UNIVERSIDADE FEDERAL DE VIÇOSA
CAMPUS DE RIO PARANAÍBA
SISTEMAS DE INFORMAÇÃO

GLEIDSON VINÍCIUS GOMES BARBOSA – 6331

ANÁLISE DE ALGORITMOS DE ORDENAÇÃO - INSERTION SORT, BUBBLE SORT, SELECTION SORT E SHELL SORT

RIO PARANAÍBA
23 de outubro de 2022

GLEIDSON VINÍCIUS GOMES BARBOSA – 6331

ANÁLISE DE ALGORITMOS DE ORDENAÇÃO - INSERTION SORT, BUBBLE SORT, SELECTION SORT E SHELL SORT

Trabalho apresentado para obtenção de créditos na disciplina SIN213 - Projeto de Algoritmos da Universidade Federal de Viçosa - Campus de Rio Paraníba, ministrada pelo Professor Pedro Moisés de Souza.

Orientador: Pedro Moisés de Souza

RIO PARANAÍBA

23 de outubro de 2022

Resumo

Os algoritmos de ordenação apresentam grande importância no estudo teórico de algoritmos, bem como em aplicações rotineiras e práticas do cotidiano. Diante dessa grande importância ao longo dos anos surgem alguns algoritmos de ordenação na literatura, onde são analisados casos em que cada algoritmo se enquadra. Sendo assim se torna necessário um estudo para analisar o comportamento desses algoritmos, bem como suas complexidades para indicar qual é o mais adequado para cada tipo de problema. Neste trabalho serão estudados os algoritmos Insertion Sort, Bubble Sort, Selection Sort, Shell Sort, onde são realizados testes de e analisadas suas complexidades com base na teoria apresentada em sala de aula e vistas em livros texto além de ferramentas encontradas na internet.

Palavras-chaves: Insertion Sort, Selection Sort, Bubble Sort, Shell Sort, algoritmos de ordenação, projeto de algoritmos.

Lista de ilustrações

Figura 1 – Código Insertion Sort 9
Figura 2 – Gráfico de tempos de execução Insertion Sort 10
Figura 3 – Código Bubble Sort 12
Figura 4 – Gráfico de tempos de execução Bubble Sort 13
Figura 5 – Código Selection Sort 16
Figura 6 – Gráfico de tempos de execução Selection Sort 17
Figura 7 – Código Shell Sort 19
Figura 8 – Gráfico de tempos de execução Shell Sort 20
Figura 9 – Gráfico de tempos de execução para todos os algoritmos 21

Lista de tabelas

Tabela 1 – Insertion Sort: Tempos por tipo e tamanho das entradas. 10
Tabela 2 – Bubble Sort: Tempos por tipo e tamanho das entradas. 13
Tabela 3 – Selection Sort: Tempos por tipo e tamanho das entradas. 16
Tabela 4 – Tempos por tipo e tamanho das entradas. 19
Tabela 5 – Complexidade dos algoritmos 21

Sumário

1 Introdução 6
2 Insertion Sort 7
2.1 Análise e Complexidade do Algoritmo 7
2.1.1 Melhor Caso 7
2.1.2 Pior Caso 8
2.1.3 Caso Médio 8
2.2 Tabela e Gráfico do Algoritmo 9
3 Bubble Sort 11
3.1 Análise e Complexidade do Algoritmo 11
3.1.1 Melhor Caso, Pior Caso e Caso Médio 11
3.2 Tabela e Gráfico do Algoritmo 12
4 Selection Sort 14
4.1 Análise e Complexidade do Algoritmo 14
4.1.1 Melhor Caso 15
4.1.2 Pior Caso 15
4.1.3 Caso Médio 15
4.2 Tabela e Gráfico do Algoritmo 15
5 Shell Sort 18
5.1 Análise e Complexidade do Algoritmo 18
5.1.1 Melhor Caso, Pior Caso e Caso Médio 18
5.2 Tabela e Gráfico do Algoritmo 18
6 Visão Geral dos Algoritmos 21
7 Conclusão 23

Referências 24

1 Introdução

Neste trabalho, objetiva-se analisar os algoritmos Insertion Sort, Bubble Sort, Selection Sort e Shell Sort realizando testes com entradas para melhor caso, pior caso e caso médio e comparar seus desempenhos analisando sua complexidade e observar se os resultados práticos condizem com a teoria apresentada. Apesar de existirem diversas ferramentas na web como (TOPTAL, 2022) e (USF, 2022) onde podemos ver um pouco da teoria além de um exemplo do funcionamento dos algoritmos, para este trabalho desenvolvemos um software em linguagem C conforme a teoria apresentada em sala de aula, vista também em (CORMEN et al., 2002) e no já citado Toptal. Neste software geramos as entradas compatíveis com cada caso para 10, 100, 1.000, 10.000, 100.000 e 1.000.000 de instâncias, executamos testes e gravamos os dados em arquivos, tanto as entradas quanto saída destes dados ordenados e o tempo de execução do algoritmo e analisamos os resultados com base na teoria.

2 Insertion Sort

2.1 Análise e Complexidade do Algoritmo

Conforme visto em sala de aula, o algoritmo usa é apresentado utilizando uma sequência A de n elementos, observamos isso na tabela a seguir:

Insertion Sort(A,n)	**Custo**	**Vezes**
1 for j <- 2 to comprimento	C1	n
2 do chave < - A[j]	C2	n-1
3 "Inserir A[j] na sequência ordenada A[1.2.....j-1]"	C3 = 0	n-1
4 i <- j-1	C4	n-1
5 while i>0 e A[i] >chave	C5	$\sum_{t=2}^{n} tj$
6 do A[i+1] <- A[i]	C6	$\sum_{t=2}^{n}(tj-1)$
7 i <- i-1	C7	$\sum_{t=2}^{n}(tj-1)$
8 A[i+1] <- chave	C8	n-1

Fonte: Algoritmo visto em sala de aula

Ao observar o algoritmo apresentado, nota-se que é possível calcular sua complexidade multiplicando o custo de cada linha de sua execução por pela quantidade de vezes que esta é executada, assim chegamos na seguinte equação:

$$C1n+C2(n-1)+C4(n+1)+C5[\sum_{t=02}^{n} tj]+C6[\sum_{t=02}^{n}(tj-1)]+C7[\sum_{t=02}^{n}(tj-1)]+C8(n-1) \quad (2.1)$$

Sendo tj o número de vezes que o loop foi executado.Observa-se que a linha 3 tem custo 0 visto que apenas apresenta um comentário. Para chegar ao melhor caso, pior caso e caso médio, basta desenvolver tal equação utilizando os seguintes valores:

2.1.1 Melhor Caso

$$tj = 1 \quad (2.2)$$

Neste caso teremos que o nosso tj = 1 e na linha 5 a condição do loop é falsa, logo as linhas 6 e 7 não são executadas. Assim ao desenvolver a equação da complexidade do algoritmo temos:

$$(C1 + C2 + C3 + C4 + C5 + C8)n - (C2 + C4 + C5 + C8) \quad (2.3)$$

que podemos simplificar para:

$$An - B \tag{2.4}$$

chegando assim a uma função linear de n que é o melhor caso, ou $\Omega(n)$.

2.1.2 Pior Caso

$$tj = j \tag{2.5}$$

Neste caso, o vetor inicial estará em ordem decrescente, o que nos fará entrar em nosso loop e comparar cada elemento do nosso vetor principal com cada elemento do subarranjo ordenado, neste caso teremos que o nosso tj = j para todo jogo de 2 até n. O que nos leva a desenvolver utilizando propriedades matemáticas os nossos somatórios, chegando a seguintes igualdades:

$$\sum_{t=02}^{n} tj = \sum_{t=2}^{n} j = \frac{n(1+n)}{2} - 1 \tag{2.6}$$

$$\sum_{t=02}^{n} (tj - 1) = \sum_{t=2}^{n} tj - \sum_{t=2}^{n} 1 = \frac{n(1+n)}{2} - 1 - (n-1) = \frac{n(n-1)}{2} \tag{2.7}$$

Assim, ao desenvolver a equação de complexidade do algoritmo aplicando esta condição temos:

$$(\frac{C5 + C6 + C7}{2})n^2 + (C1 + C2 + C4 + C8 + \frac{C5 - C6 - C7}{2})n + (-C2 - C4 - C5 - C8) \tag{2.8}$$

que podemos simplificar para:

$$An^2 + Bn + C \tag{2.9}$$

Chegando assim a uma função quadrática de n que é o pior caso, ou $\mathcal{O}(n^2)$.

2.1.3 Caso Médio

$$tj = \frac{j}{2} \tag{2.10}$$

No caso médio, assumimos que teremos nossa condição para entrar no loop cumprida em média 50% das vezes, logo temos tj = j/2, utilizando as propriedades matemáticas chegamos nas seguintes igualdades:

$$\sum_{t=02}^{n} tj = \sum_{t=2}^{n} \frac{j}{2} = \frac{1}{2}(\frac{n(1+n)}{2} - 1) = \frac{n(1+n)}{4} - \frac{1}{2} \tag{2.11}$$

$$\sum_{t=02}^{n}(tj-1)=\sum_{t=2}^{n}tj-\sum_{t=2}^{n}1=\frac{n(1+n)}{4}-\frac{1}{2}-(n-1)=\frac{n(1+n)}{4}-n+\frac{1}{2} \tag{2.12}$$

Assim, ao desenvolver a equação de complexidade do algoritmo aplicando esta condição temos:

$$(\frac{C5+C6+C7}{4})n^2+(C1+C2+C4+C8-C6-C7+\frac{C5-C6-C7}{4})n \tag{2.13}$$

$$+(-C2-C4-C5-C8-\frac{C5-C6-C7}{2}) \tag{2.14}$$

que podemos simplificar para:

$$An^2+Bn+C \tag{2.15}$$

Chegando assim a uma função quadrática de n que é o caso médio, observe que é igual ao nosso pior caso, assim observamos $\theta(n^2)$.

2.2 Tabela e Gráfico do Algoritmo

O Algoritmo foi implementado conforme apresentado em sala de aula, apenas com adaptações para a linguagem solicitada conforme a imagem a seguir:

```
void insertionSort(int *vetor, int tamanho)
{
    int i, j;
    for(i = 1; i < tamanho; i++)
    {
        int x = vetor[i];
        j = (i-1);
        while(j >= 0 && vetor[j] > x)
        {
            vetor[j+1] = vetor[j];
            vetor[j] = x;
            j--;
        }
    }
}
```

Figura 1 – Código Insertion Sort

Fonte: Autoria própria

Foram realizados testes para cada entrada que variam entre 10, 100, 1.000. 10.000, 100.000, 1.000.000 instâncias e são ordenados de três formas: Crescente, Decrescente e Aleatório. Para cada combinação de quantidades de instâncias e tipo ordenação foi realizada uma execução do algoritmo. Os resultados da ordenação e o tempo de execução são gravados em arquivos diferentes para que se possa observar o resultado da ordenação e o tempo de execução do algoritmo. Os resultados destes testes são apresentados na tabela e gráfico a seguir:

Tabela 1 – Insertion Sort: Tempos por tipo e tamanho das entradas.

	10	**100**	**1.000**	**10.000**	**100.000**	**1.000.000**
Aleatório	0.0140	0.0140	0.0160	0.0760	4.9940	562.4980
Crescente	0.0130	0.0150	0.0140	0.0150	0.0140	0.0140
Decrescente	0.0170	0.0140	0.0150	0.1310	9.8410	1084.3570

Fonte: Autoria própria

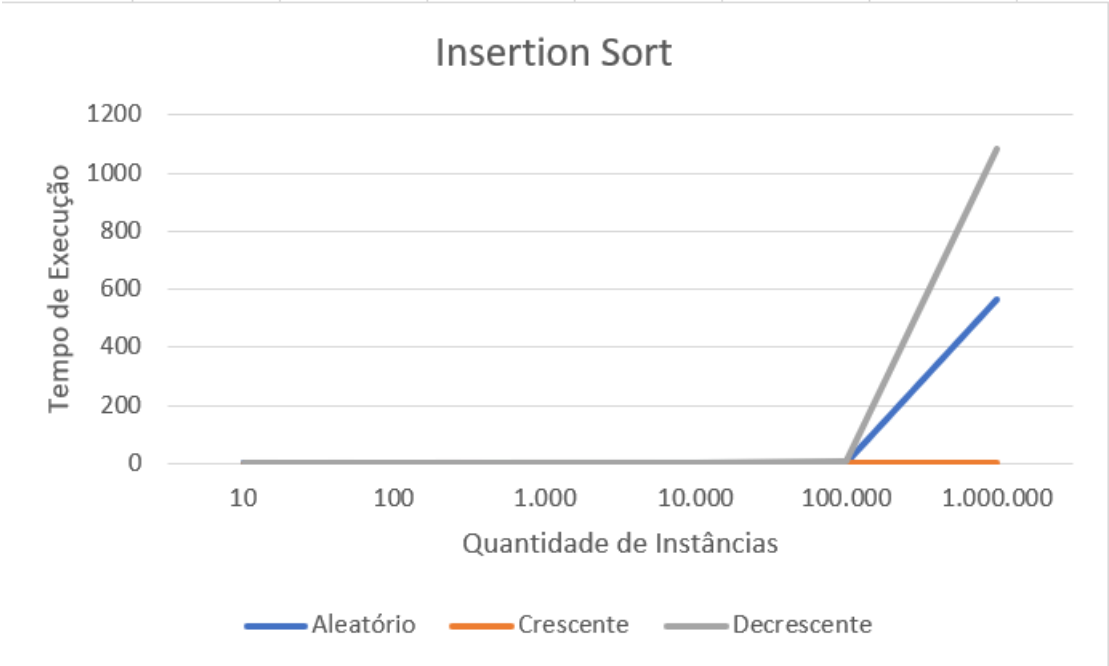

Figura 2 – Gráfico de tempos de execução Insertion Sort

Fonte: Autoria própria

Assim é possível observar que conforme a teoria nos propõe com a análise da complexidade, os resultados dos testes apresentam desempenho compatível com aquilo que foi proposto.

3 Bubble Sort

3.1 Análise e Complexidade do Algoritmo

Conforme visto em sala de aula, o algoritmo usa é apresentado utilizando uma sequência A de n elementos, observamos isso na tabela a seguir:

Bubble Sort(A,n)		**Custo**	**Vezes**
1	para i = 1 até comprimento[A]-1	C1	n-1
2	para j = 1 até comprimento[A]-i	C2	$\sum_{t=2}^{n} tj$
3	se A[j] > A[j+1] então	C3	$\sum_{t=2}^{n}(tj-1)$
4	troca A[j] <-> A[j+1]	C4	$\sum_{t=2}^{n}(tj-1)$

Fonte: Algoritmo visto em sala de aula

Ao observar o algoritmo apresentado, nota-se que é possível calcular sua complexidade multiplicando o custo de cada linha de sua execução por pela quantidade de vezes que esta é executada, assim chegamos na seguinte equação:

$$C1(n-1) + C2[\sum_{t=1}^{n-1} tj] + C3[\sum_{t=1}^{n-1}(tj-1)] + C4[\sum_{t=1}^{n-1}(tj-1)] \tag{3.1}$$

Neste caso tj é o número de vezes que o loop foi executado, também é interessando observar não temos nenhuma condiçao de parada, o que impacta diretamente nosso Melhor Caso, Pior Caso e Caso Médio.

3.1.1 Melhor Caso, Pior Caso e Caso Médio

$$tj = j \tag{3.2}$$

Neste Algoritmo observa-se um caso interessante, como não temos nenhuma condição de parada, teremos que o Melhor Caso, Pior Caso e Caso Médio são iguais, pois independente da ordenação da lista, será executada a mesma quantidade de instruções, é possível observar desenvolvendo a equação utilizando tj = j:

$$(C2 + C3 + C4)n^2 + (C1 - \frac{C2 - 3C3 - 3C4}{2})n + (-C1 + \frac{2C3 + 2C4}{2}) \tag{3.3}$$

que podemos simplificar para:

$$An^2 + Bn + C \tag{3.4}$$

Chegando assim a uma função quadrática de n que é a mesma para todos os casos. ou seja , temos $\mathcal{O}(n^2)$ e $\Omega(n^2)$, logo temos $\theta(n^2)$.

3.2 Tabela e Gráfico do Algoritmo

O Algoritmo foi implementado conforme apresentado em sala de aula, apenas com adaptações para a linguagem solicitada conforme a imagem a seguir:

```
void bubbleSort(int *vetor, int n)
{
    int i,j;
    for( i = 0; i < n; i++ )
    {
    for(j=0;j<(n-(i+1));j++)
    {
        if ( vetor[j] > vetor[j+1] )
        {
            int aux = vetor[j];
            vetor[j] = vetor[j+1];
            vetor[j+1] = aux;
        }
    }
    }
}
```

Figura 3 – Código Bubble Sort

Fonte: Autoria própria

Foram realizados testes para cada entrada que variam entre 10, 100, 1.000. 10.000, 100.000, 1.000.000 instâncias e são ordenados de três formas: Crescente, Decrescente e Aleatório. Para cada combinação de quantidades de instâncias e tipo ordenação foi realizada uma execução do algoritmo. Os resultados da ordenação e o tempo de execução são gravados em arquivos diferentes para que se possa observar o resultado da ordenação e o tempo de execução do algoritmo. Os resultados destes testes são apresentados na tabela e gráfico a seguir:

Tabela 2 – Bubble Sort: Tempos por tipo e tamanho das entradas.

	10	100	1.000	10.000	100.000	1.000.000
Aleatório	0.0130	0.0140	0.0160	0.2150	21.5520	2173.4350
Crescente	0.0130	0.0130	0.0140	0.1240	8.4450	837.4330
Decrescente	0.0130	0.0160	0.0190	0.1830	14.7900	1503.6750

Fonte: Autoria própria

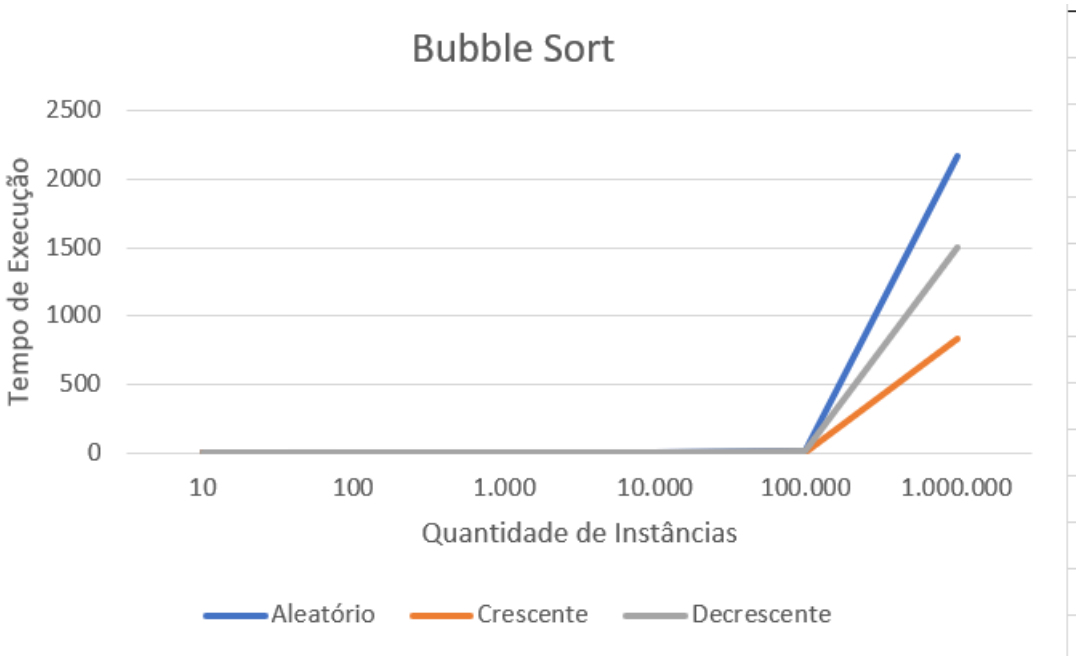

Figura 4 – Gráfico de tempos de execução Bubble Sort

Fonte: Autoria própria

Assim é possível observar que conforme a teoria vista em sala de aula nos propõe com a análise da complexidade, os resultados dos testes apresentam desempenho compatível com aquilo que foi proposto, apesar de existirem pequenas diferenças, todos os casos levam tempos elevados, mantendo o padrão quadrático, então temos $\theta(n^2)$.

4 Selection Sort

4.1 Análise e Complexidade do Algoritmo

Bubble Sort(A,n)	**Custo**	**Vezes**
1 for (i = 1 até n-1)	C1	n
2 min = i	C2	n-1
3 for (j = i+1 até n)	C3	$\sum_{t=2}^{n} tj$
4 se A[j] < A[min]	C4	$\sum_{t=2}^{n}(tj-1)$
5 min = j	C5	$\sum_{t=2}^{n}(tj-1)$
6 swap (A[i] <-> A[min])	C6	n-1

Fonte: Algoritmo apresentado pelo monitor da disciplina

Ao observar o algoritmo apresentado, nota-se que é possível calcular sua complexidade multiplicando o custo de cada linha de sua execução por pela quantidade de vezes que esta é executada, assim chegamos na seguinte equação:

$$C1n + C2(n-1) + C3[\sum_{t=1}^{n-1} tj] + C4[\sum_{t=1}^{n-1}(tj-1)] + C4[\sum_{t=1}^{n-1}(tj-1)] + C6(n-1) \quad (4.1)$$

É possível observar que as linhas 4 e 5 são executas (n-j+1) vezes. A linha 3 deve ser executada uma vez a mais, ou seja (n-j+2) vezes, assim temos tj = (n-j+2) para 2 < j < n. Ao desenvolver utilizando propriedades matemáticas os nossos somatórios, chegando a seguintes igualdades:

$$\sum_{t=02}^{n} tj = \sum_{t=2}^{n} j = \frac{n(1+n)}{2} - 1 \quad (4.2)$$

$$\sum_{t=02}^{n}(tj-1) = \sum_{t=2}^{n} tj - \sum_{t=2}^{n} 1 = \frac{n(1+n)}{2} - 1 - (n-1) = \frac{n(n-1)}{2} \quad (4.3)$$

$$\sum_{i=1}^{n-1}(n-i+1) = \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i + \sum_{t=1}^{n-1} 1 = n(n-1) - \frac{n(n-1)}{2} + n - 1 = \frac{n^2+n-2}{2} \quad (4.4)$$

$$\sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i = n(n-1) - \frac{n(n-1)}{2} = \frac{n(n-1)}{2} \quad (4.5)$$

4.1.1 Melhor Caso

Nosso pior caso ocorre quando a linha 5 não é executada, assim utilizando as igualdades matemáticas e a equação já apresentadas, chegamos na seguinte fórmula:

$$(\frac{C3+C4}{2})n^2 + (C1 + C2 + \frac{C3-C4}{2} + C6)n + (-C2 - C3 - C6) \tag{4.6}$$

que podemos simplificar para:

$$An^2 + Bn + C \tag{4.7}$$

Temos assim $\Omega(n^2)$.

4.1.2 Pior Caso

Nosso pior caso ocorre quando a linha 5 é executada todas as vezes, assim utilizando as igualdades matemáticas e a equação já apresentadas, chegamos na seguinte fórmula:

$$(\frac{C3+C4+C5}{2})n^2 + (C1 + C2 + \frac{C3-C4-C5}{2} + C6)n + (-C2 - C3 - C6) \tag{4.8}$$

que podemos simplificar para:

$$An^2 + Bn + C \tag{4.9}$$

Temos assim $\mathcal{O}(n^2)$.

4.1.3 Caso Médio

Conforme cálculos demonstrados anteriormente, temos que $\mathcal{O}(n^2)$ e $\Omega(n^2)$, logo temos $\theta(n^2)$.

4.2 Tabela e Gráfico do Algoritmo

O Algoritmo foi implementado conforme apresentado em exercícios propostos em (CORMEN et al., 2002, p.20), apenas com adaptações para a linguagem solicitada conforme a imagem a seguir:

```c
void selectionSort(int *vetor, int n)
{
    int i;
    for (i = 0; i < (n-1); i++)
    {
        int menorIndice=i;
        for(int j = (i+1); j <= (n-1); j++)
        {
            if(vetor[menorIndice] > vetor[j])
            {
                menorIndice = j;
            }
        }
        int temp = vetor[i];
        vetor[i] = vetor[menorIndice];
        vetor[menorIndice] = temp;
    }
}
```

Figura 5 – Código Selection Sort

Fonte: Autoria própria

Foram realizados testes para cada entrada que variam entre 10, 100, 1.000. 10.000, 100.000, 1.000.000 instâncias e são ordenados de três formas: Crescente, Decrescente e Aleatório. Para cada combinação de quantidades de instâncias e tipo ordenação foi realizada uma execução do algoritmo. Os resultados da ordenação e o tempo de execução são gravados em arquivos diferentes para que se possa observar o resultado da ordenação e o tempo de execução do algoritmo. Os resultados destes testes são apresentados na tabela e gráfico a seguir:

Tabela 3 – Selection Sort: Tempos por tipo e tamanho das entradas.

	10	**100**	**1.000**	**10.000**	**100.000**	**1.000.000**
Aleatório	0.0130	0.0140	0.0150	0.1010	7.2960	733.3190
Crescente	0.0130	0.0170	0.0150	0.1020	7.2870	732.1820
Decrescente	0.014	0.0120	0.0150	0.1130	8.1750	831.0870

Fonte: Autoria própria

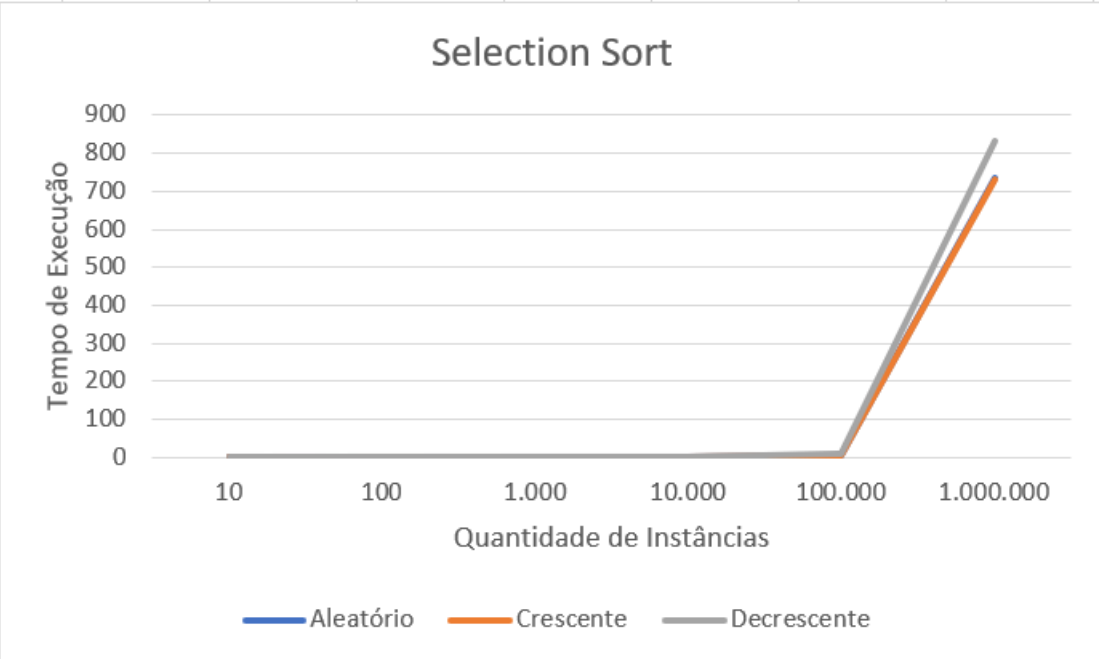

Figura 6 – Gráfico de tempos de execução Selection Sort

Fonte: Autoria própria

Assim é possível observar que conforme a teoria apresentada em exercícios vistos em (CORMEN et al., 2002, p.20) nos propõe ao escrever o pseudo-código da ordenação por seleção e análise de sua complexidade, os resultados dos testes apresentam desempenho compatível com aquilo que foi proposto, apesar de existirem pequenas diferenças, todos os casos levam tempos elevados, mantendo o padrão quadrático, assim como visto no algoritmo Bubble Sort, então temos $\theta(n^2)$.

5 Shell Sort

5.1 Análise e Complexidade do Algoritmo

Este algoritmo é uma extensão do Insertion sort porém ele possibilita a troca de registros distantes um do outro, porém é um caso estranho, pois apesar de ser o melhor dentro todos os algoritmos apresentados até o momento, a razão de seus ótimos resultados não é conhecida, de acordo com (ZIVIANI et al., 2004) ninguém foi capaz de prová-lo, então não conseguimos demonstrar com cálculos tal como os anteriores, existem apenas aproximações ou conjecturas. Apesar de toda sua vantagem em relação aos anteriores, deve-se observar que existe problemas, por não ser um método estável e ser sensível a ordem inicial do arquivo.

5.1.1 Melhor Caso, Pior Caso e Caso Médio

Como citado anteriormente, ninguém conseguiu provar sua complexidade, assim temos apenas as seguintes conjecturas:
Conjectura 1: $C(n) = \mathcal{O}(n^{1,25})$
Conjectura 1: $C(n) = \mathcal{O}(n(\ln n)^2)$

5.2 Tabela e Gráfico do Algoritmo

O Algoritmo foi implementado com base nos algoritmos apresentados em (TOPTAL, 2022) e (ZIVIANI et al., 2004, p.219), apenas com adaptações para a linguagem solicitada conforme a imagem a seguir:

```c
void shellSort(int *vetor, int n)
{
    int i , j , x;
    int h = 1;

    do
    {
        h = 3*h+1;
    } while(h < n);

    do
    {
        h /= 3;
        for(i = h; i < n; i++)
        {
            x = vetor[i];
            j = i - h;

            while (j >= 0 && x < vetor[j])
            {
                vetor[j + h] = vetor[j];
                j -= h;
            }
            vetor[j + h] = x;
        }
    }while(h > 1);
}
```

Figura 7 – Código Shell Sort

Fonte: Autoria própria

Foram realizados testes para cada entrada que variam entre 10, 100, 1.000. 10.000, 100.000, 1.000.000 instâncias e são ordenados de três formas: Crescente, Decrescente e Aleatório. Para cada combinação de quantidades de instâncias e tipo ordenação foi realizada uma execução do algoritmo. Os resultados da ordenação e o tempo de execução são gravados em arquivos diferentes para que se possa observar o resultado da ordenação e o tempo de execução do algoritmo. Os resultados destes testes são apresentados na tabela e gráfico a seguir:

Tabela 4 – Tempos por tipo e tamanho das entradas.

	10	**100**	**1.000**	**10.000**	**100.000**	**1.000.000**
Aleatório	0.0130	0.0140	0.0150	0.0170	0.0350	0.2700
Crescente	0.0130	0.0140	0.0120	0.0140	0.0180	0.0740
Decrescente	0.0140	0.0140	0.0130	0.0150	0.0170	0.0580

Fonte: Autoria própria

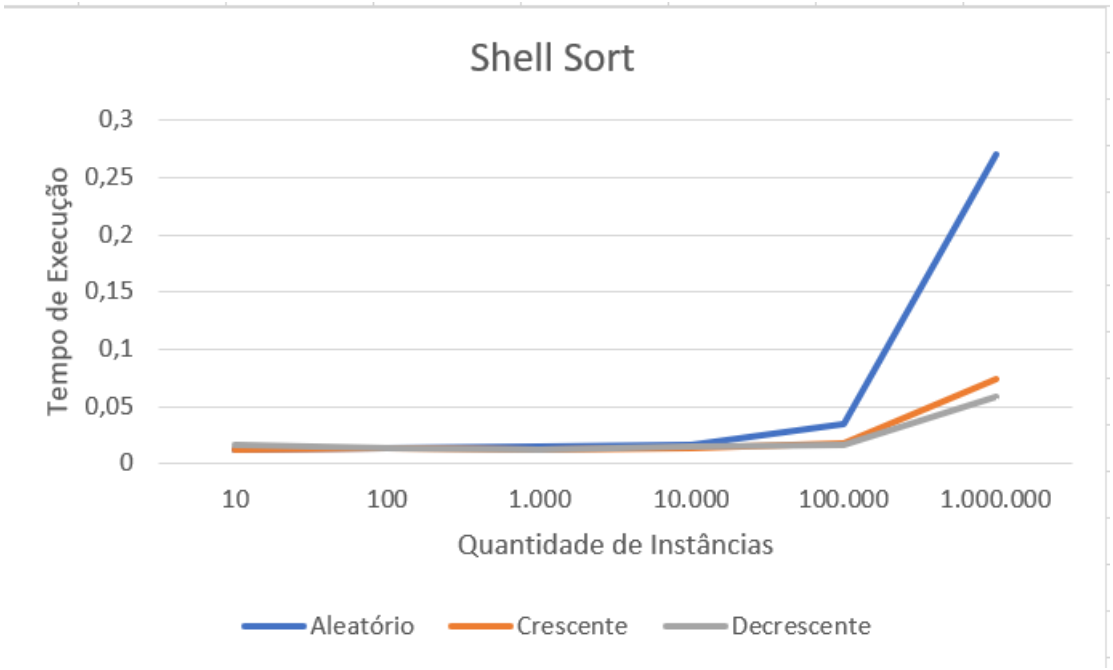

Figura 8 – Gráfico de tempos de execução Shell Sort

Fonte: Autoria própria

Observamos que até então este foi o caso mais efetivo dentre todos os algoritmos de ordenação, se mantem constante em todas as execuções, com alterações mínimas.

6 Visão Geral dos Algoritmos

A seguir temos a tabela apresentando as complexidades para todos os casos dos algoritmos vistos juntamente com o gráfico comparando os tempos de execuções para cada um deles. Deve-se observar que o algoritmo shell sort trabalha apenas com conjecturas ou aproximações visto que sua complexidade não foi provada, então será apresentado apenas a complexidade de seu pior caso conjecturada em (ZIVIANI et al., 2004).

Tabela 5 – Complexidade dos algoritmos

Algoritmo	**Melhor Caso (Ω)**	**Pior Caso ($\mathcal{O}$)**	**Caso Médio (θ)**
Insertion Sort	$\Omega(n)$	$\mathcal{O}(n^2)$	$\theta(n^2)$
Bubble Sort	$\Omega(n^2)$	$\mathcal{O}(n^2)$	$\theta(n^2)$
Selection Sort	$\Omega(n^2)$	$\mathcal{O}(n^2)$	$\theta(n^2)$
Shell Sort	-	$\mathcal{O}(n(\ln n)^2)$	-

Fonte: Autoria própria

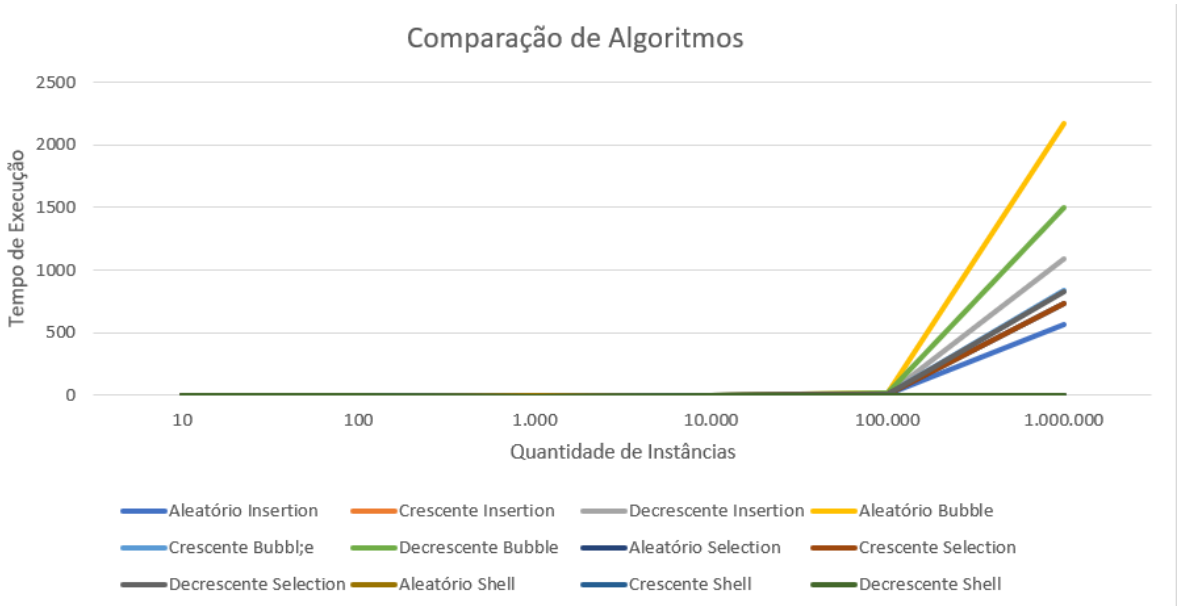

Figura 9 – Gráfico de tempos de execução para todos os algoritmos

Fonte: Autoria própria

Ao observar as complexidades de todos os algoritmos vistos até o momento, é possível verificar que a complexidade do caso médio, exceto pelo shell sort, é denotada por $\theta(n^2)$ que nos mostra que são de complexidade quadrática. Como variação da complexidade temos o Shell Sort, que como citado não teve sua complexidade provada, então

temos apenas aproximações e ao observá-la é possível notar a enorme diferença de desempenho para os demais. Também nota-se a diferença de complexidade para o melhor caso do algoritmo Insertion Sort, onde observamos que é de ordem linear, assim conseguindo também grande vantagem sobre os demais.

7 Conclusão

Portanto, levando em consideração os estudos e os testes feitos métodos de Ordenação Insertion Sort, Bubble Sort e Selection ao longo deste trabalho pode-se perceber que em uma ordenação com 10, 100 e 1.000 instâncias, o algoritmo possui o mesmo desempenho, tanto para crescente, decrescente e randômico, a partir de 10.000 instâncias já se pode perceber uma pequena variação no tempo de execução que logo se torna uma grande diferença com 100.000 e 1.000.000 de instâncias. No caso do algoritmo Shell Sort, observa-se uma constância em seus resultados para todos os tipos e tamanhos de entrada, sendo este o mais efetivo e rápido algoritmo visto até o momento. Assim confirma-se que conforme a teoria os desempenhos alcançam desempenho compatível a complexidade apresentada para todos os casos.

Referências

CORMEN, T. H. et al. Algoritmos: teoria e prática. **Editora Campus**, v. 2, p. 296, 2002. Citado 3 vezes nas páginas 6, 15 e 17.

TOPTAL. **Toptal**. 2022. Disponível em: <https://www.toptal.com/developers/sorting-algorithms/>. Acesso em: 28 sep. 2022. Citado 2 vezes nas páginas 6 e 18.

USF. **USF Computer Science Department**. 2022. Disponível em: <https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html>. Acesso em: 28 sep. 2022. Citado na página 6.

ZIVIANI, N. et al. **Projeto de algoritmos: com implementações em Pascal e C**. [S.l.]: Thomson Luton, 2004. v. 2. Citado 3 vezes nas páginas 18 e 21.