Análise e comparação assintótica de algoritmos de ordenação

Disciplia:

Projeto e Análise de Algoritmos

Grupo constituido por:

01 Lucas Evangelista Freire

02 João Gabriel Alves de Souza

03 Larissa Mitie Curi Hirai

04 Fernanda Menezes Plessim de Melo

05 Welton Santana de Andrade Junior

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.

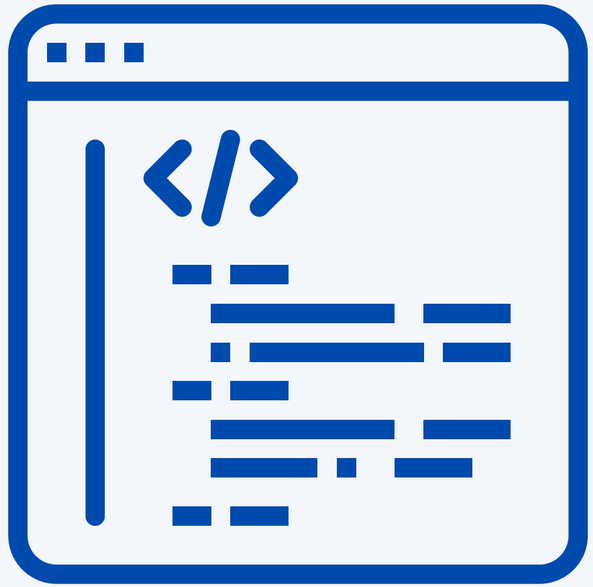
Ao longo do artigo pontos como a relação de recorrência de cada algoritmos, complexidade e tempo de execução serão comparadas visando mostrar como eles se comportam dadas entradas já ordenadas de forma crescente, aleatórias e ordenadas de forma decrescente, assim obtendo os tempos de melhor caso, caso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

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



Algoritmos:

Bubble sort, Insertion sort, Quick sort, Merge sort, Shell sort, Comb sort, Heap sort, Selection sort.

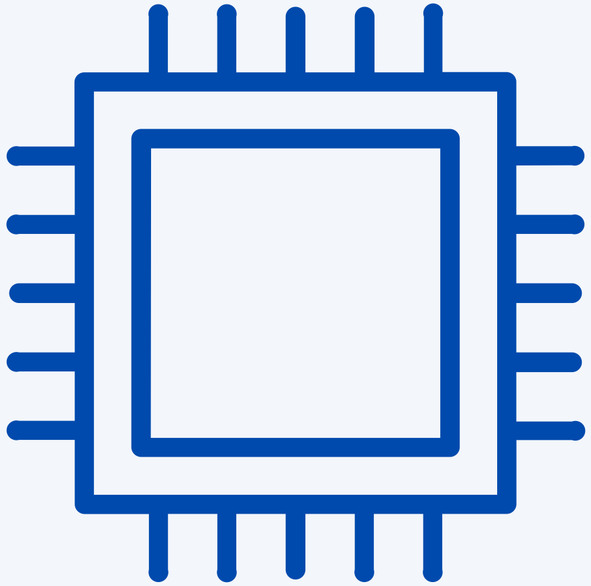
Especificações de hardware

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
RAM : 512 Mb



Bubble sort

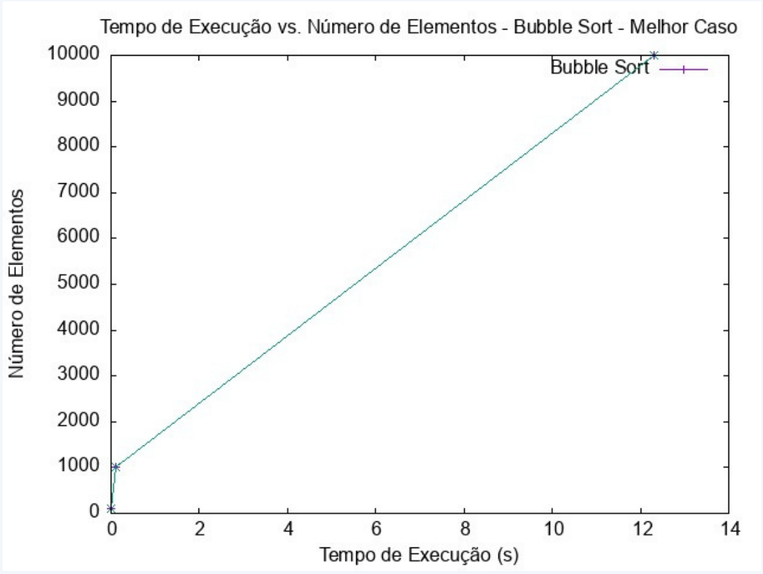
```
function bubbleSort (int array) {
    int m, n, swap;
    for (m = 0; m < array - m; m ++) {
        for (n = 0; n < array - (m + 1); n ++)
            if (array [n] > array [n + 1]) {
                swap = array [n];
                array [n] = array [n + 1];
                array [n + 1] = swap
            }
        }
    }
    return array;
}
```

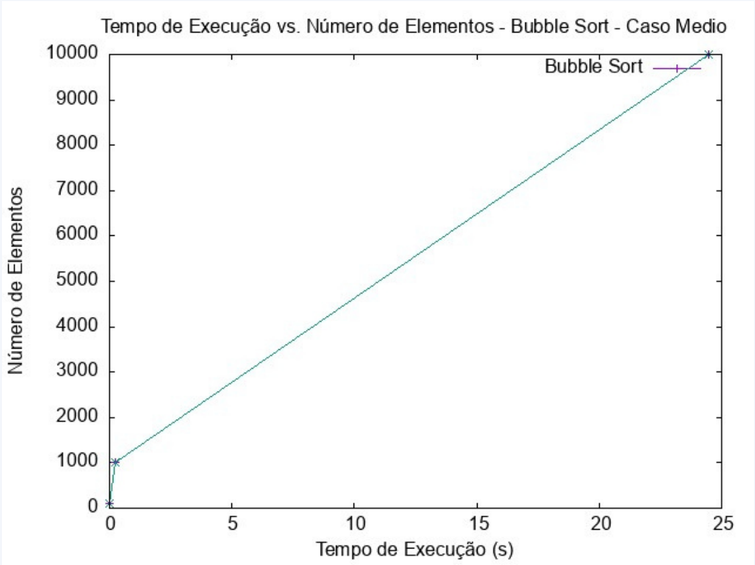
- Um algoritmo de ordenação simples e intuitivo.
- Baseia-se na comparação de pares adjacentes de elementos.
- Fácil de entender e implementar.
- Estável, ou seja, preserva a ordem de elementos com mesmo valor
- O algoritmo no pior caso (array invertido) e no caso médio (array aleatório) tem complexidade 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 O(n).

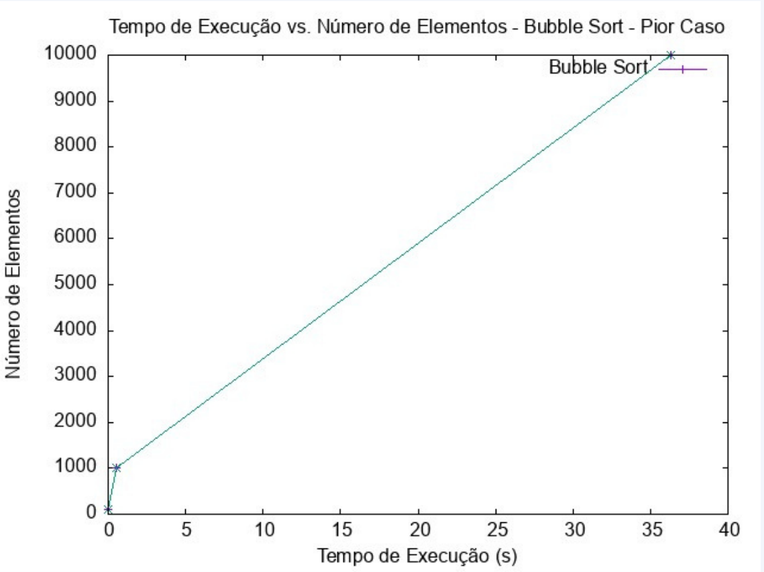
Bubble sort

Casos	Tempo de execução	Número de trocas
Melhor caso com 100 posições	0.0017514200000800863	0
Melhor caso com 1000 posições	0.27296823500000755	0
Melhor caso com 10000 posições	31.093445957000768	0
Caso médio com 100 posições	0.0025462390000366768	2333
Caso médio com 1000 posições	0.5836652469997716	250251
Caso médio com 10000 posições	57.047481122000136	24966640
Pior caso com 100 posições	0.0037950099999521426	4950
Pior caso com 1000 posições	0.7202099539999836	499500
Pior caso com 10000 posições	79.82543029200042	49995000

Bubble sort


Tempo de Execução vs. Número de Elementos - Bubble Sort - Melhor Caso
Bubble Sort
Número de Elementos
Tempo de Execução (s)
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
0
2
4
6
8
10
12
14


Tempo de Execução vs. Número de Elementos - Bubble Sort - Caso Medio
Bubble Sort
Número de Elementos
Tempo de Execução (s)
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
0
5
10
15
20
25


Tempo de Execução vs. Número de Elementos - Bubble Sort - Pior Caso
Bubble Sort
Número de Elementos
Tempo de Execução (s)
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
0
5
10
15
20
25
30
35
40

Selection Sort

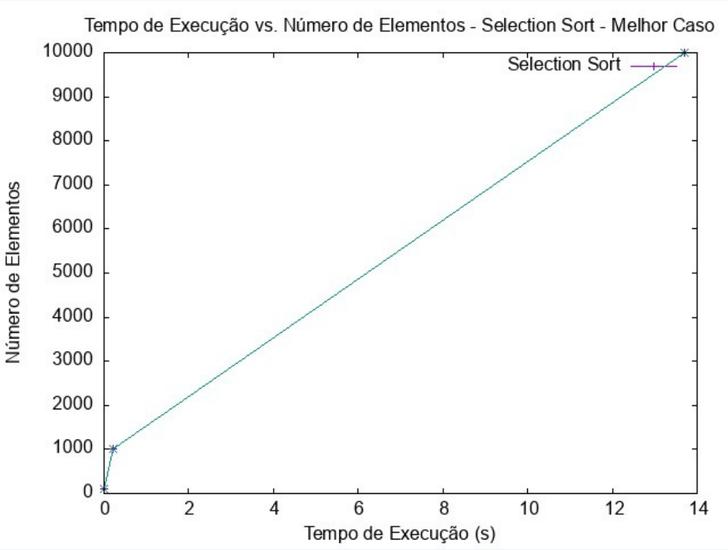
```
Function SelectionSort(array[]):
    int i, j, menor, aux;
    for (i ← 0; i < array; i ++) do
        menor ← i;
        for (j ← i + 1; j < array; j ++) do
            if (array [j] < array [menor]) then
                menor ← j;
            end
        end
        if (array [i] ! = array [menor])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

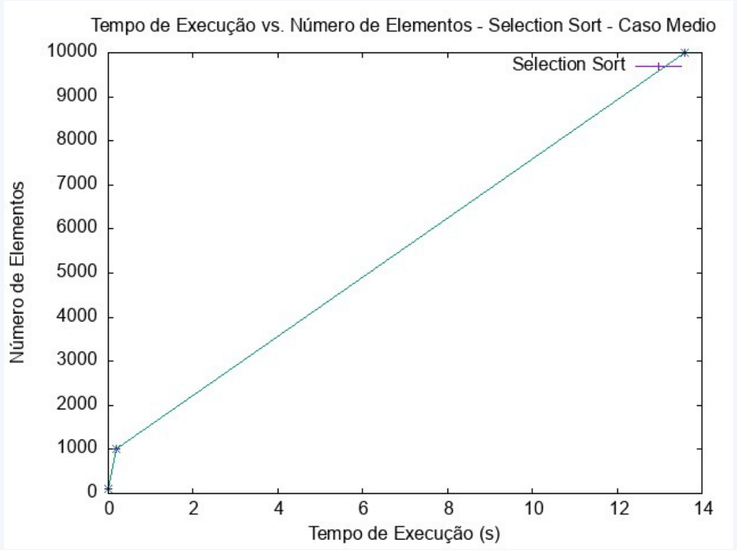
- Um algoritmo de ordenação simples que seleciona o elemento de menor valor (ou maior valor) e o coloca na posição correta na lista
- Não requer movimentação repetitiva de elementos na lista.
- O algoritmo no pior caso (array invertido), caso médio (array aleatório) e melhor caso (array ordenado) tem complexidade n2 e tem por relação de recorrência o seguinte :
    - T (n) = T (n - 1) + n - 1

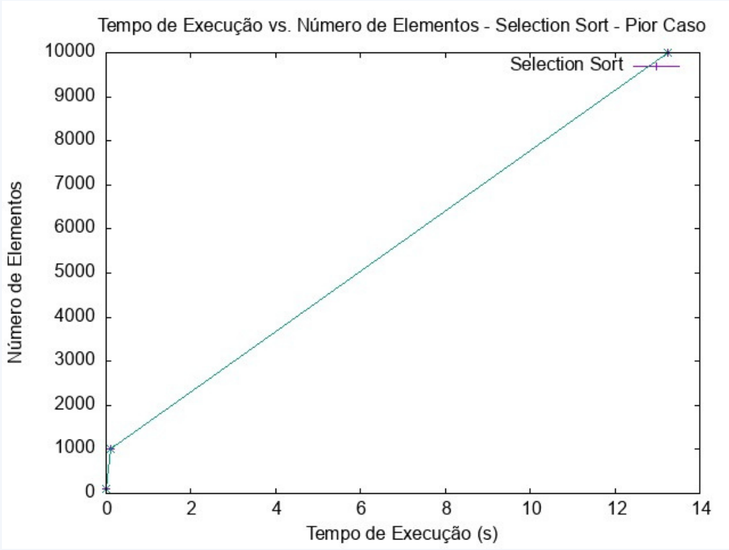
Selection Sort

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

Selection Sort







Insertion sort

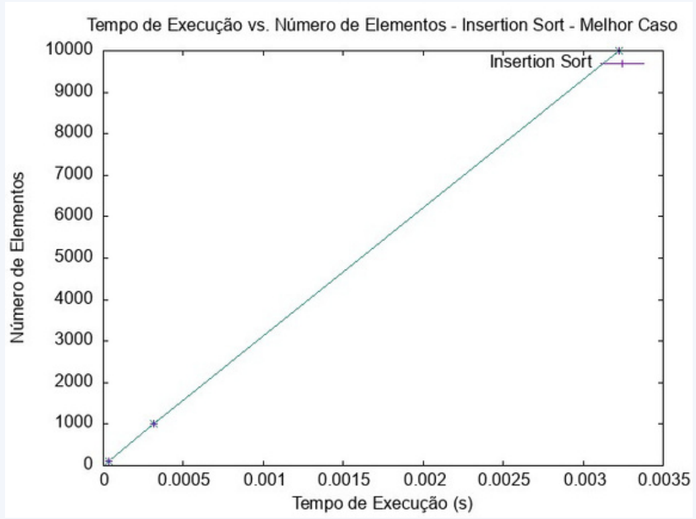
```
Function InsertionSort(array[]):
    int i, j, key;
    for (i ← 1; i < array; i ++) do
        key ← array [i];
        j ← i - 1;
        while j ≥ 0 and key < array [j] do
            array [j + 1] ← array [j];
            j ← j - 1;
            array [j + 1] ← key;
        end
    end
    return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

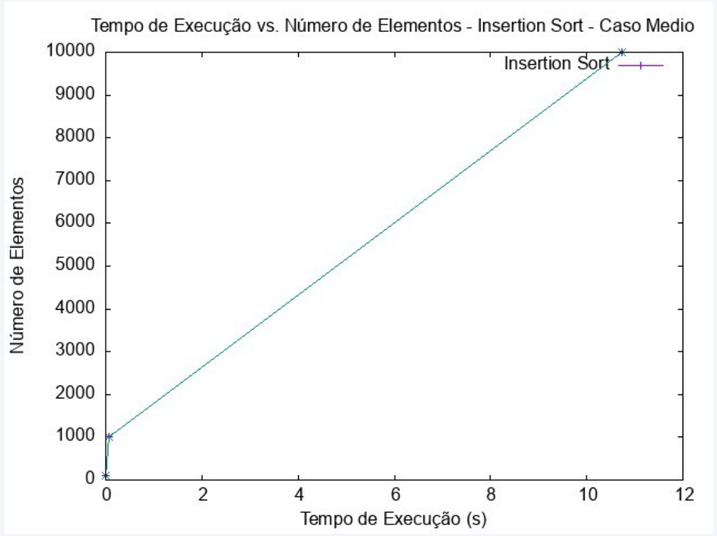
- Um algoritmo de ordenação que se baseia na ideia de ordenar elementos como se estivesse organizando um baralho de cartas.
- Estável, ou seja, preserva a ordem de elementos com mesmo valor
- Fácil de entender e implementar.
- Eficiente para pequenas listas ou listas parcialmente ordenadas.
- O algoritmo 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 O(n)

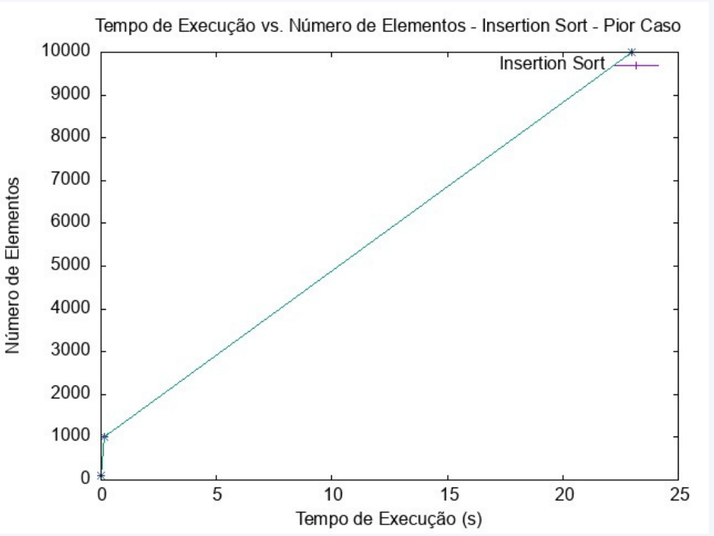
Insertion sort

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

Insertion sort







```
Function QuickSort(Array[], Inicio, 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
```

```
Function QuickSortIterative(Array[], Inicio, Fim):
    Stack stack;
    stack.push((Inicio, Fim));
    while stack is not 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
Function Partition(Array[], Inicio, Fim):
    pivo ← Array[Fim];
    i ← Inicio − 1;
    for j ← Inicio to Fim − 1 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;
```

Quick sort

- Um algoritmo eficiente de ordenação baseado na estratégia "dividir e conquistar".
- Amplamente utilizado na prática devido à sua velocidade e simplicidade.

Quick sort

- 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 ≤ n/2, representado pela relação:

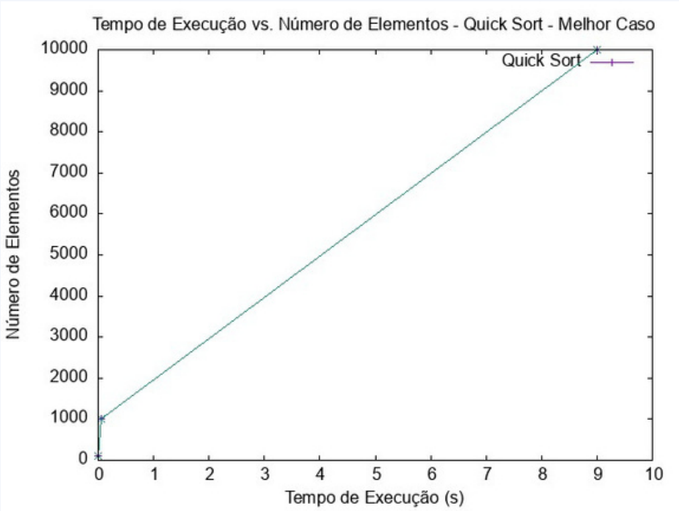
$$T(n) <= 2T(n/2) + \theta(n)$$

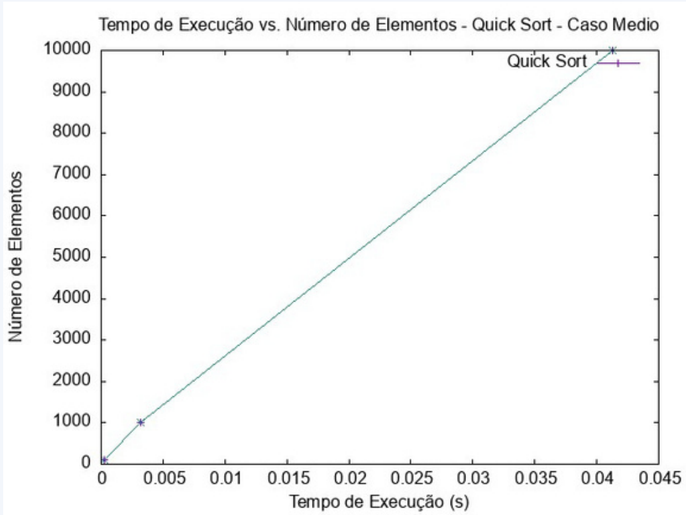
- Portanto, A complexidade do quickSort é O(n2), no pior caso e O(nlogn) , no melhor caso e no caso medio.

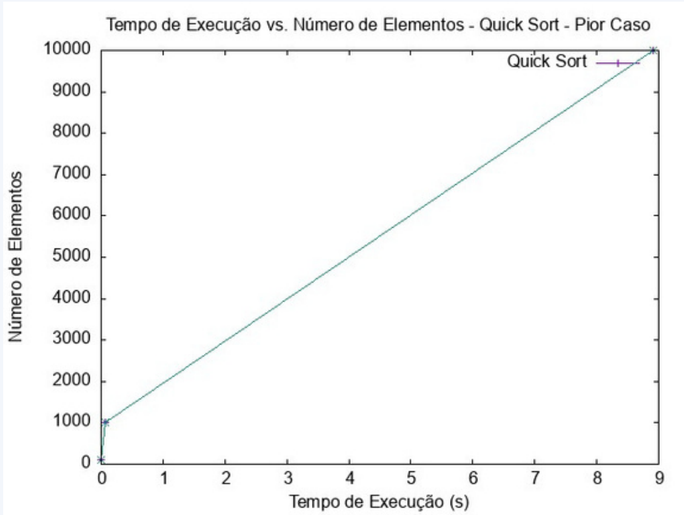
Quick sort

Casos	Tempo de execução	Número de trocas
Melhor caso com 100 posições Melhor caso com 1000 posições Melhor caso com 10000 posições	0.0009836200006247964 0.180424365000036274 21.367593666000175	99 999 9999
Caso médio com 100 posições Caso médio com 1000 posições Caso médio com 10000 posições	0.002989118998812046 0.0029637789994012564 0.10647092100043665	161 2364 30721
Pior caso com 100 posições Pior caso com 1000 posições Pior caso com 10000 posições	0.0010493189984117635 0.19795522300046287 20.90247191399976	99 999 9999

Quick sort


Tempo de Execução vs. Número de Elementos - Quick Sort - Melhor Caso
Quick Sort
Número de Elementos
Tempo de Execução (s)
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
0
1
2
3
4
5
6
7
8
9
10


Tempo de Execução vs. Número de Elementos - Quick Sort - Caso Medio
Quick Sort
Número de Elementos
Tempo de Execução (s)
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
0
0.005
0.01
0.015
0.02
0.025
0.03
0.035
0.04
0.045


Tempo de Execução vs. Número de Elementos - Quick Sort - Pior Caso
Quick Sort
Número de Elementos
Tempo de Execução (s)
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
0
1
2
3
4
5
6
7
8
9

```
Function MergeSort(array[]):
    int mid, left, right, i, j, k;
    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
```

Merge sort

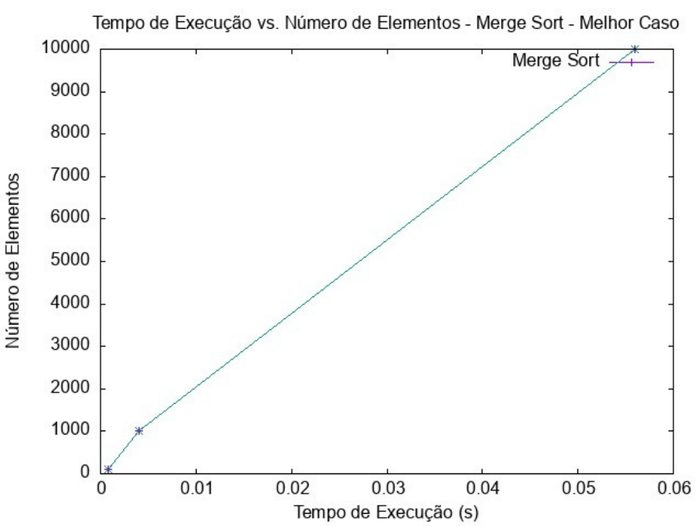
- Assim como o quick sort se baseia na estratégia de "dividir para conquistar".
- Eficiente para grandes quantidades de dados.
- A complexidade do merge sort para o pior caso (array invertida), caso médio (array aleatória) e melhor caso (array ordenada) é nlogn e sua relação de recorrência para todos os casos é de :

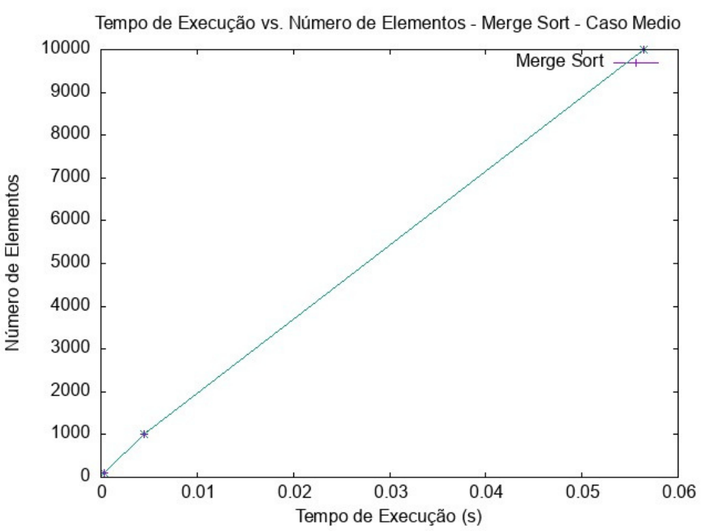
$$T(n) = 2 \times T(n/2) + O(n)$$

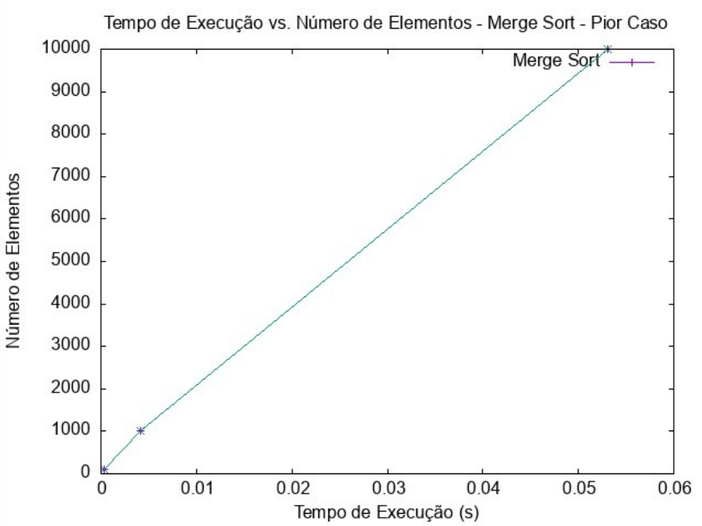
Merge Sort

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

Merge Sort


Tempo de Execução vs. Número de Elementos - Merge Sort - Melhor Caso
Merge Sort
Número de Elementos
Tempo de Execução (s)
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
0
0.01
0.02
0.03
0.04
0.05
0.06


Tempo de Execução vs. Número de Elementos - Merge Sort - Caso Medio
Merge Sort
Número de Elementos
Tempo de Execução (s)
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
0
0.01
0.02
0.03
0.04
0.05
0.06


Tempo de Execução vs. Número de Elementos - Merge Sort - Pior Caso
Merge Sort
Número de Elementos
Tempo de Execução (s)
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
0
0.01
0.02
0.03
0.04
0.05
0.06

```
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);

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;
```

Heap sort

- Um algoritmo que se baseia na estrutura de dados 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.

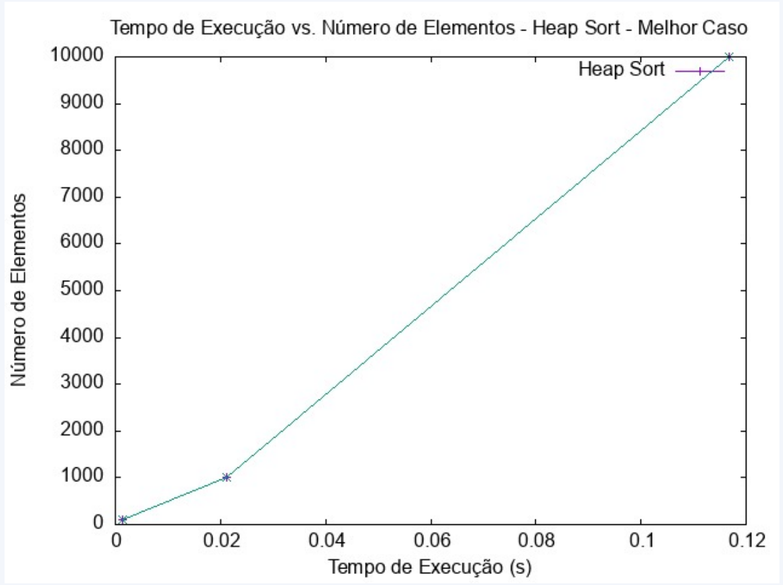
- A complexidade do merge sort para o pior caso (array invertida), caso médio (array aleatória) e melhor caso (array ordenada) é nlogn e sua relação de recorrência para todos os casos é de :

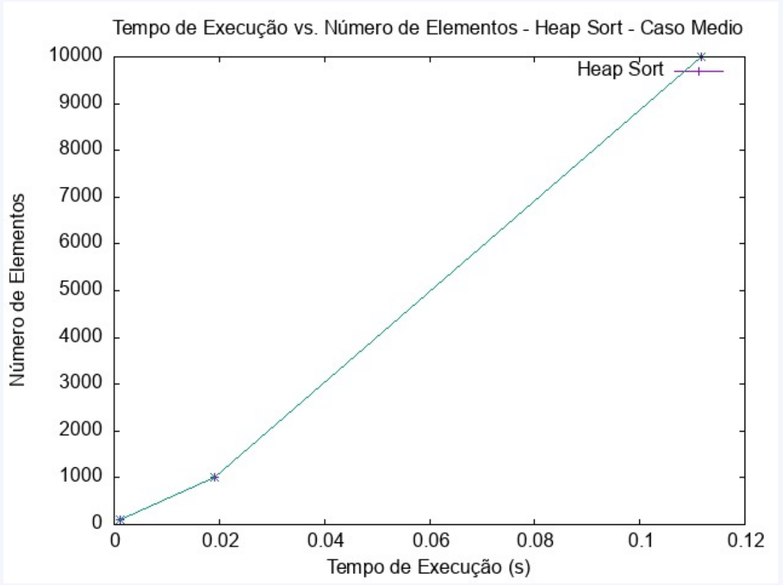
$$T(n) = 2 \times T(n/2) + O(\log n)$$

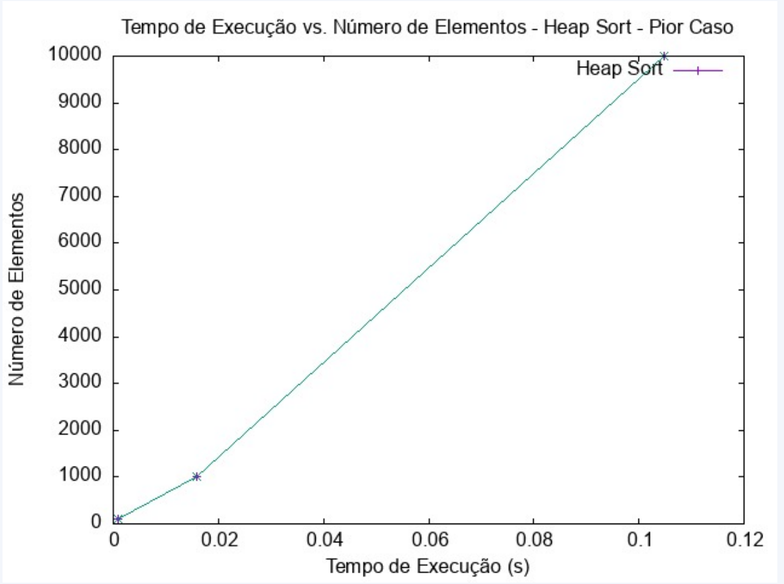
Heap Sort

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

Heap Sort


Tempo de Execução vs. Número de Elementos - Heap Sort - Melhor Caso
Heap Sort
Número de Elementos
Tempo de Execução (s)
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
0
0.02
0.04
0.06
0.08
0.1
0.12


Tempo de Execução vs. Número de Elementos - Heap Sort - Caso Medio
Heap Sort
Número de Elementos
Tempo de Execução (s)
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
0
0.02
0.04
0.06
0.08
0.1
0.12


Tempo de Execução vs. Número de Elementos - Heap Sort - Pior Caso
Heap Sort
Número de Elementos
Tempo de Execução (s)
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
0
0.02
0.04
0.06
0.08
0.1
0.12

```
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;
```

Shell sort

- um algoritmo de ordenação que combina a estratégia do Insertion Sort com a técnica "dividir para conquistar"
- instável, o que significa que não preserva a ordem relativa de elementos com o mesmo valor.
- Mais eficiente do que algoritmos de ordenação quadráticos

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(n2) :

$$T(n) = 2 \times T(n/2.2) + O(n)$$

- 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 n2 e nlogn. No caso desde algoritmo com gap de 2,2 a complexidade está mais próxima de Θ(n2).

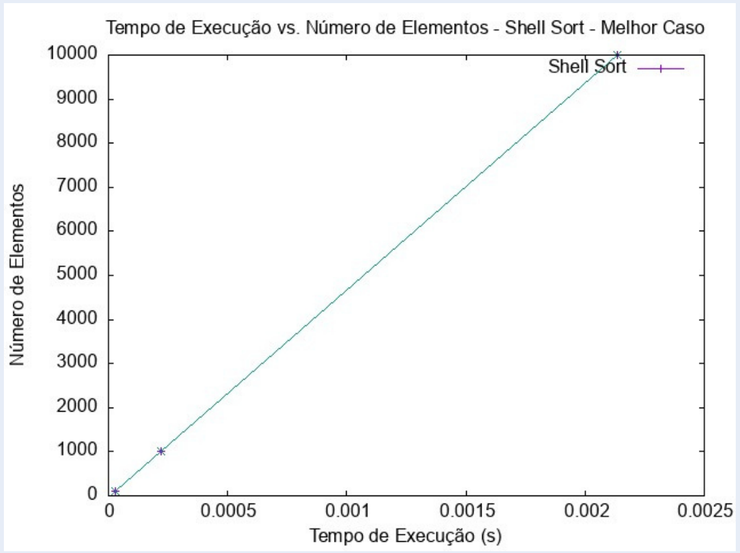
- A complexidade do shell sort no melhor caso (array ordenada) é Ω(nlogn). Não ocorre nenhuma troca e sua relação de recorrência é :

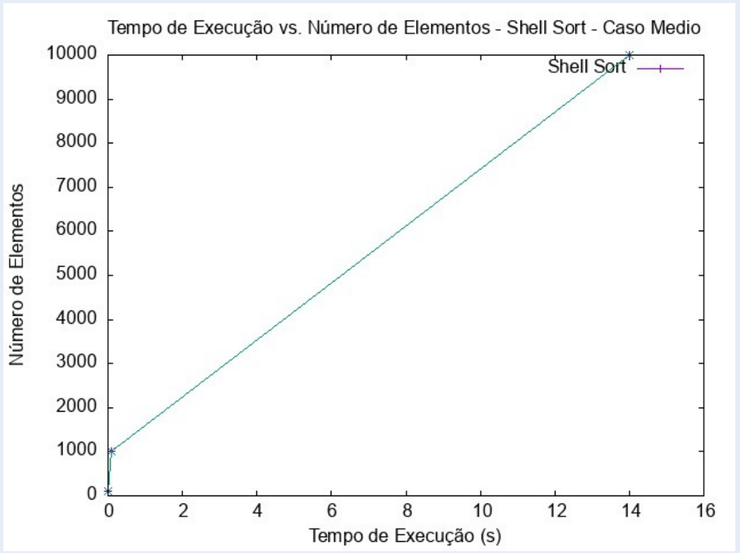
$$T(n) = 2 \times T(n/2.2) + O(1)$$

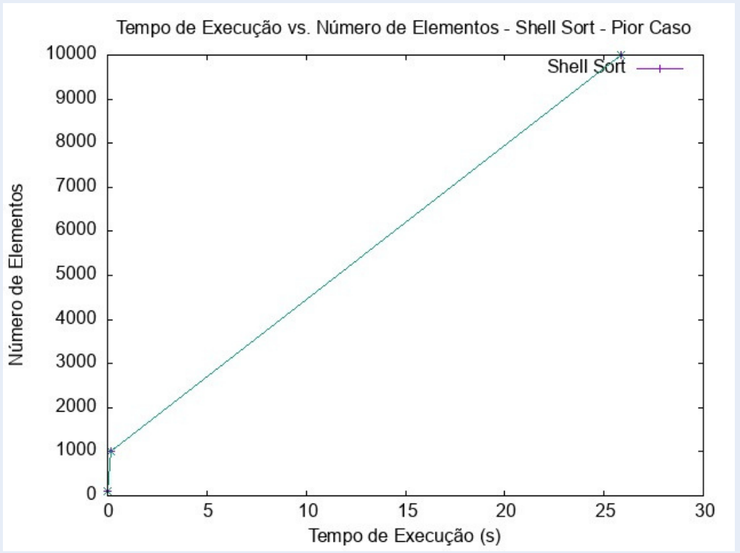
Shell sort

Casos	Tempo de execução	Número de trocas
Melhor caso com 100 posições Melhor caso com 1000 posições Melhor caso com 10000 posições	2.7889000193681568 0.0005072999992989935 0.0036803689999942435	0 0 0
Caso médio com 100 posições Caso médio com 1000 posições Caso médio com 10000 posições	0.0013505490005627507 0.5037145500009501 30.09744681200027	2489 259850 24751426
Pior caso com 100 posições Pior caso com 1000 posições Pior caso com 10000 posições	0.0024792000000379933 0.6976805189988227 55.59779411499949	4950 499500 49995000

Shell sort


Tempo de Execução vs. Número de Elementos - Shell Sort - Melhor Caso
Shell Sort
Número de Elementos
Tempo de Execução (s)
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
0
0.0005
0.001
0.0015
0.002
0.0025


Tempo de Execução vs. Número de Elementos - Shell Sort - Caso Medio
Shell Sort
Número de Elementos
Tempo de Execução (s)
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
0
2
4
6
8
10
12
14
16


Tempo de Execução vs. Número de Elementos - Shell Sort - Pior Caso
Shell Sort
Número de Elementos
Tempo de Execução (s)
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
0
5
10
15
20
25
30

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

Comb sort

- Um algoritmo de ordenação que basicamente é uma melhoria do Bubble Sort.
- Tem uma diferença crucial que impacta no desempenho: a utilização de um gap (espaçamento) definido que diminui a cada iteração.

Comb sort

- O comb sort no pior caso (array invertido) tem complexidade O(n2) 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 Θ(n2) e sua relação de recorrência é:

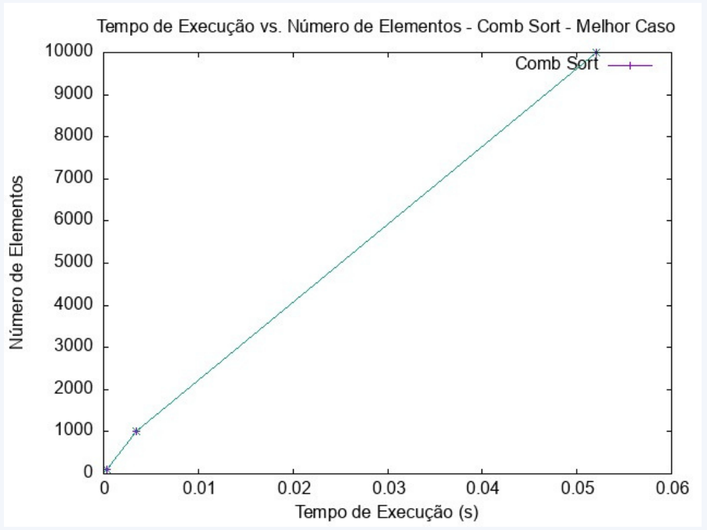
$$T(n) = T(n-1) + T(n-2) + O(n)$$

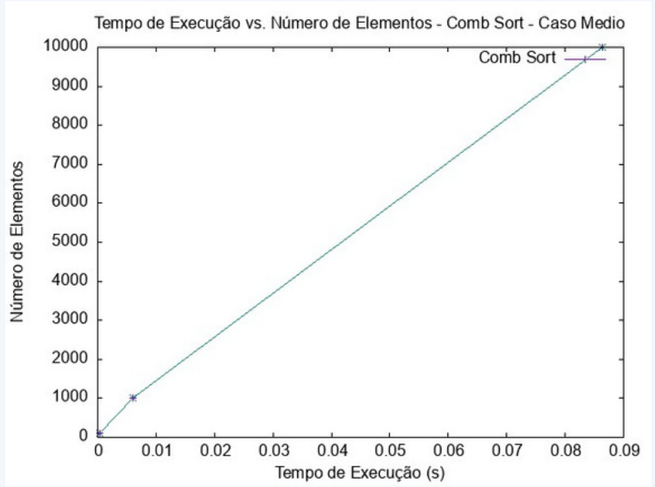
- No melhor caso (array ordenado) não são feitas comparações e sua complexidade é Ω(nlogn).

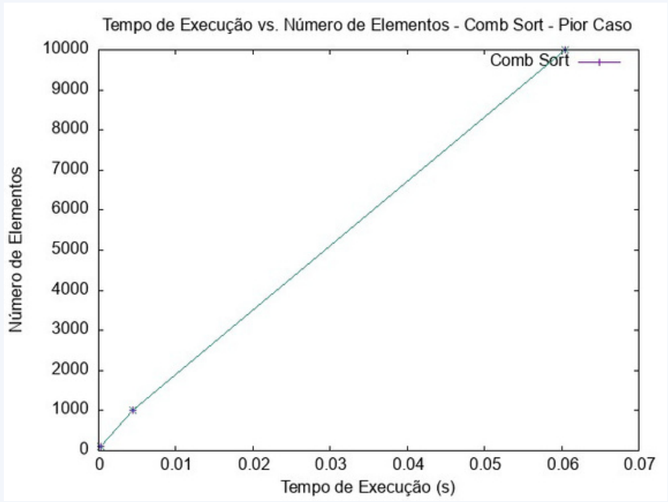
Comb sort

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

Comb sort


Tempo de Execução vs. Número de Elementos - Comb Sort - Melhor Caso
Comb Sort
Número de Elementos
Tempo de Execução (s)
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
0
0.01
0.02
0.03
0.04
0.05
0.06


Tempo de Execução vs. Número de Elementos - Comb Sort - Caso Medio
Comb Sort
Número de Elementos
Tempo de Execução (s)
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
0
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08
0.09


Tempo de Execução vs. Número de Elementos - Comb Sort - Pior Caso
Comb Sort
Número de Elementos
Tempo de Execução (s)
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
0
0.01
0.02
0.03
0.04
0.05
0.06
0.07

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.

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

Tempo de execução de cada algoritmo sob os mesmos vetores de 100 posições

Algoritmo	Melhor Caso	Caso Médio	Pior Caso
Bubble Sort	0.0030403300006582867	0.0033711889991536736	0.0042503599999787179
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	0.0007292700011021225	0.0006809000005887356
Shell Sort	3.4160000723204575	2.8329999622656032	2.764000055321958
Comb Sort	0.00024913999914133456	0.00023983999926713295	0.0002730699982101843

Tempo de execução de cada algoritmo sob os mesmos vetores de 1000 posições

Algoritmo	Melhor Caso	Caso Médio	Pior 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	0.0052158590006001 76	0.0050900389978778 55
Heap Sort	0.009027899002830964	0.01108194900000722	0.009569058998749824
Shell Sort	0.0003303100020275451	0.00033041999995475635	0.0003327199992781 9706
Comb Sort	0.056768414000544	0.0051228000011178665	0.004421278998052003

Tempo de execução de cada algoritmo sob os mesmos vetores de 10000 posições

Algoritmo	Melhor Caso	Caso Médio	Pior Caso
Bubble Sort	34.23319528799948	55.964998411000124	70.19898077499965
Insertion Sort	0.003694589999213349	0.00533240000000429805	0.004612770000676392
Quick Sort	21.902357894001398	13.633132177999869	21.19884775799983
Selection Sort	30.187859120998837	28.484823288999905	31.691810364000048
Merge Sort	0.1914418000014848	0.1749050519993034	0.1276131859995075 5
Heap Sort	0.6150853649996861	0.3385325350009225	0.3501250830013305
Shell Sort	0.0029185399998823414	0.003976488998887362	0.003118220000033034
Comb Sort	0.19023469999956433	0.18981027999871003	0.10567796899886162

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.