Algoritmos de Ordenação: Um estudo comparativo da eficiência entre os algoritmos de ordenação: Insertion Sort, Merge Sort e Quick Sort

Arthur Anthony da Cunha Romão E Silva

[1]Departamento de Computação e Tecnologia
Universidade Federal do Rio Grande do Norte (UFRN)
Rua Joaquim Gregório, S/N – 59.300-000 – Caicó – RN – Brasil

arthuranthony@ufrn.edu.br

Resumo. *O presente trabalho tem como objetivo abordar a análise da complexidade dos algoritmos de ordenação: Insertion Sort, Merge Sort e Quick Sort em relação ao crescimento das instâncias. Isto foi feito por meio analítico e prático, na prática foi realizado um experimento utilizando a linguagem de programação Python, salvando as soluções em arquivos texto e gerando os gráficos. Também foi realizado o cálculo de complexidade destes algoritmos de ordenação, melhor caso, caso médio e pior caso. Este trabalho é parte da avaliação da primeira unidade da disciplina de Estrutura de Dados ministrada pelo professor João Paulo durante o período suplementar 2020.6.*

1. Introdução

Um algoritmo é um conjunto de etapas bem definidas para se atingir um objetivo. Portanto o conceito de algoritmo se estende além da computação, por exemplo, atravessar a rua e fazer café são objetivos que para realiza-los é necessário um conjunto de passos bem definidos. Em computação um algoritmo é um conjunto de procedimentos computacionais finitos e não ambíguos, que recebe uma instância ou entrada de dados, processa a instância e retorna algo como saída. [Szwarcfiter and Markenzon 1994].

É utilizado para se resolver algum problema computacional, por exemplo, encontrar um determinado elemento em um vetor, remover elementos repetidos em um vetor, ou ordenar um conjunto de dados por meio de permutações com a finalidade de retorna-los de modo ordenado, que será o tema abordado neste trabalho.

O processo de desenvolvimento de um algoritmo leva algumas etapas, pois é necessário realizar algumas perguntas sobre o problema computacional em mente, para que se utilize da melhor estratégia para soluciona-lo. Para isto nós realizamos o projeto de um algoritmo que segue algumas etapas, como é apresentado na figura abaixo.

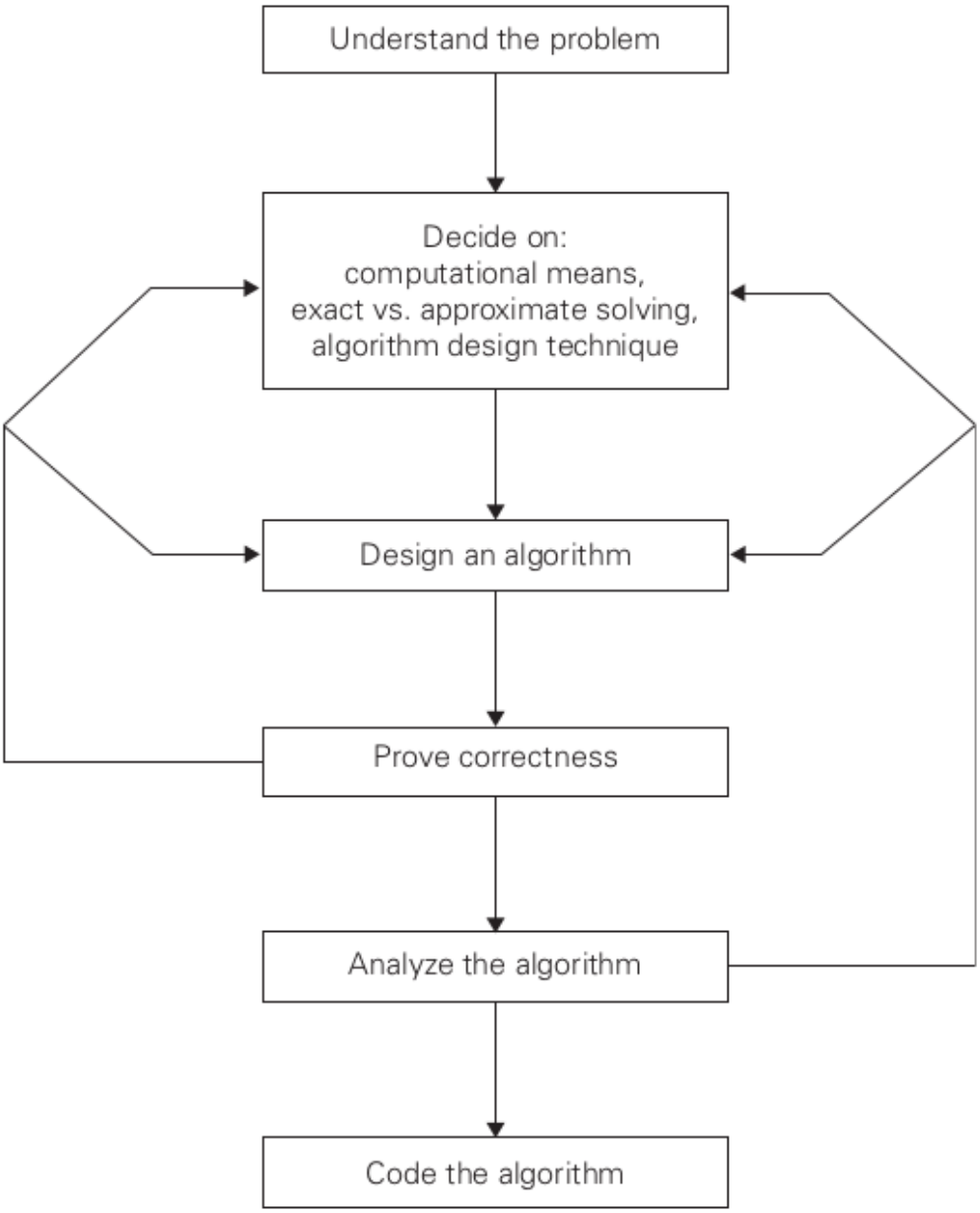


Figura 1. Etapas que são necessárias durante a elaboração do projeto de um algoritmo
Fonte: [Levitin 2012]

Primeiramente é necessário entender o problema computacional em questão, de modo que o problema abstrato seja convertido em procedimentos de entrada, processamento e saída de dados. É necessário ao entender o problema computacional, verificar se o mesmo pode ser resolvido utilizando uma solução exata ou aproximada.

Na litetura nós temos problemas computacionais comuns que são ensinados em Cursos da Computação, como por exemplo busca binária, busca utilizando estruturas de dados, como árvore, ordenação de vetores, problemas envolvendo grafos. Porém há problemas que a solução exata não é possível, sua complexidade torna isto inviável, como os problemas NP-Arduo, que neste caso é utilizado metaheurísticas, como por exemplo algoritmos estocásticos (algoritmo genético, otimização por enxame de partículas, dentre outros), que é o caso do famoso problema do caixeiro viajante. [Levitin 2012]

É necessário realizar a prova de corretude, se porventura o algoritmo proposto é correto, dado uma instância, retorne a saída pretendida, e incorreto, se para uma determinada instância, a saída não seja como o esperado. [Cormen et al. 2009] Após nós chegamos ao passo em que abordaremos neste trabalho, a análise do algoritmo. É realizado cálculos que estimam o consumo de tempo para que o algoritmo retorne a saída esperada, isto é, a complexidade do algoritmo. Podemos possuir um algoritmo eficaz para um resolver um problema computacional, porém ineficiente, propõe resolver o problema, mas de maneira ineficiente. No âmbito da ordenação de dados temos algoritmos de ordem quadrática, ordem linear, ordem logarítmica, dentre outros. E por fim a implementação do algoritmo, que o passo em que é codificado, pressupondo que as melhores estratégias foram utilizados, e sua modelagem é correta.

Sendo assim, é de suma importância realizar o projeto de um algoritmo, para que se tenha uma certeza e um viés positivo para que seja resolvido o problema computacional de maneira inteligente e eficiente, utilizando-se dos recursos disponíveis na litetura. Portanto neste trabalho será abordado especificadamente o tópico da análise de complexidade de três algoritmos de ordenação: Insertion Sort, Merge Sort e Quick Sort por meio de um experimento prático e um meio analítico.

2. Algoritmo de Ordenação

Em computação, um algoritmo de ordenação é uma solução para um problema computacional que dado uma instância, seja esta númerica ou lexicográfica, deve ser retornado como saída os valores desta instância de modo ordenado, seja de modo crescente ou decrescente. Os algoritmos de ordenação são utilizados frequentemente como estratégia de melhorar a instância de qualquer outro problema computacional, para se obter uma solução mais eficiente e incorporado na maioria dos sistemas, como por exemplo, a ordenação dos nomes dos alunos em uma lista de presença escolar, a ordenação dos contatos em uma lista telefônica em ordem alfabética. [Honorato 2013]

Em resumo, o problema computacional de ordenar uma conjunto de dados como apresentado na figura abaixo, consiste em uma entrada, uma ordenação de modo crescente ou decrescente ou reordenação, utilizamos em suma para beneficiar a legibilidade dos dados, passo intermediário em algoritmos de outros problemas computacionais, como buscas por elementos em uma amostra de dados quaisquer e como saída a permutação destes dados baseando-se em estratégias como o transformar para conquistar, que é reduzir o tempo de custo que aquela instância custaria caso não estivesse ordenada, a simplificação da mesma e a mudança de representação. [do Lago Pereira et al.]

Entrada: Uma sequência de n números $\langle a_1, a_2, \ldots, a_n \rangle$
Saída: Uma permutação (reordenação) de n números $\langle a'_1, a'_2, \ldots, a'_n \rangle$ da sequência de entrada, tal que $a'_1 \leq a'_2 \leq \cdots \leq a_n$.

Na literatura possuímos diversos algoritmos de ordenação, cada um projetado para uma situação especifica em que se encontram os dados ou o dispositivo que irá processar estes dados, por exemplo, se no problema computacional em questão ter como requisito o menor custo de memória possível, ou se os dados estão pre-ordenados ou uniformente distribuídos no vetor, ou se o vetor possui um conjunto extenso de dados, temos para isso uma boa solução para cada um destes casos.

Há diversas técnicas para se obter o resultado esperado, utilizando-se de várias estratégias, como: divisão e conquista, no caso do merge sort, na qual baseia-se em quebrar a instância em pequenas partes e ordenando-as, depois unindo-as de modo que seja ordenado de modo intercalado, força bruta, na qual é analisado todos os casos possíveis para se chegar ao conjunto de dados ordenados, neste caso temos como exemplo o selection sort, diminuir e conquistar, baseando-se em reduzir a instância para uma versão melhor de forma que auxilie na ordenação, tornando o problema de ordenação mais eficaz, como por exemplo o algoritmo de ordenação por inserção, input-enhancement, que utiliza-se de preprocessar a instância, com a finalidade de guardar informações adicionais, como a utilização de vetores auxiliares, de modo que acelere a busca pela solução, no caso o counting-sort, há dentre destas estratégias o uso de soluções recursivas ou iterativas. [Cormen et al. 2009]

2.1. Insertion Sort

Um dos algoritmos estudados no presente trabalho é o Insertion Sort, utiliza-se um pivô e sua comparação com os demais elementos já ordenados do vetor, e permutados de modo que o vetor seja complementado ordenado, seu funcionamento é analogamente explicado como a ordenação das cartas de baralho por uma pessoa em sua mão, como apresentado na figura abaixo, um conjunto de dados é ordenado, neste caso as cartas dois, três e quatro, e a carta sete nesta ilustração seria o pivô nesta iteração, seria permutado com todas as cartas fossem maiores que a mesma, neste caso o objetivo é ordena-las de modo crescente, isto é, do menor para o maior. Como veremos no experimento, o Insertion Sort é eficiente para instâncias pequenas, além de sua fácil implementação.

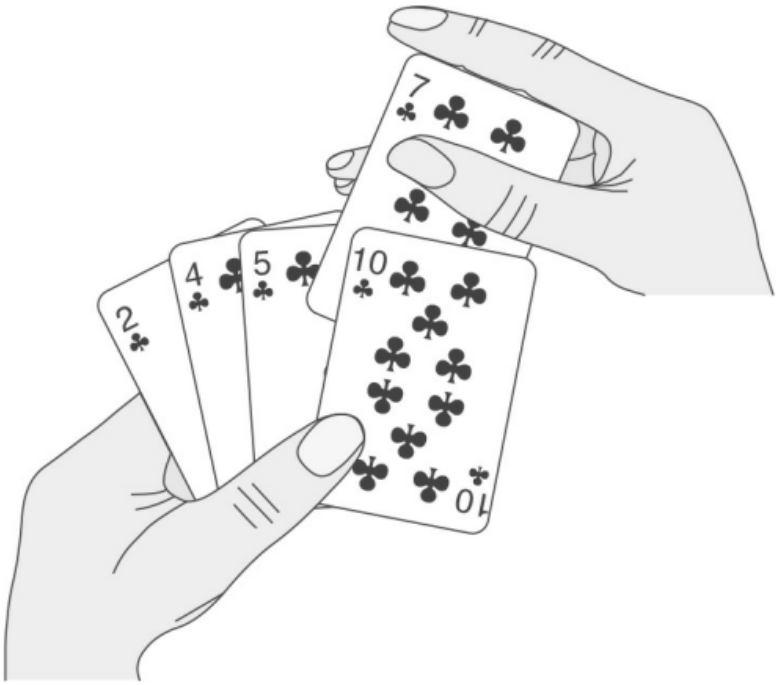


Figura 2. O processo de ordenação do algoritmo de ordenação Insertion Sort se assemelha a ordenação manual humana de cartas de baralho em sua mão.
Fonte: [Cormen et al. 2009]

Como vemos na figura abaixo, assumimos que a primeira posição está ordenada, e a próxima posição é determinada como o pivô, e para cada iteração, este pivô varre o vetor de forma que a condição de que o elemento anterior a este seja maior do que ele mesmo, ele permuta com o elemento, de maneira que mais um fragmento do vetor seja classificado como ordenado, assim realizando a estratégia, como antes comentada de diminuir e conquistar, é realizado estes procedimentos até que o vetor esteja totalmente ordenado.

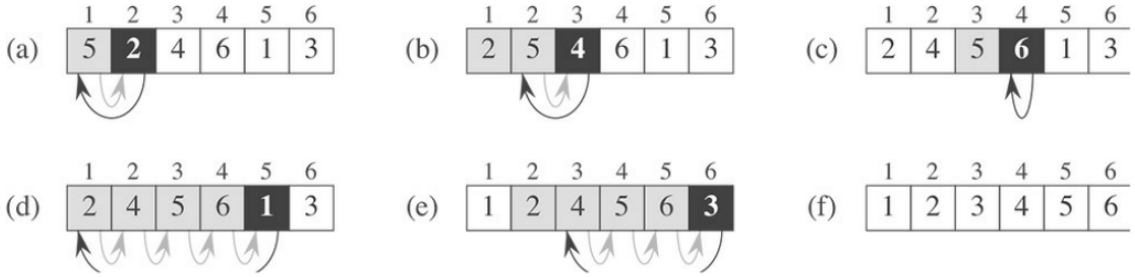


Figura 3. Processo de ordenação de um vetor por meio do algoritmo de ordenação Insertion Sort.
Fonte: [Cormen et al. 2009]

A figura abaixo ilustra o pseudocódigo do algoritmo de ordenação Insertion Sort, é utilizado duas estruturas de repetição, fácil implementação, uma boa opção para pequenas instâncias, neste presente trabalho é calculo a complexidade, que em seu melhor caso é um comportamento assintótico linear, e em seu médio e pior caso possui um comportamento assintótico quadrático.

INSERTION-SORT(A)	*cost*	*times*
1: **for** $j = 2$ **to** $A.length$	c_1	n
2: $key = A[j]$	c_2	$n-1$
3: // Insert $A[j]$ to the sorted sequence $A[1..j-1]$	0	$n-1$
4: $i = j - 1$	c_4	$n-1$
5: **while** $i > 0$ **and** $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6: $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7: $i = i - 1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
8: $A[i+1] = key$	c_8	$n-1$

Figura 4. Pseudocódigo do algoritmo de ordenação Insertion Sort.
Fonte: [Cormen et al. 2009]

2.1.1. Cálculo de Complexidade do Insertion Sort

Arthur Anthony da Cunha Romão E Silva

Insertion-Sort

$$T_b(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1) =$$

$$c_3 n + c_2 n - c_2 + c_3 n - c_3 + c_4 n - c_4 + c_7 n - c_7 =$$

$$n(c_1 + c_2 + c_3 + c_4 + c_7) + (-c_2 - c_3 - c_4 - c_7)$$

a $\quad$ b

$$\boxed{T_b(n) = an + b}$$

$$T_w(n) = c_3 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=2}^{n} i + \sum_{i=1}^{n-1} i\, c_5 + c_6 \sum_{i=1}^{n-1} i + c_7(n-1)$$

$$T_w(n) = c_3 n + c_2 n - c_2 + c_3 n - c_3 + c_7 n - c_7 + c_4 \sum_{i=2}^{n} + c_5 \sum_{i=1}^{n-1} + c_6 \sum_{i=1}^{n-1}$$

$$T_w(n) = n(c_1 + c_2 + c_3 + c_7) + (-c_2 - c_3 - c_7) + \sum_{i=1}^{n-1} i(c_5 + c_6) + \sum_{i=2}^{n} i c_4$$

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

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

$$T_w(n) = n(c_1 + c_2 + c_3 + c_7) + (-c_2 - c_3 - c_7) + (n^2+n-2n)\left(\frac{c_5}{2} + \frac{c_6}{2}\right) + c_4\left(\frac{n^2+n-2}{2}\right)$$

$$T_w(n) = n(c_1 + c_2 + c_3 + c_7) + (-c_2 - c_3 - c_7) + \frac{c_5 n^2}{2} + \frac{c_6 n^2}{2} + \frac{c_5 n}{2} + \frac{c_6 n}{2} - \frac{2c_5 n}{2} -$$

$$\frac{2c_6 n}{2} + \frac{c_4 n^2}{2} + \frac{c_4 n}{2} - \frac{2c_4}{2}$$

$$T_w(n) = n(c_1 + c_2 + c_3 + c_7) + (-c_2 - c_3 - c_7) + \frac{c_5 n^2}{2} + \frac{c_6 n^2}{2} + \frac{c_5 n}{2} + \frac{c_6 n}{2} - c_5 n - c_6 n - c_4$$

$$\frac{c_4 n^2}{2} + \frac{c_4 n}{2}$$

Arthur Anthony da Cunha Ramão E Silva

Insertion-Sort (Continuação)

$T_w(n) = \frac{1}{2}(C_5n^2 + C_6n^2 + C_4n^2) + n(C_1+C_2+C_3+\frac{C_4}{2}+C_5+C_6) + (-C_2-C_3-C_4-C_7$

$T_w(n) = n^2\underbrace{\left(\frac{C_5+C_6+C_4}{2}\right)}_{a} + n\underbrace{(C_1+C_2+C_3+\frac{C_4}{2}+C_5+C_6)}_{b} + \underbrace{(-C_2-C_3-C_4-C_7}_{c}$

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

$T_a(n) = C_1n + (C_2+C_3+C_7)(n-1) + C_4\sum_{j=2}^{n}\frac{j}{j-1} + (C_5+C_6)\sum_{j=2}^{n}\frac{j}{j-1}$

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

$T_a(n) = C_1n + (C_2+C_3+C_7)(n-1) + C_4\sum_{j=2}^{n-1}\frac{j+1}{2} + (C_5+C_6)\sum_{j=2}^{n-1}\frac{j+1}{2} =$

$C_1n + (C_2+C_3+C_7)(n-1) + C_4\left(\frac{n^2+3n-4}{4}\right) + (C_5+C_6)\left(\frac{n^2+n-2}{4}\right)$

$\sum_{j=2}^{n-1}\frac{j+1}{2} = \frac{1}{2}\sum_{j=2}^{n-1}(j+1) = \frac{1}{2}\left(\sum_{j=2}^{n-1}j + \sum_{j=2}^{n-1}1\right) = \frac{n(n+1)}{2} - 1 + (n-1) =$

$\frac{n^2+n}{2} - \frac{2}{2} + \frac{n}{1} = \frac{n^2+n+2n-4}{2} = \frac{n^2+3n-4}{2}\cdot\frac{1}{2} = \frac{n^2+3n-4}{2\cdot 2} = \frac{n^2+3n-4}{4}$

$\sum_{j=2}^{n-1}\frac{j+1}{2} = \frac{1}{2}\left(\sum_{j=2}^{n-1}j+1\right) = \frac{1}{2}\left(\sum_{j=2}^{n-1}j + \sum_{j=2}^{n-1}1\right) = \frac{1}{2}\left(\frac{n(n+1)}{2} - n + (n-1) - n\right) =$

$\frac{n^2+n}{4} - 2n + n - 1 = \frac{n^2+n-2n-2}{2} = \frac{n^2-n-2}{2}\cdot\frac{1}{2} = \frac{n^2-n-2}{4}$

Arthur Anthony da Cunha Romão E Silva

Insertion-Sort (Continuação 2)

$T_a(n) = C_1 n + (C_2 + C_3 + C_4)(n-1) + C_4\left(\frac{n^2+3n-4}{4}\right) + (C_5+C_6)\left(\frac{n^2-n-2}{4}\right)$

$T_a(n) = C_1 n + (C_2 + C_3 + C_4)(n-1) + \frac{C_4}{4}(n^2+3n-4) + \frac{1}{4}(C_5+C_6)(n^2-n-2)$

$T_d(n) = C_1 n + C_2 n - C_2 + C_3 n - C_3 + C_4 n - C_4 + \frac{C_4 n^2}{4} + \frac{3C_4 n}{4} - \frac{4C_4}{4} + \frac{C_5 n^2}{4}$

$-\frac{C_5 n}{4} - \frac{2C_5}{4} - \frac{C_6 n^2}{4} - \frac{C_6 n}{4} - \frac{2C_6}{4}$

$T_a(n) = n^2\underbrace{\left(\frac{C_4}{4} + \frac{C_5}{4} + \frac{C_6}{4}\right)}_{a} + n\underbrace{\left(C_1 + C_2 + C_3 + C_4 + \frac{3C_4}{4} + \frac{C_5}{4} + \frac{C_6}{4}\right)}_{b} + \underbrace{\left(-C_2 - C_3 - C_4 - C_7 - \frac{1}{2}C_5 - \frac{C_6}{2}\right)}_{c}$

$\boxed{T_a(n) = an^2 + bn + c}$

2.2. Merge Sort

O algoritmo de ordenação merge sort utiliza-se da estratégia dividir para conquistar, que divide o vetor em subestruturas, de modo que ordene pequenas partes, como o principio da estratégia é dividir o problema em problemas menores, sendo assim com o objetivo ordenar vetores por meio da divisão, intercalação e união dos elementos contidos no vetor. Será recursivamente dividido o vetor em pequenas partes, depois serão organizados as pequenas partes seja em ordem crescente ou decrescente, após isso os subvetores serão fundidos, ordenados por meio da intercalação, este processo ocorre até que o vetor seja totalmente ordenado.

```
algorithm  merge-sort(A, s, e)
 1 if s < e then
 2    m ← ⌊(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 ← s
 2 j ← m + 1
 3 for k ← 1 to e − s + 1 do
 4    if A[i] < A[j] and i ≤ m or j > e then
 5       B[k] ← A[i]
 6       i ← i + 1
 7    else
 8       B[k] ← A[j]
 9       j ← j + 1
10 for k ← 1 to e − s + 1 do
11    A[s + k − 1] ← B[k]
```

Figura 5. Algoritmo de Ordenação Merge Sort
Fonte: [Cormen et al. 2009]

Como apresentado acima o pseudocódigo do merge sort, sua implementação é intermediária, visto que utiliza-se de uma divisão recursiva do vetor, de modo que as partes sejam ordenadas e unidas por meio da função merge, sua complexidade em qualquer caso é n log n, todavia por ser um algoritmo recursivo e utilizar um vetor auxiliar, tende a consumir mais memória, e seu fluxo é executado mesmo se o vetor esta ordenado, em alguns casos não é indicado, com o viés da recursos, pode consumir um tempo de execução de memória não viável para o computador que irá processar, o tamanho da instância ou a configuração predefinida da instância.

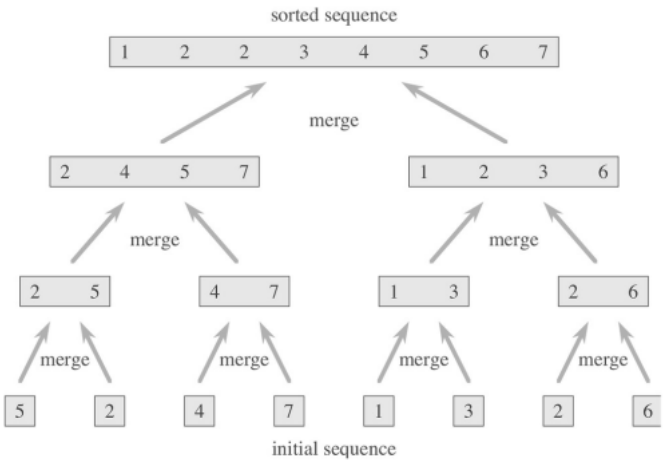


Figura 6. Processo de ordenação de um vetor por meio do Merge Sort
Fonte: [Cormen et al. 2009]

A figura acima ilustra o funcionamento do merge sort, a instância é dividida recursivamente em duas metades, estas subestruturas são ordenadas e dividas até o ponto em que não seja mais possível dividi-las, depois são intercaladas novamente e unidas gerando a configuração da instância original de forma ordenada, seja de modo crescente ou decrescente.

2.2.1. Cálculo de Complexidade do Merge Sort

Arthur Anthony da Cunha Romão E Silva

Merge Sort:

$T(1) = d \; ; \; T_m(n) = dn + b$

$T(n) = c + T_m(n) + 2T\left(\frac{n}{2}\right)$

$T\left(\frac{n}{2}\right) = c + T_m\left(\frac{n}{2}\right) + 2T\left(\frac{n}{4}\right)$

$T(n) = c + T_m(n) + 2\left[c + T_m\left(\frac{n}{2}\right) + 2T\left(\frac{n}{4}\right)\right]$

$T(n) = c + T_m(n) + 2c + 2T_m\left(\frac{n}{2}\right) + 4T\left(\frac{n}{4}\right)$

$T(n) = 3c + T_m(n) + 2T_m\left(\frac{n}{2}\right) + 4T\left(\frac{n}{2}\right)$

$T\left(\frac{n}{4}\right) = c + T_m\left(\frac{n}{4}\right) + 2T\left(\frac{n}{8}\right)$

$T(n) = 3c + T_m(n) + 2T_m\left(\frac{n}{2}\right) + 4\left[c + T_m\left(\frac{n}{4}\right) + 2T\left(\frac{n}{8}\right)\right]$

$T(n) = 7c + T_m(n) + 2T_m\left(\frac{n}{2}\right) + 4T_m\left(\frac{n}{4}\right) + 8T\left(\frac{n}{8}\right)$

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

$T(n) = \left(2^{\log_2 n} - 1\right) + 2^{\log_2 n} + T\left(\frac{n}{2^{\log_2 n}}\right) + \sum_{i=0}^{(\log_2 n)-1} 2^i T_m\left(\frac{n}{2^i}\right)$

$T(n) = (n-1)c + nT\left(\frac{n}{n}\right) + \sum_{i=0}^{(\log_2 n)-1} 2^i T_m\left(\frac{n}{2^i}\right)$

$T(n) = (n-1)c + nd + \sum_{i=0}^{(\log_2 n)-1} 2^i T_m\left(\frac{n}{2^i}\right)$

Arthur Anthony da Cunha Ramão E Silva

Merge Sort (Continuação):

$$\sum_{i=0}^{(\log_2 m)-1} 2^i \left(a\frac{m}{2^i} + b\right)$$

$$\sum_{i=0}^{(\log_2 m)-1} am + 2^i b = \sum_{i=0}^{(\log_2 m)-1} am + \sum_{i=0}^{(\log_2 m)-1} 2^i b = am \sum_{i=0}^{(\log_2 m)-1} + b \sum_{i=0}^{(\log_2 m)-1} 2^i =$$

$$am \sum_{i=0}^{(\log_2 m)-1} 1 = am \cdot \log_2 m$$

$$b \sum_{i=0}^{(\log_2 m)-1} = b(2^{\log_2 m} - 1) = b(m-1)$$

$$T(m) = (m-1)c + md + am \log_2 m + b(m-1)$$

$$T(m) = m(b+c+d) + am\log_2 m + (-b-c)$$

$$T(n) = an + \beta n \log n + \gamma$$

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

2.3. Quick Sort

O Quick Sort utiliza-se da mesma estratégia do algoritmo Merge Sort, que é o de dividir para conquistar, dividir o problema em subproblemas, para melhorar a eficiência da ordenação, sua complexidade é quadrática em seu pior caso, e n log n em seu melhor caso e caso médio. É selecionado um pivô que separará o vetor em duas subestruturas, na qual a esquerda do pivô ficaram os valores menores e a direita os maiores, o mesmo é feito nas subestruturas, de modo que o vetor seja totalmente ordenado. A figura abaixo ilustra o comportamento do pivô com o particionamento da estrutura e o funcionamento da recursão para ordenação.

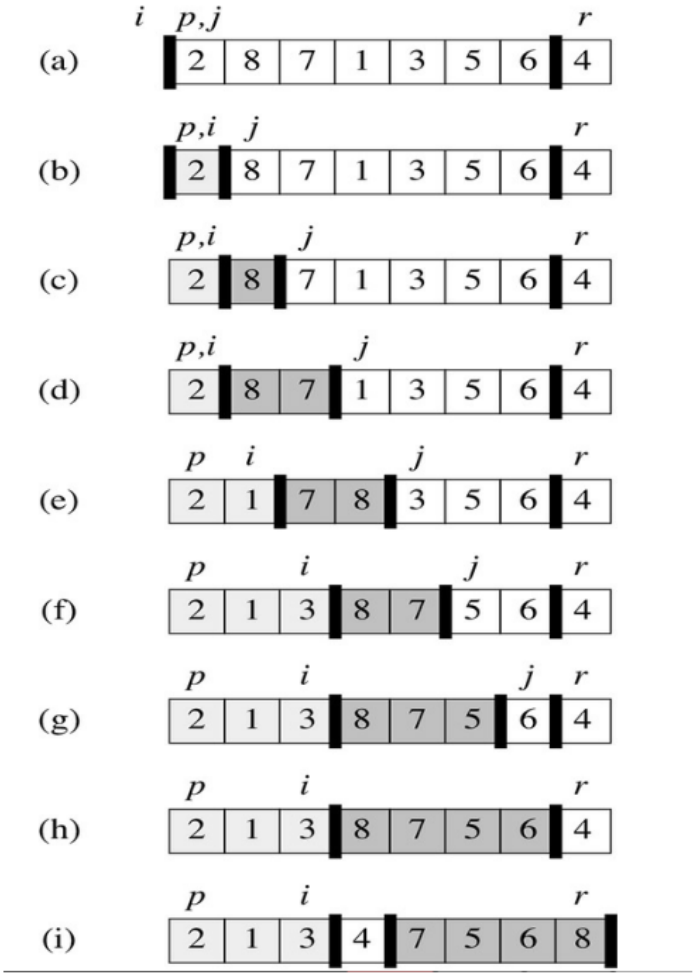


Figura 7. Processo de ordenação de um vetor por meio do Quick Sort
Fonte: [Cormen et al. 2009]

O algoritmo é de intermidiária implementação, pois utiliza-se da recursão como estratégia de ordenar as subestruturas no vetor, não utiliza-se de vetor auxiliar como acontece no Merge Sort, indicado quando o tamanho da instância é grande, superior ao Insertion Sort neste quesito.

```
algorithm  quick-sort(A, s, e)
 1 if s < e then
 2    p ← partition(A, s, e)
 3    quick-sort(A, s, p − 1)
 4    quick-sort(A, p + 1, e)
algorithm  partition(A, s, e)
 1 k ← A[e]
 2 i ← s − 1
 3 for j ← s to e − 1 do
 4    if A[j] ≤ k then
 5       i ← i + 1
 6       A[i] ↔ A[j]
 7 A[i + 1] ↔ A[e]
 8 return  i + 1
```

Figura 8. Psedocódigo do Algoritmo de Ordenação Quick Sort
Fonte: O Próprio Autor (2020) **Fonte:** [Cormen et al. 2009]

2.3.1. Cálculo de Complexidade do Quick Sort

Arthur Anthony da Cunha Romão E Silva

Quick-Sort:

Caso base:

$T_w(1) = c1$

$T_w(n) = c1 + c2 + T_p(n) + c3 + T_w(n-1) + c4 + T_w(0)$

$T_w(n) = c + T_p(n) + T_w(n-1)$

$T_w(n-1) = c + T_p(n-1) + T_w(n-2)$

$T_w(n) = c + T_p(n) + [c + T_p(n-1) + T_w(n-2)]$

$T_w(n) = 2c + T_p(n) + T_p(n-1) + T_w(n-2)$

$T_w(n-2) = c + T_p(n-2) + T_w(n-3)$

$T_w(n) = 2c + T_p(n) + T_p(n-1) + [c + T_p(n-2) + T_w(n-3)]$

$T_w(n) = 3c + T_p(n) + T_p(n-1) + T_p(n-2) + T_w(n-3)$

$T_w(n) = xc + T_w(n-x) + [T_p(n) + T_p(n-1) + T_p(n-2) + \dots + T_p(n-(x-1))]$

$T_w(n) = xc + T_w(n-x) + \sum_{i=0}^{x-1} T_p(n-i)$ $\quad n-x=1$

$T_w(n) = (n-1)c + T_w(n-(n-1)) + \sum_{i=0}^{(n-1)-1} T_p(n-i)$ $\quad x=n-1$

$T_w(n) = (n-1)c + T_w(1) + \sum_{i=0}^{n-2} T_p(n-i)$ $\quad n=x+1$

$T_w(n) = (n-1)c + d + \sum_{i=0}^{n-2} (a(n-i)+b)$ $\quad n-1=x$

$T_p(n) = an + b$

$T_w(n) = (n-1)c + d + \sum_{i=0}^{n-2} an - ai + b$

$T_w(n) = (n-1)c + d + \sum_{i=0}^{n-2} an + \sum_{i=0}^{n-2} ai + \sum_{i=0}^{n-2} b$

$T_w(n) = cn + d - c + a\sum_{i=0}^{n-2} n + a\sum_{i=0}^{n-2} i + \sum_{i=0}^{n-2} b$

Arthur Anthony da Cunha Ramão E Silva

$$T_w(n) = cn - c + d + an(n-2) + \left(-a\left(\frac{n(n+1)}{2} - n - (n-1)\right)\right) + b(n-2)$$

$$T_w(n) = cn - c + d + an^2 - 2a + \left(-a\left(\frac{n^2+n}{2} - n - n + 1\right)\right) + bn - 2b$$

$$T_w(n) = cn - c + d + an^2 - 2a + \left(-a\left(\frac{n^2+n}{2} - 2n + 1\right)\right) + bn - 2b$$

$$T_w(n) = cn - c + d + an^2 - 2a + \left(-a\left(\frac{n^2+n-4n+4}{2}\right)\right) + bn - 2b$$

$$T_w(n) = cn - c + d + an^2 - 2a + \left(\frac{-a}{2}(n^2+n-4n+4)\right) + bn - 2b$$

$$T_w(n) = cn - c + d + an^2 - 2a + \left(\frac{-an^2}{2} - \frac{an}{2} + \frac{4an}{2} - \frac{4a}{2}\right) + bn - 2b$$

$$T_w(n) = cn - c + d + an^2 - 2a - \frac{an^2}{2} - \frac{an}{2} + 2an - 2a + bn - 2b$$

$$T_w(n) = n^2\underbrace{\left(\frac{a}{2} + a\right)}_{a} + n\underbrace{\left(c - \frac{a}{2} + 2a + b\right)}_{b} + \underbrace{\left(-c + d - 2a - 2a - 2b\right)}_{c}$$

$$\boxed{T_w(n) = an^2 + bn + c \approx O(n^2)}$$

Arthur Anthony da Cunha Ramão E Silva

Quick-Sort (Continuação):

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

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

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

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

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

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

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

$T_b(n) = c + 2c + 2T_p\left(\frac{n-1}{2}\right) + 4c + 4T_p\left(\frac{n-3}{4}\right) + 8T_b\left(\frac{n-7}{8}\right)$

$T_b(n) = 7c + 2T_p\left(\frac{n-1}{2}\right) + 4T_p\left(\frac{n-3}{4}\right) + 8T_b\left(\frac{n-7}{8}\right)$

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

$T_b(n) = (2^{\log_2 n} - 1)c + 2^{\log_2 n} d + \sum_{i=0}^{\log_2 n - 1} 2^i T_p\left(\frac{n-2i+1}{2i}\right)$

$T_b(n) = (n-1)c + nd + \sum_{i=0}^{\log_2 n - 1} 2^i T_p\left(\frac{n-2i+1}{2i}\right)$

$\sum_{i=0}^{\log_2 n - 1} 2^i \left(a\frac{(n-2i+1)}{2i} + b\right)$

$a\frac{(n-2i+1)}{2} + 2^i b$

$\sum_{i=0}^{\log_2 n - 1} an - a2i + a + 2^i b = an\sum_{i=0}^{\log_2 n - 1} + (-a\sum_{i=0}^{\log_2 n} 2i) + 1\sum_{i=0}^{\log_2 n - 1} + b\sum_{i=0}^{\log_2 n - 1} 2i$

Arthur Anthony Da Cunha Ramão E Silva

Quick-Sort (Continuação 2):

$an\log_2 n + (-a(n-1)] + a(n-1) + b(n-1)$

$an\log_2 n - an + a + an - a + bn - b$

$cn - c + n] + an\log_2 n - an + a + an - a + bn - b$

$n(\underbrace{c+d}_{k}) + an\log_2 n + \underbrace{(-c-b)}_{z}$

$nk + an\log_2 n + z$

$Tb(n) = nk + an\log_2 n + z$

$\boxed{Tb(n) \approx \Omega(n\log_2 n)}$

Arthur Anthony La Cuesta Ramos E Sidh

Quick-Sort (Continuación 3):

$$T_a(n) = \sum_{i=1}^{n-1} \frac{[T(i) + T(n-i)]}{n} + T_p(n)$$

$$T_a(n) = \frac{2}{n} \sum_{j=1}^{n-1} T(j) + T_p(n)$$

$$T_a(n) = \frac{2}{n} \sum_{j=1}^{n-1} (aj \log j + b) + T_p(n)$$

$$T_a(n) = \frac{2a}{n} \sum_{j=1}^{n-1} j \log j + \frac{2b}{n}(n-1) + T_p(n)$$

$$T_a(n) = \frac{2a}{n}\left(\frac{1}{2} n^2 \log n - \frac{1}{8} n^2\right) + \frac{2b}{n}(n-1) + T_p(n)$$

$$T_a(n) = an \log n - \frac{a}{4} n + 2b + T_p(n)$$

$$T_a(n) = an \log n + b + \left(b - \frac{a}{4} n + T_p(n)\right)$$

$$T_a(n) = an \log_2 n + b$$

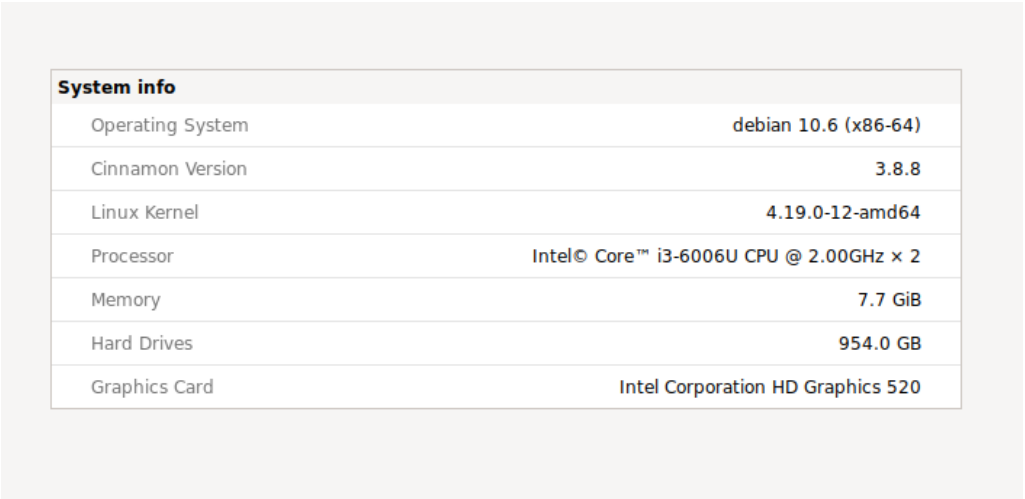
$$T_a(n) \approx \Theta(n \log_2 n)$$

3. Resultados e Conclusões

Para se analisar e comparar a eficiência entre os algoritmos de ordenação foi necessário realizar um experimento, medindo os tempos de execução em cada algoritmo, para diversos tamanhos instâncias, geradas de modo aleatorio, para cada algoritmo, e armazenando os resultados em um arquivo texto.

Vale salientar que o experimento foi realizado em uma máquina, que há inúmeros fatores que podem pesar no confiabilidade dos dados da execução, visto que há outros processos rodando em conjunto com os algoritmos. Foi utilizado uma linguagem de programação livre e interpretada, o Python, apesar de ser uma linguagem de alto nível, a eficiência para uma quantidade significativa de instâncias mostrou-se eficaz, além da simplicidade de implementação da linguagem e sua legibilidade. Os tamanhos das instâncias foram entre 1 e 1000. Todavia foi realizado a cada iteração de ordenação de dada instância, uma amostra de 1000 ordenações de instâncias que continham valores entre -100 e 100, para que fosse realizado uma média e armazenado então no arquivo, visto que foi realizado testes sem estes fatores, e o gráfico não representava de modo correto o comportamento do algoritmo, ficou de modo grosseiro, após a aplicação destas médias, o gráfico ficou mais suave. Dentro desta estrutura foi capturado o tempo antes da ordenação, e o tempo após a ordenação, foi realizado uma diferença e armazeno em um arquivo texto, que posteriormente foi utilizado para a criação dos gráficos.

A figura abaixo mostra detalhadamente as configurações da máquina que realizou este experimento, vale salientar que os três algoritmos foram executados nas mesmas circustâncias, isto é, apenas com o programa aberto, com a tentativa de evitar gargalhos durante a medição. Em resumo, uma máquina rodando Linux com um processador Intel Core 3 e 8 GB de RAM. Foram utilizados o editor de texto Sublime, para codificar e testar os algoritmos, e a ferramente de criação de gráficos livre, o GNUPlot.



System info	
Operating System	debian 10.6 (x86-64)
Cinnamon Version	3.8.8
Linux Kernel	4.19.0-12-amd64
Processor	Intel© Core™ i3-6006U CPU @ 2.00GHz × 2
Memory	7.7 GiB
Hard Drives	954.0 GB
Graphics Card	Intel Corporation HD Graphics 520

3.1. Gráfico de comportamento do Insertion Sort

A figura abaixo demonstra o comportamento do Insertion Sort, seu crescimento com base no tamanho da instância, sendo esta com um intervalo entre 1 e 1000.

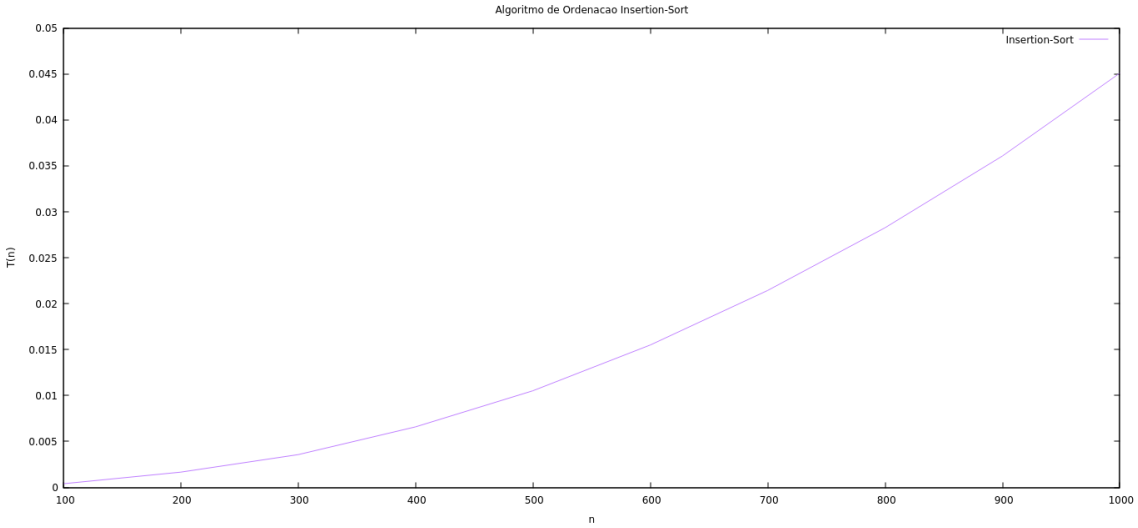


Figura 9. Gráfico de crescimento de tempo do Insertion Sort
Fonte: O Próprio Autor (2020)

A complexidade do Insertion Sort é quadrática, um algoritmo de simples implementação, mostrou-se eficiência para instâncias pequenas, além da simplicidade possui poucas comparações, as listas em ordem decrescente qualificam o pior caso, listas em ordem crescente como melhor caso. Dentre todos foi o pior, por sua ordem de crescimento ser alta.

3.2. Gráfico de comportamento do Quick Sort

A figura abaixo demonstra o comportamento do Quick Sort, seu crescimento com base no tamanho da instância, sendo esta com um intervalo entre 1 e 1000.

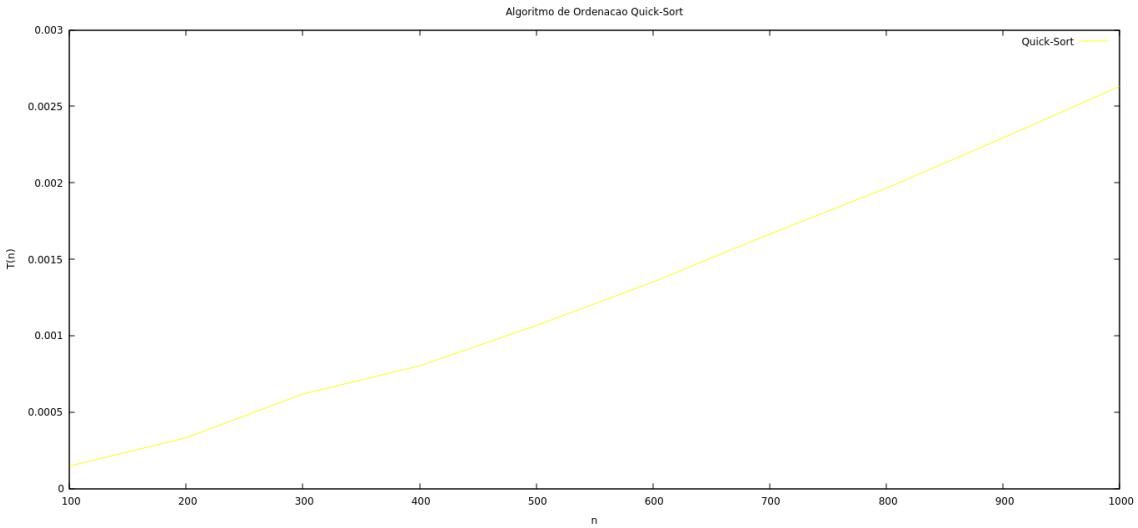


Figura 10. Gráfico de crescimento de tempo do Quick Sort
Fonte: O Próprio Autor (2020)

O algoritmo quicksort consome menos memória que o merge sort, fragmenta o vetor em duas partes sem a utilização de vetores auxiliares, e ordenando os vetores a partir de um pivô, melhorando a eficácia da ordenação, porém há muitas comparações e permutações, isso eleve um pouco seu custo. Dentre os três apresentados foi o melhor, mostrou-se eficiente para instâncias grandes, e mais rápido que merge sort, apesar de sua complexidade ser quadrática no pior caso.

3.3. Gráfico de comportamento do Merge Sort

A figura abaixo demonstra o comportamento do Merge Sort, seu crescimento com base no tamanho da instância, sendo esta com um intervalo entre 1 e 1000.

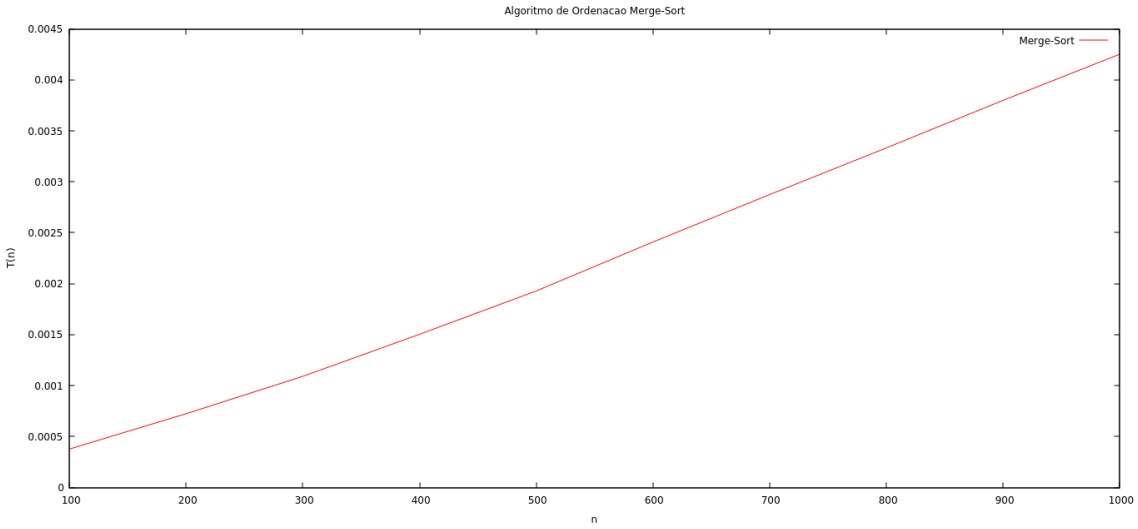


Figura 11. Gráfico de crescimento de tempo do Merge Sort
Fonte: O Próprio Autor (2020)

O Merge Sort utiliza-se da estratégia de dividir pra conquistar, sua recursividade é um fator imporante quando se trata em estruturas grandes e esparsas, porém sua desvantagem esta nas estruturas uniformemente distribuidas, seja em ordem crescente ou decrescente, também não se mostrou eficiente para pequenas instâncias, porém superior ao Inserion Sort. Apesar do consumo de memória e a recursividade pesar em determinadas circustâncias ficou como segundo melhor, sua complexidade é nlog n para qualquer caso.

3.4. Gráfico de comparação do comportamento do Insertion Sort, Merge Sort e Quick Sort

A figura abaixo demonstra a comparação do comportamento do Insertion Sort, Merge Sort e Quick Sort, o crescimento com base no tamanho da instância, sendo esta com um intervalo entre 1 e 1000.

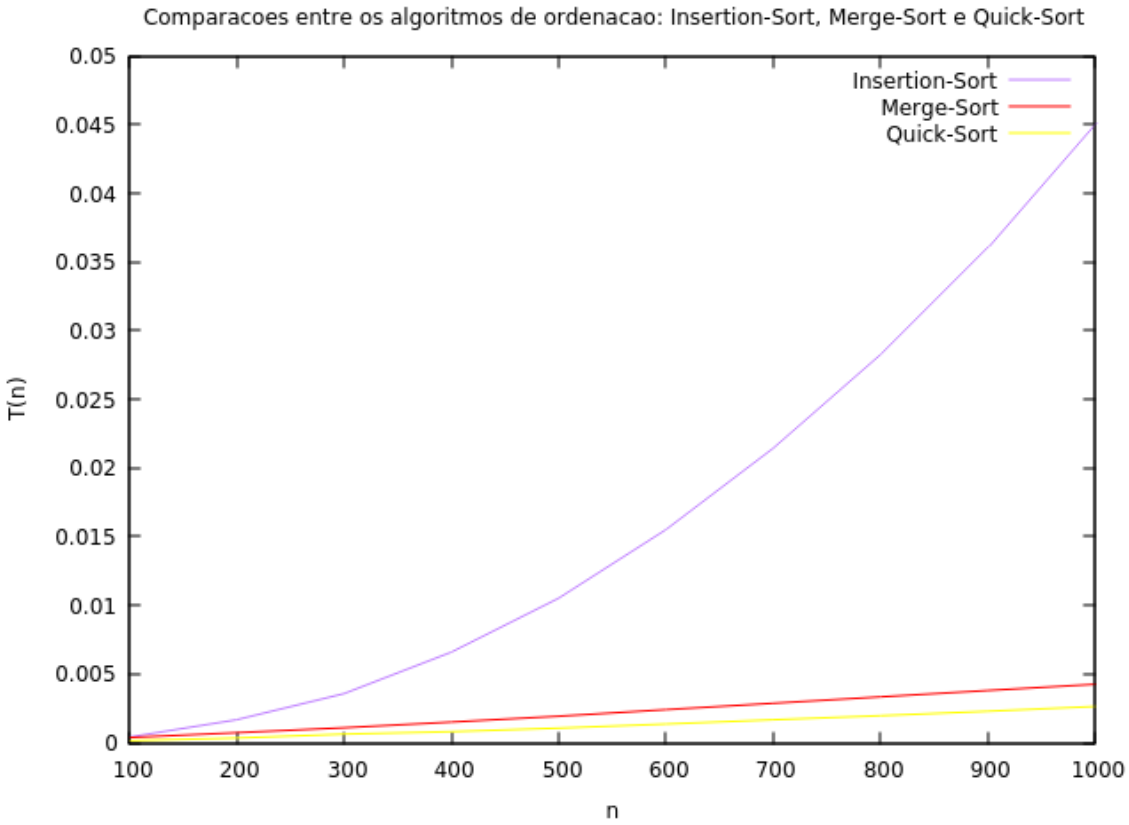


Figura 12. Gráfico de crescimento de tempo do Merge Sort
Fonte: O Próprio Autor (2020)

Como esperado analiticamente o Insertion Sort foi o pior, devido sua ordem quadrática, porém uma certa vantagem para pequenas instâncias, o Merge Sort ficou em segundo lugar, devido a intercalação e a recursividade, que são aspectos importantes, em certas configurações não são bons, como estruturas pre-ordenadas. E por fim o Quick Sort foi o vencedor, sua estrategia de divisão, inserções diretas, nestas configurações se apresentou superior aos demais.

O presente trabalho apresentou importantes conceitos no estudo de projeto e análise de algoritmos, uma introdução a cerca de problemas computacionais, com ênfase em problemas de ordenação de instâncias, foi realizado comparações experimentais, utilizando a linguagem Python, constatando os pontos negativos e positivos a cerca de cada algoritmo e suas respectivas estratégias, também foi realizado o projeto analítico a cerca dos três algoritmos de ordenação. Este trabalho é parte da atividade avaliativa da primeira unidade da disciplina de Estrutura de Dados ministrado pelo professor João Paulo.

Referências

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to algorithms*. MIT press.

do Lago Pereira, S., Linguagem, C., and Pelles, C. Estruturas de dados-1º sem/2020.

Honorato, B. (2013). Algoritmos de ordenação: análise e comparação.

Levitin, A. (2012). *Introduction to the design & analysis of algorithms*. Boston: Pearson,.

Szwarcfiter, J. L. and Markenzon, L. (1994). *Estruturas de Dados e seus Algoritmos*, volume 2. Livros Tecnicos e Cientificos.