Análise e comparação assintótica de algoritmos de ordenação

Asymptotic analysis and comparison of sorting algorithms

Lucas Evangelista Freire
Welton Santana Júnior
João Gabriel Alves de Souza
Larissa Mitie Curi Hirai
Fernanda Menezes Plessim de Melo

2023

Resumo

Conforme solicitado pelo professor Dr. Warley Gramacho este artigo visa realizar uma comparação de eficiência entra diferentes algoritmos de ordenação (sorting algorithms) tendo como base a complexidade de cada um deles.

Ao longo do artigo pontos como a relação de recorrência de cada algoritmos, complexidade e tempo de execução serão comparadas visando mostrar como eles se comportam dadas entradas já ordenadas de forma crescente, aleatórias e ordenadas de forma decrescente, assim obtendo os tempos de melhor caso, caso médio e pior caso.

Palavras-chave: ordenação. algoritmos. análise assintótica. quick sort. bubble sort. insertion sort.

Abstract

As requested by teacher Dr. Warley Gramacho this article aims to compare the efficiency of different sorting algorithms based on the complexity of each one of them.

Throughout the article, points such as the recurrence ratio of each algorithm, complexity and execution time will be compared in order to show how they behave given inputs already sorted in ascending order, random and sorted in descending order, thus obtaining the best times case, average case and worst case. **Keywords**: ordination.

algorithms. asymptotic analysis. fast sort. bubble sort. insertion sort.

1 Introdução

Este trabalho visa realizar a comparação de diferentes algoritmos de ordenação buscando explicar a teoria por trás da complexidade de cada um mostrando suas relações de recorrência.

Ao final deste artigo bucando retratar de forma mais palpável a eficiência de cada algoritmo serão realizadas uma série de comparações entre os algoritmos de forma mais empírica, apresentando o tempo de execução de cada um quando submetidos ao mesmo vetor. Todas as implementações serão realizadas em python

2 Especificações de hardware

Por mais que haja a complexidade de cada algoritmo o hardware em que ele é executado é uma variável no resultado final visto que quanto mais atual o hardware mais rápido o algoritmo será executado.

Vale lembrar que mesmo que hajam dois hardwares com especificações diferentes dependendo do algoritmo, em dado momento se submetidos ao mesmo caso o hardware mais fraco pode terminar a execução primeiro.

Segue abaixo a especificação do hardware utilizado, tendo em vista que foi-se utilizado o plano gratuito do site replit :

Cpu : 0,5 vCPU (intel)
RAM : 512 Mb

3 Algoritmos

3.1 Bubble sort

O algoritmo bubble sort é o mais intuitivo dentre os algoritmos de ordenação de tal modo que é possível uma pessoa que nunca viu seu pseudo-código consiga implementá-lo com pequenas divergências com o pseudo-código padrão.

Uma de suas características é que ele é estável, ou seja, preserva a ordem de elementos com mesmo valor.

Abaixo consta o pseudo-código do bubble sort :

```
Function BubbleSort(int array[]):
    int m, n, swap;
    for (m ← 0;m ≤ array - m; m ++) do
        for (n ← 0; n < array - (m + 1); n ++) do
            if (array [n] > array [n + 1]) then
                swap ← array [n];
                array [n] ← array [n + 1];
                array [n + 1] ← swap;
            end
        end
    end
    return array;
```

Algorithm 1: Algoritmo Bubble Sort

Dado um vetor qualquer, para cada posição há uma comparação com a posição que a sucede e caso a posição atual seja maior que sua sucessora as duas trocam de posição com a ajuda de uma variável auxiliar.

Como essa ordenação ocorreria apenas uma vez caso houvesse apenas um laço de repetição é necessário a adição de mais uma laço externo fazendo o algoritmo realizar essa ordenação quantas vezes for necessária, que no caso é a diferença do tamanho do vetor com o valor do passo atual m.

Para que não hajam operações obsoletas visto que a cada iteração há uma ordenação parcial o laço interno é executado tendo como parada a diferença entre o tamanho do vetor menos a soma de $m + 1$.

Visto que o algoritmo tem dois laços de repetição um dentro do outro considerando que o vetor tenha um tamanho n serão realizadas n^2 operações na maioria dos casos.

O algoritmo no pior caso (array invertido) e no caso médio (array aleatório) tem complexidade n^2 e tem por relação de recorrência o seguinte :

$$T(n) = T(n-1) + n - 1$$

No melhor caso o algoritmo recebe um vetor ordenado e nunca é executado visto que não há necessidade de uma ordenação, logo sua complexidade é $\Omega(n)$.

3.1.1 Tempo de execução e número de trocas

Tabela 1 – Tempo de execução e número de trocas do bubble sort

Casos	Tempo de execução	Número de trocas
Melhor caso com 100 posições	0.001751420000800863	0
Melhor caso com 1000 posições	0.27296823500000755	0
Melhor caso com 10000 posições	31.093445957000768	0
Caso médio com 100 posições	0.002546239000366768	2333
Caso médio com 1000 posições	0.5836652469997716	250251
Caso médio com 10000 posições	57.047481122000136	24966640
Pior caso com 100 posições	0.003795009999521426	4950
Pior caso com 1000 posições	0.7202099539999836	499500
Pior caso com 10000 posições	79.82543029200042	49995000

3.2 Insertion sort

O algoritmo insertion sort funciona como que uma pessoa que ao pegar cartas, para ordenar o baralho olha onde a carta deverá ser posta antes de inserí-la no conjunto de cartas que ela já possui.

Segue abaixo o pseudo-código do algoritmo insertion sort :

```
Function InsertionSort(int array[]):
    int i, j, key;
    for (i ← 1; i < array; i ++) do
        key ← array [i];
        j ← i - 1;
        while j ≥ 0 and key < array [j] do
            array [j + 1] ← array [j];
            j ← j - 1;
            array [j + 1] ← key;
        end
    end
    return array;
```

Algorithm 2: Algoritmo Insertion Sort

Considerando todo o vetor no laço de repetição mais externo, para cada posição nele faremos uma comparação com um pseudo subvetor, e compararemos cada posição desse vetor com o subvetor ordenado no laço mais interno, percorrendo-o da esquerda para a direita. Uma vez achada a posição do elemento do vetor no subvetor ordenado ele é inserido no subvetor.

O algoritmo no pior caso (array invertido) e no caso médio (vetor aleatório) tem complexidade n^2 e tem por relação de recorrência o seguinte :

$$T(n) = T(n-1) + n - 1$$

No melhor caso o algoritmo recebe um vetor ordenado, não há nenhuma troca e sua complexidade é $\Omega(n)$.

3.2.1 Tempo de execução e número de trocas

Tabela 2 – Tempo de execução e número de trocas do insertion sort

Casos	**Tempo de execução**	**Número de trocas**
Melhor caso com 100 posições	7.101999995029473	0
Melhor caso com 1000 posições	0.0005271600000469334	0
Melhor caso com 10000 posições	0.0037117099998340564	0
Caso médio com 100 posições	0.0030275200001597113	2927
Caso médio com 1000 posições	0.29654498300010346	234812
Caso médio com 10000 posições	22.487308944000006	24969176
Pior caso com 100 posições	0.0025175299999773415	4950
Pior caso com 1000 posições	0.5348003090000475	499500
Pior caso com 10000 posições	41.90451112100004	49995000

3.3 Quick sort

O Quick Sort é um algoritmo eficiente de ordenação, baseado na ideia de divisão e conquista. O Quick Sort possui as caracteristicas de ser instavel e "in place", ou seja, não preserva a ordem dos elementos de mesmo valor, mas não exige a utilização de estruturas de dados auxiliares. O algoritmo pode ser implementado de forma recursiva e iterativa.

Pseudo-código do Quicksort Recursivo:

```
Function QuickSort(int Array[], int Inicio, int Fim):
    int Baixo, Alto, pivo, swap;
    Baixo ← Inicio;
    Alto ← Fim;
    pivo ← Array[(Inicio + Fim) / 2];
    while (Baixo ≤ Alto) do
        while (Array[Baixo] < pivo) do
            Baixo ← Baixo + 1;
        end
        while (Array[Alto] > pivo) do
            Alto ← Alto - 1;
        end
        if (Baixo ≤ Alto) then
            swap ← Array[Baixo];
            Array[Baixo] ← Array[Alto];
            Array[Alto] ← swap;
            Baixo ← Baixo + 1;
            Alto ← Alto - 1;
        end
    end
    if (Inicio < Alto) then
        QuickSort (Array, Inicio, Alto);
    end
    if (Baixo < Fim) then
        QuickSort (Array, Baixo, Fim);
    end
```

Algorithm 3: Algoritmo QuickSort Recursivo

Pseudo-código de Quicksort Iterativa:

```
Function QuickSortIterative(int Array[], int Inicio, int Fim):
    Stack stack;
    stack.push((Inicio, Fim));
    while (stack != empty) do
        (Inicio, Fim) ← stack.pop();
        pivo ← Partition(Array, Inicio, Fim);
        if (Inicio < pivo - 1) then
            stack.push((Inicio, pivo - 1));
        end
        if (pivo + 1 < Fim) then
            stack.push((pivo + 1, Fim));
        end
    end
```

Algorithm 4: Algoritmo QuickSort Iterativo

```
Function Partition(int Array[], int Inicio, int Fim):
    pivo ← Array[Fim];
    i ← Inicio - 1;
    for (j ← Inicio; j ≤ Fim - 1; j ++) do
        if (Array[j] ≤ pivo) then
            i ← i + 1;
            swap ← Array[i];
            Array[i] ← Array[j];
            Array[j] ← swap;
        end
    end
    swap ← Array[i + 1];
    Array[i + 1] ← Array[Fim];
    Array[Fim] ← swap;
    return i + 1;
```

Algorithm 5: Algoritmo QuickSort Iterativo

O procedimento do algoritmo é baseado em sucessivas execuções de particionamento, uma rotina que escolhe um pivô e realiza subtituições entre valores de forma que no final de cada rotina, os elementos menores ou iguais ao pivot estão à sua esquerda e os maiores estão à sua direita.

A complexidade de tempo do algoritmo depende do valor do pivot em cada rotina. A escolha do pivot afeta como a lista sera particionada. O particinamento irá criar duas sublistas, que podem ser balanceadas, quando o seus tamanhos são proximos, ou desbalanceadas, quando uma sublista possui a grande maioria dos elementos enquanto a outra possui poucos.

O pior caso se torna aquele em que o particionamento de todas as rotina gera uma sublista de tamanho $n-1$ e outra de tamanho 0. Resultando na seguinte relação de recorrência:

$$T(n) = T(n-1) + T(0) + \theta(n)$$

O melhor caso acontece quando o particionamento de cada rotina gera duas sublistas de tamanho $\leq n/2$, representado pela relação:

$$T(n) \leq 2T(\frac{n}{2}) + \theta(n)$$

Portanto, A complexidade do quickSort é $O(n^2)$, no pior caso e $nlogn$, no melhor caso e no caso medio.

3.3.1 Tempo de execução e número de trocas

Tabela 3 – Tempo de execução e número de trocas do quick sort

Casos	Tempo de execução	Número de trocas
Melhor caso com 100 posições	0.0009836200006247964	99
Melhor caso com 1000 posições	0.18042436500036274	999
Melhor caso com 10000 posições	21.367593666000175	9999
Caso médio com 100 posições	0.002989118998812046	161
Caso médio com 1000 posições	0.0029637789994012564	2364
Caso médio com 10000 posições	0.10647092100043665	30721
Pior caso com 100 posições	0.0010493189984117635	99
Pior caso com 1000 posições	0.19795522300046287	999
Pior caso com 10000 posições	20.90247191399976	9999

3.4 Selection Sort

O SelectionSort é um algritmo de ordenação simples, baseada na seleção do elemento de menor valor, ou maior valor no caso de ordenação inversa. O Selection Sort, divide a lista em duas partes: a parte ordenada e a parte não ordenada.

O algoritmo procura o menor elemento na parte não ordenada e o coloca na posição correta na parte ordenada. Esse processo é repetido até que a parte não ordenada esteja vazia e a lista esteja completamente ordenada.

Pseudo-codigo do SelectionSort:

```
Function SelectionSort(int array[]):
    int i, j, menor, aux;
    for (i ← 0; i < array; i ++) do
        menor ← i;
        for (j ← i + 1; j < array; j ++) do
            if (array [j] < array [menor]) then
                menor ← j;
            end
        end
        if (array [i] ! = array [menor]) then
            aux ← array [i];
            array [i] ← array [menor];
            array [menor] ← aux;
        end
    end
    return array;
```

Algorithm 6: Algoritmo Selection Sort

A implementação clássica do Selection Sort é instável e in-place, ou seja, não preserva a ordem relativa de elementos iguais, mas não exige estrutura de dados adicionais.

O algoritmo no pior caso (array invertido), caso médio (array aleatório) e melhor caso (array ordenado) tem complexidade n^2 e tem por relação de recorrência o seguinte :

$$T(n) = T(n-1) + n - 1$$

3.4.1 Tempo de execução e numero de trocas

Tabela 4 – Tempo de execução e número de trocas do selection sort

Casos	**Tempo de execução**	**Número de trocas**
Melhor caso com 100 posições	0.0016296699996019015	4950
Melhor caso com 1000 posições	0.48282380100044975	499500
Melhor caso com 10000 posições	27.297785023999495	49995000
Caso médio com 100 posições	0.0016296699996019015	4950
Caso médio com 1000 posições	0.41332481000063126	499500
Caso médio com 10000 posições	29.122214330000133	49995000
Pior caso com 100 posições	0.0017672200010565575	4950
Pior caso com 1000 posições	0.30363024300004327	499500
Pior caso com 10000 posições	30.80261730699931	49995000

3.5 Merge Sort

O algoritmo merge sort é estável e assim como o quick sort se baseia na estratégia de "dividir para conquistar". Considerando que podemos imaginar que essa divisão pode ser representada como uma árvore ela basicamente divide uma array qualquer várias vezes até que não seja mais possível.

Quando a divisão não é mais possível os elementos dentro de cada fatia do array são ordenados entre si e o resultado dessa ordenação é passado para o nível superior. Quando o resultado é passado para o nível superior tudo nele está ordenado, então podemos considerar que esse resultado é um grande bloco ordenado, logo não será necessário fazer verificações dentro dele, apenas entre ele e o outro bloco.

Isso se repete até que se chegue na raíz da árvore resultando numa array ordenada. Abaixo consta o pseudo código do merge sort :

```
Function MergeSort(int array[]):
    int mid, i, j, k;
    int left[], right[];
    if (array > 1) then
        mid ← array/2;
        left ← array [:mid]; //recebe a primeira parte da array
        right ← array [mid:]; // recebe a segunda parte da array
        MergeSort (left);
        MergeSort (right);
        i ← j ← k ← 0;
    end
    while (i = 0; i < left and j < r) do
        if (left [i] ≤ right [i]) then
            array [k] ← left [i];
            i ← i + 1;
        end
        array [k] ← right [j];
        j ← j + 1;
        k ← k + 1;
    end
    while (i < left) do
        array [k] ← left [i];
        i ← i + 1;
        k ← k + 1;
    end
    while (j < right) do
        array [k] ← right [j];
        j ← j + 1;
        k ← k + 1;
    end
    return array;
```

Algorithm 7: Algoritmo Merge Sort

A complexidade do merge sort para o pior caso (array invertida), caso médio (array aleatória) e melhor caso (array ordenada) é $nlogn$ e sua relação de recorrência para todos os casos é de :

$$T(n) = 2 \times T(\tfrac{n}{2}) + O(n)$$

3.5.1 Tempo de execução e numero de trocas

Tabela 5 – Tempo de execução e número de trocas do merge sort

Casos	Tempo de execução	Número de trocas
Melhor caso com 100 posições	0.0005619100011244882	0
Melhor caso com 1000 posições	0.005540949001442641	0
Melhor caso com 10000 posições	0.20440335899911588	0
Caso médio com 100 posições	0.0005567500011238735	205
Caso médio com 1000 posições	0.056981145000463584	3323
Caso médio com 10000 posições	0.2876178810001875	40667
Pior caso com 100 posições	0.0006723500009684358	316
Pior caso com 1000 posições	0.007618838997586863	4932
Pior caso com 10000 posições	0.22137081699838745	64608

3.6 Heap Sort

O algoritmo heap sort é instável e se baseia na estrutura heap que age como uma árvore binária organizada num vetor.

Ao receber um array uma operação de heapify será aplicada nela até que a array se comporte como uma heap.

Em sequência o maior elemento é localizado e colocado no início de outra array e removido da heap. A operação de heapify é aplicada novamente na heap que perdeu o maior elemento para que a estrutura da heap seja preservada. Essa operação é feita até que não que não haja mais elementos a serem removidos da heap.

Quando a operação for feita a array estará em ordem. Vale lembrar que o heap sort tem o desempenho bom mas consome muita memória devido a utilização da heap.

Segue o pseudo código do heap sort

```
Function Heapify(int array[], int n, int i):
    int largest ← i, swap;
    int left ← 2 × i + 1, right ← 2 × i + 2;
    if (left < n and array [i] < array [left]) then
        largest ← left;
    end
    if (right < n and array [largest] < array [right]) then
        largest ← right;
    end
    if (largest != i) then
        swap ← array [i];
        array [i] ← array [largest];
        array [largest] ← array [i];
    end
    return Heapify (arr, n, largest);
```

Algorithm 8: Algoritmo Heapify

Function `HeapSort(`*int array[]*`):`

```
int n ← length (array), i, swap;
for (i ← n/2−1; i ≤ -1; i –) do
    Heapify (array, n, i);
end
for (i ← n - 1; i ≤ 0; i –) do
    swap = array [i];
    array [i] ← array [0];
    array [0] ← swap;
    Heapify (array, n, i);
end
return array;
```

Algorithm 9: Algoritmo Heap Sort

A complexidade do merge sort para o pior caso (array invertida), caso médio (array aleatória) e melhor caso (array ordenada) é $nlogn$ e sua relação de recorrência para todos os casos é de :

$$T(n) = 2 \times T(\tfrac{n}{2}) + O(logn)$$

3.6.1 Tempo de execução e numero de trocas

Tabela 6 – Tempo de execução e número de trocas do heap sort

Casos	Tempo de execução	Número de trocas
Melhor caso com 100 posições	0.0007792299984430429	247
Melhor caso com 1000 posições	0.01083201900109998	2496
Melhor caso com 10000 posições	0.3897250690024521	24997
Caso médio com 100 posições	0.04823076500179013	238
Caso médio com 1000 posições	0.05125076600234024	2318
Caso médio com 10000 posições	0.31345455499831587	23949
Pior caso com 100 posições	0.0006795010012865532	197
Pior caso com 1000 posições	0.007393488998786779	1996
Pior caso com 10000 posições	0.19963725399793475	19997

3.7 Shell Sort

O shell sort é um algoritmo de ordenação instável e o mais eficiente entre os algoritmos de ordenação quadráticos. Ele faz a utilização da estratégia do insertion sort mas ao invés de considerar a array como um grande elemento realiza a aplicação da estratégia de "dividir para conquistar".

O algoritmo ao receber uma array a divide em partes menores sendo que esse tamanho chamado de "gap"pode variar de acordo com o problema.

Ao usar esse gap para fazer essa divisão em cada parte resultante dela o algoritmo do insertion sort é aplicado. Quando isso chegar ao final o valor do gap deve ser reduzido da maneira que for conveniente (aqui o dividimos por 2,2) e o processo deve ser repetido até que o valor do gap atinja um valor mínimo.

Ao final do passo acima mais uma vez o insertion sort é aplicado no array como um todo para garantir que nada ficou desordenado.

Segue abaixo o algoritmo do shell sort :

```
Function ShellSort(int array[]):
    int h ← 1, n ← length (array), i, c, j;
    while (h > 0) do
        for (i ← h; i < n; i ++) do
            c ← array [i];
            j ← i;
            while (j  h and c < array [j - h]) do
                array [j] ← array [j - h];
                j ← j - h;
                array [j] ← c;
            end
        end
        h ← h/2,2;
    end
    return array;
```

Algorithm 10: Algoritmo Shell Sort

A complexidade do shell sort vai variar de acordo com o gap do algoritmo que no caso é 2,2. Para o pior caso (array invertida) a complexidade é de $O(n^2)$:

$$T(n) = 2 \times T(\tfrac{n}{2,2}) + O(n)$$

A complexidade do algoritmo no caso médio (array aleatório) pode variar de acordo com o gap que geralmente, mas mesmo assim a complexidade costuma variar entre n^2 e $nlogn$. No caso desde algoritmo com gap de 2,2 a complexidade está mais próxima de $\Theta(n^2)$.

A complexidade do shell sort no melhor caso (array ordenada) é $\Omega(nlogn)$. Não ocorre nenhuma troca e sua relação de recorrência é :

$$T(n) = 2 \times T(\tfrac{n}{2,2}) + O(1)$$

3.7.1 Tempo de execução e numero de trocas

Tabela 7 – Tempo de execução e número de trocas do shell sort

Casos	Tempo de execução	Número de trocas
Melhor caso com 100 posições	2.7889000193681568	0
Melhor caso com 1000 posições	0.0005072999992989935	0
Melhor caso com 10000 posições	0.0036803689999942435	0
Caso médio com 100 posições	0.0013505490005627507	2489
Caso médio com 1000 posições	0.5037145500009501	259850
Caso médio com 10000 posições	30.09744681200027	24751426
Pior caso com 100 posições	0.002479200000379933	4950
Pior caso com 1000 posições	0.6976805189988227	499500
Pior caso com 10000 posições	55.59779411499949	49995000

3.8 Comb Sort

O algoritmo comb sort é como que uma melhoria do bubble sort com uma diferença crucial que impacta no desempenho, um gap (espaçamento) definido que a cada vez diminui de tamanho.

O gap faz com que dada uma posição, há uma comparação com a posição da frente e em seguida a posição atual somada ao valor do gap fornece a posição do próximo elemento que será comparado com seu sucessor.

A cada iteração o gap diminui até um tamanho desejado, normalmente até que a posição atual somada ao gap resulte na posição sucessora. A redução pode ser adaptável, mas no algoritmo abaixo ele vale 1,3 :

```
Function GetNextGap(int gap):
    gap ← (gap×10)/13;
    if (gap < 1) then
        return 1;
    end
    return gap;
```

Algorithm 11: Algoritmo Get Next Gap

```
Function CombSort(int array[]):
    int n ← lenght (array);
    int gap ← n, swapped ← true, i, swap;
    while (gap != 1 or swapped == 1) do
        gap = GetNextGap (gap);
        swapped = False;
        for (i ← 0; i < n - gap; i ++) do
            if (array [i] > array [i + gap]) then
                swap ← array [i];
                array [i] ← array [i + gap];
                array [i + gap] ← swap;
            end
        end
    end
    return array;
```

Algorithm 12: Algoritmo Comb Sort

O comb sort no pior caso (array invertido) tem complexidade $O(n^2)$ e sua relação de recorrência é :

$$T(n) = T(n-1) + T(n-2) + O(n)$$

No caso médio (array aleatório) o algoritmo tem complexidade $\Theta(n^2)$ e sua relação de recorrência é :

$$T(n) = T(n-1) + T(n-2) + O(n)$$

No melhor caso (array ordenado) não são feitas comparações e sua complexidade é $\Omega(nlogn)$.

3.8.1 Tempo de execução e numero de trocas

Tabela 8 – Tempo de execução e número de trocas do comb sort

Casos	Tempo de execução	Número de trocas
Melhor caso com 100 posições	0.00014797999938309658	0
Melhor caso com 1000 posições	0.004542329999821959	0
Melhor caso com 10000 posições	0.18482989999938582	0
Caso médio com 100 posições	0.0003504800006339792	227
Caso médio com 1000 posições	0.005528989000595175	4192
Caso médio com 10000 posições	0.29971279699930164	59997
Pior caso com 100 posições	0.00033552000058989506	122
Pior caso com 1000 posições	0.004736650000268128	1582
Pior caso com 10000 posições	0.21666921700125386	20078

4 Comparações gerais

Em cada algoritmo no final foi-se mostrado uma tabela onde o algoritmo era submetido às entradas de melhor, pior e caso médio porém isoladamente. Vale lembrar que nessas comparações individuais a cada tabela eram gerados novos vetores aleatórios para o caso médio.

Neste tópico teremos uma tabela de comparação de complexidades e tempos de execução de uma forma geral.

Tabela 9 – Complexidade de caso médio de cada algoritmo

Algoritmo	Complexidade
Bubble sort	$O(n^2)$
Insertion sort	$O(n^2)$
Quick sort	$O(nlogn)$
Selection sort	$O(n^2)$
Merge sort	$O(nlogn)$
Heap sort	$O(nlogn)$
Shell sort	$O(n^2)$ ou $O(nlogn)$
Comb sort	n^2

Tabela 10 – Tempo de execução de cada algoritmo sob os mesmos vetores de 100 posições

Algoritmo	Pior caso	Caso médio	Melho caso
Bubble sort	0.0030403300006582867	0.0033711889991536736	0.004250359999787179
Insertion sort	4.263099981471896	0.00010435900003358256	4.277100015315227
Quick sort	0.0011450600013631629	0.0007421600002999185	0.0009756999988894677
Selection sort	0.0015285600002243882	0.0019408499993005535	0.0013760700003331294
Merge sort	0.00048257999878842384	0.0004475999994610902	0.0004970400004822295
Heap sort	0.0007292700011021225	0.0006705400010105222	0.0006809000005887356
Shell sort	3.4160000723204575	2.8329999622656032	2.764000055321958
Comb sort	0.00024913999914133456	0.00023983999926713295	0.00027306999982101843

Tabela 11 – Tempo de execução de cada algoritmo sob os mesmos vetores de 1000 posições

Algoritmo	**Pior caso**	**Caso médio**	**Melho caso**
Bubble sort	0.452837783999712	0.6000362499980838	0.6095789489991148
Insertion sort	0.0003869700012728572	0.00048098000115714967	0.0006285900017246604
Quick sort	0.19558338999922853	0.17557772200234467	0.21362827800112427
Selection sort	0.28916351000225404	0.23413935600183322	0.27402310199977364
Merge sort	0.009515379002550617	0.005215859000600176	0.005090038997877855
Heap sort	0.009027899002830964	0.0110819490000722	0.009569058998749824
Shell sort	0.0003303100020275451	0.00033041999995475635	0.00033271999927819706
Comb sort	0.056768414000544	0.0051228000011178665	0.004421278998052003

Tabela 12 – Tempo de execução de cada algoritmo sob os mesmos vetores de 10000 posições

Algoritmo	**Pior caso**	**Caso médio**	**Melho caso**
Bubble sort	34.23319528799948	55.964998411000124	70.19898077499965
Insertion sort	0.003694589999213349	0.0053324000000429805	0.004612770000676392
Quick sort	21.902357894001398	13.633132177999869	21.19884775799983
Selection sort	30.187859120998837	28.484823288999905	31.691810364000048
Merge sort	0.1914418000014848	0.1749050519993034	0.12761318599950755
Heap sort	0.6150853649996861	0.3385325350009225	0.3501250830013305
Shell sort	0.0029185399998823414	0.003976488998887362	0.003118220000033034
Comb sort	0.19023469999956433	0.18981027999871003	0.10567796899886162

5 Graficos

Todos os gráficos neste tópico foram feitos utilizando o gnu plot para fazer uma comparação doas algoritmos com tempo x tamanho do vetor.

6 Conclusão

Ao longo deste artigo tivemos a oportunidade de aprender um pouco mais sobre alguns algoritmos de ordenação, tendo uma visão geral sobre eles.

Vale ressaltar que por mais que alguns algoritmos sejam mais ineficientes eles não são obsoletos visto que podem cumprir o que se pede em projetos simples e possuem fins didáticos uma vez que algoritmos ais eficientes tem uma curva de aprendizado maior e requerem que quem os implementa tenha uma experiência prévia sobre alguns assuntos.

7 Referências bibliográficas

Shell Sort (2022) Wikipedia. Available at: https://pt.wikipedia.org/wiki/Shell$_sort(Accessed : 31May2023)$.

Comb sort (2018) Wikipedia. Available at: https://pt.wikipedia.org/wiki/Comb$_sort(Accessed : 31May2023)$.

(No date) INE 5384 - Estruturas de Dados. Available at: http://www.inf.ufsc.br/ r.mello/ine5384/ (Accessed: 31 May 2023).

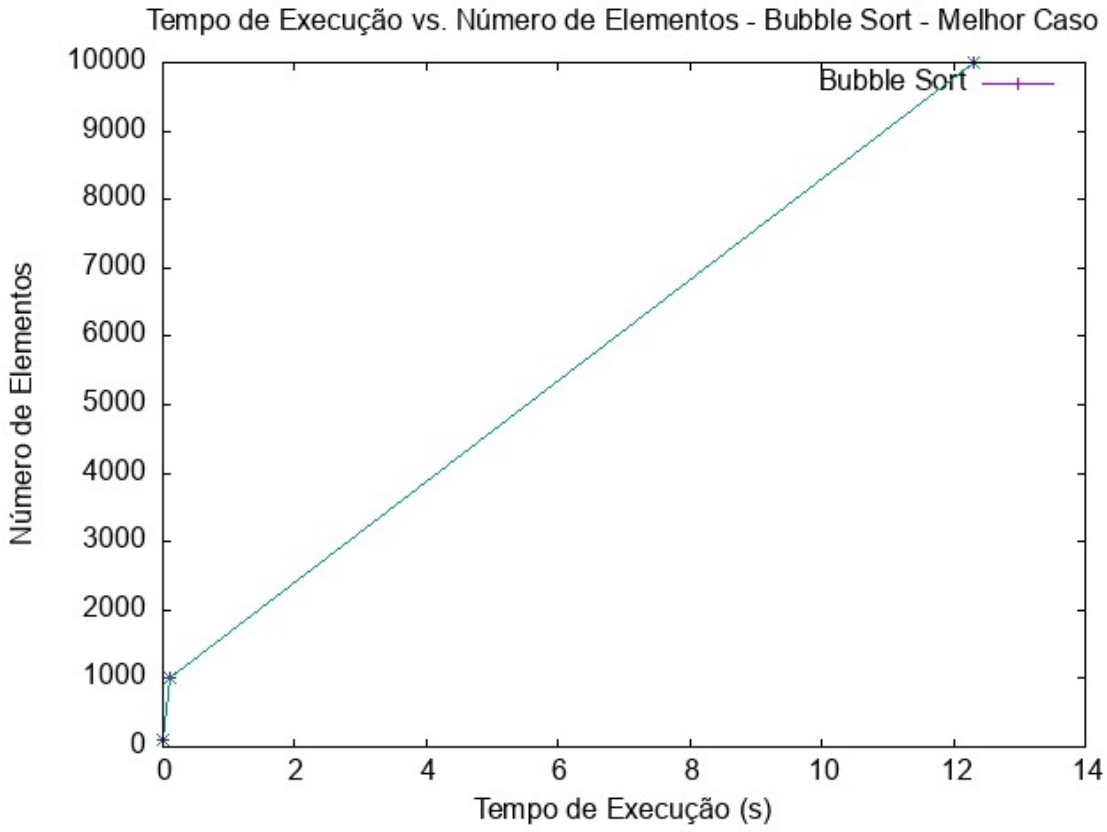

Figura 1 – Bubble sort melhor caso

Merge sort (2023) Wikipedia. Available at: https://en.wikipedia.org/wiki/Merge$_sort$: : $text = In$

Recursive selection sort questions and answers (2019) Sanfoundry. Available at: https://www.sanfoundry.com/recursive-selection-sort-multiple-choice-questions-answers-mcqs/ (Accessed: 31 May 2023).

Selection sort (2023) Wikipedia. Available at: https://pt.wikipedia.org/wiki/Selection$_sort(Accessed$: $31May2023)$.

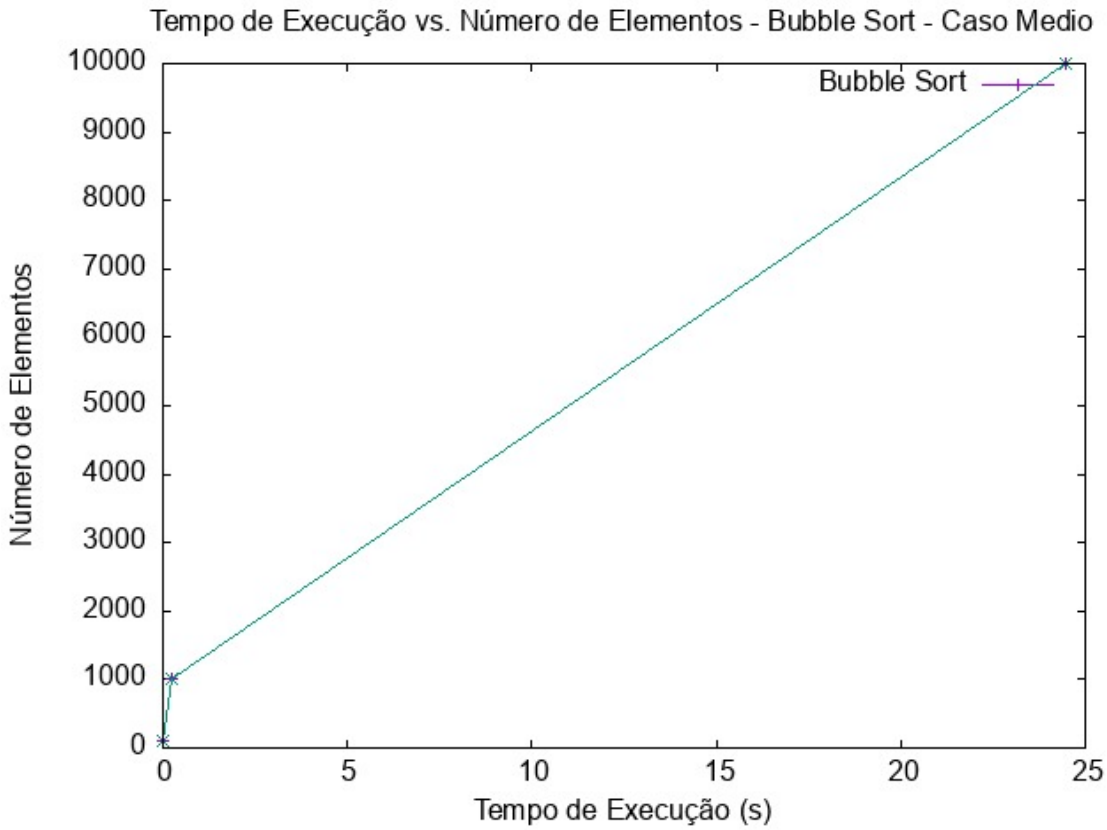

Figura 2 – Bubble sort caso médio

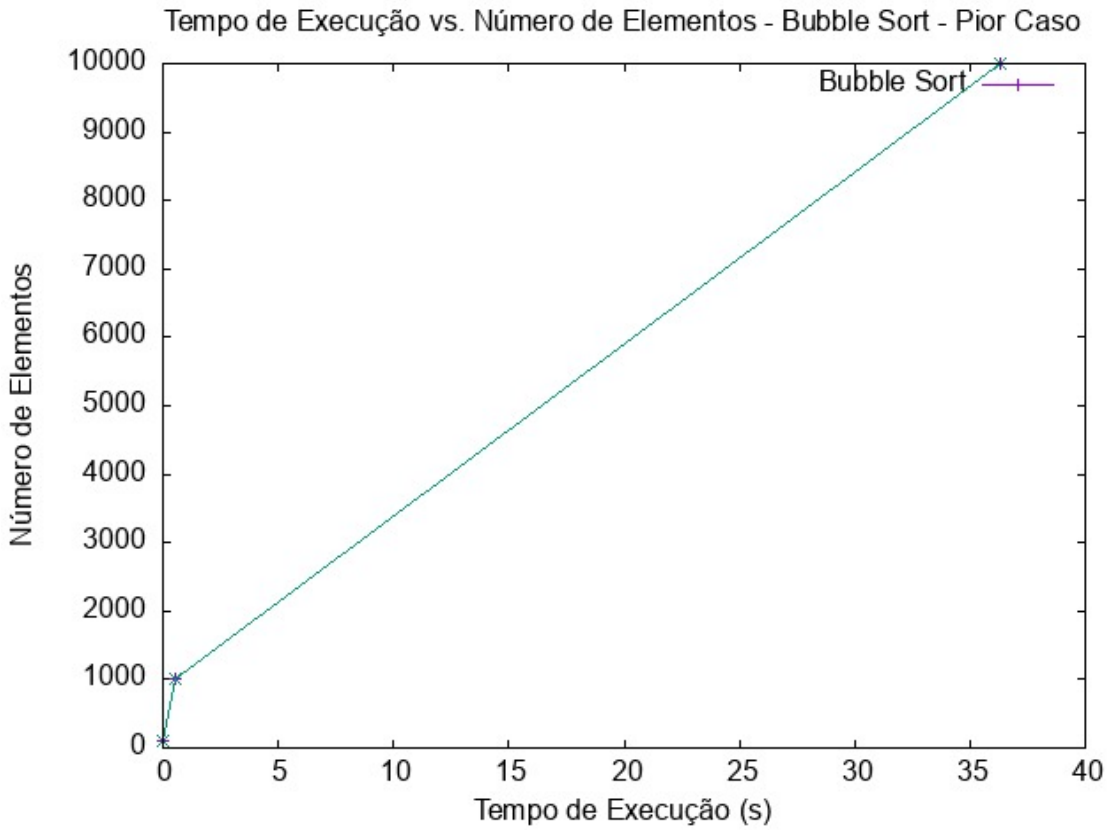

Figura 3 – Bubble sort pior caso

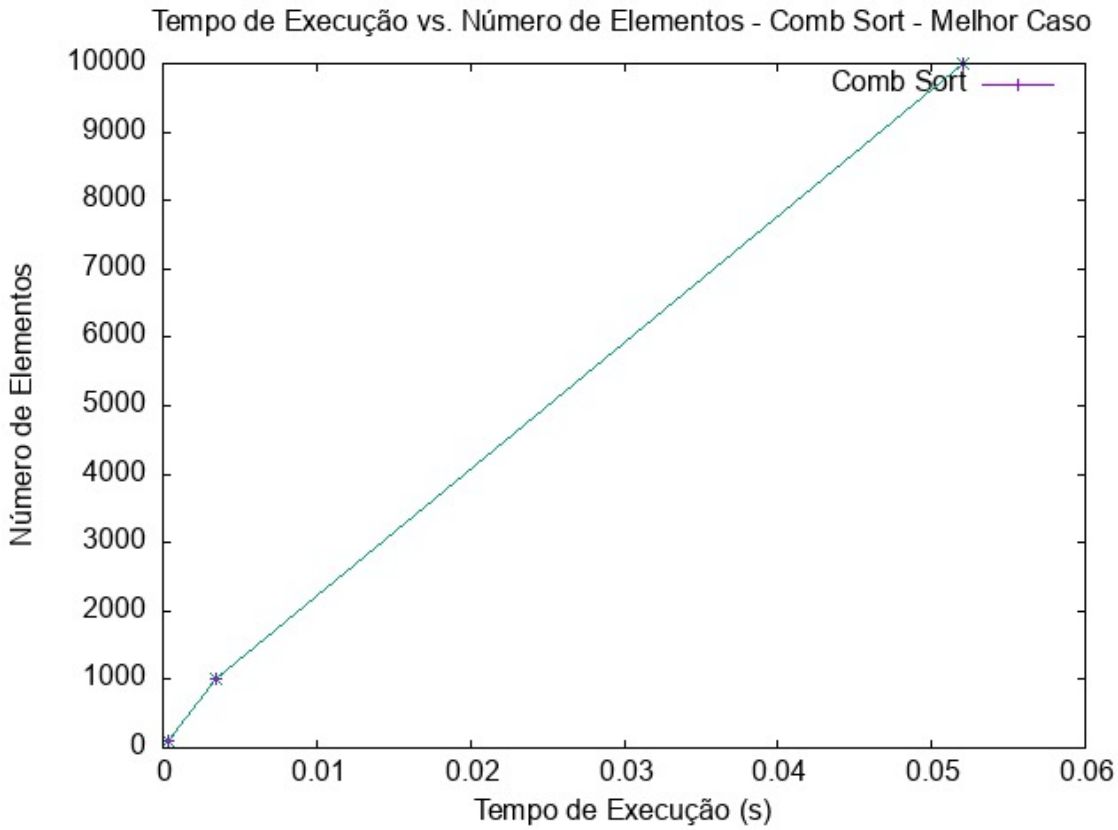

Figura 4 – Comb sort melhor caso

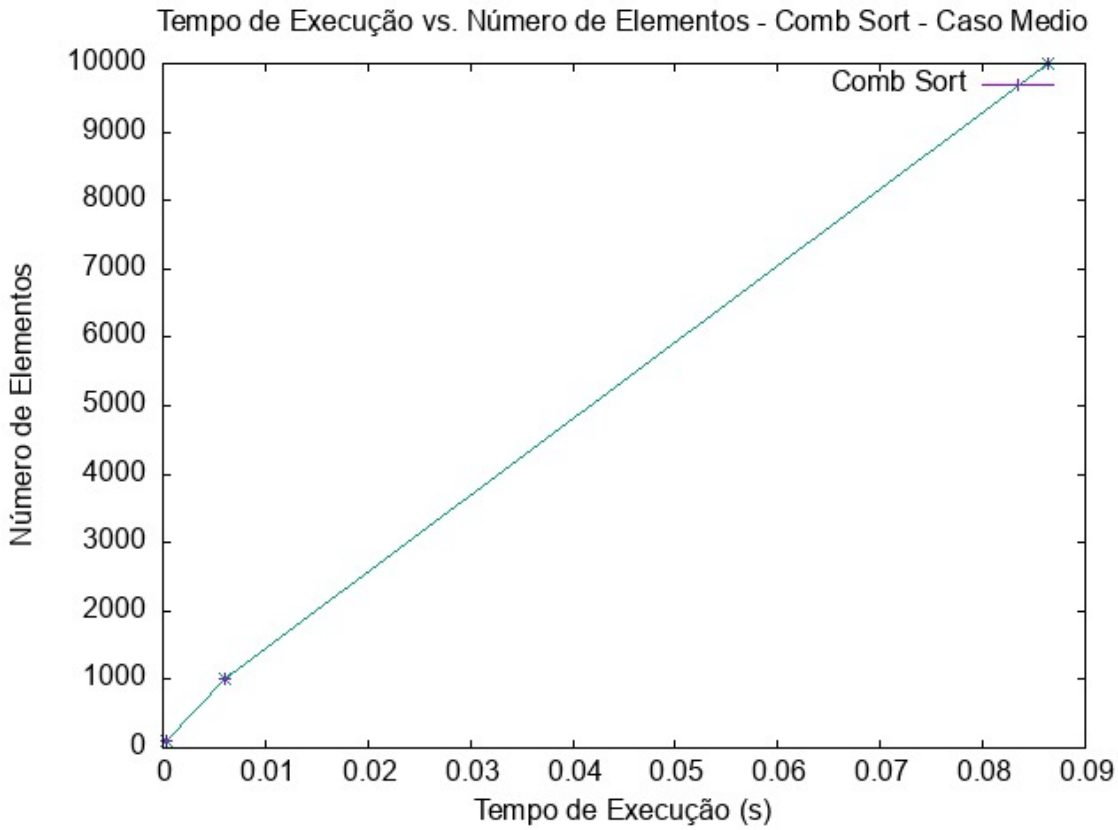

Figura 5 – Comb sort caso médio

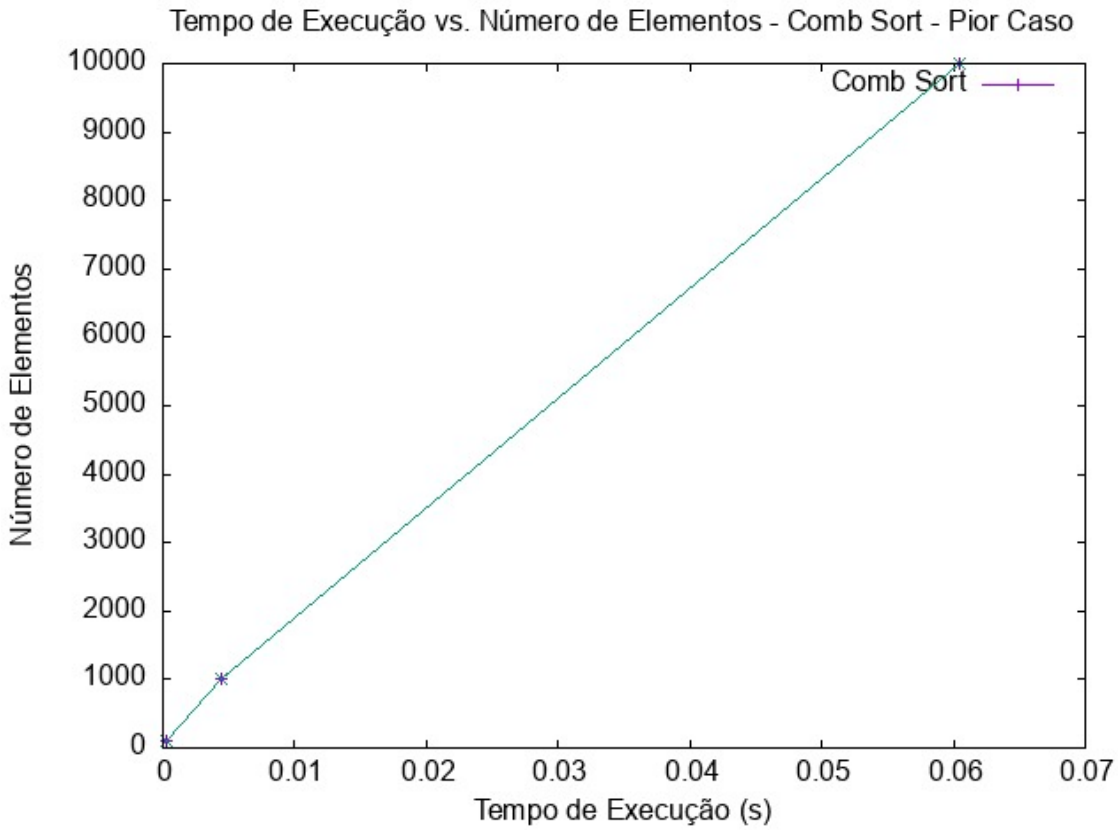

Figura 6 – Comb sort pior caso

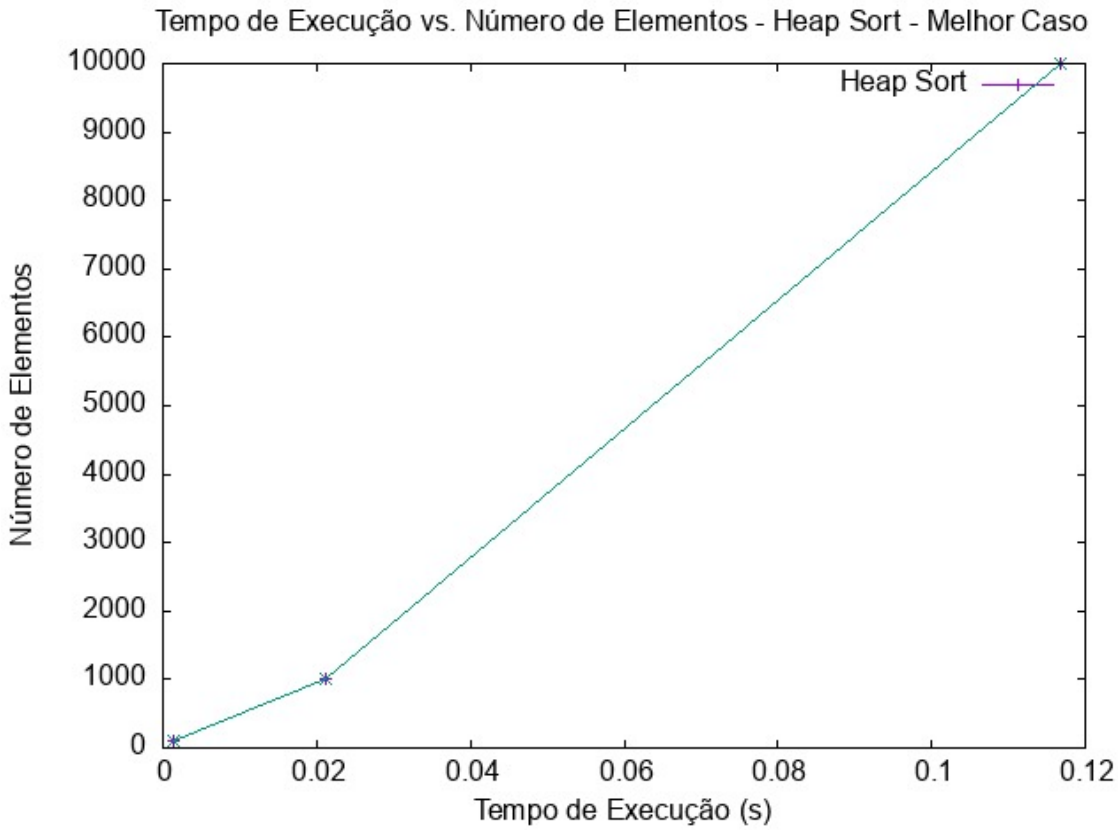

Figura 7 – Heap sort melhor caso

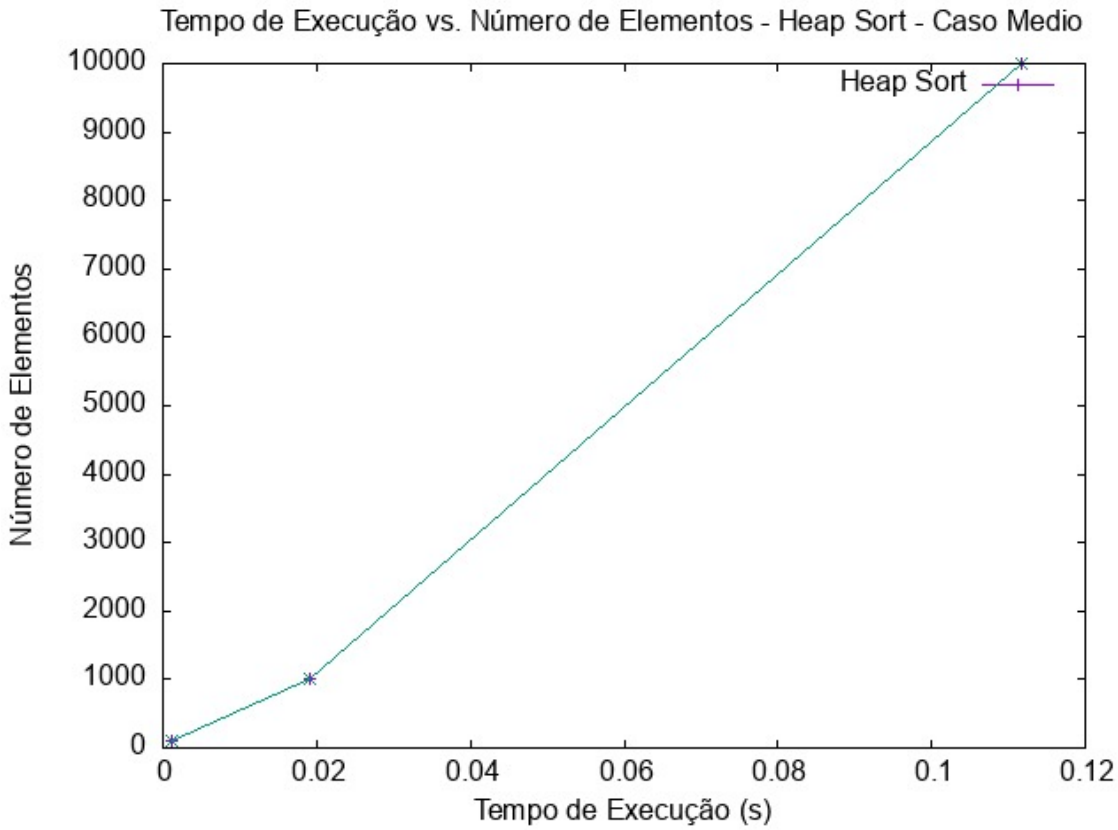

Figura 8 – Heap sort caso médio

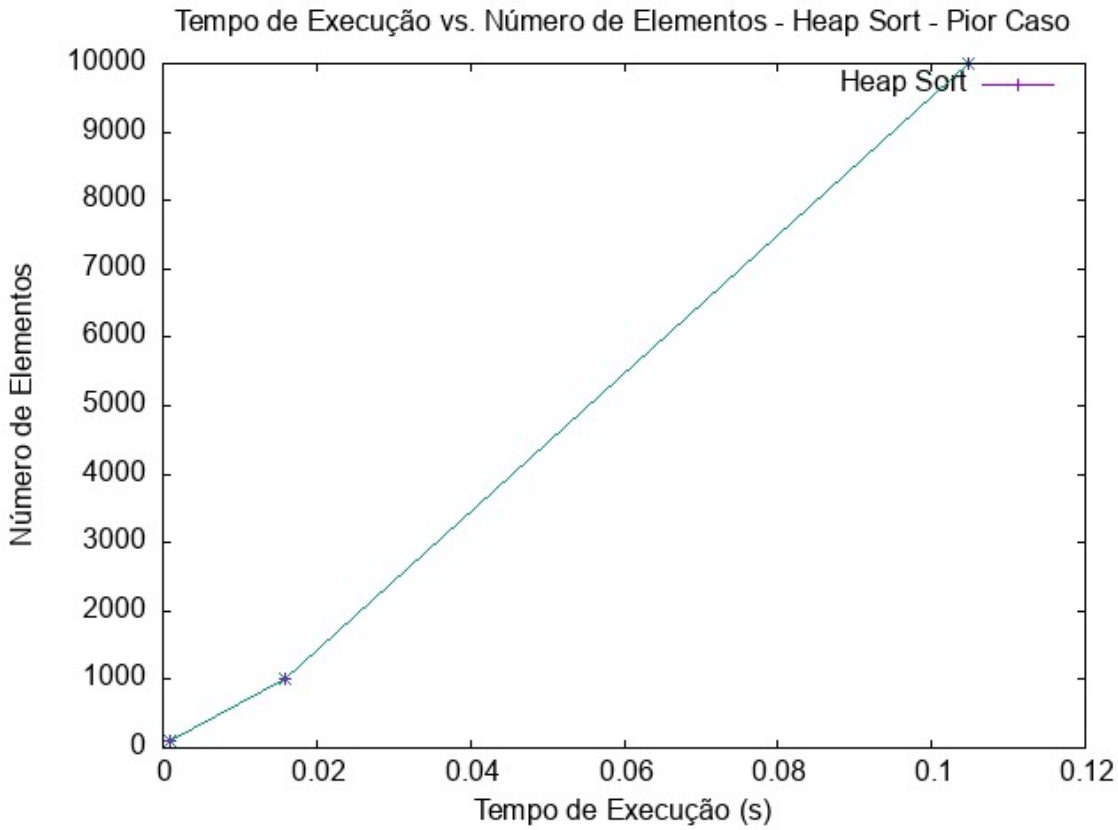

Figura 9 – Heap sort pior caso

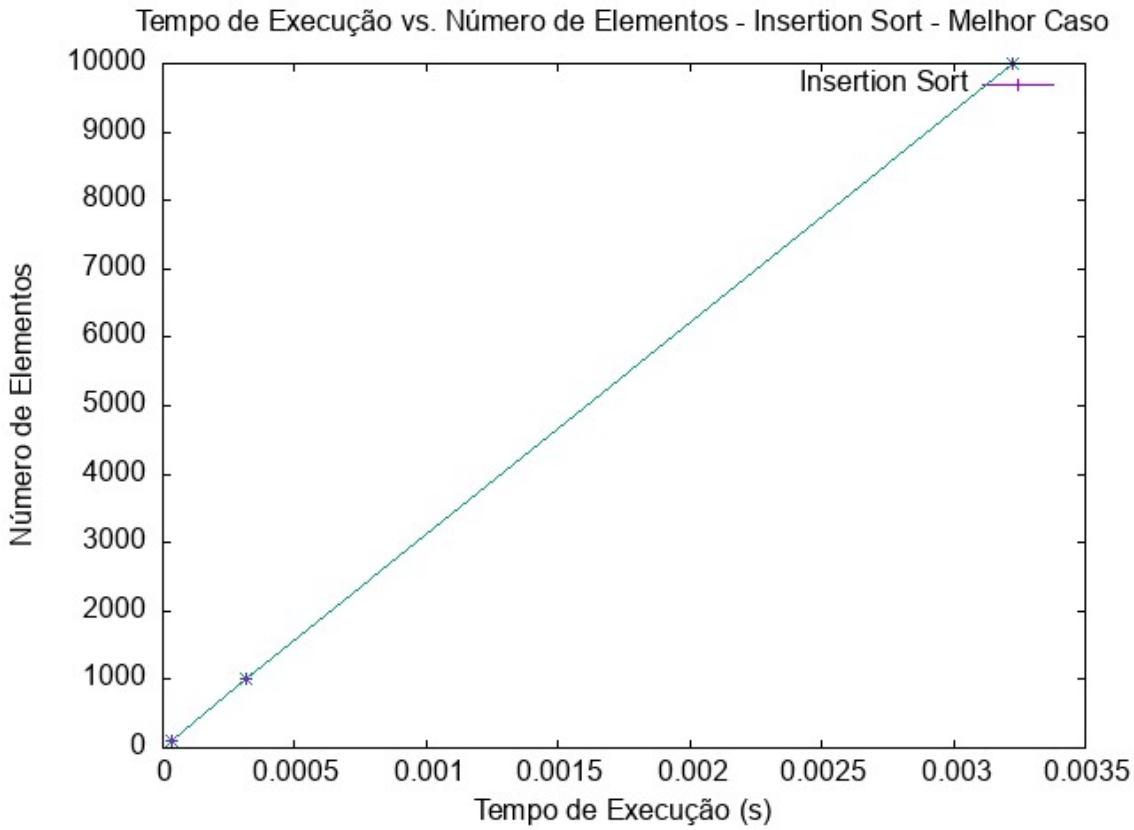

Figura 10 – Insertion sort melhor caso

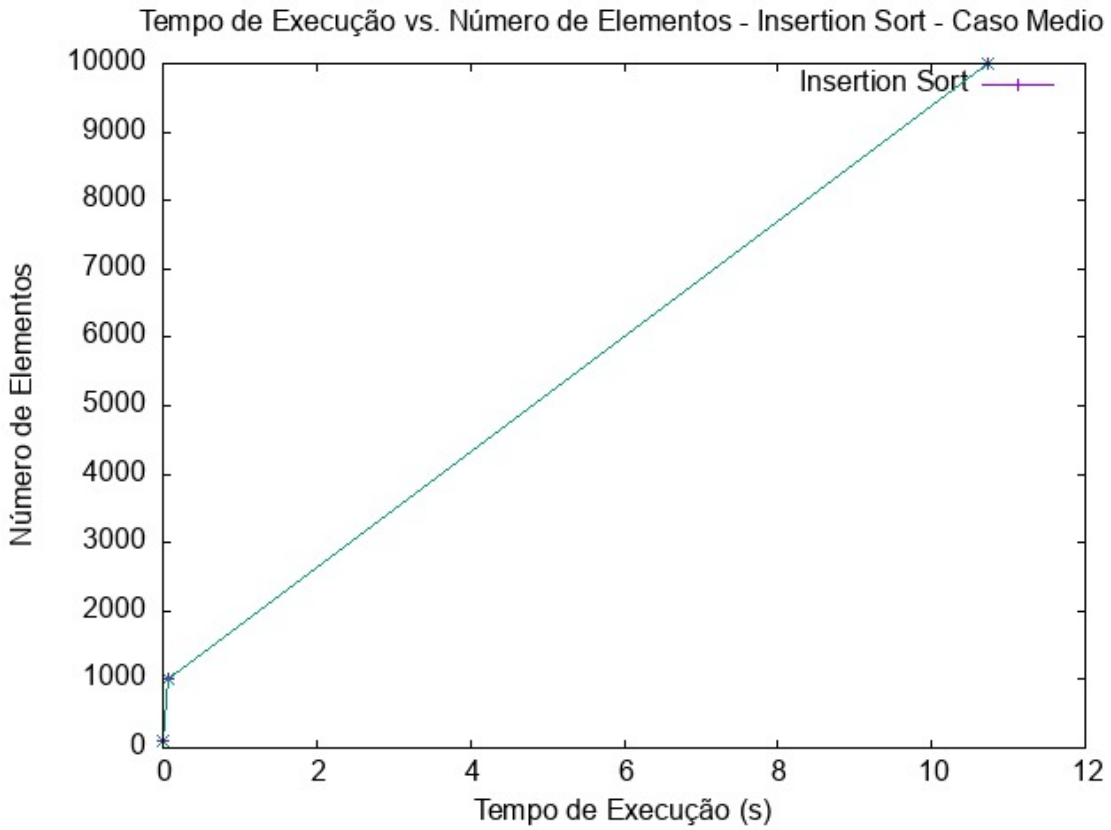

Figura 11 – Insertion sort caso médio

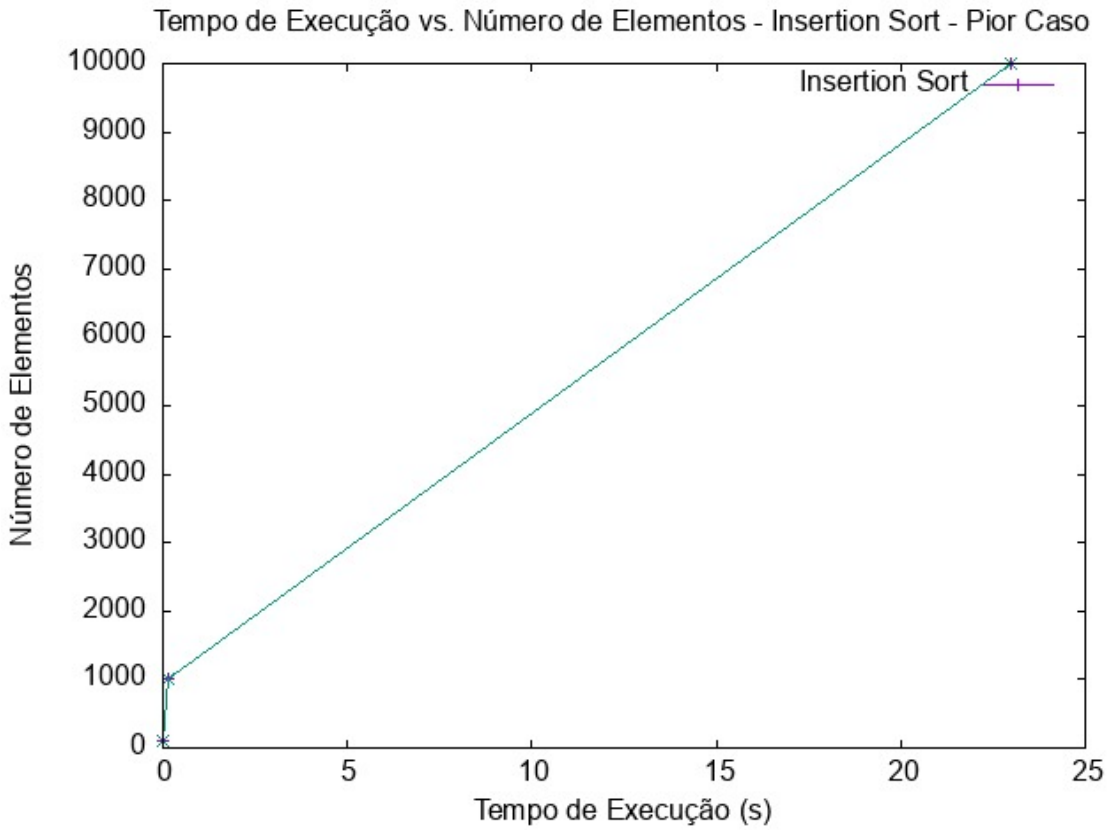

Figura 12 – Insertion sort pior caso

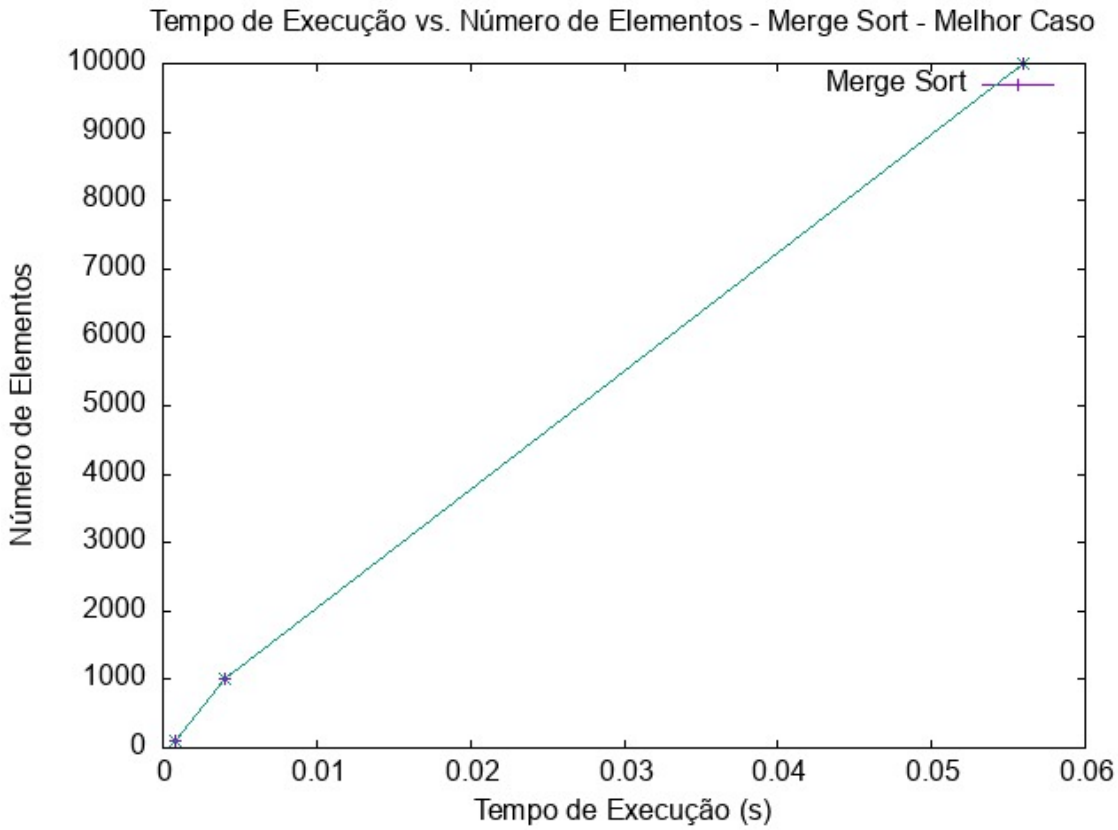

Figura 13 – Melhor sort melhor caso

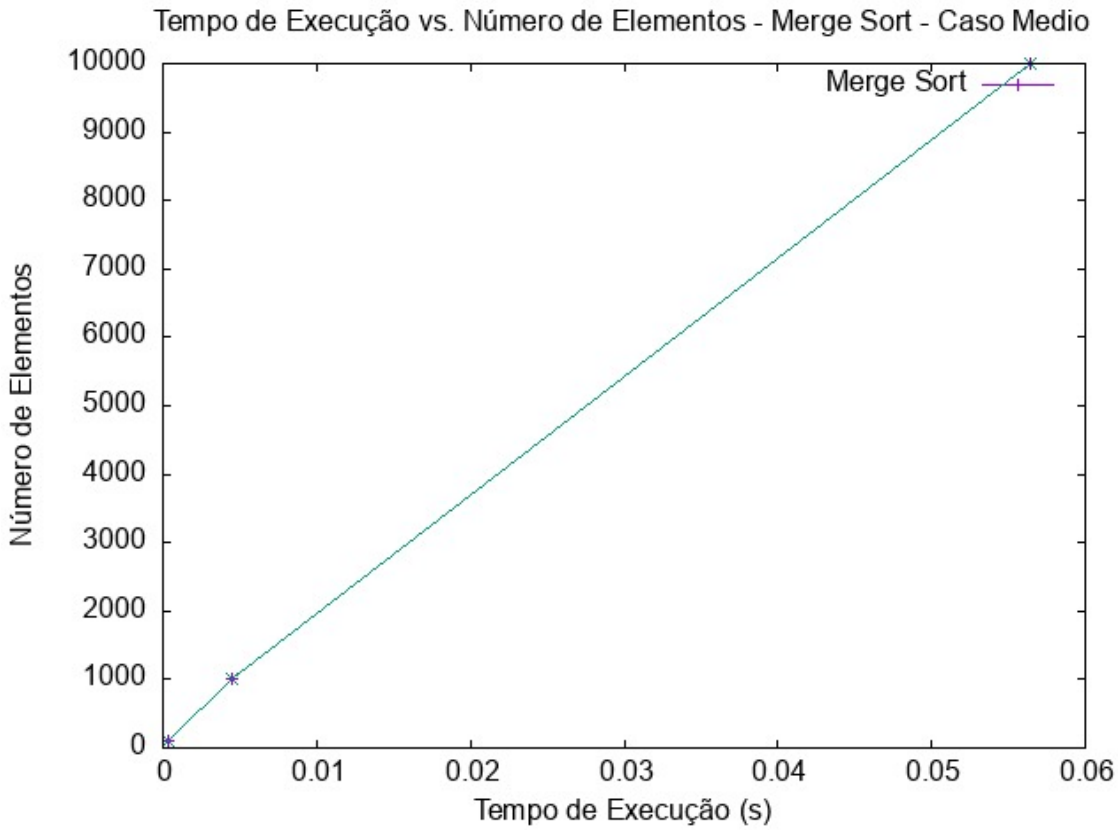

Figura 14 – Melhor sort caso médio

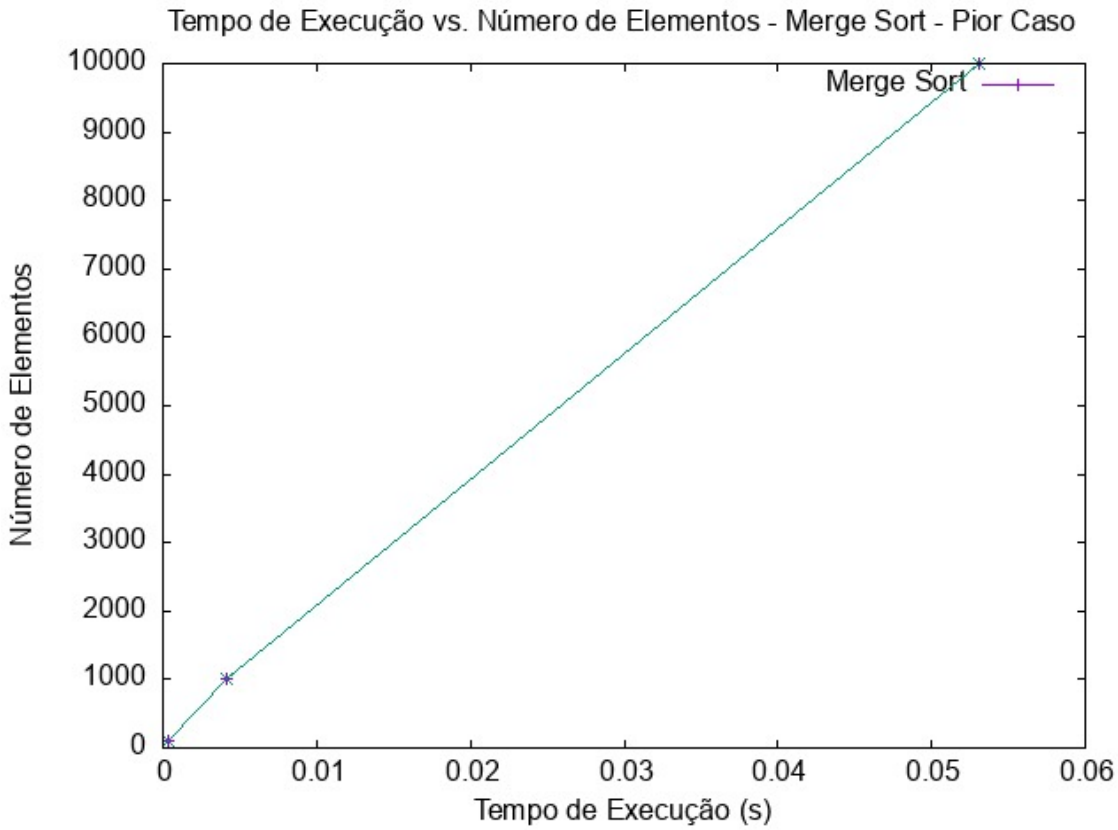

Figura 15 – Melhor sort pior caso

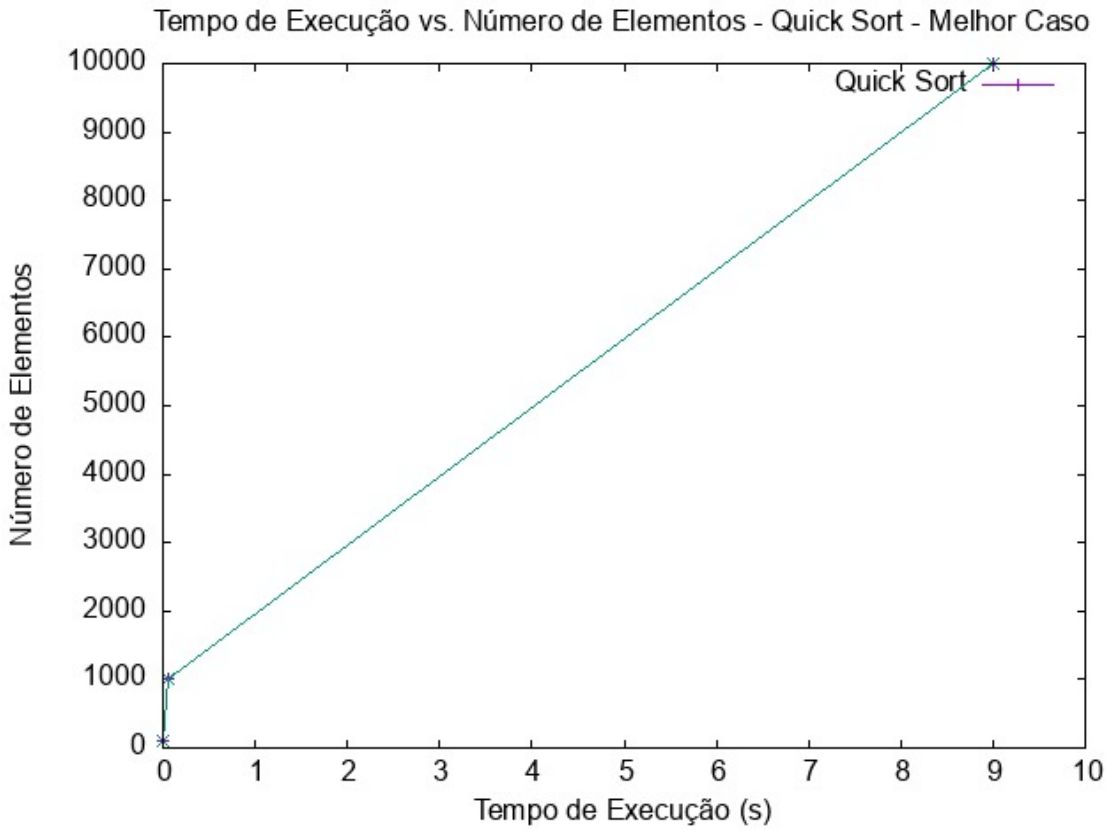

Figura 16 – Quick sort melhor caso

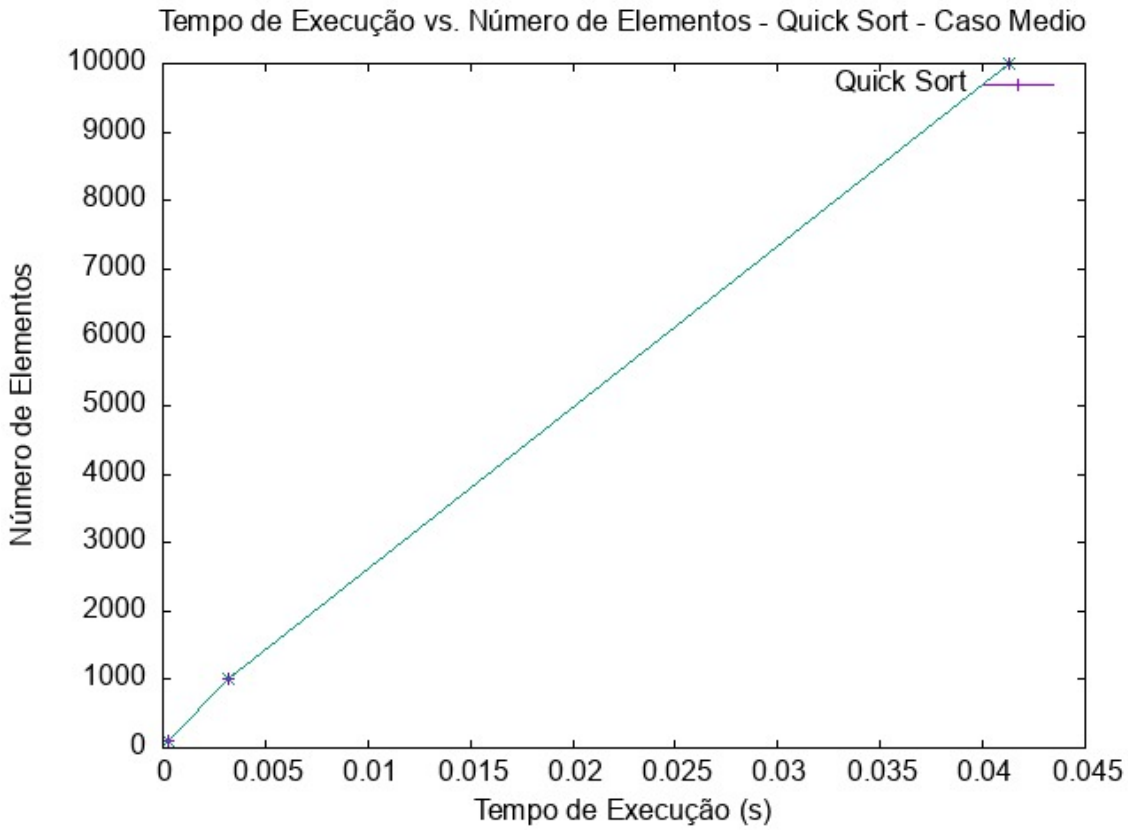

Figura 17 – Quick sort caso médio

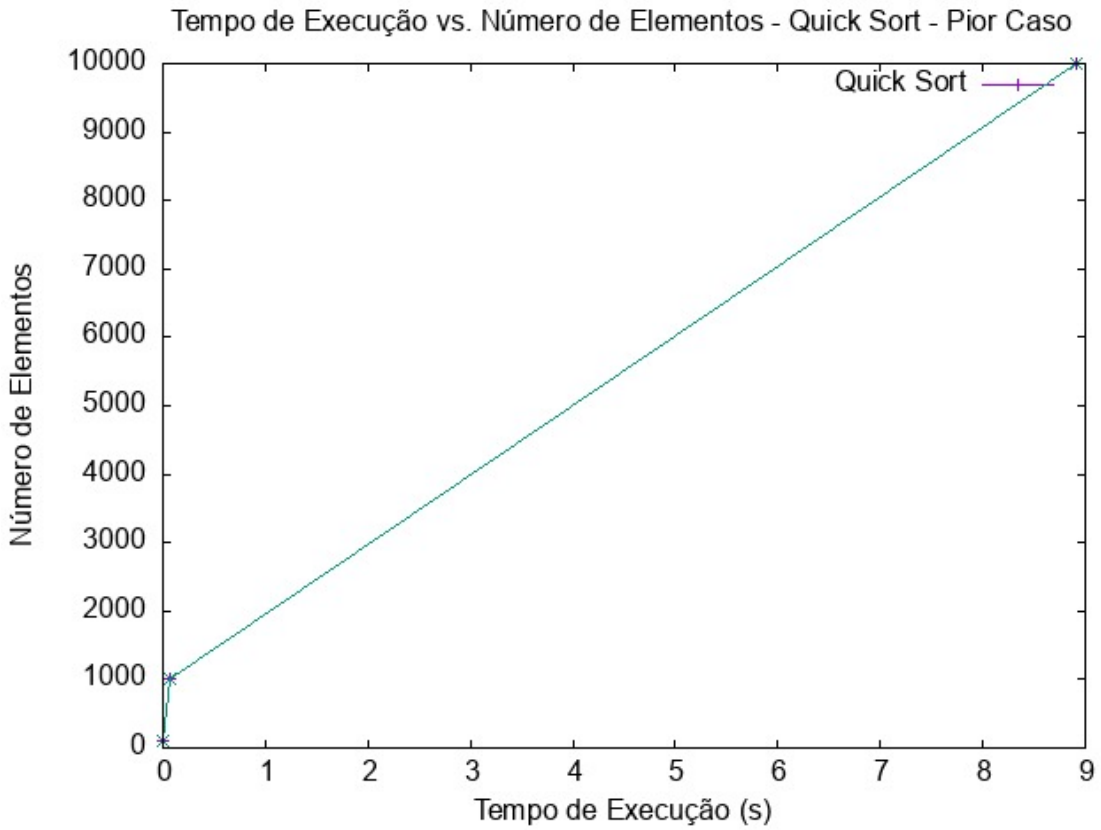

Figura 18 – Quick sort pior caso

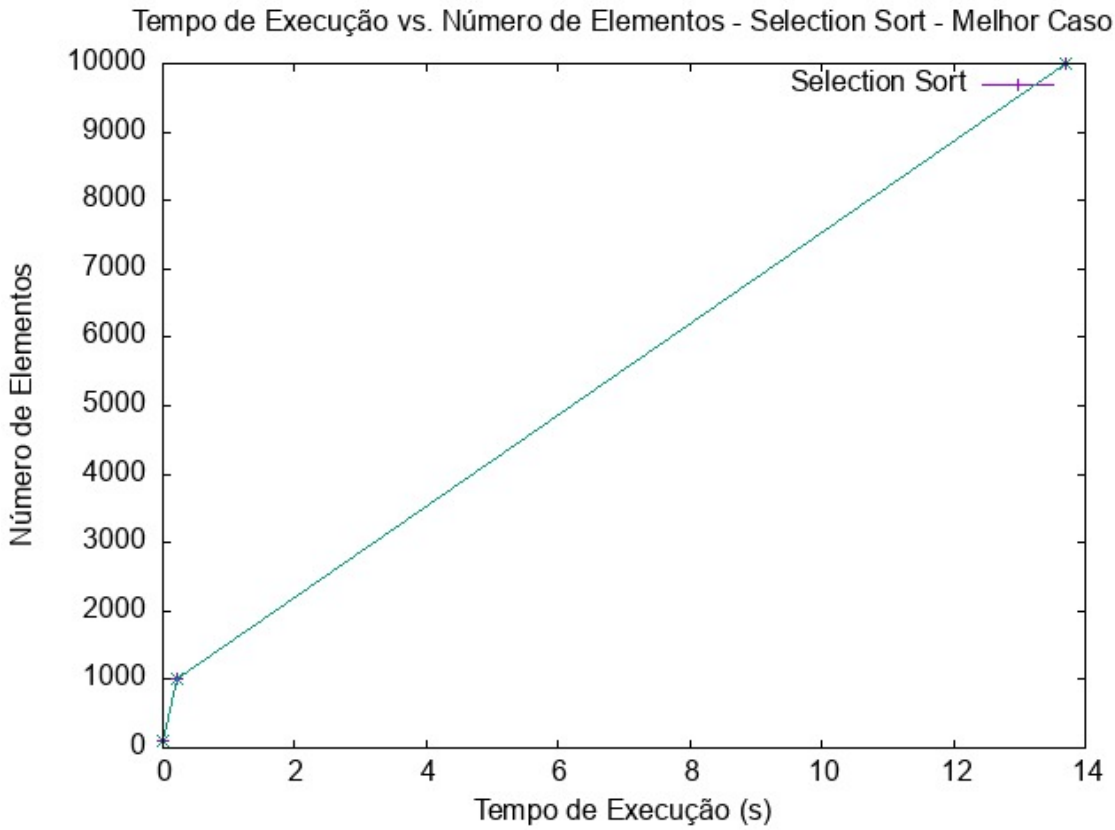

Figura 19 – Selection sort melhor caso

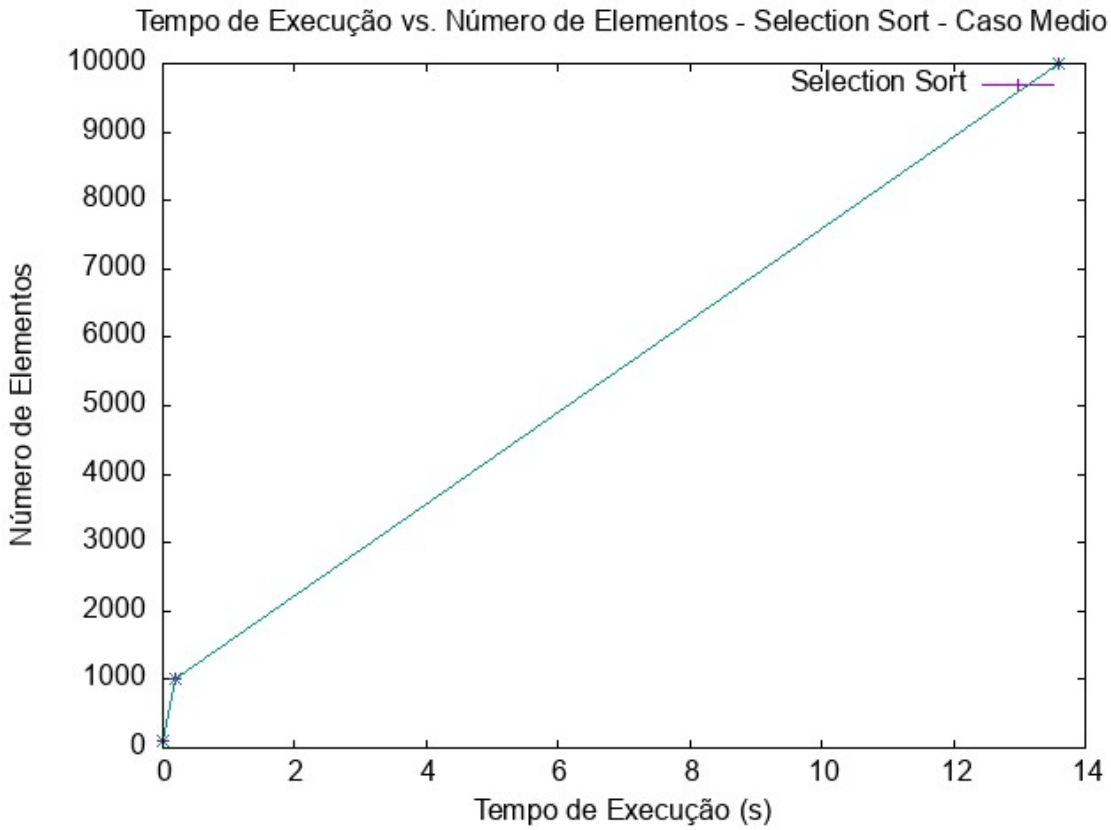

Figura 20 – Selection sort caso médio

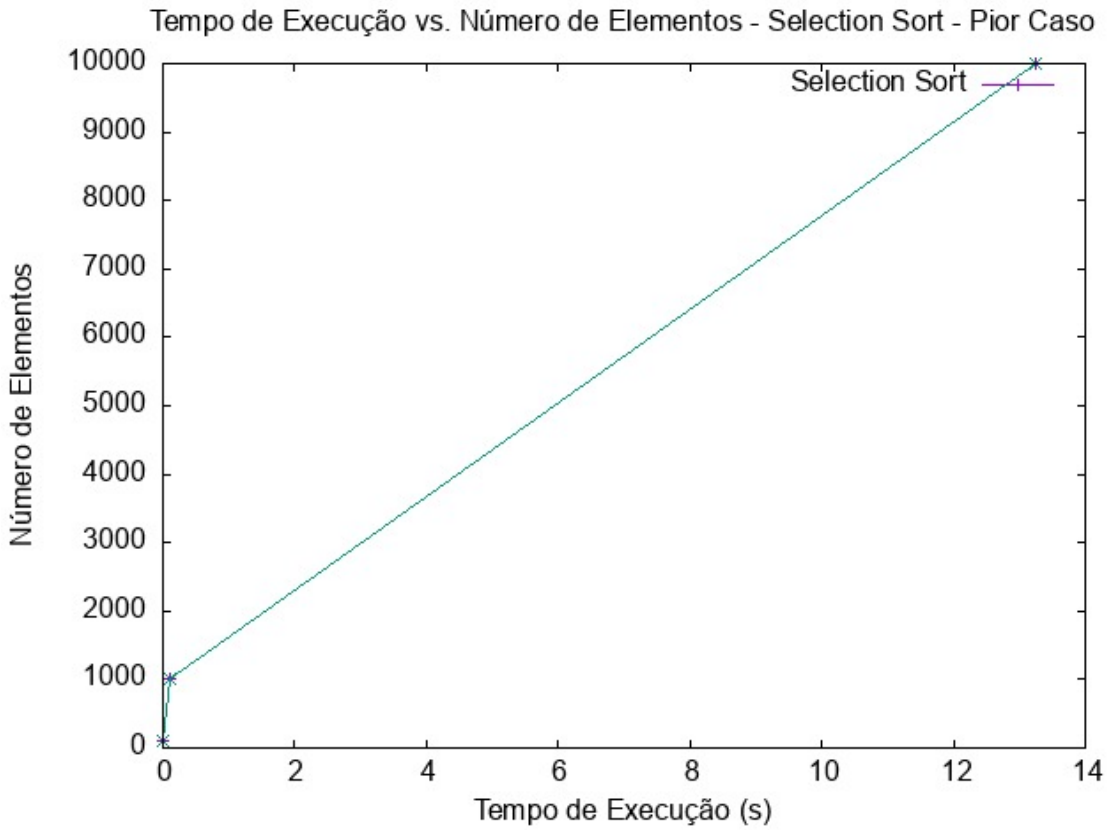

Figura 21 – Selection sort pior caso

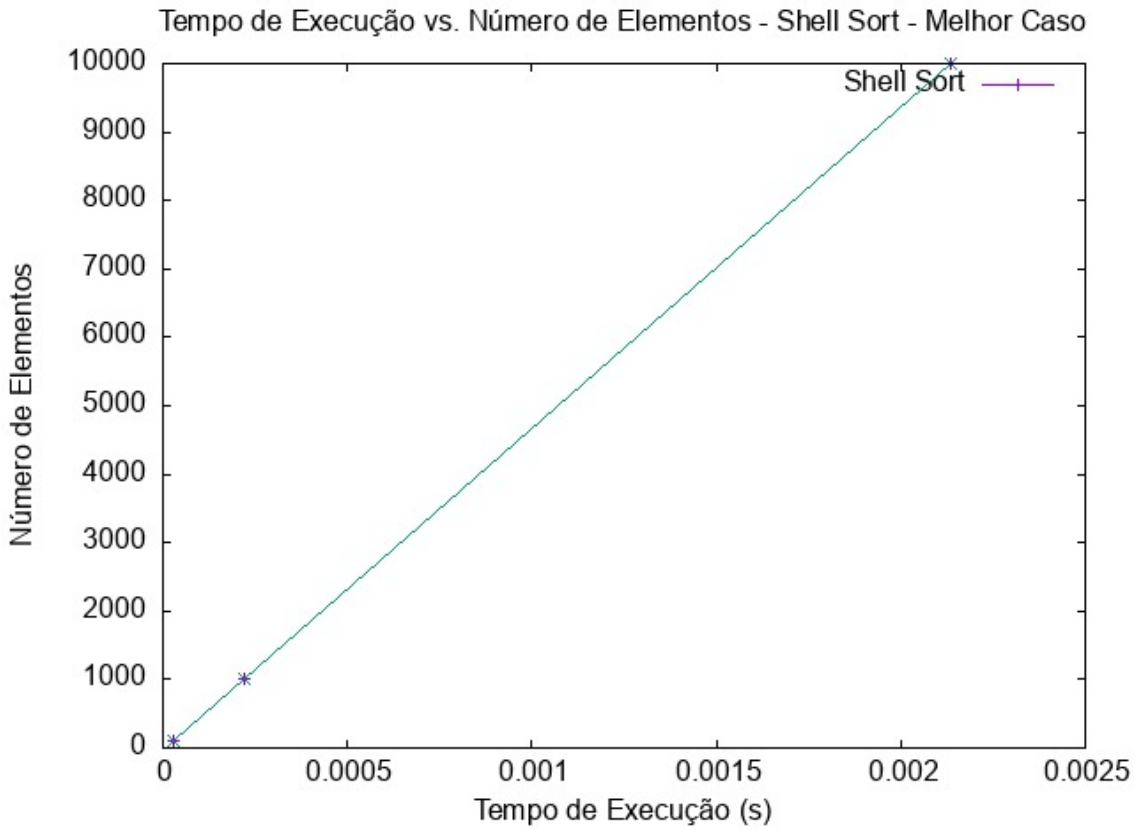

Figura 22 – Shell sort melhor caso

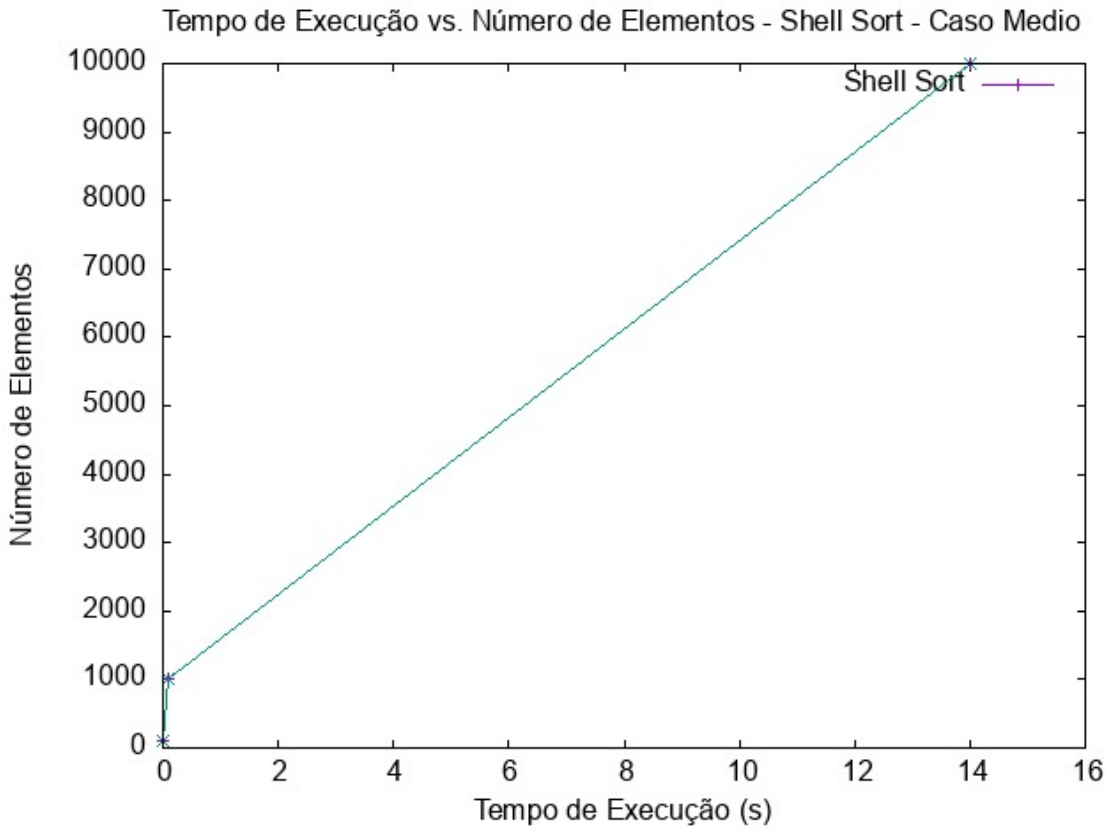

Figura 23 – Shell sort caso médio

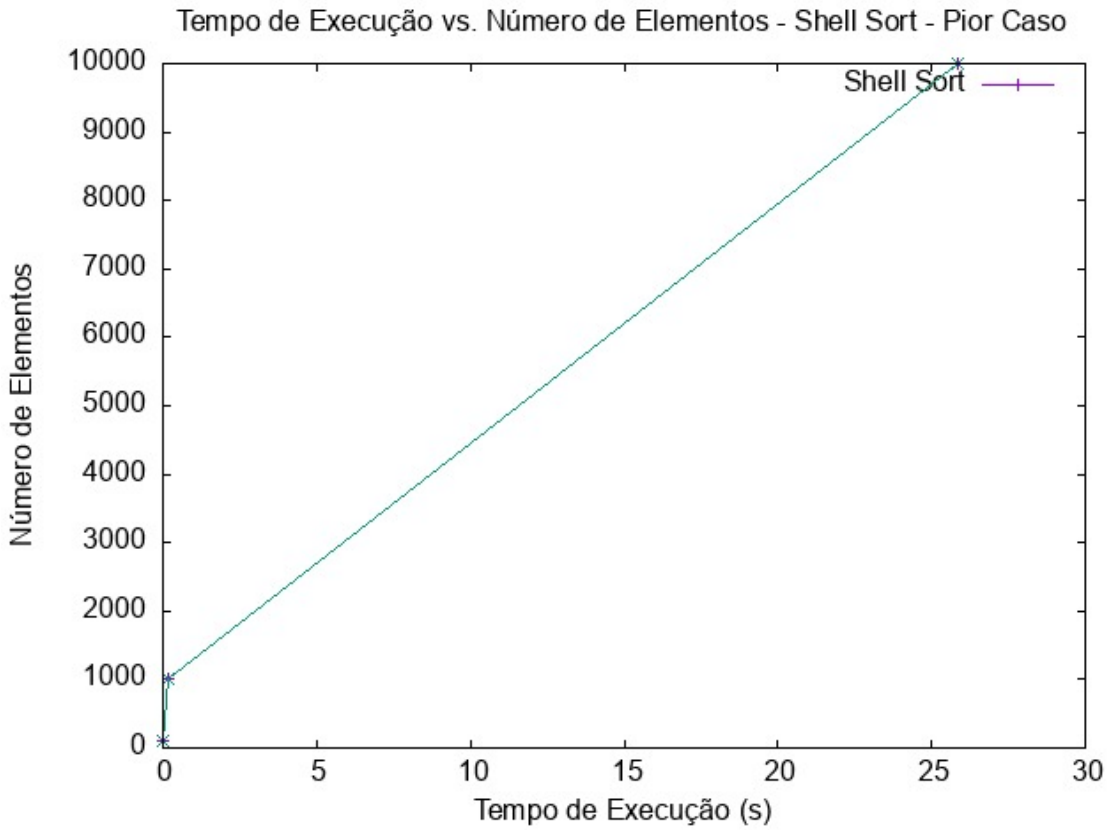

Figura 24 – Shell sort pior caso