UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE ENSINO SUPERIOR DO SERIDÓ
DEPARTAMENTO DE COMPUTAÇÃO E TECNOLOGIA
BACHARELADO EM SISTEMAS DE INFORMAÇÃO

Trabalho da Primeira Unidade: Análise de tempo de execução de algoritmos de ordenação

Luiz Miguel Santos Silva

Caicó - RN
Abril de 2024

Luiz Miguel Santos Silva

Trabalho da Primeira Unidade: Análise de tempo de execução de algoritmos de ordenação

Relatório técnico apresentado ao curso de Bacharelado em Sistemas de Informação do Departamento de Computação e Tecnologia da Universidade Federal do Rio Grande do Norte como requisito parcial para a obtenção de nota na disciplina de Estrutura de Dados.

Orientador(a): Prof.Dr João Paulo de Souza Medeiros.

Caicó - RN
Abril de 2024

RESUMO

Este trabalho tem como objetivo apresentar uma análise dos tempos de execução dos algoritmos de ordenação: distribution-sort(), insertion-sort(), merge-sort(), quick-sort() e selection-sort().

Palavras-chave:Análises, Algoritmos, Tempos de execuções, Ordenação.

ABSTRACT

This work aims to present an analysis of the execution times of the sorting algorithms: distribution-sort(), insertion-sort(), merge-sort(), quick-sort() and selection-sort().

Keywords: Analysis, Algorithms, Execution Times, Sorting.

SUMÁRIO

LISTA DE FIGURAS

1 INTRODUÇÃO

Trabalho da disciplina de Estrutura de Dados, referente à primeira unidade. Este trabalho se pauta no estudo dos temas e algoritmos discutidos em sala de aula, com objetivo de fornecer uma análise detalhada dos mesmos.

2 DISTRIBUTION - SORT

O algoritmo Distribution-Sort é uma abordagem eficiente para ordenar um conjunto de números inteiros não negativos de forma linear. Ele opera dividindo os elementos em vários arrays auxiliares com base em seus valores e, em seguida, ordenando cada um desses arrays individualmente. Após a ordenação dentro dos arrays, os elementos são combinados para formar a sequência ordenada final. Contudo, para vetores que contém valores muito altos o algoritmo pode exigir muito espaço na memória para a criação dos arrays auxiliares.

```python
def distributionSort(vetor: list[int], n):
    maiorElemento = max(vetor)
    vetorAux = [0] * (maiorElemento + 1)
    vetorResultante = [0] * n

    for i in range(n):
        vetorAux[vetor[i]] += 1

    for i in range(1, maiorElemento + 1):
        vetorAux[i] += vetorAux[i - 1]

    for i in range(n - 1, -1, -1):
        vetorResultante[vetorAux[vetor[i]] - 1] = vetor[i]
        vetorAux[vetor[i]] -= 1

    for i in range(n):
        vetor[i] = vetorResultante[i]
```

Figura 1 – Distribution-Sort feito em Python

2.1 ANÁLISE DE TEMPO DE EXECUÇÃO DO ALGORITMO

max → PERCORRE O VETOR EM BUSCA DO MAIOR ELEMENTO = LINEAR

$$T(n) = C_1 + an + b + C_1 + C_3 + nC_4 + (n-1)C_5 + KC_6 + (K-1)C_7 + nC_8 + (n-1)((C_9 + C_{10})) + nC_{11} + (n-1)C_{12}$$

$$T(n) = n(C_4 + C_5 + C_8 + C_9 + C_{10} + C_{11} + C_{12}) + K(C_6 + C_7) + (C_1 + C_2 + C_3 - C_4 - C_7 - C_9 - C_{10} - C_{12}) + an + b$$

$$T(n) = nd + Kg + h + an + b$$

$$T(n) = nj + Kg + l \qquad O(n+K)$$

Figura 2 – Cálculo analítico do Distribution-Sort

2.2 GRÁFICO DE TEMPO DE EXECUÇÃO DO DISTRIBUTION

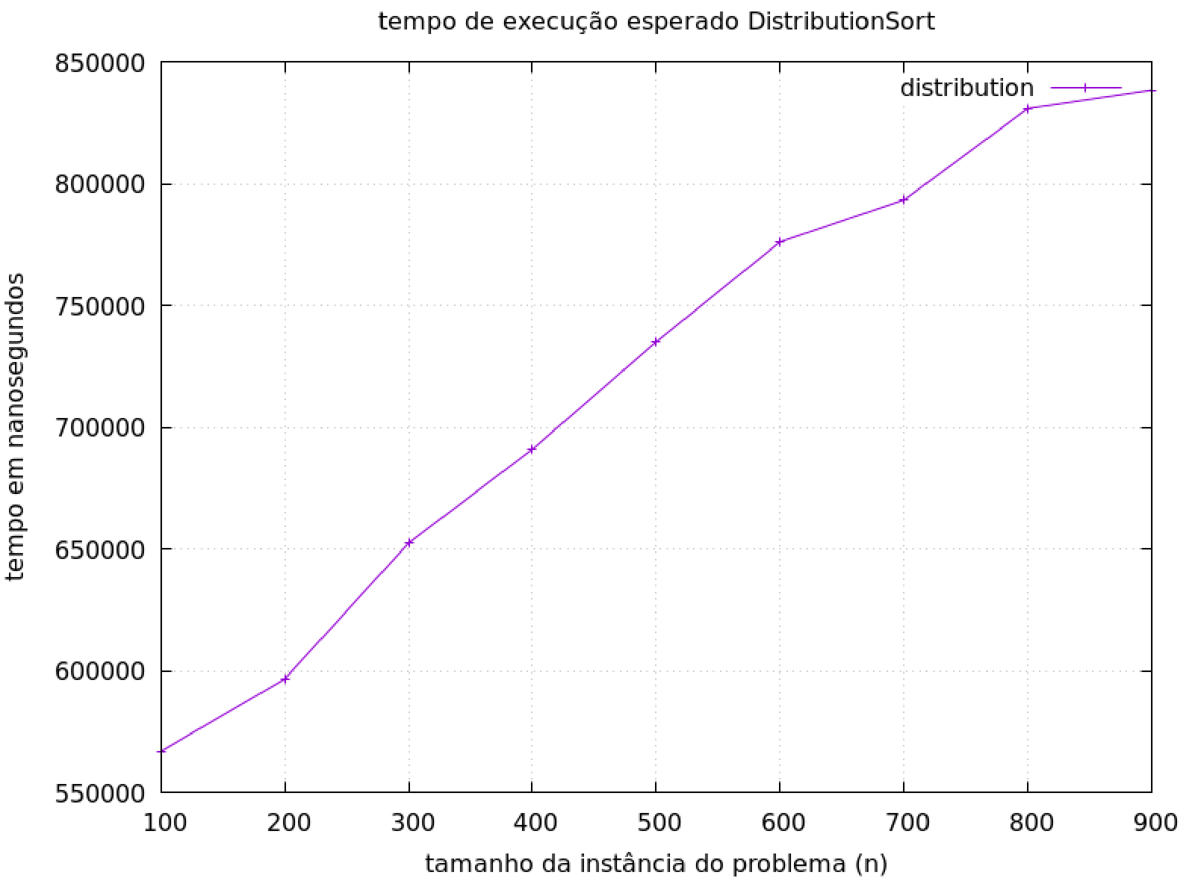

Figura 3 – Tempo de execução esperado do Distribution-Sort

2.3 COMPLEXIDADE

O Distribution-Sort ou Counting-sort não possui melhor ou pior caso, sua execução será sempre linear, podendo ter seu tempo de execução alterado mediante o tamanho do vetor. Quanto maior o tamanho do vetor ou quanto maiores forem os valores do vetor, maior será o tempo de execução do algoritmo.

2.4 ANÁLISE ASSINTÓTICA DO DISTRIBUTION-SORT

Caso	**Complexidade**
Caso Médio	$\mathcal{O}(n+k)$

2.5 ANÁLISE DE CUSTOS DO DISTRIBUTION-SORT

Como mencionado anteriomente, o algoritmo Distribution-Sort não apresenta melhor ou pior caso já que seu tempo de execução tende a ser semelhante em qualquer configuração inicial. Isso ocorre porque o método de ordenação do Distribution-Sort consiste em criar vetores auxiliares e colocar cada elemento dentro desse vetor correspondente com base em alguma característica dos dados, como seu valor numérico, por exemplo. Em seguida, os arrays são percorridos em ordem, combinando os elementos de volta em uma única sequência ordenada.

3 INSERTION - SORT

O algoritmo Inserion-sort percorre um vetor, a partir do segundo elemento, da esquerda para a direita, e compara o elemento atual com o elemento anterior, à medida que avança o algoritmo realiza trocas quando necessário para encontrar a posição correta do elemento. Esse processo se repete para cada elemento subsequente, ordenando assim todos os elementos do vetor.

```python
def insertionSort(vetor: list[int], n):
    for i in range(1, n):
        x = i
        while (x > 0) and vetor[x - 1] > vetor[x]:
            vetor[x], vetor[x - 1] = vetor[x - 1], vetor[x]
            x -= 1
```

Figura 4 – Insertion-Sort feito em Python

3.1 ANÁLISE DE TEMPO DE EXECUÇÃO DO ALGORITMO

$$T_b(n) = nC_1 + (n-1)C_2 + (n-1)C_3$$

$$T_b(n) = nC_1 + nC_2 + nC_3 + (-C_2 - C_3)$$

$$T_b(n) = an - b \qquad O(n)$$

$$T_w(n) = nC_1 + (n-1)C_2 + C_3\left(\sum_{i=2}^{n} i\right) + (C_4+C_5)\left(\sum_{i=1}^{n-1} i\right)$$

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

$$T_w(n) = n(C_1+C_2) + (C_3-C_2) + \frac{n^2C_3}{2} + \frac{nC_3}{2} - C_3 + \frac{n^2(C_4+C_5)}{2} - n\left(\frac{C_4+C_5}{2}\right)$$

$$T_w(n) = n^2\left(\frac{C_3+C_4+C_5}{2}\right) + n\left(C_1+C_2+\frac{C_3}{2}-\frac{C_4+C_5}{2}\right) + C_2$$

$$T_w(n) = an^2 + bn + c \qquad O(n^2)$$

Figura 5 – Cálculo analítico do Insertion-Sort

3.2 GRÁFICO DE TEMPO DE EXECUÇÃO DO INSERTION

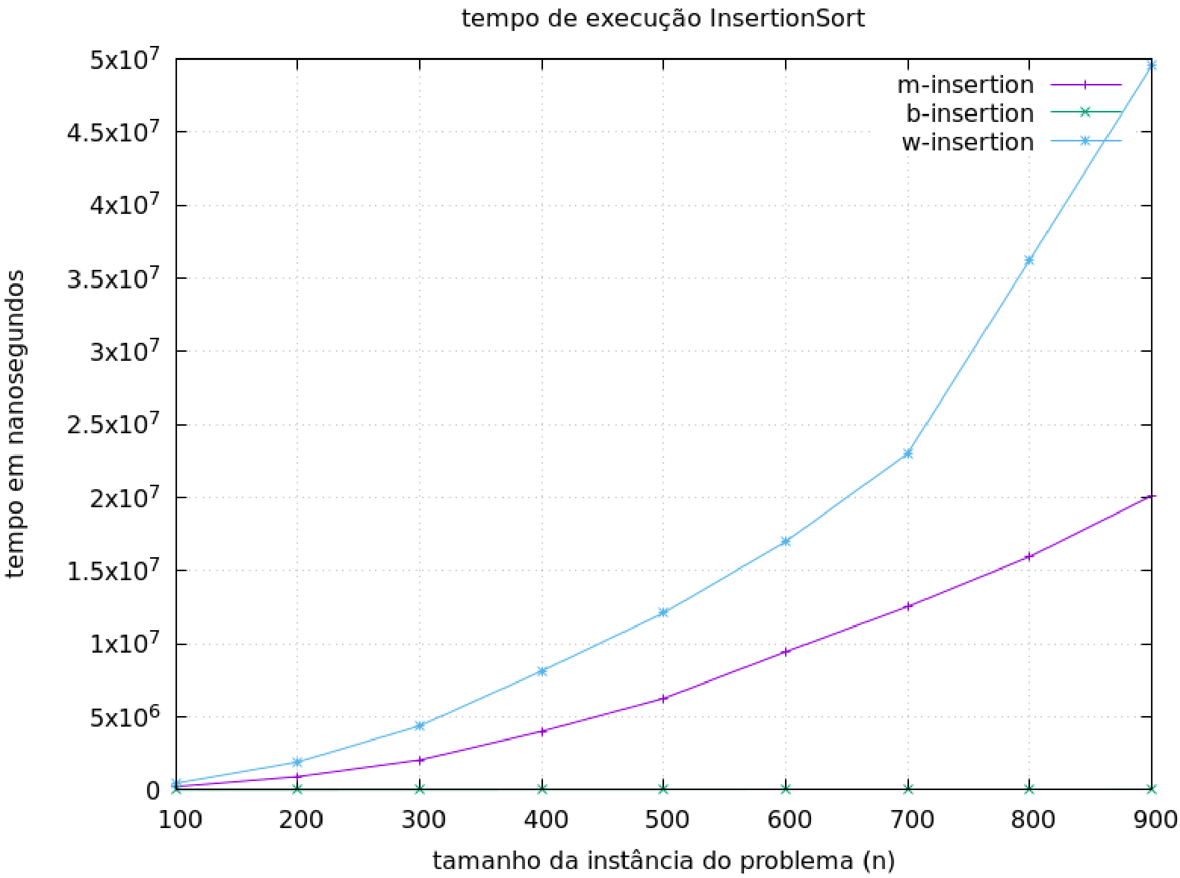

Figura 6 – Tempo de execução esperado do Insertion-Sort

3.3 COMPLEXIDADE

3.3.1 MELHOR CASO

A configuração inicial para que o Insertion-Sort execute seu melhor caso é o vetor já ordenado, já que desse modo o algoritmo só percorrerá todo o vetor sem fazer alterações, o que caracteriza o tempo de execução linear.

3.3.2 CASO MÉDIO

A configuração inicial para que o Insertion-Sort execute seu caso médio é o vetor com elementos aleatoriamente ordenados, desse modo o algoritmo percorrerá todo o vetor e cada elemento, em média, será comparado com metade dos elementos anteriores até ser inserido na posição correta. Nesse cenário, o tempo de execução esperado do algoritmo é quadrático.

3.3.3 PIOR CASO

A configuração inicial para que o Insertion-Sort execute seu pior caso é o vetor ordenado em ordem decrescente, já que desse modo o algoritmo percorrerá todo o vetor realizando todas as trocas, e consequentemente, entrando sempre nas duas estruturas de repetição e executando-as 'n' vezes, o que caracteriza o tempo de execução quadrático.

3.4 ANÁLISE ASSINTÓTICA DO INSERTION-SORT

Caso	**Complexidade**
Melhor Caso	$\mathcal{O}(n)$
Caso Médio	$\mathcal{O}(n^2)$
Pior Caso	$\mathcal{O}(n^2)$

3.5 ANÁLISE DE CUSTOS DO INSERTION-SORT

3.5.1 MELHOR CASO

No melhor caso do Insertion seu tempo de execução esperado é linear, o custo nesse caso é extremamente inferior aos outros casos.

3.5.2 CASO MÉDIO

No caso médio do algoritmo de inserção seu tempo de execução esperado é quadrático. O custo neste caso é mais baixo do que no pior caso, porém, é consideravelmente superior ao custo do melhor caso.

3.5.3 PIOR CASO

No pior caso do algoritmo seu tempo de execução esperado é quadrático. Nessa situação, o custo é muito superior ao melhor caso e ligeiramente superior ao caso médio.

4 MERGE - SORT

No algoritmo Merge-Sort, dividimos o array ao meio até que cada sub-array contenha apenas um elemento. Após isso, o algoritmo compara os elementos em pares e mescla os sub-arrays criando um novo array já ordenado. Essa lógica se repete até que os dois sub-arrays finais se unam e formem o array final já ordenado.

```python
def merge(v: list[int], s: int, m: int, e: int) -> list[int]:
    i = s
    j = m + 1
    w: list[int] = []
    for k in range(0, (e - s + 1)):
        if (j > e) or ((i <= m) and (v[i] < v[j])):
            w.append(v[i])
            i += 1
        else:
            w.append(v[j])
            j += 1
    for k in range(0, (e - s + 1)):
        v[(s + k)] = w[k]


def mergeSort(v: list[int], s: int, e: int) -> list[int]:
    if (s < e):
        m: int = (s + e) // 2
        mergeSort(v, s, m)
        mergeSort(v, (m + 1), e)
        merge(v, s, m, e)
```

Figura 7 – Merge-Sort feito em Python

4.1 ANÁLISE DE TEMPO DE EXECUÇÃO DO ALGORITMO

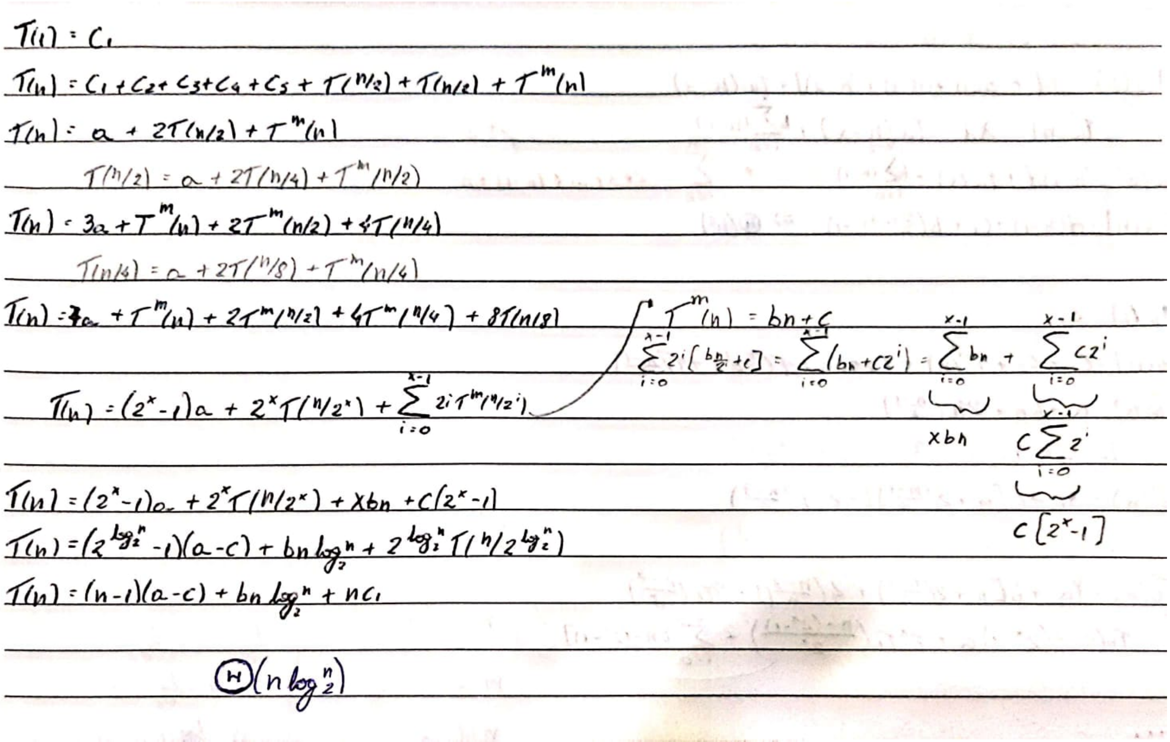

Figura 8 – Cálculo analítico do Merge-Sort

4.2 GRÁFICO DE TEMPO DE EXECUÇÃO DO MERGE

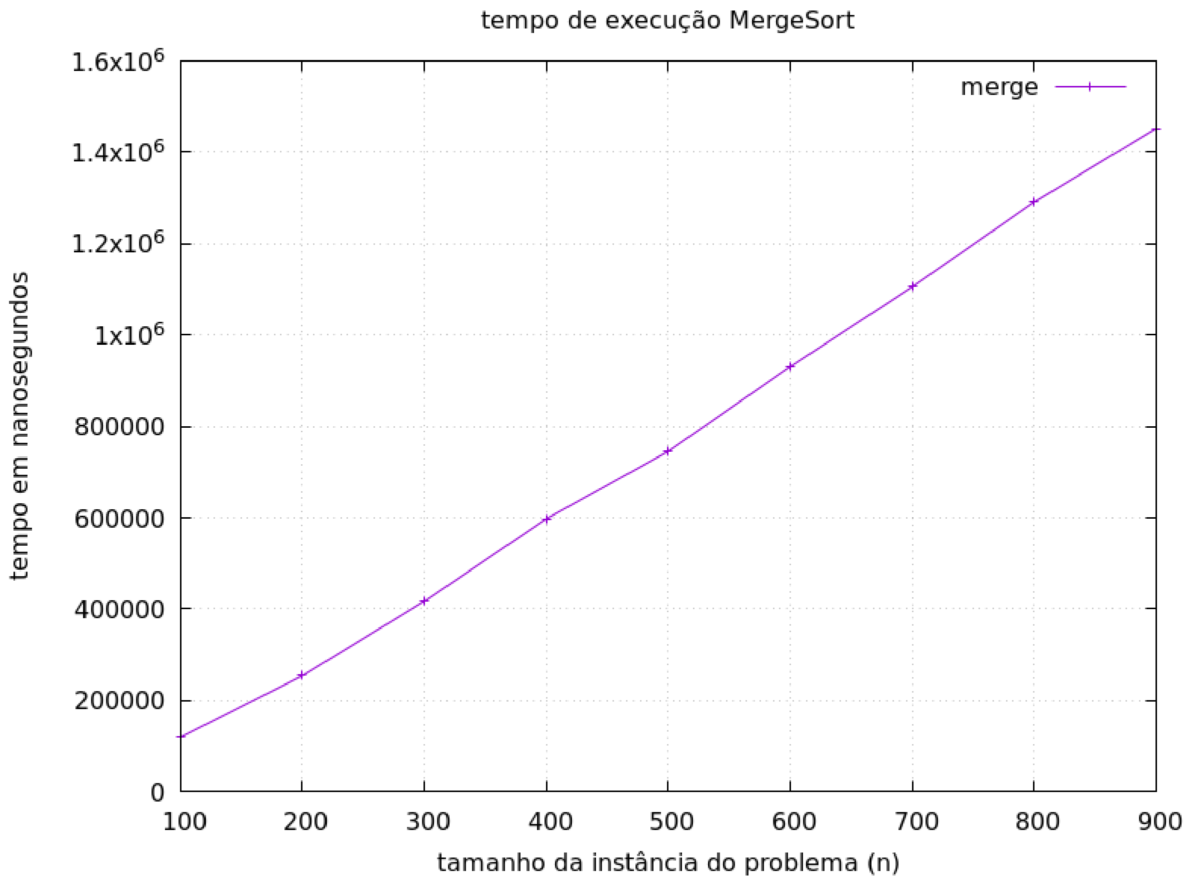

Figura 9 – Tempo de execução esperado do Merge-Sort

4.3 COMPLEXIDADE

O Merge-Sort não possui melhor ou pior caso, sua execução será sempre polilogarítmica. Independente da configuração inicial do vetor, o algoritmo segue a mesma lógica de dividir recursivamente a lista em duas metades até que cada sublista contenha um único elemento. E em seguida as sublistas são repetidamente combinadas de volta em listas maiores de maneira ordenada.

4.4 ANÁLISE ASSINTÓTICA DO MERGE-SORT

Caso	**Complexidade**
Caso Médio	$\mathcal{O}(n.\log n)$

4.5 ANÁLISE DE CUSTOS DO MERGE-SORT

Como mencionado anteriormente, o algoritmo Merge-Sort não apresenta melhor ou pior caso já que seu tempo de execução tende a ser semelhante em qualquer configuração inicial. Isso ocorre devido a lógica de ordenação utilizada pelo algoritmo, a qual divide a lista ao meio até que cada sublista contenha um único elemento, o número de divisões necessárias para reduzir a lista a sublistas unitárias é $\log_2 n$, onde n é o número de elementos na lista. Após a divisão, ocorre o processo de junção dos arrays de forma ordenada que é uma tarefa de tempo linear. Por essa razão, o Merge-Sort trabalha com tempo de execução polilogarítmo.

5 QUICK - SORT

O algoritmo Quick-Sort se baseia na ordenação em sucessivas execuções de particionamento. Ele escolhe um pivô e o posiciona no array de forma que os elementos menores ou iguais ao pivô fiquem à sua esquerda e os maiores à sua direita. A partir do pivô, o algoritmo divide o array principal em dois sub-arrays e realiza o mesmo procedimento recursivamente até que cada sub-array seja unitário. Após isso, as sublistas ordenadas são combinadas para formar o array ordenado final.

```python
def partition(v: list[int], s: int, e: int) -> int:
    d = s - 1
    for j in range(s, e):
        if v[j] <= v[e]:
            d += 1
            v[j], v[d] = v[d], v[j]
    v[e], v[(d + 1)] = v[(d + 1)], v[e]
    return (d + 1)


def quickSort(v: list[int], s: int, e: int):
    if (s < e):
        p = partition(v, s, e)
        quickSort(v, s, (p - 1))
        quickSort(v, (p + 1), e)
```

Figura 10 – Quick-Sort feito em Python

5.1 ANÁLISE DE TEMPO DE EXECUÇÃO DO ALGORITMO

$$T_w(1) = T_w(0) = c_1$$

$$T_w(n) = c_1 + c_2 + T^p(n) + c_3 + c_4 + \underbrace{T_w(0)}_{c_1} + T_w(n-1)$$

$$T_w(n) = a + \underbrace{T^p(n)} + T_w(n-1)$$

$$T_w(n) = a + bn + c + T_w(n-1)$$

$$T_w(n) = d + bn + T_w(n-1)$$

$$T_w(n-1) = d + b(n-1) + T_w(n-2)$$

$$T_w(n) = 2d + b[n + (n-1)] + T_w(n-2)$$

$$T_w(n-2) = d + b(n-2) + T_w(n-3)$$

$$T_w(n) = 3d + b[n + (n-1) + (n-2)] + T_w(n-3)$$

$$T_w(n) = xd + T_w(n-x) + b\sum_{i=0}^{x-1}(n-i)$$

$$T_w(n) = (n-1)d + T_w(1) + b\sum_{i=0}^{n-2}(n-i) \longrightarrow \sum_{i=2}^{n} i = 2 + 3 + \ldots + (n-1) + n \rightarrow \frac{n}{2}(n+1) - 1$$

$$T_w(n) = d(n-1) + c_1 + b\left(\frac{n}{2}(n+1) - 1\right) \Rightarrow \Theta(n^2)$$

$$T_b(1) = c_1$$

$$T_b(n) = c_1 + c_2 + T^p(n) + c_3 + c_4 + T\left(\frac{n-1}{2}\right) + T\left(\frac{n-1}{2}\right)$$

$$T_b(n) = a + bn + 2T_b\left(\frac{n-1}{2}\right)$$

$$T_b\left(\frac{n-1}{2}\right) = a + b\left(\frac{n-1}{2}\right) + 2T_b\left(\frac{n-3}{4}\right)$$

$$T_b(n) = 3a + b\left[n + 2\left(\frac{n-1}{2}\right)\right] + 4T_b\left(\frac{n-3}{4}\right)$$

$$T_b\left(\frac{n-3}{4}\right) = a + b\left(\frac{n-3}{4}\right) + 2T_b\left(\frac{n-7}{8}\right)$$

$$T_b(n) = 7a + b\left[n + 2\left(\frac{n-1}{2}\right) + 4\left(\frac{n-3}{4}\right)\right] + 8T_b\left(\frac{n-7}{8}\right)$$

$$T_b(n) = (2^x - 1)a + 2^x T_b\left(\frac{n-(2^x-1)}{2^x}\right) + \sum_{i=0}^{x-1}[n-(2^i-1)]$$

$$\sum_{i=0}^{x-1}(n-(2^i-1)) = n + (n-1) + (n-3) + (n-7) \ldots [n-(2^{x-1}-1)]$$

$$\frac{n-(2^x-1)}{2^x} = 1$$

$$n - 2^x + 1 = 2^x$$

$$n + 1 = 2^{x+1}$$

$$\log_2(n+1) = \log_2 2^{(x+1)}$$

$$\log_2 2^{x+1} = \log_2(n+1)$$

$$x + 1 = \log_2(n+1)$$

$$x = \log_2(n+1) - 1$$

...

$$\Theta(n \log_2 n)$$

Figura 11 – Cálculo analítico do Quick-Sort

5.2 GRÁFICO DE TEMPO DE EXECUÇÃO DO QUICK

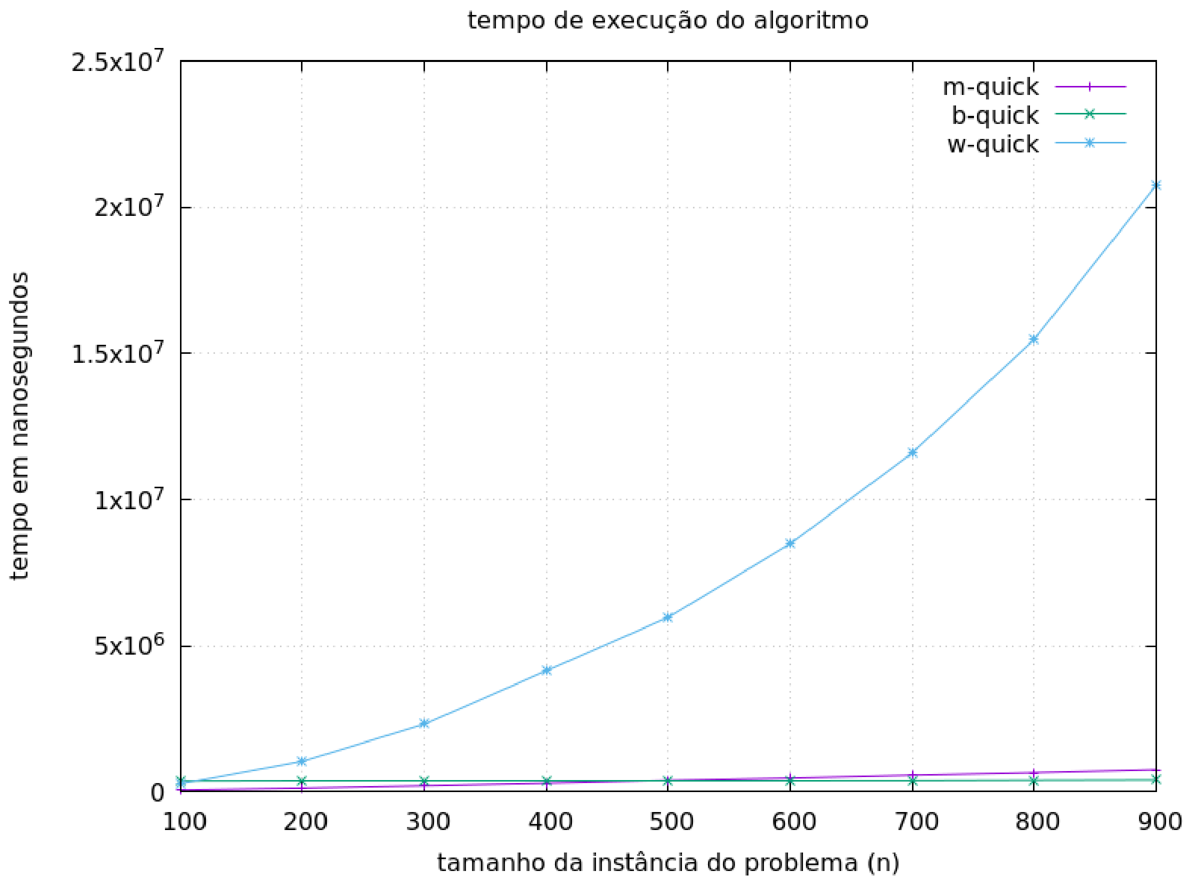

Figura 12 – Tempo de execução esperado do Quick-Sort

5.3 COMPLEXIDADE

5.3.1 MELHOR CASO

A configuração inicial para que o Quick-Sort execute seu melhor caso ocorre quando o pivô divide o array em duas partes aproximadamente iguais em cada chamada recursiva, já que desse modo o número de níveis de recursão é minimizado. Isso resulta em uma árvore de recursão balanceada, onde a altura da árvore é $\log_2 n$, sendo n o número de elementos do array.

5.3.2 CASO MÉDIO

A configuração inicial para que o Quick-Sort execute seu caso médio é o vetor com elementos aleatoriamente ordenados. Nesse cenário, o tempo de execução esperado do algoritmo é ligeiramente superior ao seu melhor caso.

5.3.3 PIOR CASO

A configuração inicial para que o Quick-Sort execute seu pior caso é o vetor já ordenado em qualquer sentido, já que desse modo o pivô escolhido vai ser sempre o menor ou o maior elemento do vetor, isso faz com que o algoritmo divida o array inicial em uma parte com todos os elementos com excessão do pivô e outra parte vazia, o que gera recursões excessivas, impactando assim, negativamente no desempenho do algoritmo.

5.4 ANÁLISE ASSINTÓTICA DO QUICK-SORT

Caso	**Complexidade**
Melhor Caso	$\mathcal{O}(n.\log n)$
Caso Médio	$\mathcal{O}(n.\log n)$
Pior Caso	$\mathcal{O}(n^2)$

5.5 ANÁLISE DE CUSTOS DO QUICK-SORT

5.5.1 MELHOR CASO

Como mencionado anteriormente, o cenário ideal para o algoritmo Quick-Sort ocorre quando o pivô divide o array em duas partes aproximadamente iguais em cada chamada recursiva. Nessa situação, o custo de execução é o mais baixo entre os possíveis casos, sendo ele $n. \log n$.

5.5.2 CASO MÉDIO

A configuração inicial para que o Quick-Sort execute seu caso médio é o vetor com elementos aleatoriamente ordenados. Nesse cenário, ocorre uma alternância entre particionamentos bons e ruins. Um particionamento é considerado bom quando o pivô está próximo do meio do array, enquanto um particionamento ruim ocorre quando o pivô está longe do meio. Nestes casos, o custo computacional do algoritmo é ligeiramente superior ao seu melhor caso.

5.5.3 PIOR CASO

A configuração inicial para que o Quick-Sort execute seu pior caso é o vetor ordenado em qualquer sentido, já que desse modo o pivô escolhido vai ser sempre o menor ou o maior elemento do vetor. Neste caso, o custo de execução aumenta de forma quadrática, tornando-se consideravelmente superior aos outros dois casos.

6 SELECTION - SORT

O algoritmo Selection-Sort utiliza uma lógica simples e intuitiva para ordenar um conjunto de dados. Ele opera selecionando o elemento com o menor valor do vetor e colocando-o na primeira posição, depois ele procura pelo segundo menor item e o coloca na segunda posição, e assim sucessivamente até que todos os elementos estejam ordenados.

```python
def selectionSort(vetor: list[int], n: int):
    for i in range(0, (n - 1)):
        m = i
        for j in range(i + 1, n):
            if vetor[j] < vetor[m]:
                m = j
        vetor[i], vetor[m] = vetor[m], vetor[i]
```

Figura 13 – Selection-Sort feito em Python

6.1 ANÁLISE DE TEMPO DE EXECUÇÃO DO ALGORITMO

$$T(n) = nc_1 + (n-1)c_2 + c_3\left[\frac{n}{2}(n+1) - 1\right] + c_4\left[\frac{n}{2}(n+1) - n\right] + (n-1)c_6$$

$$T(n) = (c_3 + c_4)\left(\frac{n^2 - n}{2}\right) + (c_2 + c_6)(n-1) + nc_1$$

$$T(n) = an^2 + bn + c \qquad O(n^2)$$

Figura 14 – Cálculo analítico do Selection-Sort

6.2 GRÁFICO DE TEMPO DE EXECUÇÃO DO SELECTION

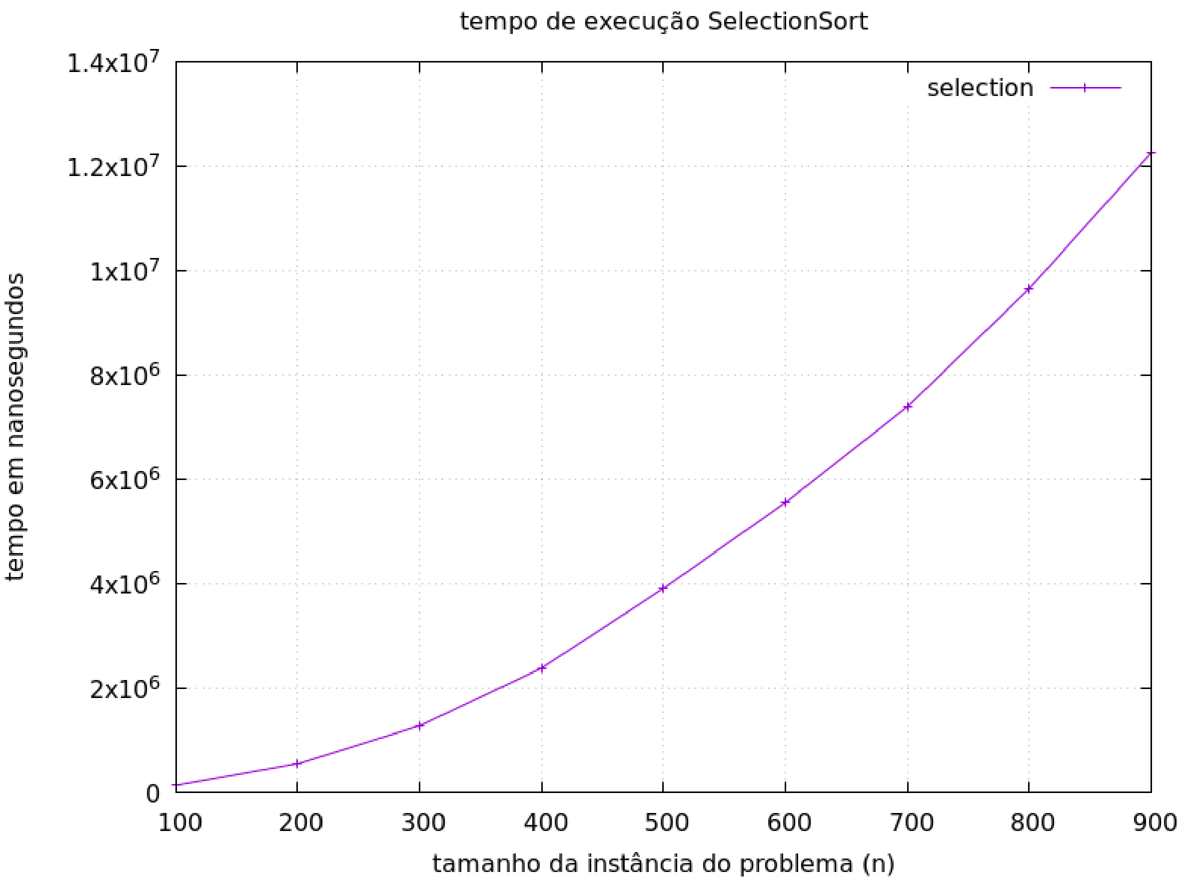

Figura 15 – Tempo de execução esperado do Selection-Sort

6.3 COMPLEXIDADE

O Selection-Sort é um algoritmo de ordenação simples que possui um comportamento de tempo de execução previsível, tanto no melhor quanto no pior caso. Ele sempre realiza uma quantidade fixa de operações comparativas e trocas, resultando em um tempo de execução quadrático. Isso significa que, independentemente da disposição inicial dos elementos, o tempo de execução do Selection-Sort cresce quadráticamente com o aumento do tamanho do vetor.

6.4 ANÁLISE ASSINTÓTICA DO SELECTION-SORT

Caso	**Complexidade**
Caso Médio	$\mathcal{O}(n^2)$

6.5 ANÁLISE DE CUSTOS DO SELECTION-SORT

O custo do Selection-Sort independente do caso em questão será sempre o mesmo. Isso ocorre devido a lógica de ordenação do algoritmo, que se baseia em realizar uma quantidade fixa de operações comparativas e trocas, resultando em um tempo de execução esperado $\mathcal{O}(n^2)$, onde n é o número de elementos do array. Isso significa que, independentemente da disposição inicial dos elementos, o tempo de execução do Selection-Sort cresce quadráticamente com o aumento do tamanho do vetor. O que exige um custo significativo de processamento e tempo, especialmente para vetores muito grandes.

7 CONCLUSÃO DA ANÁLISE

O gráfico abaixo exibe uma comparação dos tempos médios de execução dos algoritmos de ordenação citados no trabalho. A partir dele podemos concluir que o Insertion-Sort apresentou o pior resultado, muito provavelmente devido à sua lógica de comparar cada elemento à, em média, metade do elementos anteriores. O Selection-Sort vem logo na sequência do Insertion como o segundo "*pior*" algoritmo analisado no trabalho. Apesar de também possuir tempo de execução esperado de ordem quadrática, a estratégia do Selection o permite ser superior ordenando vetores muitos grandes. O Merge-Sort ficou na terceira posição sendo ligeiramente inferior ao Distribution em razão de sua lógica de criar vetores auxiliares e juntá-los novamente, o que demanda um pouco a mais de tempo e processamento. O Distribution-Sort aparece logo após o Merge, na segunda posição, e apesar de possuir tempo de execução esperado $\mathcal{O}(n+k)$ ficou muito próximo do terceiro colocado e do campeão Quick-Sort que possuem tempo de execução esperado $\mathcal{O}(n.\log n)$. Porém o algoritmo deve ser usado com cautela, já que pode haver um uso excessivo de memória para vetores que contenham valores muito altos. E por último e em primeiro lugar fica o Quick-Sort, que é o mais eficiente e consequentemente um dos algoritmos de ordenação mais utilizados no mundo.

OBS: Vale ressaltar que cada um dos algoritmos foram executados 900 vezes, tornando possível a criação de uma comparação gráfica confiável.

Comparação entre todos os algoritmos (casos médios)

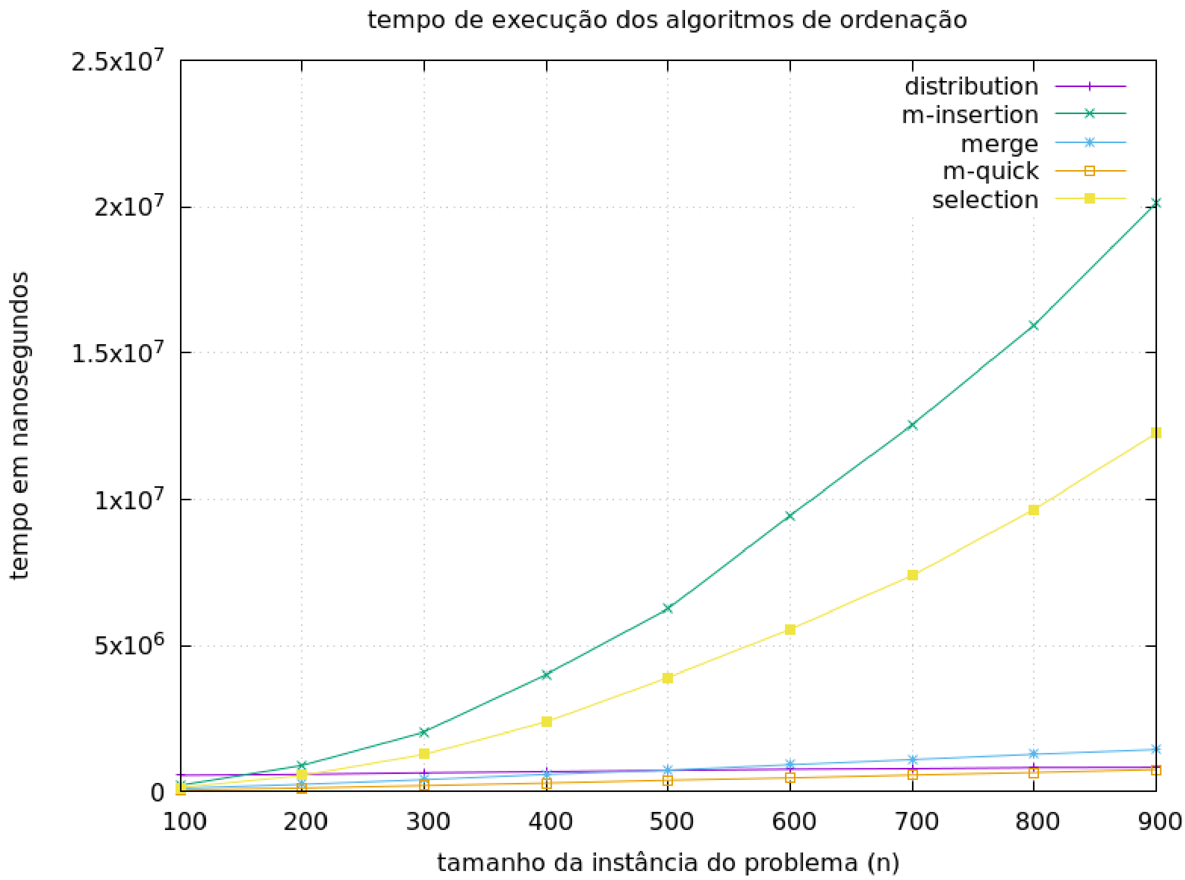

Figura 16 – Gráfico que compara o tempo de execução esperado de todos os algoritmos analisados

Classificação	**Algoritmo**	**Tempo de Execução Esperado**
1°	Quick	$\mathcal{O}(n \log n)$
2°	Distribution	$\mathcal{O}(n + k)$
3°	Merge	$\mathcal{O}(n \log n)$
4°	Selection	$\mathcal{O}(n^2)$
5°	Insertion	$\mathcal{O}(n^2)$

8 APÊNDICE

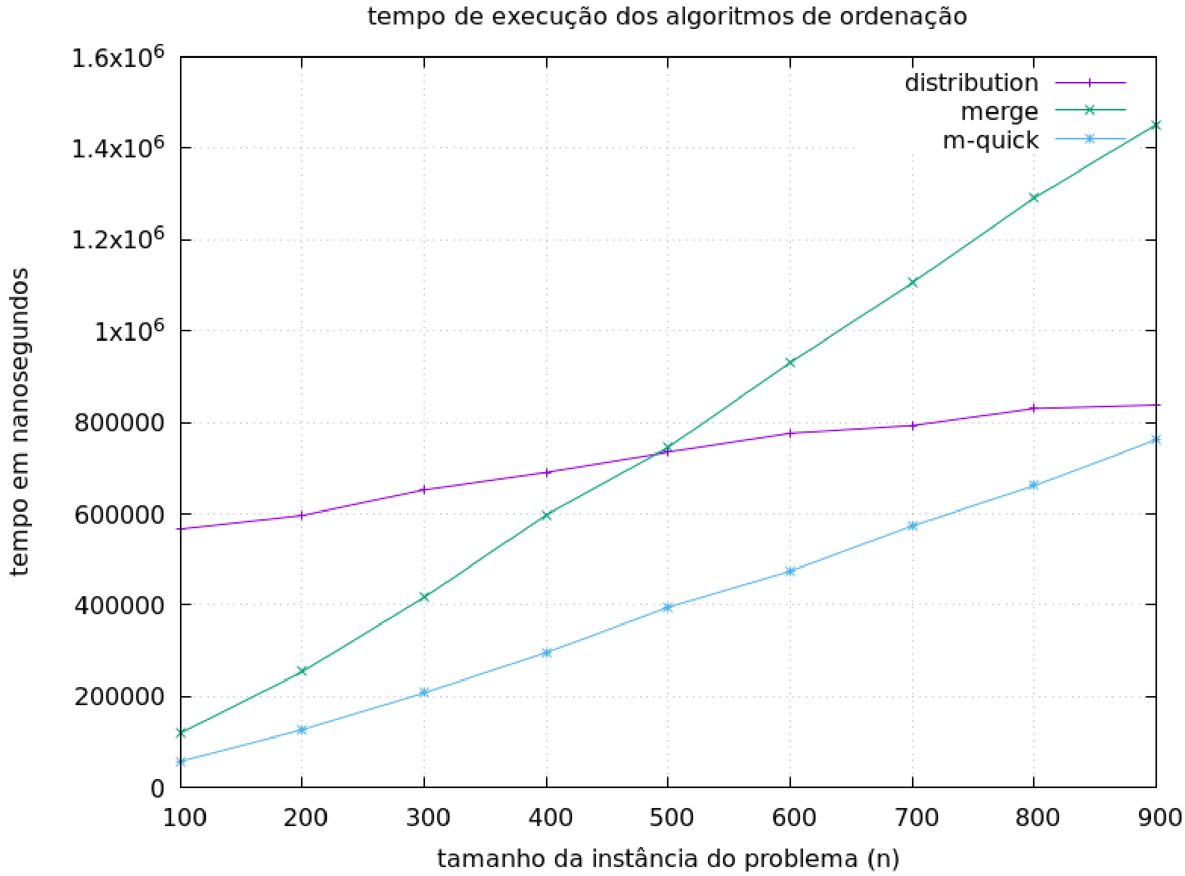

Figura 17 – Gráfico que compara os três melhores algoritmos analisados