint(0, 1000) for x in range(n)]
18     first_time = time.time_ns()
19     v = selection_sort(v, n)
20     final_time = time.time_ns()
21     total_time = final_time - first_time
22     print(total_time)
```

Figura 3.1: Código do Selection Sort

Insertion

Um pouco melhor que o algoritmo anterior, o Insertion Sort percorre toda a lista, mas diferentemente do selection, ele faz isso somente quando necessário. Ao percorrer o vetor com um índice, ele faz comparações com o elemento anterior. Ao indentificar se o elemento anterior é maior que o atual e que é um índice válido, ele faz com que esse indice anterior substitua a posição atual, e então compara com o elemento anterior ao elemento anterior. Isso se repete até que não haja mais indices válidos ou números maiores. Com esse processo, ele consegue fazer uma lista de itens maiores que vão crescendo dentro da própria lista.

```
1 import sys
2 import time
3 import random as rm
4
5 def insertion_sort(v, n):
6     for i in range(1, n):
7         k = v[i]
8         j = i-1
9         while(j>=0 and v[j] > k):
10             v[j+1] = v[j]
11             j = j-1
12         v[j+1] = k
13     return v
14
15 # médio caso: vetor aleatório
16 if __name__ == "__main__":
17     n = int(sys.argv[1])
18     v = [rm.randint(0, 1000) for x in range(n)]
19     first_time = time.time_ns()
20     v = insertion_sort(v, n)
21     final_time = time.time_ns()
22     total_time = final_time - first_time
23     print(total_time)
```

Figura 3.2: Código do Insertion Sort

```
1 import sys
2 import time
3 from insertion_sort import insertion_sort
4
5 # Melhor caso: vetor já ordenado
6 if __name__ == "__main__":
7     n = int(sys.argv[1])
8     v = list(range(n))
9     first_time = time.time_ns()
10    v = insertion_sort(v, n)
11    final_time = time.time_ns()
12    total_time = final_time - first_time
13    print(total_time)
```

Figura 3.3: Código do Insertion Sort no seu melhor caso

```
1 import sys
2 import time
3 from insertion_sort import insertion_sort
4
5 # Pior caso: vetor ordenado inversamente
6 if __name__ == "__main__":
7     n = int(sys.argv[1])
8     v = list(range(n))
9     v.reverse()
10    first_time = time.time_ns()
11    v = insertion_sort(v, n)
12    final_time = time.time_ns()
13    total_time = final_time - first_time
14    print(total_time)
```

Figura 3.4: Código do Insertion Sort no seu pior caso

Merge

O Merge-sort é um tipo de algoritmo de custo computacional alto e resultados excepcionais. Esse algoritmo ordena o vetor escolhido criando sub-listas menores até que não seja possível mais dividi-las, geralmente quando se chega a 2 elementos cada lista. Após isso, essas sub-listas são ordenadas das menores até as maiores, comparando os elementos e reordenando-os na lista principal. Esse processo chega de duas posição na lista, para quatro, para oito, e assim em diante.

```

1 import sys
2 import time
3 import random as rm
4
5 def merge_sort(A, start, end):
6     if start < end:
7         middle = (start + end)//2
8
9         merge_sort(A, start, middle)
10        merge_sort(A, middle+1, end)
11        merge(A, start, middle, end)
12    return A
13
14 def merge(A, start, middle, end):
15     i = start
16     j = middle + 1
17
18     v = [0]*len(A)
19     for k in range(0, end-start):
20         if (i <= middle) and ((j>end) or (A[i] < A[j])):
21             v[k] = A[i]
22             i += 1
23         else:
24             v[k] = A[j]
25             j += 1
26     for k in range(0, end-start):
27         A[start+k] = v[k]
28
29 # médio caso: vetor aleatório
30 if __name__ == "__main__":
31     n = int(sys.argv[1])
32     v = [rm.randint(0, 1000) for x in range(n)]
33     first_time = time.time_ns()
34     v = merge_sort(v, 0, n-1)
35     final_time = time.time_ns()
36     total_time = final_time - first_time
37     print(total_time)

```

Figura 3.5: Código do Merge Sort

Quick

Embora não seja tão rápido quanto o algoritmo anterior, o Quick-sort possui lógica “parecida” e menor custo computacional. Estabelecendo um elemento como pivô: é o elemento central do algoritmo, ou seja, acredita-se que ele será exatamente o elemento do meio. Assim, a lista é organizada com base nos elementos que são menores que o pivô e que são maiores que ele. Não necessariamente essa organização gera a lista já ordenada, então, o processo é repetido novamente, mas dessa vez tendo o conhecimento que o pivô inicial já está ordenado, ou seja, é estabelecido um novo pivô para o restante da lista desorganizadas, e como criou-se novas listas menores ou maiores que ele, estabelecemos um novo pivô para cada uma delas. A vantagem do quick-sort é que ele não cria sub-listas como o merge-sort, ao invés disso organiza a mesma lista do início.

```

1 import sys
2 import time
3 import random as rm
4
5 def partition(A, start, end):
6     pivot = A[end]
7     i = start - 1
8     for j in range(start, end):
9         if A[j] <= pivot:
10             i += 1
11             A[i], A[j] = A[j], A[i]
12     A[i + 1], A[end] = A[end], A[i + 1]
13     return i + 1
14
15 def quick_sort(A, start, end):
16     if start < end:
17         pivot = partition(A, start, end)
18         quick_sort(A, start, pivot - 1)
19         quick_sort(A, pivot + 1, end)
20     return A
21
22 # médio caso: vetor aleatório
23 if __name__ == "__main__":
24     n = int(sys.argv[1])
25     v = [rm.randint(0, 1000) for x in range(n)]
26     first_time = time.time_ns()
27     v = quick_sort(v, 0, n-1)
28     final_time = time.time_ns()
29     total_time = final_time - first_time
30     print(total_time)

```

Figura 3.6: Código do Quick Sort

```

1 import sys
2 import time
3 from quick_sort import quick_sort
4
5 # pior caso: o vetor já está ordenado
6 if __name__ == "__main__":
7     n = int(sys.argv[1])
8     v = list(range(n))
9     first_time = time.time_ns()
10    v = quick_sort(v, 0, n-1)
11    final_time = time.time_ns()
12    total_time = final_time - first_time
13    print(total_time)

```

Figura 3.7: Código do Quick Sort em seu pior caso

Distribution

O Distribution Sort organiza o vetor dado com uma série de transformações de vetores, que reconhecem a posição de cada elemento da lista progressivamente de maneira ordenada. Criando no algoritmo um vetor que é a diferença entre o maior e menor valor do vetor original. Esse vetor criado recebe, inicialmente, 0 em todas as suas posições. Após isso, de o vetor original é percorrido novamente, e o valor de cada índice será subtraído com o menor valor encontrado no início do algoritmo, e essa conta resultante será a posição em que será adicionado um no vetor criado. A partir disso, o vetor criado é percorrido novamente, e então começando da posição inicial+1, ele soma com o valor anterior até o fim do vetor. O resultado é um vetor criado com números que vão de 1 até n, em ordem crescente. Finalmente, criamos um outro vetor com posições 0 de tamanho do vetor original, e esse percorre o vetor original, pegando cada índice atual, subtraindo com o menor valor, e com essa conta encontrando o valor no vetor criado anterior e subtraindo um do que foi encontrado, por fim colocando isso na posição do vetor para ser ordenado e recebendo o valor atual do vetor original. Além disso, é subtraído -1 do vetor criado anterior para que não haja repetições de números iguais em mesmo índice.

```
1 import sys
2 import time
3 import random as rm
4
5 def distribution_sort(v, n):
6     s = min(v)
7     e = max(v)
8     k = e-s
9
10    c, z = [], []
11    for i in range(k+1):
12        c.append(0)
13    for i in range(n):
14        c[v[i]-s] = c[v[i]-s]+1
15    z.append(0)
16    for i in range(1,k+1):
17        c[i] = c[i]+c[i-1]
18    for i in range(n):
19        d = v[i]-s
20        z[c[d]-1] = v[i]
21        c[d] = c[d]-1
22
23    return z
24
25 if __name__ == "__main__":
26     n = int(sys.argv[1])
27     v = [rm.randint(0, 1000) for x in range(n)]
28     first_time = time.time_ns()
29     v = distribution_sort(v, n)
30     final_time = time.time_ns()
31     total_time = final_time - first_time
32     print(total_time)
```

Figura 3.8: Código do Distribution Sort

3.1.2 Tempo de Execução

Todos os elementos dessa seção contam com os gráficos resultante dos algoritmos ditados acima utilizando da ferramenta gnuplot. Em caso de algoritmos com tempo de execução variado em casos, todos os gráficos estarão juntos em uma mesma imagem.

Selection

Análise Analítica do Selection Sort

09 / 06 / 24

algorithm selection-sort (V, n):

- 1 for i from 1 to $(n-1)$ do
- 2 min $\leftarrow i$
- 3 for j from $(i+1)$ to n do
- 4 if $V[j] < V[min]$ then
- 5 min $\leftarrow j$
- 6 $V[i] \leftrightarrow V[min]$

O algoritmo não põe um "melhor" ou "pior" caso - Para executar todas as interações (ou $\frac{n(n-1)}{2}$ trocas)

$T(n) = nC_1 + (n-1)C_2 + C_3 \sum_{i=1}^{n-1} i + C_4 \sum_{i=1}^{n-1} i^2 + (n-1)C_6$

$T(n) = nC_1 + (n-1)(C_2 + C_6) + C_3 \left(\frac{n(n+1)}{2} - 1 \right) + C_4 \left(\frac{n(n+1)}{2} - n \right)$

$\sum_{i=1}^{n-1} i = \frac{n(n+1)}{2} - 1$

$\sum_{i=1}^{n-1} i^2 = \frac{n(n+1)(2n+1)}{6}$

$\Theta(n^2)$

$\Theta(n^2)$

$\Theta(n^2)$

Alex Cam Yalcin

$T(n) = nC_1 + (n-1)(C_2 + C_6) + C_3 \left(\frac{n(n+1)}{2} - 1 \right) + C_4 \left(\frac{n(n+1)}{2} - n \right)$

Reciclado com embalagem da Tetra Pak

Figura 3.9: Selection Sort cálculo do tempo analítico

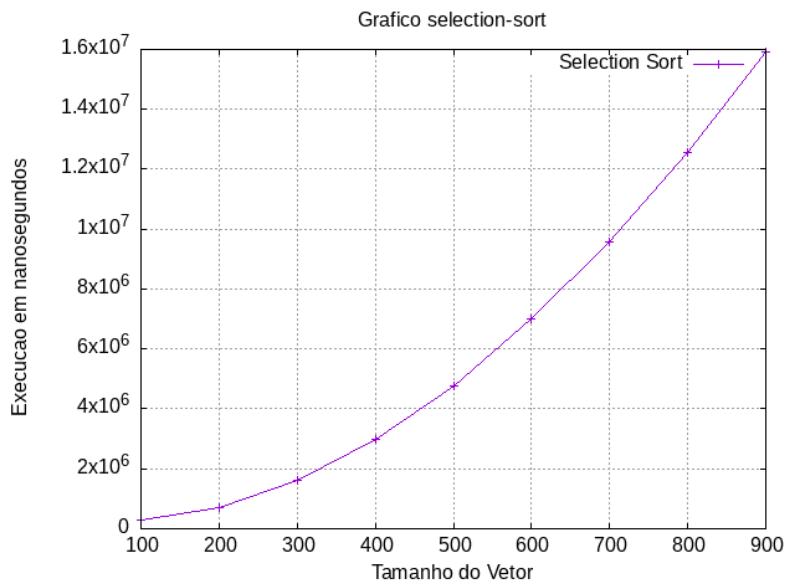


Figura 3.10: Selection Sort gráfico de tempo de execução

Insertion

04/06/24 Análise Analítica do Inserção Sert

algoritmo inserção_sert(V, h):

1. for i from 2 to n do:
2. $k \leftarrow V[i]$
3. $j \leftarrow i - 1$
4. while ($j \geq 1$ and $V[j] > k$) do
 5. $V[j+1] \leftarrow V[j]$
 6. $j \leftarrow j - 1$
7. $V[j+1] \leftarrow k$

(*) algoritmo para inserção de um elemento no vetor de origem.

$T_L(n) = nC_1 + (n-1)(C_2 + C_3 + C_7)$

$T_B(n) = n(C_1 + (n-1)(C_2 + C_3 + C_7))$

$T_W(n) = nC_1 + (n-1)(C_2 + C_3 + C_7) + C_4 \sum_{i=1}^{(n-1)} i + (C_5 + C_6) \sum_{i=1}^{(n-2)} i$

$\begin{array}{rccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \text{a} & * & * & * & * & & \\ \text{l} & 0 & & & & & \\ \text{b} & 1 & & & & & \\ \text{w} & 1 & 1 & 2 & 2 & & \\ \vdots & \vdots & \vdots & \vdots & \vdots & & \end{array}$

$T_W = nC_1 + (n-1)(C_2 + C_3 + C_7) + C_4 \left(\frac{n}{2}(n+1) - n \right) + (C_5 + C_6) \left(\frac{n}{2}(n+1) - 2n + 1 \right)$

$\sum_{i=1}^{(n-1)} i = \frac{n}{2}(n+1) - n$

$\sum_{i=1}^{(n-2)} i = \frac{n}{2}(n+1) - 2n + 1$

$O(1)$
 $O(n^2)$
 $\Theta(n)$

$T_W = nC_1 + (n-1)(C_2 + C_3 + C_7) + C_4 \left(\frac{n}{2}(n+1) - n \right) + (C_5 + C_6) \left(\frac{n}{2}(n+1) - 2n + 1 \right)$



Figura 3.11: Insertion Sort cálculo do tempo analítico

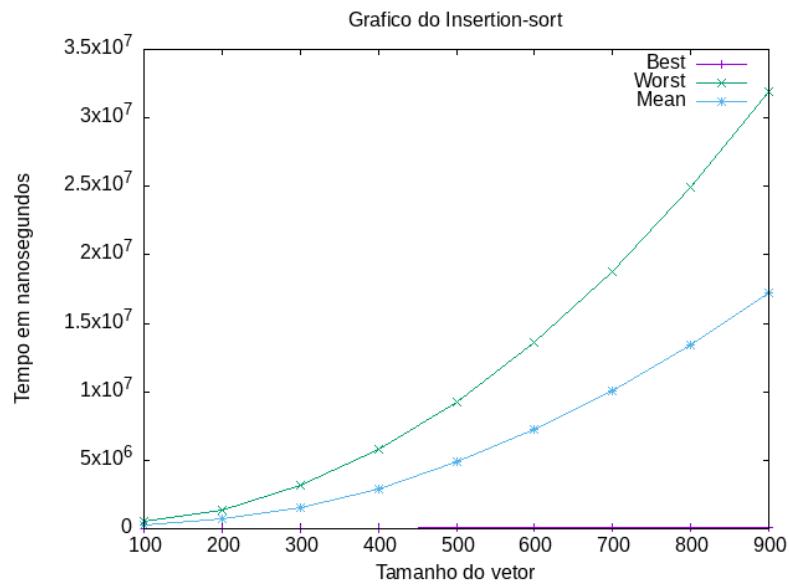


Figura 3.12: Insertion Sort gráfico de tempo de execução

Merge

Análise Analítica Merge Sort

algorithm merge-sort (A, s, e)

1. if $s < e$ then
2. | $m \leftarrow (s+e)/2$
3. | merge-sort (A, s, m)
4. | merge-sort ($A, m+1, e$)
5. | merge (A, s, m, e)

algorithm merge (A, s, m, e)

1. $i \leftarrow s$
2. $j \leftarrow m+1$
3. for $k \leftarrow 1$ to $e-s+1$ do
 1. if ($A[i] < A[j]$) and $i \leq m$ or ($j > e$) then
 1. $B[k] \leftarrow A[i]$
 2. $i \leftarrow i+1$
 - else
 1. $B[k] \leftarrow A[j]$
 2. $j \leftarrow j+1$
4. for $k \leftarrow 1$ to $e-s+1$ do ($n/4$)
 1. $A[s+k] \leftarrow B[k]$

$T(1) = C_1 \cdot a$

$T(n) = C_1 + C_2 + C_3 + C_4 + C_5 + T(n/2) + T(n/2) + T_m(n)$

$T(n) = a + 2T(n/2) + T_m(n)$

$T(n/2) = a + 2T(n/4) + T_m(n/2)$

$T(n) = a + T_m(n) + 2 \cdot [a + 2T(n/4)]$

$T(n) = 3a + T_m(n) + 2T_m(n/2) + 4[a + 2T(n/8) + T_m(n/8)]$

$T(n) = 7a + T_m(n) + 2T_m(n/2) + 4T_m(n/8) + 8T(n/8)$

$T(n) = (2^{x-1})a + 2^x T(n/2^x) + \sum_{j=0}^{x-1} 2^j T_m(n/2^j)$

 Reciclado com embalagem da Tetra Pak

 PROTEGE
Tetra Pak
O QUE É BOM

Figura 3.13: Merge sort cálculo do tempo analítico 1

1 / 1 $\frac{n}{2} = 1 \quad 2^x = n$

$$T_m(n) = bn + c$$

$$T(n) = (2^x - 1)a + 2^x T(\frac{n}{2}) + xbn + c(2^x - 1)$$

$$\hookrightarrow \frac{n}{2^x} = 1 \Rightarrow 2^x = n \Rightarrow \log_2 n = \log_2 2^x \Rightarrow \log_2 n = x$$

$$T(n) = (2^{\log_2 n} - 1)a + 2^{\log_2 n} T(1) + \log_2 n bn + c(2^{\log_2 n} - 1)$$

$$T(n) = (n-1)a + nC_1 + \log_2 n bn + C(n-1)$$

$$T(n) = (a+c)(n-1) + nC_1 + \log_2 n bn$$

$$T(n) = b \cdot (\frac{n}{2}) + c$$

$$\sum_{i=0}^{x-1} 2^i \cdot T_m(\frac{n}{2^i}) \Rightarrow \sum_{i=0}^{x-1} 2^i \cdot \sum_{j=0}^{x-1} T_m(\frac{n}{2^j})$$

$$= \sum_{i=0}^{x-1} 2^i \cdot \sum_{j=0}^{x-1} (bn + c)$$

$$= \sum_{i=0}^{x-1} 2^i \cdot \sum_{j=0}^{x-1} b(\frac{n}{2^j}) + c$$

$$= \sum_{i=0}^{x-1} 2^i \cdot \left(\sum_{j=0}^{x-1} b(\frac{n}{2^j}) \right) + \sum_{i=0}^{x-1} c$$

$$T(n) = (a+c)(n-1) + nC_1 + \log_2 n bn$$


PROTEGE
Tetra Pak®
OQUE E BOM

 Reciclado com embalagem da Tetra Pak

Figura 3.14: Merge Sort cálculo do tempo analítico 2

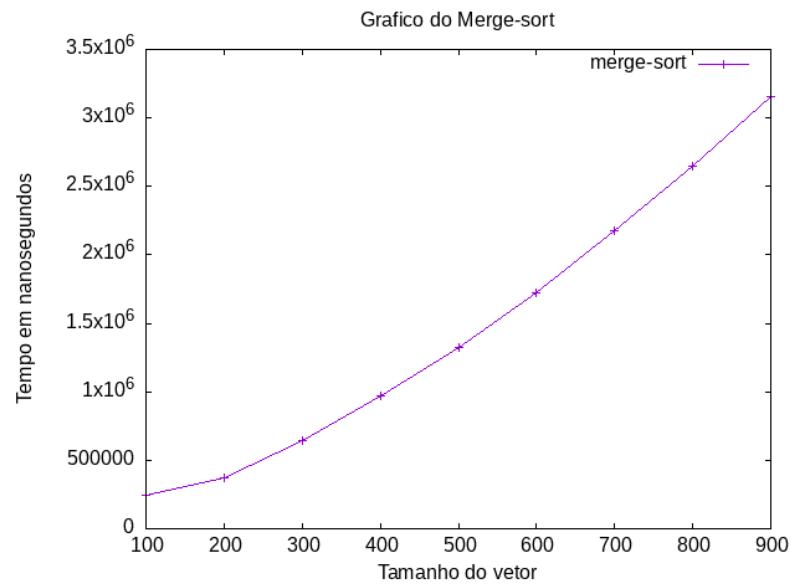


Figura 3.15: Merge Sort gráfico de tempo de execução

Quick

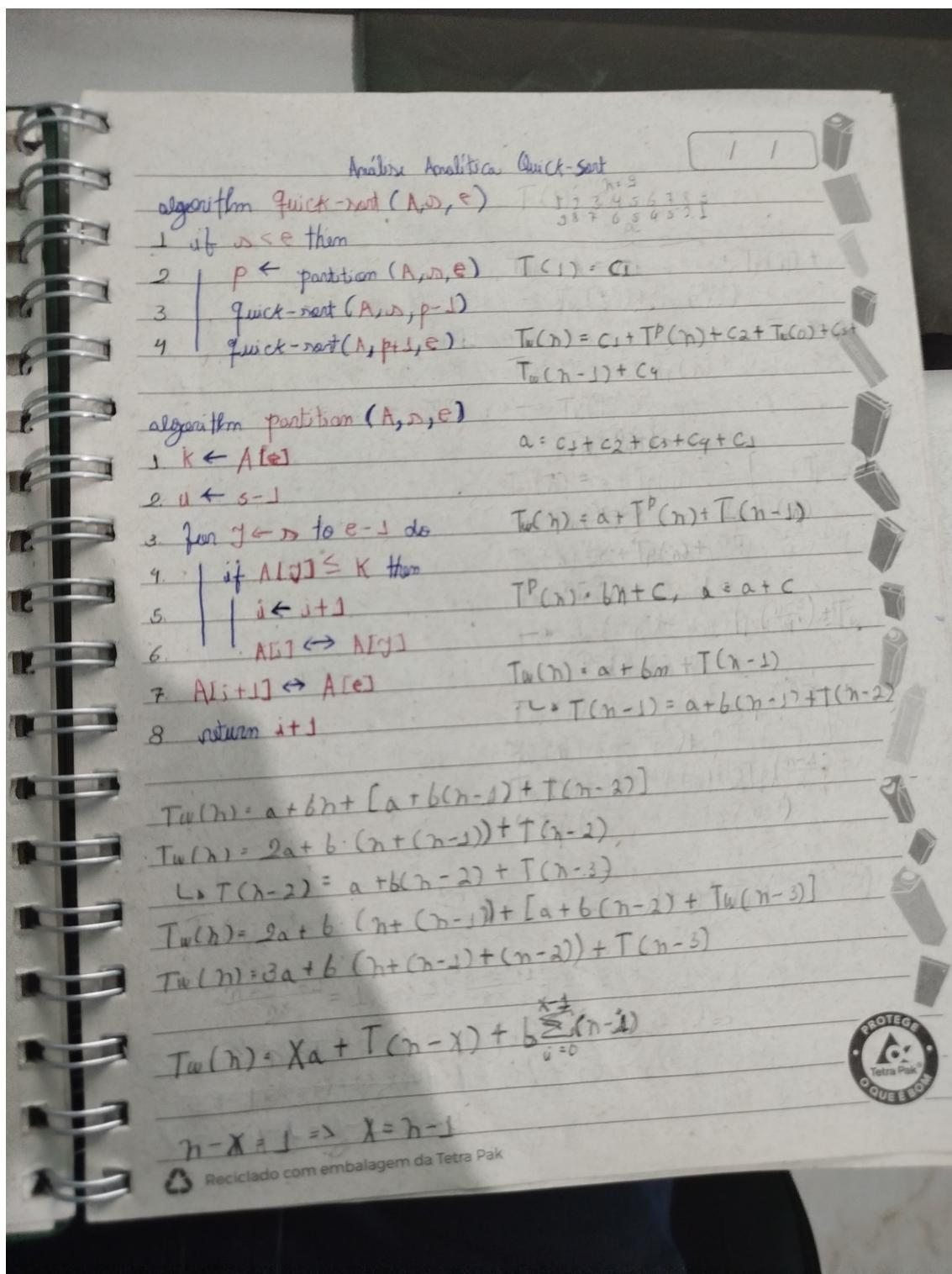


Figura 3.16: Quick Sort cálculo do tempo analítico 1

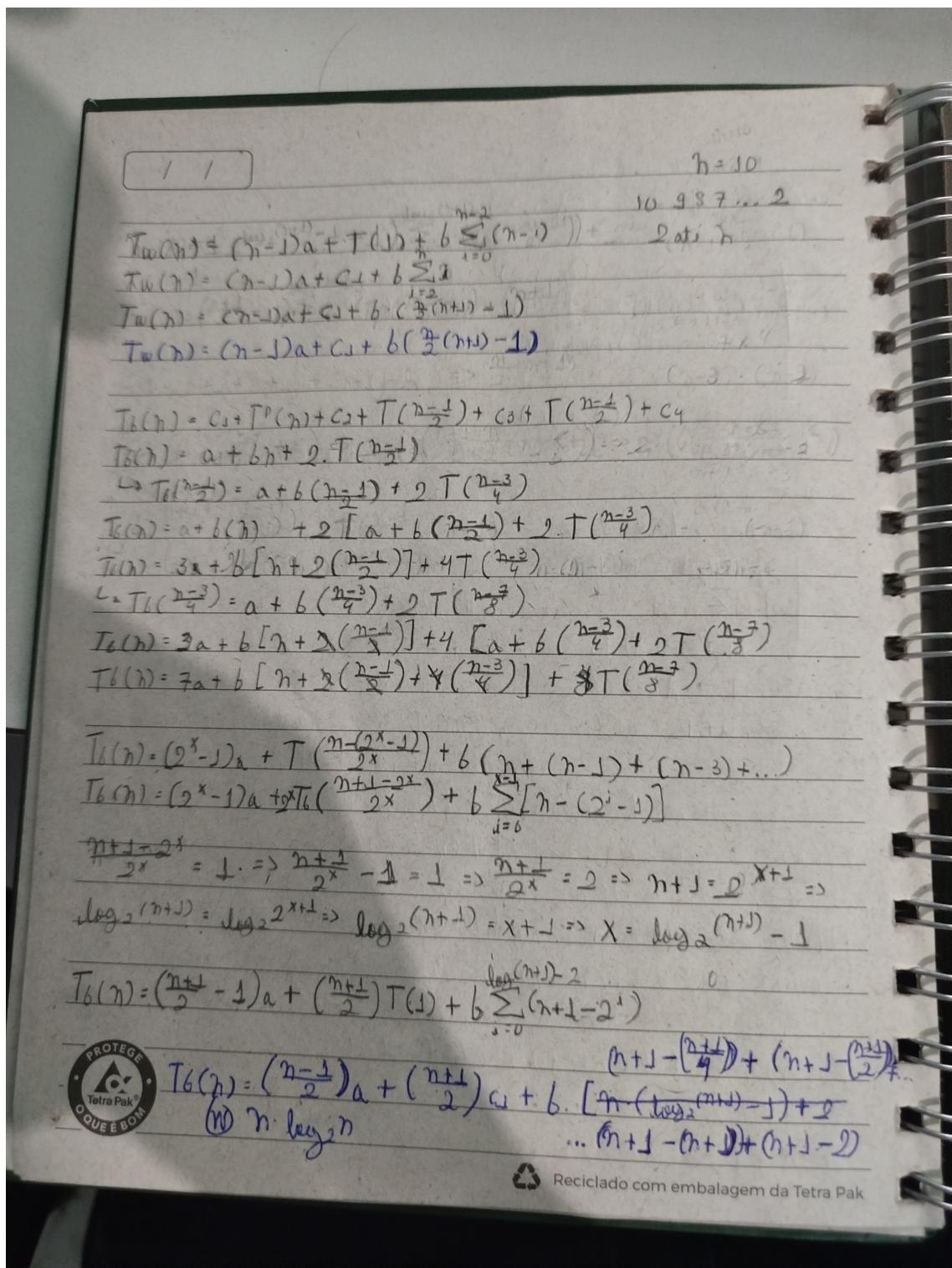


Figura 3.17: Quick Sort cálculo do tempo analítico 2



Reciclado com embalagem da Tetra Pak

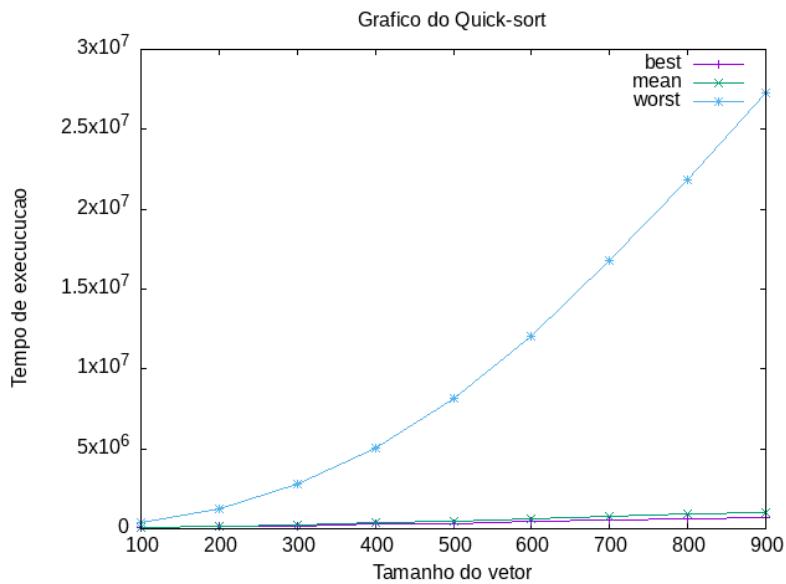


Figura 3.18: Quick Sort gráfico de tempo de execução

Distribution

Análise Analítica Distribution-Sort

1 / 1

algorithm distribution-sort(V, n)

1. $l \leftarrow \min(V, n)$
2. $b \leftarrow \max(V, n)$
3. for $i \leftarrow 0$ to $(b-1)$ do:
4. $W[i] \leftarrow 0$
5. for $j \leftarrow 0$ to $(n-1)$ do
6. $W[V[j]-1] \leftarrow W[V[j]-1] + 1$
7. for $j \leftarrow 0$ to $(b-1)$ do
8. $W[i] \leftarrow W[i] + W[i-1]$
9. for $j \leftarrow 0$ to $(n-1)$ do
10. $Z[W[V[j]-1]] \leftarrow V[j]$
11. $W[V[j]-1] \leftarrow W[V[j]-1] - 1$

$T(n, k) = T_{\min}(n) + C_1 + T_{\max}(n) + C_2 + (k+1)C_3 + k(C_4 + C_5) + (n-1)C_6 + kC_7 + (k-1)C_8 + nC_9 + (n-1)(C_{10} + C_{11})$

$a = C_1 + C_2 + C_3 - C_6 - C_8 - C_{10} - C_{11}$

$T(n, k) = a + k(C_3 + C_4 + C_7 + C_8) + n(C_6 + C_8 + C_9 + C_{10} + C_{11}) + T_{\min}(n) + T_{\max}(n)$

$b = C_3 + C_4 + C_7 + C_8, c = (C_5 + C_6 + C_9 + C_{10} + C_{11})$

$T(n, k) = a + b \cdot k + c \cdot n + T_{\min}(n) + T_{\max}(n)$

 Reciclado com embalagem da Tetra Pak


PROTEGE
Tetra Pak®
O QUE É BOM

Figura 3.19: Distribution Sort cálculo do tempo analítico 1

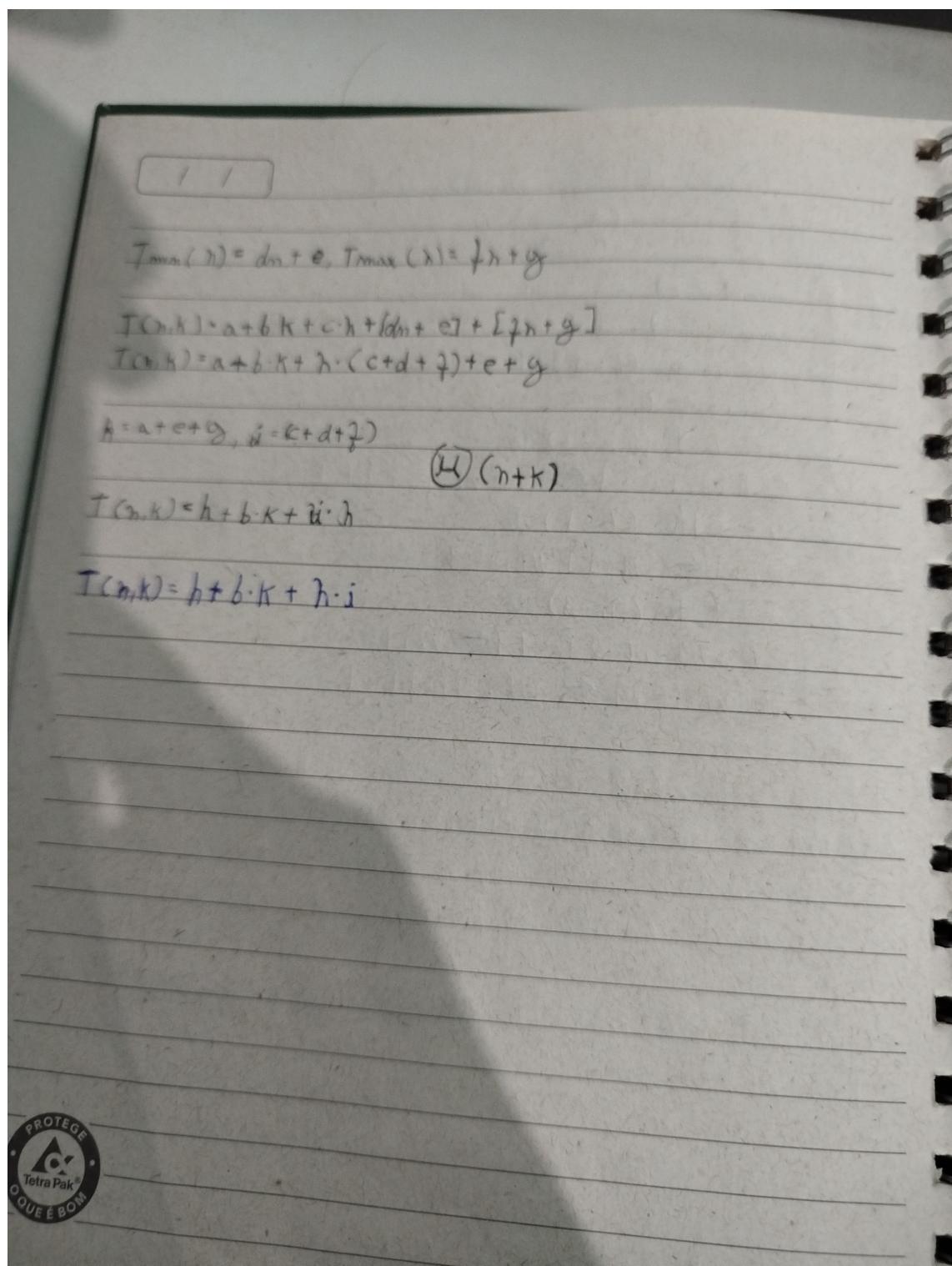


Figura 3.20: Distribution Sort cálculo do tempo analítico 2

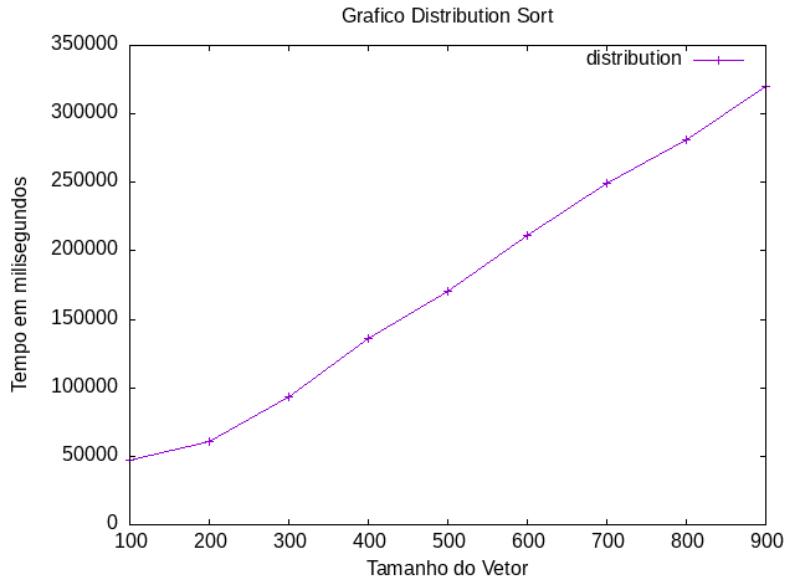


Figura 3.21: Distribution Sort gráfico de tempo de execução

3.2 Discussão dos Resultados

Com base em todos esses dados, é possível fazer o comparativo entre os algoritmos. De maneira geral, os algoritmos foram organizados em ordem dos piores para os melhores. A análise dos gráficos facilita a visualização e oferece ideia do crescimento escalar de cada um dos algoritmos. É possível perceber que o tempo de execução dos algoritmos é quase proporcional ao seu custo de memória, com exceção do merge-sort. Abaixo, uma revisão geral sobre o tempo de execução e memória de cada algoritmo

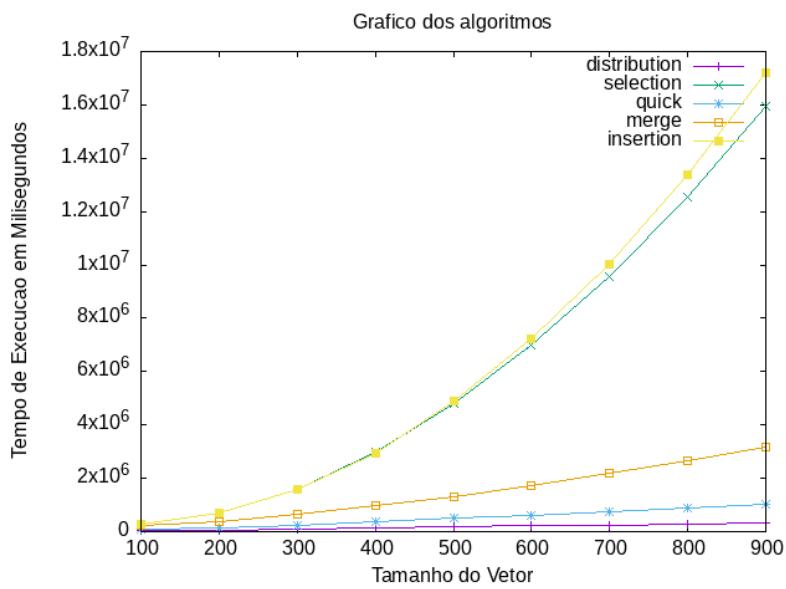


Figura 3.22: Gráfico de todos os algoritmos em seus tempos médios

Selection Sort

O algoritmo mais simples de todos, funciona na ideia de percorrer o vetor por inteiro, sendo assim o mais fácil de ser implementado ao mesmo tempo que é o pior de todos em tempo de execução, por escolar em n^2 independente dos termos no vetor. O seu custo de memória é simples por usar estruturas básicas de repetição e comparação.

Insertion Sort

Um pouco mais sofisticado que o selection-sort, ele não percorre o vetor duas vezes o tempo inteiro, fazendo esse tipo de repetição dupla somente quando encontrado valores que sejam maiores que o índice atual, ele se mostra um pouco melhor em tempo de execução. Seu custo de memória também se mostra menor, por usar estruturas que não se repetem constantemente. Mas depende muito dos valores no vetor para funcionar, podendo ter o mesmo tempo de execução do selection-sort em seu pior caso.

Merge Sort

Com uma velocidade bem acima dos algoritmos anteriores, ele supera até mesmo o próximo algoritmo dessa seção. O merge-sort funciona com a criação de sub-vetores, organizando-os e os montando. Devido a sua alta criação de sub-vetores e de estruturas de retorno constantes para isso, seu custo em memória é um dos mais altos dos algoritmos aqui estudados, não sendo indicado para usar em máquinas mais antigas. Contudo, sua velocidade compensa.

Quick Sort

Também rápido, o quick-sort utiliza de uma ideia parecida do merge-sort. Mas ao invés de criar sub-vetores para juntá-los, ele organiza sub-vetores dentro do vetor original, separando-os com base em um pivô, que é considerado o elemento central da lista. Não é tão rápido quanto o merge-sort mas utiliza eficientemente memória, podendo ser implementado em máquinas menos potentes. Isso se deve ao fato de todas as operações serem feitas em um mesmo vetor o tempo inteiro.

Distribution Sort

O algoritmo melhor custo-benefício em todos os sentidos. O distribution-sort mostrou que com uma lógica sofisticada é possível organizar vetores de maneira muito simples. Com suas estruturas de repetição sozinhas, e através de contas e organização de vetor e índice, o seu tempo de execução linear mostrou-se não só rápido como eficiente, tendo o custo de memória mais baixo de toda a lista, por usar estruturas de repetição simples e sem dupla repetitividade.

4. Conclusões

Neste trabalho foram analisados algoritmos de ordenação com Python. Servindo como experiência essencial para se entender a importância da análise analítica e teste da teoria sobre a criação de algoritmos eficientes. Com base nos resultados, foi possível perceber a evolução e destaque dos algoritmos em diferentes quantidades e situações. Mostrando que cabe ao programador decidir qual o melhor para se implementar quando estiver fazendo o seu código, embora haja sempre opções que podem ser melhores.

Por fim, o uso de ferramentas e adaptação dos códigos originais em C para Python foram grandes experiências que demonstram a versatilidade das linguagens de programação e os progressos e regressos feitos na produção dessas ferramentas, que podem facilitar a produção dos algoritmos ao mesmo tempo que dificultam o processamento das máquinas.