

**Algoritmos e Estruturas de Dados**

**Sorting Methods**

**Professores:**

Tomás Oliveira e Silva ([tos@ua.pt](mailto:tos@ua.pt))  
Pedro Lavrador ([plavrador@ua.pt](mailto:plavrador@ua.pt))

**Pedro** **Sobral**, 98491 – XX%

**André Freixo**, 98495 – XX%

**Marta Fradique**, 98626 – XX%

22/01/2021

**Índice**

[**1 - Introdução** 3](#_Toc60173588)

[**2 - Introdução ao Problema** 4](#_Toc60173589)

[2.1 - Resumo 4](#_Toc60173590)

[2.2 – Compilação e Execução 4](#_Toc60173591)

[**3 - Resultados** 5](#_Toc60173592)

[3.1 – Bubble Sort 5](#_Toc60173593)

[3.2 – Shaker Sort 6](#_Toc60173594)

[3.3 – Insertion Sort 7](#_Toc60173595)

[3.4 – Shell Sort 8](#_Toc60173596)

[3.5 – Quick Sort 10](#_Toc60173597)

[3.6 – Merge Sort 12](#_Toc60173598)

[3.7 – Heap Sort 14](#_Toc60173599)

[3.8 – Rank Sort 15](#_Toc60173600)

[3.9 – Selection Sort 16](#_Toc60173601)

[3.10 – Resultados Totais 17](#_Toc60173602)

[**Conclusão** 17](#_Toc60173603)

[**Bibliografia** 18](#_Toc60173604)

# **1 - Introdução**

No âmbito da unidade curricular de AED, foi-nos apresentada a realização deste trabalho prático, sendo este relatório o resultado do problema “Sorting Methods”. Todo o código fonte e informações deste trabalho prático podem ser encontradas neste [repositório do GitHub](https://github.com/TheScorpoi/AED_Trabalho02)1 (mais informações, ler README.md do repositório).

O trabalho prático, consiste essencialmente em estudar os tempos de execução de uma série de rotinas de ordenação. Os algoritmos de ordenação foram implementados em C, sendo os mesmos fornecidos pelos docentes da unidade curricular.

A linguagem de programação C, é uma linguagem muito poderosa, pois dá ao programador um controlo íntegro de todo o processo programado, sendo uma linguagem onde o programador tem de lidar com todos os pormenores, torna-se consideravelmente eficiente e otimizada. Desta forma, conseguiremos execuções mais eficientes, pois toda a implementação é feita em C, como já referido anteriormente.

Com a realização deste trabalho prático, esperamos veemente alargar os nossos conhecimentos em C, e principalmente em conhecimento sobre algoritmos de ordenação. Esperamos também conseguir concluir com êxito todos os objetivos que são propostos no início (em comentário) do programa sorting\_methods.c.

1Por motivos de privacidade o repositório encontra-se privado, para visualização é favor entrar em contacto com os autores do trabalho prático.

# **2 - Introdução ao Problema**

## 2.1 - Resumo

A realização deste trabalho visa de forma genérica, estudar e tirar conclusões à cerca de diversas estratégias de ordenação. Desta forma, iremos correr o ficheiro *sorting\_methods.c*, de modo a que todos os resultados provenientes dessa execução sejam guardados num ficheiro novo *output.txt*, e todos os gráficos e conclusões à cerca das rotinas de ordenação serão feitas a partir dessa fonte de informação.

## 2.2 – Compilação e Execução

Para compilar o programa é necessário à partida ter um compilador de C instalado na máquina, por exemplo o *gcc*.

Posto isto, para compilar usamos o *makefile*:

make sorting\_methods

O programa está capacitado com um teste, para executar o teste basta passar a seguinte linha de código:

./sorting\_methods -test

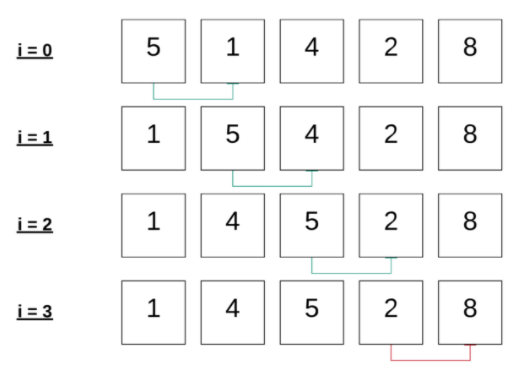
Para correr o programa, e guardar os resultados dos tempos de execução, num ficheiro .txt, executamos a seguinte linha:

./sorting\_methods -measure | tee output.txt

# **3 - Resultados**

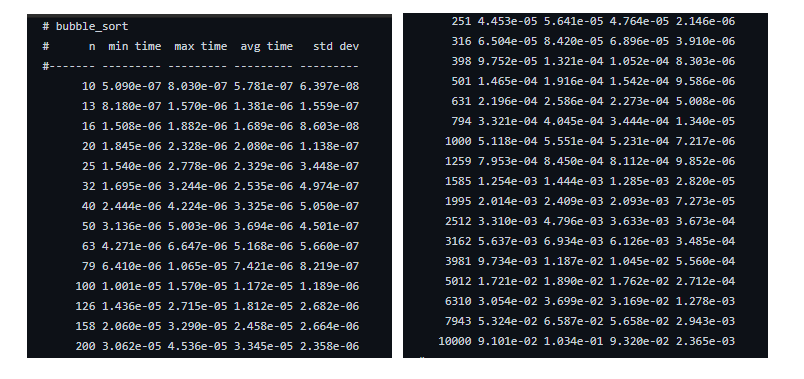
Como já referido anteriormente, os algoritmos e toda a implementação foi fornecida pelos docentes da unidade curricular, sendo então aqui no ponto 3 do relatório, que iremos expor os nossos resultados para cada rotina de ordenação.

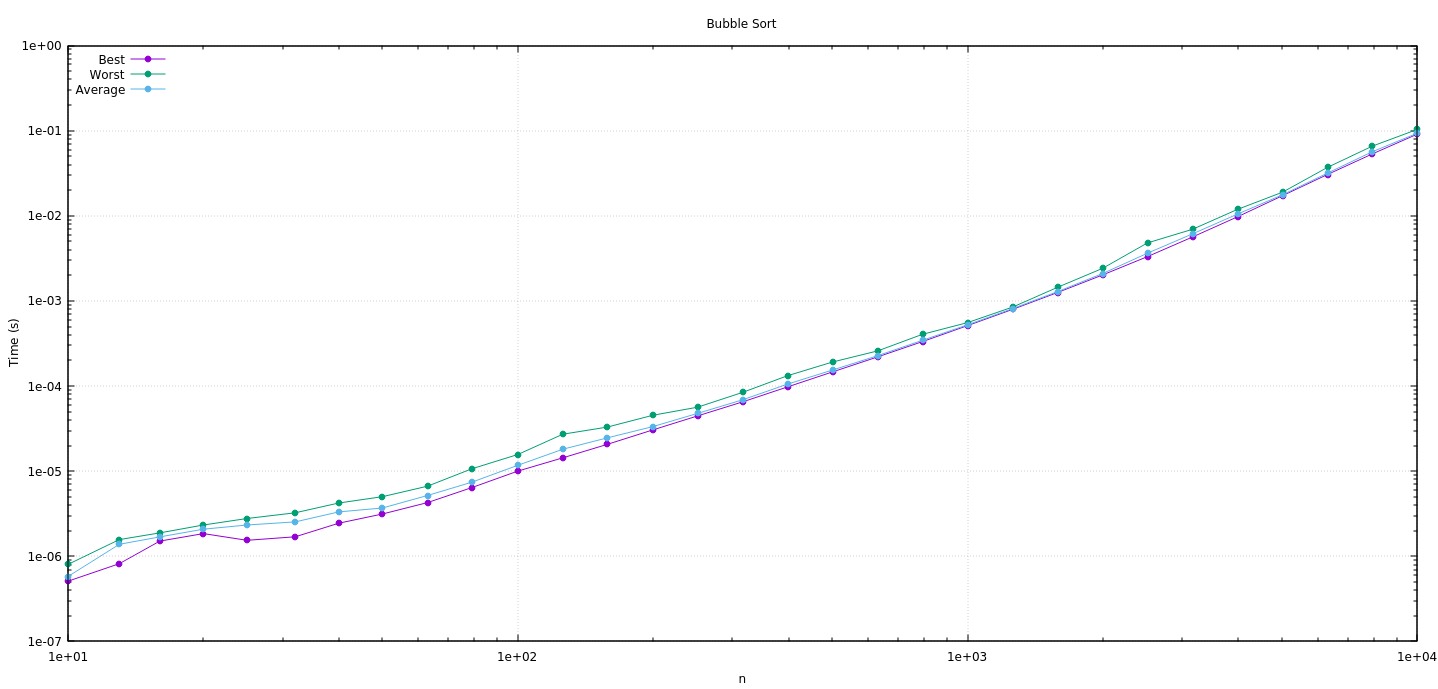
## 3.1 – Bubble Sort

*Bubble sort* é um algoritmo de ordenação bastante simples, este fundamenta-se em trocar a ordem dos elementos até estes estarem numa ordem correta. Por exemplo:

Ainda, o *bubble sort* em termos de complexidade computacional, no melhor caso é de **O(n)**, no médio de **O()** e no seu pior caso de **O()** também.

Ao analisarmos o gráfico com todas as rotinas de ordenação, podemos concluir que o *buble sort* tem o pior tempo de execução. Uma das maiores desvantagens deste gráfico é a quantidade exacerbada de tempo que demora a fazer a ordenação, ainda o tempo aumenta de forma exponencial á medida que o número de dados a analisar também aumenta.

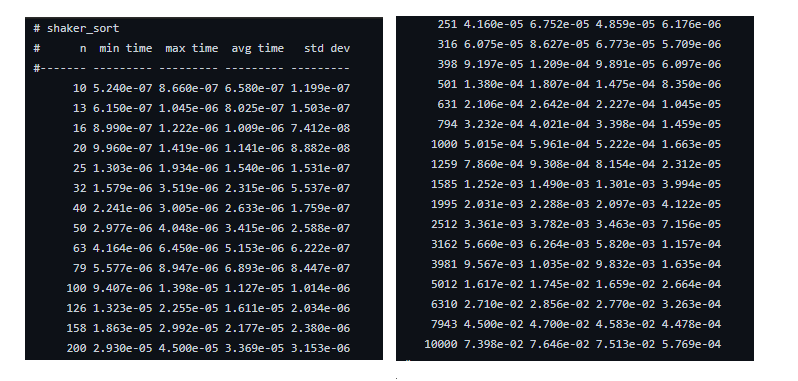


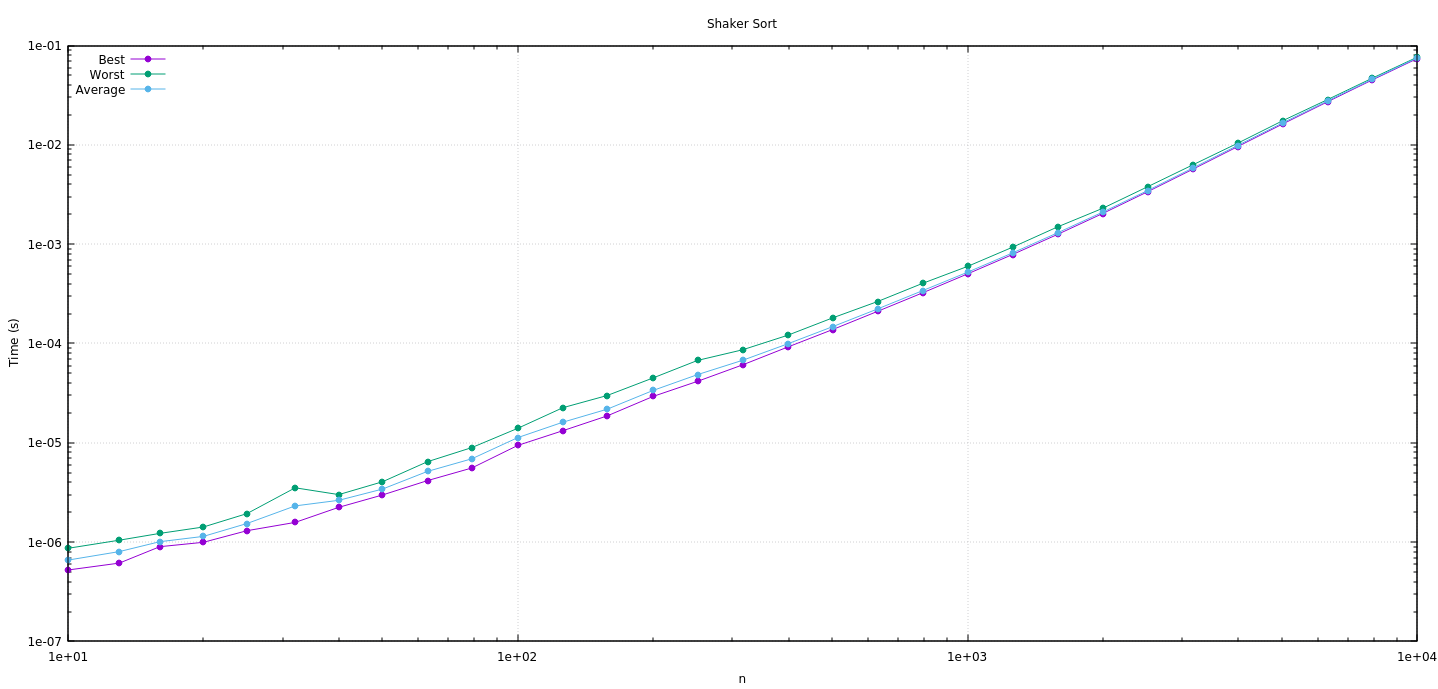


## 3.2 – Shaker Sort

O algoritmo *bubble sort* muda os elementos da direita para a sua posição correta. Ora, o algoritmo inicialmente funciona da esquerda para a direita, onde os itens são comparados dois a dois, se o da direita for maior eles trocam e assim sucessivamente, assim no fim da primeira iteração o maior número vai estar colocado na última posição. Na próxima iteração, o *array* é percorrido na direção oposta até à primeira posição, os itens também são comparados e invertem de posição se for necessário.

Ainda, o *shaker sort* em termos de complexidade computacional, no melhor caso é de **O(n)**, no médio de **O()** e no seu pior caso de **O()** também.

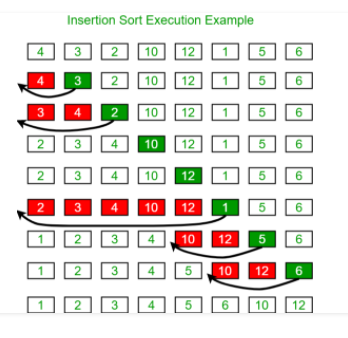




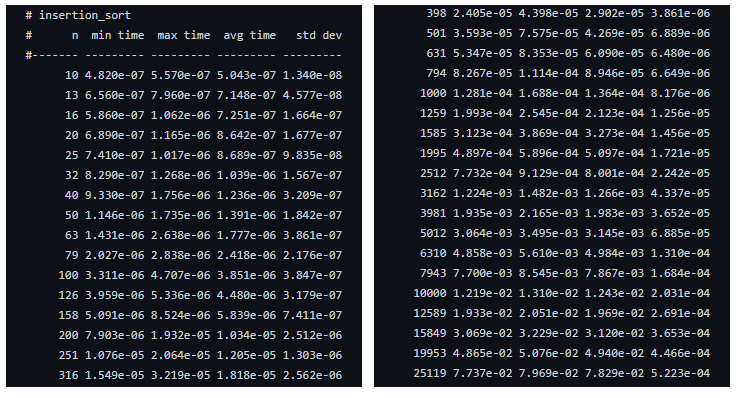
## 3.3 – Insertion Sort

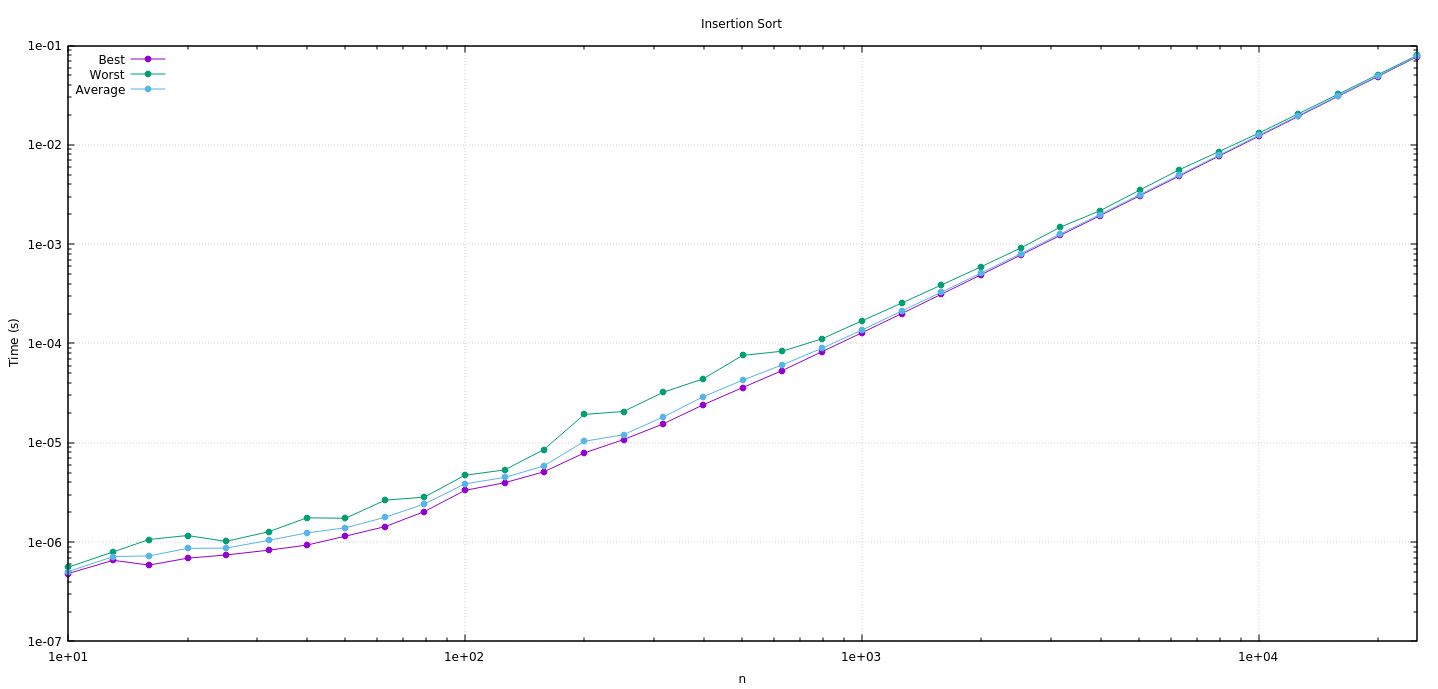
*Insertion sort* é um algoritmo de ordenação bastante acessível, este consiste em dividir um *array* numa parte ordenada e numa desordenada, de seguida selecionamos um valor da parcela desordenada e colocamos esta na parte ordenada na sua respetiva posição. Para tal, efetuamos os seguintes passos: comparamos o elemento atual com o seu anterior, se o elemento atual for inferior vamos movê-lo uma posição e trocar os elementos, por exemplo, se *arr[i]* for menor que *arr[i-1]*, trocamos as posições um pelo outro.

Ainda, o *insertion sort* em termos de complexidade computacional, no melhor caso é de **O(n)**, no médio de **O()** e no seu pior caso de **O()** também.



Ora, ao analisarmos o desempenho de todas as técnicas de *sorting* em conjunto podemos concluir que o *insertion sort* XXXXXXX. Assim, quando o array já se encontra parcialmente ordenado a técnica do *insertion sort* deve ser utilizada por apresentar o melhor desempenho, caso tenhamos um *array* maior ou menos ordenado é mais conveniente utilizar técnicas como o *merge sort.*





## 3.4 – Shell Sort

A estratégia de ordenação, *Shell Sort* (Fig.XX), é de forma genérica uma sucessiva aplicação da estratégia de ordenação, *Insertion Sort*, onde são criados sub-arrays do array original. A ideia do *Shell Sort*, é que seja possível a troca de itens distantes , criamos o array-h para um grande valor de h, e vamos reduzindo o valor de h, até que este se torne 1

A nível da complexidade computacional desta rotina de ordenação, é um pouco manhosa, pois ainda não é conhecida para nenhum dos 3 casos (best, worst e average), depende muito da sequência de *strides* (passos) que são usadas. São conhecidos alguns valores para h, de modo a que o algoritmo seja **o(n2),** por exemplo, quando h = 9 \* 2s - 9 \* 2s/2 + 1 , para *s* par.

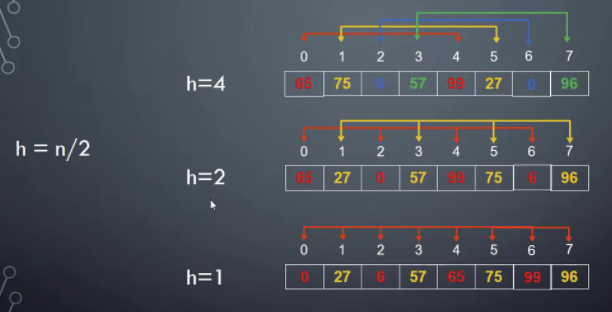


Figura - Exemplo demonstrativo do funcionamento do Shell Sort

A tabela com os tempos de execução obtida foi a seguinte(Fig.XX):

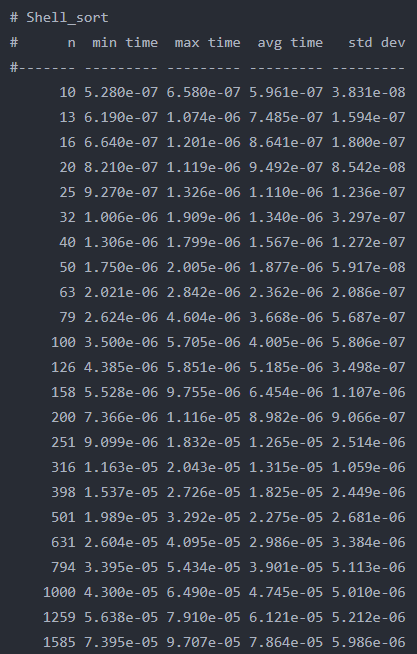
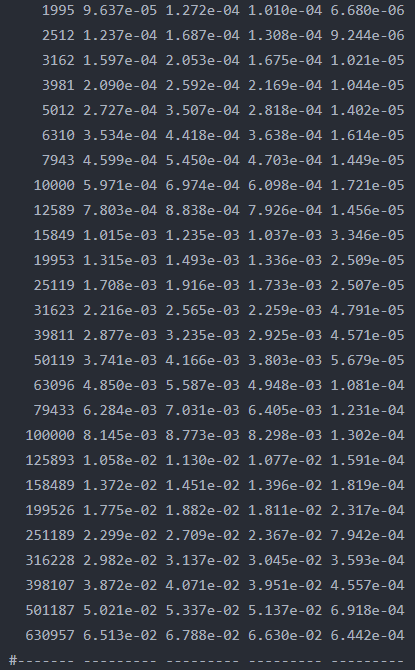


Figura - Tabela Shell Sort

O gráfico obtido para esta estratégia de ordenação foi o seguinte(Fig.XX):

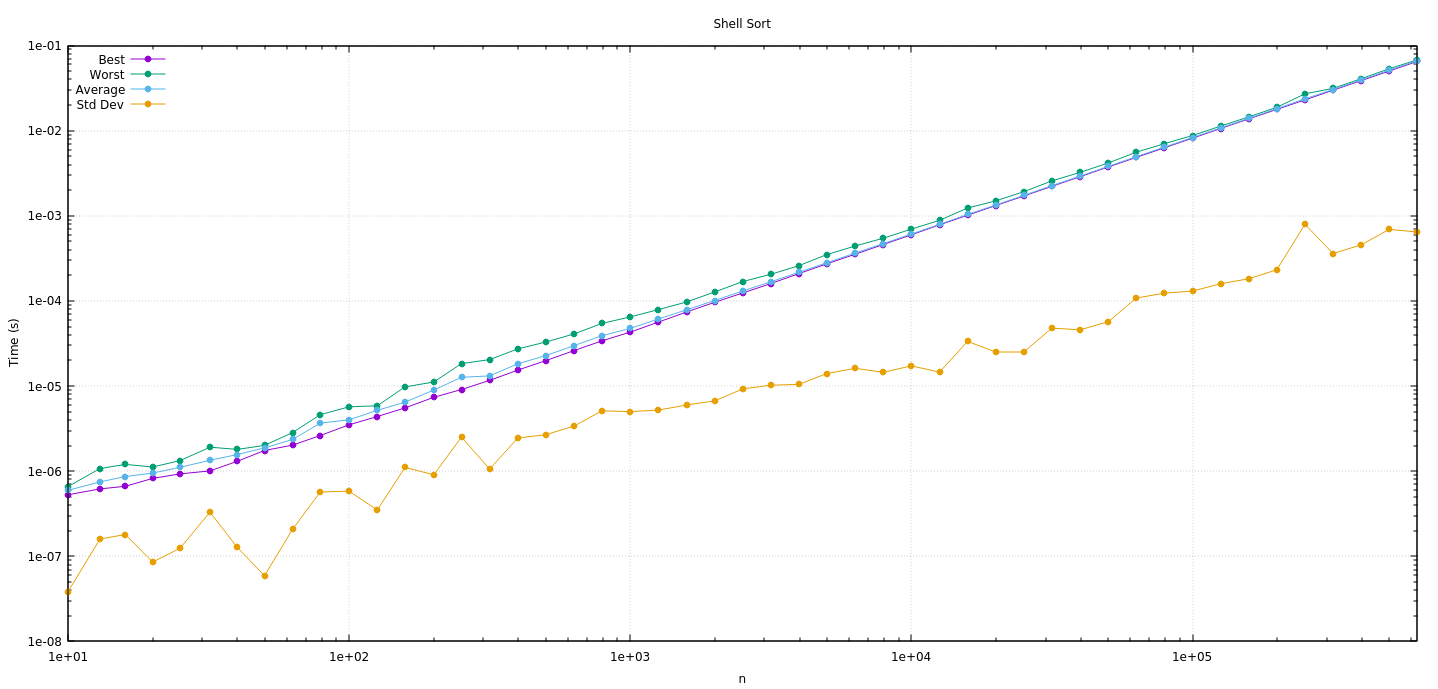
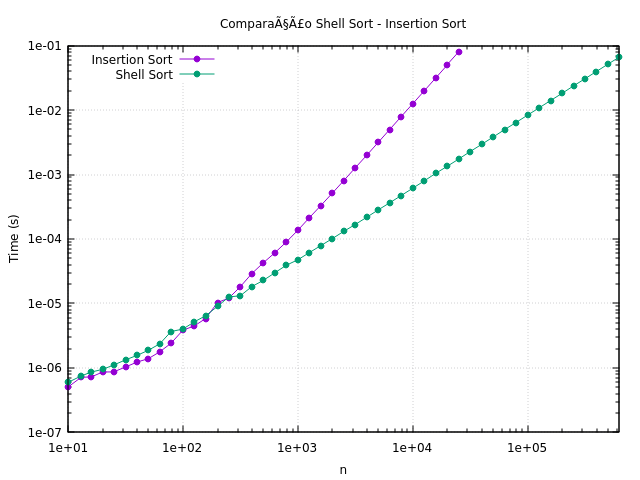


Figura - Gráfico Shell Sort

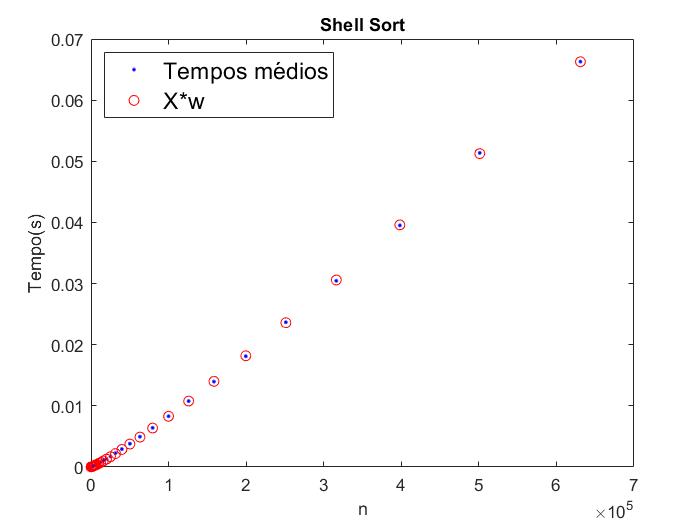
Estando no gráfico os tempos numa escala logarítmica, podemos concluir que os mesmos crescem de forma exponencial, salvo a parte inicial onde isso não se verifica assim tão bem, porém à medida que o *n* cresce, a o crescimento exponencial vai se retratando cada vez mais. O desvio padrão vai oscilando ao longo do *n*, havendo momentos em é considerável.

Como dito no início da introdução a esta estratégia de ordenação, o Shell Sort, passa por ser uma sucessiva aplicação do Insertion Sort. Sendo assim, a proxima figura (Fig.XX), apresenta a comparação entre as duas estratégias de ordenação.



Na comparação direta entre as duas rotinas, podemos concluir que o Shell Sort, apresenta vantagens significativas a nível de tempo, comparado ao Insertion Sort, sobretudo para valores de n superiores 100.

Criamos ainda o seguinte gráfico, onde é feita a *least squares fit,* esta implementação é muito poderosa, pois permite reduzir numa certa escala o número de informação e observar de forma nítida a tendência da curva. No Shell Sort os valores médios acompanham todos a tendência de crescimento de **n2,**o que comprova mais uma vez o crescimento desta rotina.



## 3.5 – Quick Sort

A rotina de ordenação Quick Sort (Fig.XX), é uma rotina que como o próprio nome indica é rápida, é implementada através de um método recursivo, que usa um algoritmo *Divide and Conquer*.

O algoritmo para ordenar arrays mais pequenos (menores que 20 a 30, discutível) usa a rotina *Insertion Sort*, pois torna-se mais eficiente, para arrays com tamanho maior, o algoritmo vai escolher um pivot, pivot esse que é escolhido aleatoriamente, neste contexto, faz-se uma passagem pelo array, e para a esquerda do pivot, colocamos os valores mais pequenos que o array, e para a direita os valores maiores, sendo a parte dos valores igual ao pivot discutível de que lado do pivot é que ficam. De seguida, é feita a mesma estratégia para cada sub-array, um da parte esquerda do pivot, outro da parte direita. O Quick Sort, tem uma complexidade computacional para o *Best e para o Average de* ***O(n log n)*,** porém para *Worst* ***O(n2)*.**

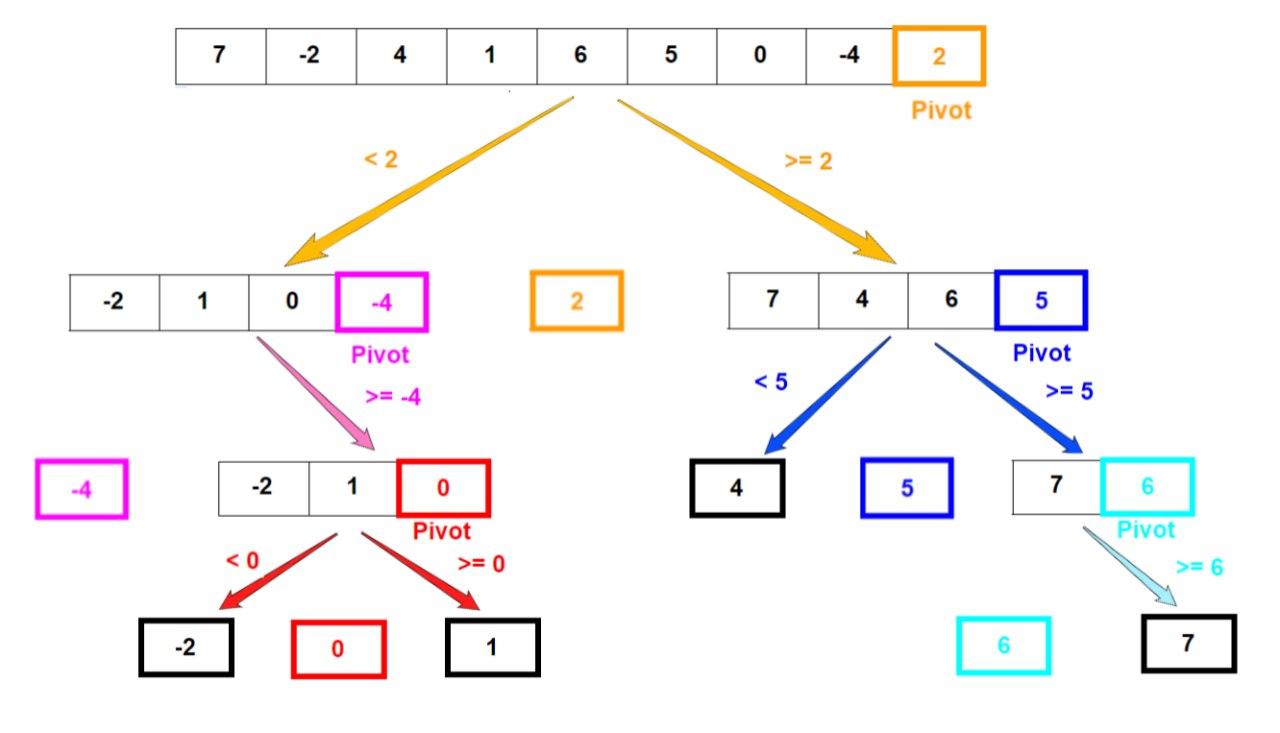


Figura - Demonstração do funcionamento do Quick Sort

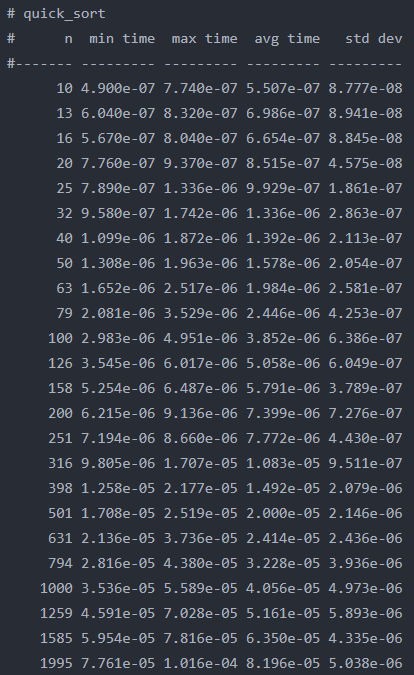
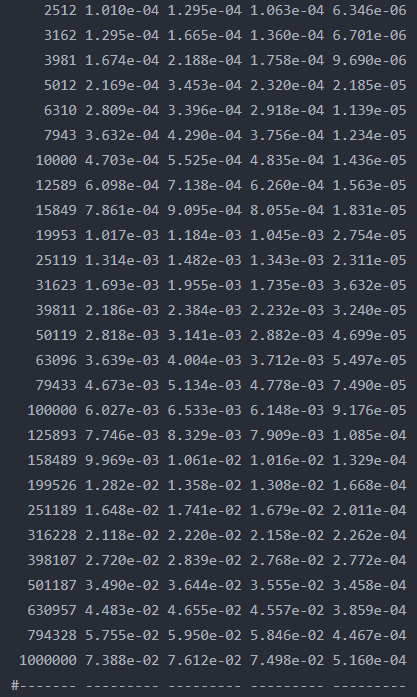
A tabela com os tempos de execução é a seguinte(Fig.XX):

Figura - Tabela Quick Sort

O gráfico com os tempos de execução para o Quick Sort, é o seguinte:

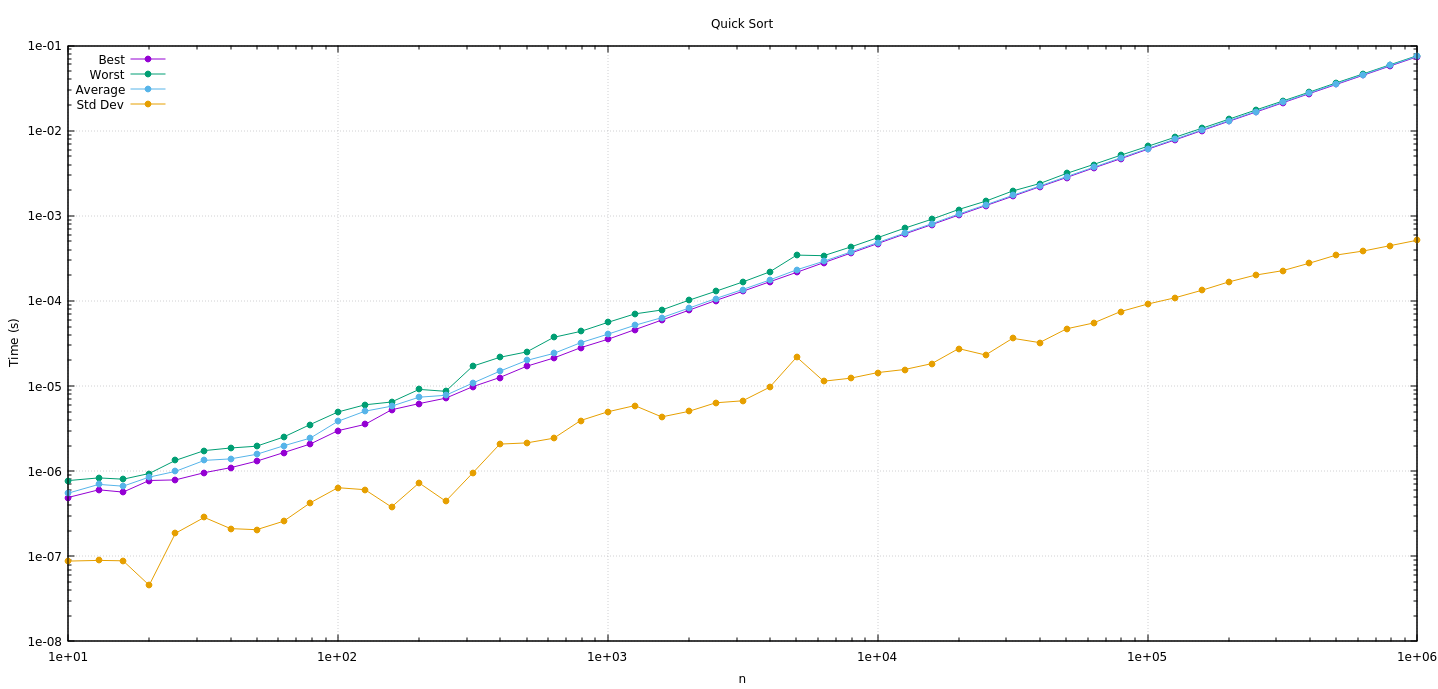
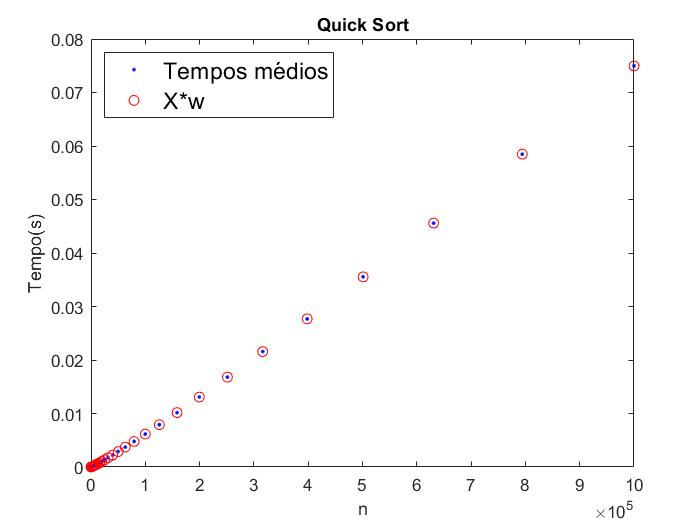


Figura - Gráfico Quick Sort

Esta rotina, como se pode observar no gráfico, apresenta um crescimento temporal exponencial, crescimento esse que no início apresenta algumas divergências, porém à medida que o n aumenta a reta torna-se mais linear, e também é notótio realçar que as diferenças de tempos de execução entre Best, Worst, e Average tornam-se muito pequenas, como a variação do desvio padrão o comprova.

Criamos ainda o seguinte gráfico, onde é feita a *least squares fit,* esta implementação é muito poderosa, pois permite reduzir numa certa escala o número de informação e observar de forma nítida a tendência da curva. No Quick Sort os valores médios acompanham todos a tendência de crescimento de **nlog(n).**



## 3.6 – Merge Sort

O Merge Sort (Fig.XX) é um algoritmo de ordenação, que usa uma estratégia de *Divide and Conquer,* é implementado de forma recursiva, e em traços gerais pode ser descrita como: Primeiramente divide-se o array em 2 partes iguais (ou, parcialmente igual, se o tamanho do mesmo for um número ímpar), divide-se recursivamente até que fiquemos com cada elemento sozinho, e a partir dessemomento, entra a parte de *Conquer*, onde se liga as partes dos sub-arrays ordenados. Esta rotina de ordenação tem uma complexidade computacional igual para os três casos, *Best, Avevage, e Worst*, **O(n log 10)**.



Figura - Demonstração do funcionamento do Merge Sort

A tabela com os tempos de execução da rotina Merge Sort, é a seguinte (Fig.XX):

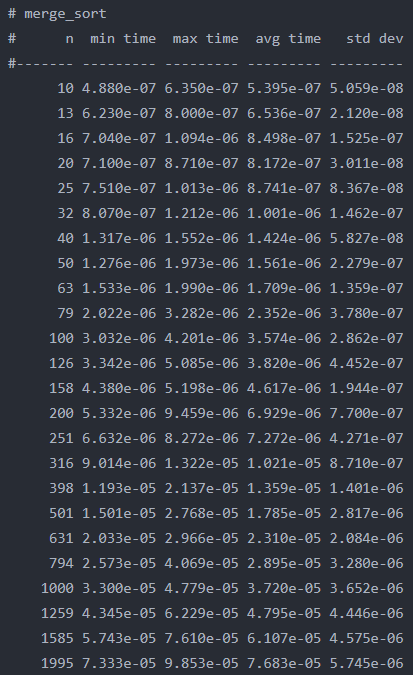
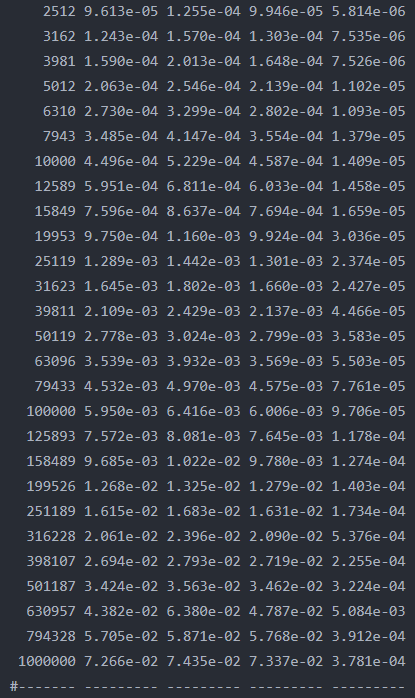


Figura - Tabela Merge Sort

O gráfico com os tempos de execução da rotina Merge Sort, é a seguinte (Fig.XX):

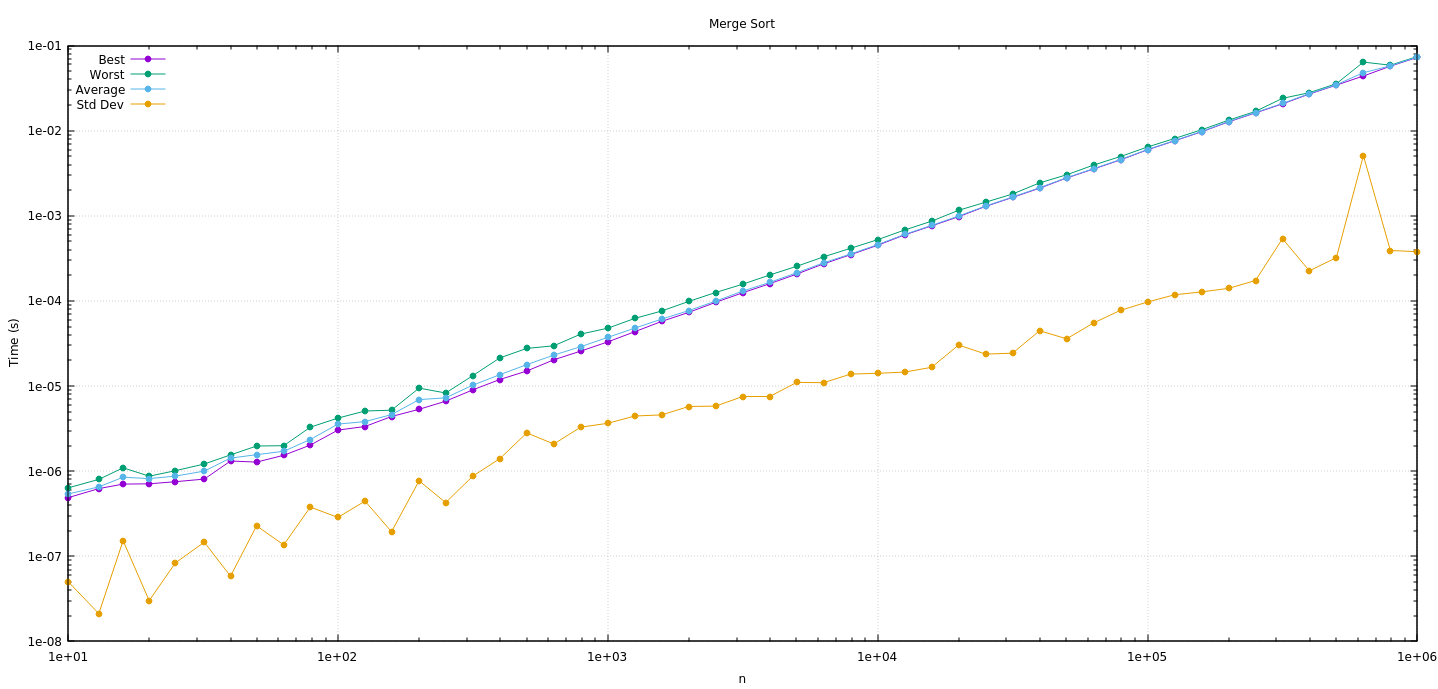
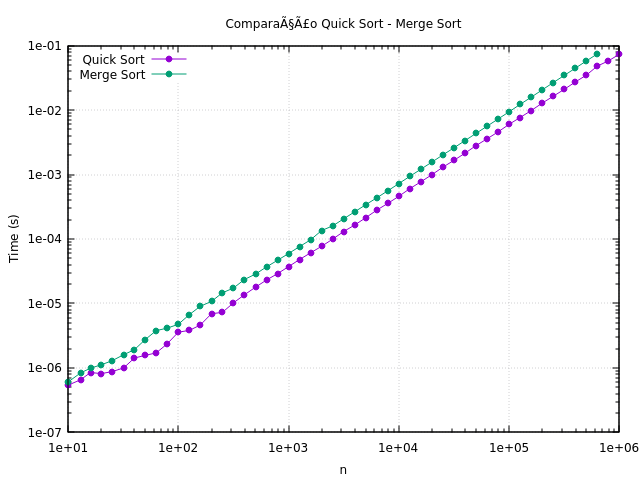
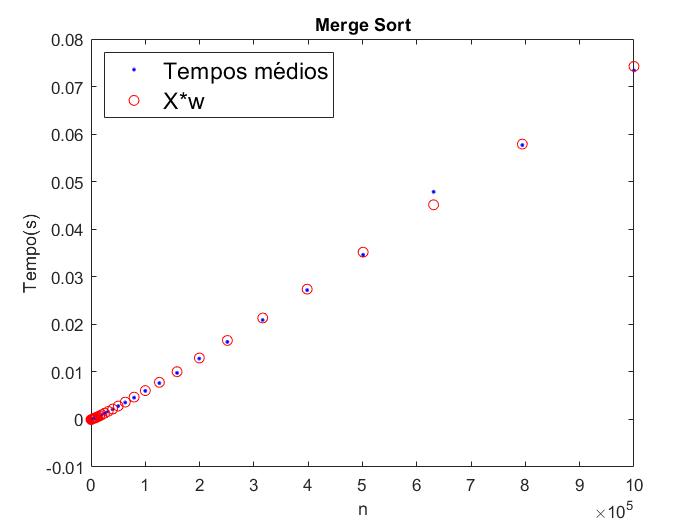


Figura - Gráfico Merge Sort

De acordo com o gráfico da (Fig.XX), a estratégia de ordenação, Merge Sort, apresenta no seu todo um crescimento exponencial, tal como nos outros algoritmos no início é mais linear que exponencial, no entanto à medida que o n aumenta o o nível de crescimento exponencial acompanha esse aumento de forma cada vez mais assente. É visível que, por análise do gráfico, o Worst Case, tem algumas oscilações, para valores de n pequenos, e algumas para valores de n mais elevados.

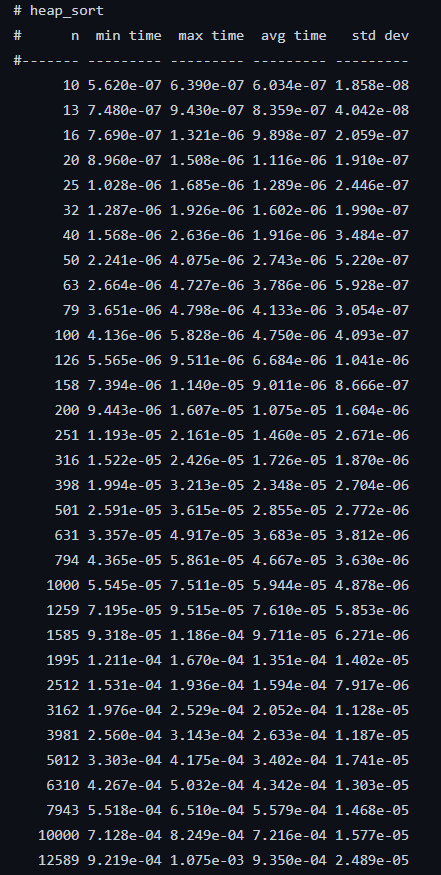
Fizemos ainda uma comparação do Merge Sort com o Merge Sort, pois achamos pertinente a comparação entre estas 2 rotinas, ambas usam Divide and Conquer, e têm tempos de execução parecidos. Na comparação direta consegue-se ver que o Quick Sort é em toda a escala melhor a nível de tempo de execução que o Merge Sort, a diferença não é muito substancial, porém é notável…

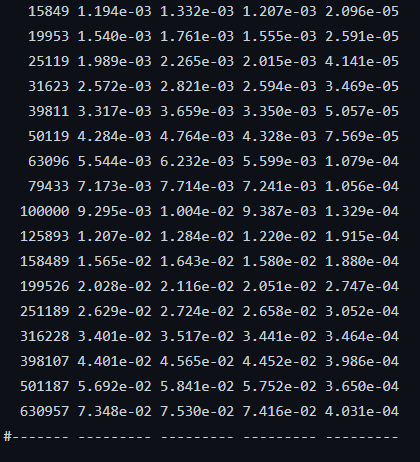
Criamos ainda o seguinte gráfico, onde é feita a *least squares fit,* esta implementação é muito poderosa, pois permite reduzir numa certa escala o número de informação e observar de forma nítida a tendência da curva. No Merge Sort os valores médios acompanham quase todos todos a tendência de crescimento de **nlog(n)**, temos alguns pontos a não seguir tão a risca o crescimento “normal”, no entanto o crescimento dos tempos de execução mantem-se.



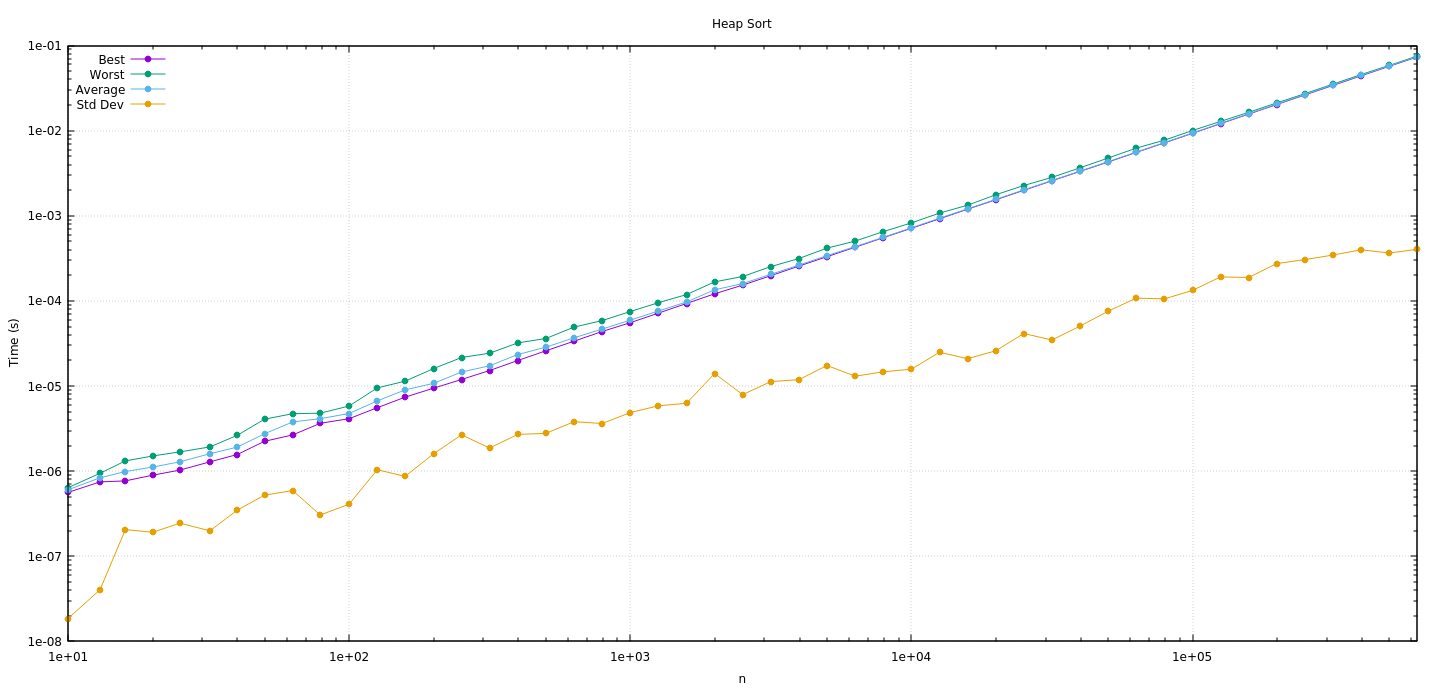
## 3.7 – Heap Sort

A tabela com os tempos de execução obtida foi a seguinte (Fig. XX):



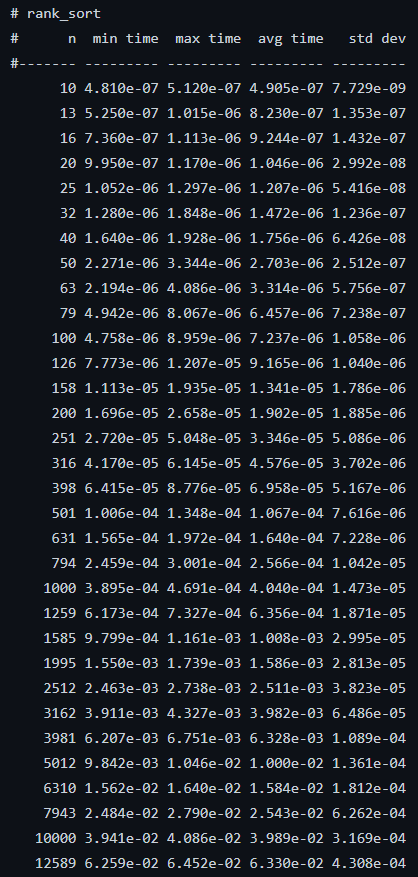


O gráfico obtido para esta estratégia de ordenação foi o seguinte (Fig.XX):

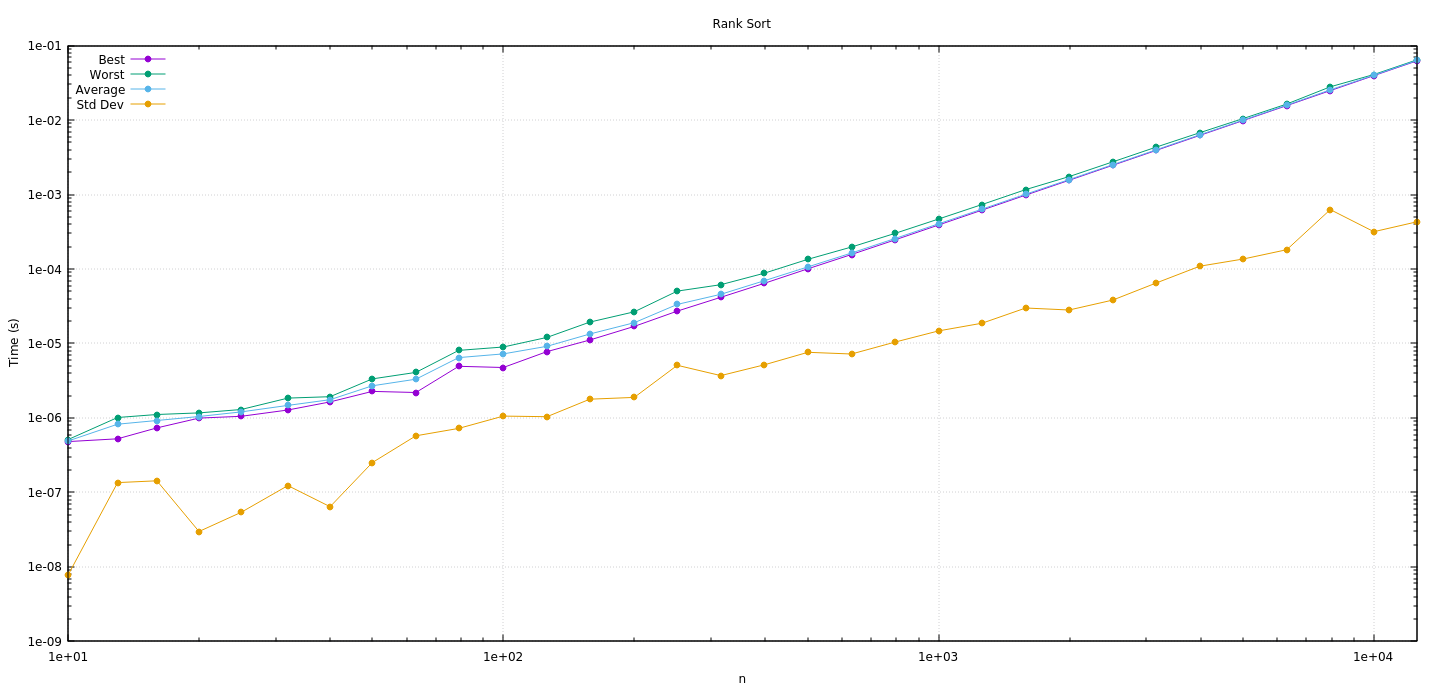


## 3.8 – Rank Sort

A tabela com os tempos de execução obtida foi a seguinte (Fig. XX):

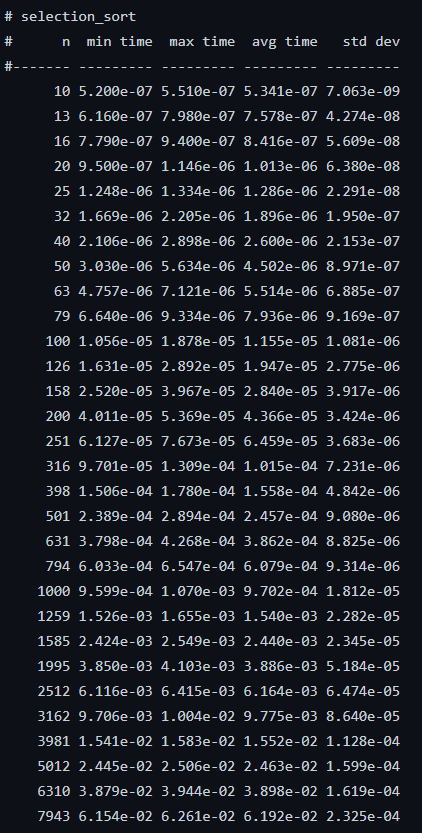


O gráfico obtido para esta estratégia de ordenação foi o seguinte (Fig.XX):

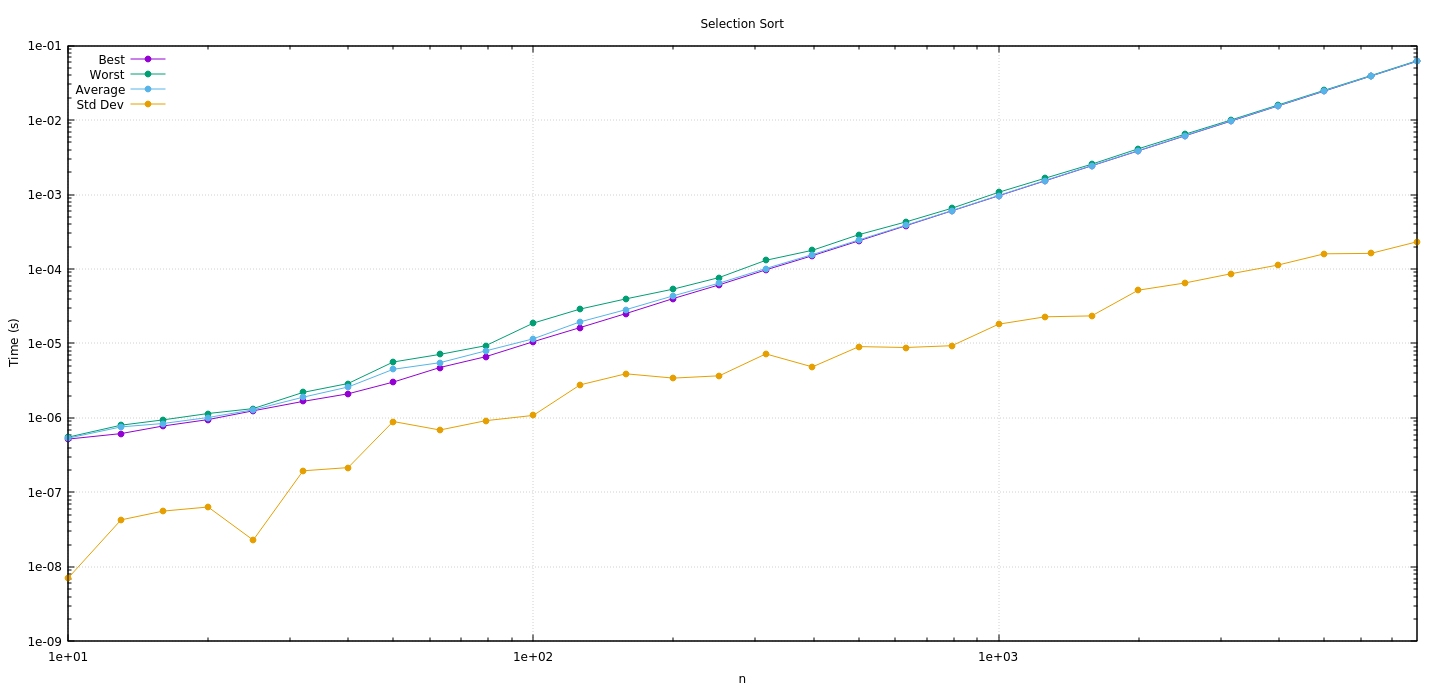


## 3.9 – Selection Sort

A tabela com os tempos de execução obtida foi a seguinte (Fig. XX):



O gráfico obtido para esta estratégia de ordenação foi o seguinte (Fig.XX):



## 3.10 – Resultados Totais

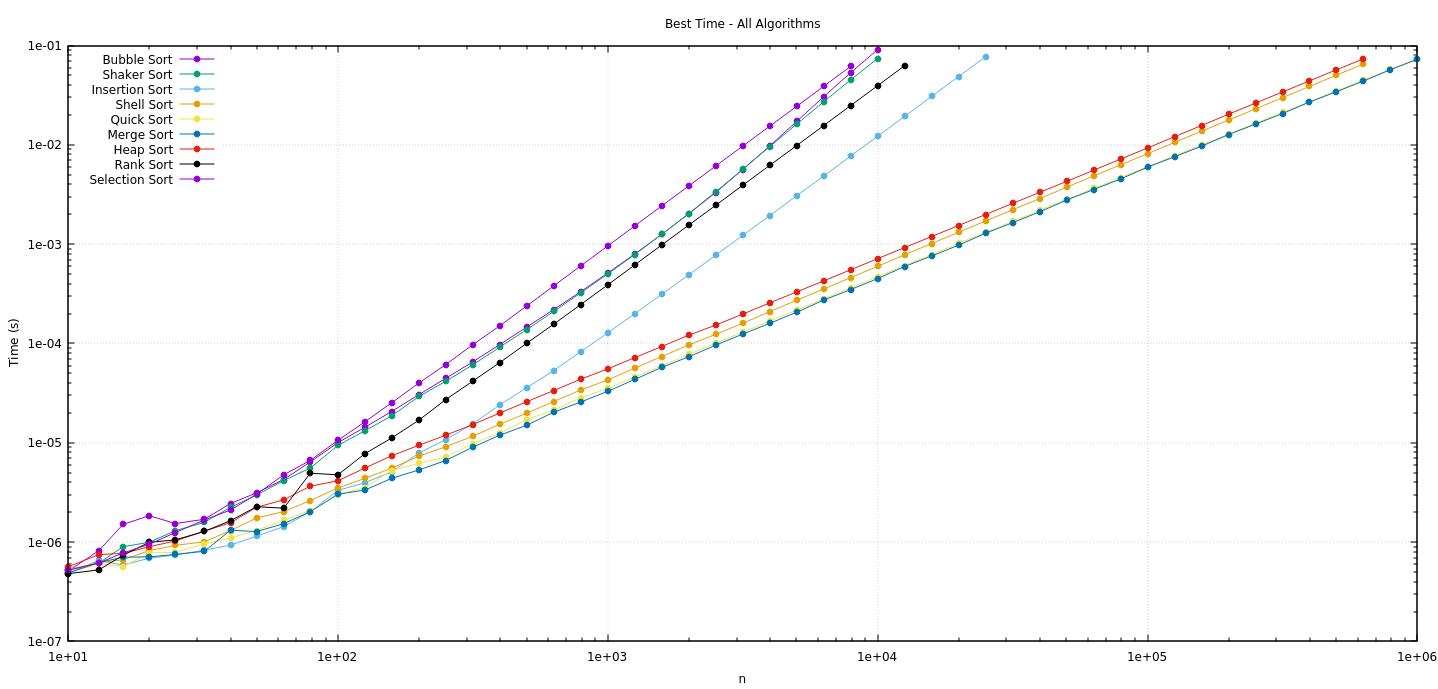
 No que toca à observação de todas as estratégias em simultâneo, criamos o seguinte gráfico (Fig.XX), o gráfico é referente ao melhor tempo de todas as rotinas de ordenação.

Figura - Melhor tempo de todas as estratégias

De acordo com o gráfico, em cima, podemos concluir que a melhor rotina de ordenação, pois é a mais rápida, é a estratégia *Merge Sort*. Pode parecer e até confundir-se um pouco com o gráfico com os tempos do Quick Sort, no entanto e como a Fig.XX, o demonstra, o Merge Sort consegue ter tempos mais pequenos.

Com a análise deste gráfico, conseguimos retirar conclusões relativamente a um determinado “grupo” de rotinas de ordenação, estratégias como *Bubble Sort, Rank Sort, Shaker Sort*, entre outras, que são computacionalmente más, pois o seu tempo de execução para uma pequena quantidade de números (menores que 100), chegam a ser semelhantes com as restantes rotinas, porém para valores maiores que esses, são significativamente más rotinas, pois são muito lentas. Como é visível no gráfico, por exemplo enquanto o *Heap Sort*, consegue quase ordenar um milhão de valores, o *Rank Sort,* consegue ordenar pouco mais de dez mil números.

FALAR DAS ESTRATEGIAS ALGORITMICAS TIPO DIVIDE AND CONQUER E ISSO, E VER SE HÁ ALGUMA RELAÇÃO DISSO E SER MAIS EFICIENTE

# **Conclusão**

# **Bibliografia**

Silva, Tomás Oliveira e. Lecture notes: Algorithms and Data Structute (AED - Algoritmos e Estruturas de Dados), LEI, MIEC, 2020/2021