Universidade de São Paulo

Instituto de Ciências Matemáticas e de Computação

Análise da complexidade de tempo entre algoritmos de ordenação

Thierry de Souza Araújo

30 de outubro de 2021

Resumo

Este relatório possui o intuito de comparar a complexidade de tempo e espaço de memória entre os algoritmos de ordenação Bubble Sort, Insertion Sort e Merge Sort, demonstrando seus pontos positivos e negativos.

---

1 Introdução

Relatório realizado para a disciplina SCC0220 - Laboratório de Introdução à Ciência da Computação II, no Instituto de Ciências Matemáticas e de Computação, USP, Campus São Carlos, ministrada por Fernando Pereira dos Santos. Serão apresentados os resultados produzidos pela análise da complexidade de tempo e espaço de 3 algoritmos de ordenação. São eles: Bubble Sort, Insertion Sort e Merge Sort, desenvolvidos em linguagem C. A partir dos dados gerados, será possível evidenciar as qualidades e problemas de cada algoritmo para determinados ambientes de utilização.

2 Metodologia e desenvolvimento

A notação *T(n)* será usada a fim de representar as funções de tempo produzidas a partir da contagem de operações necessárias para a realização do algoritmo em seu pior caso, sendo n o tamanho da entrada (quantidade de dados). A partir delas, será abstraído o conceito de notação assintótica, que analisa a taxa de crescimento da função, para isso, encontra uma função $g(n)$, tal que, para determinado N, $T(n) \leq c * g(n)$, $\forall\ n > N$. Desse modo, é possível analisar a escalabilidade da complexidade de tempo, mediante variação de n, independente da máquina ou linguagem em que o algoritmo está sendo executado. A notação $\mathcal{O}$(g(n)) representa o pior caso (conjunto de dados que geram o limite superior de $T(n)$), já Ω(h(n)), representa o melhor caso (conjunto de dados que geram o limite inferior de $T(n)$), onde g(n) e h(n) são as funções que representam os limites.

Em relação a complexidade de espaço, a análise será simplificada, verificando apenas se o algoritmo utiliza mais espaço de memória do que a fornecida pelos parâmetros da função - conceito no sentido computacional, ou seja, conjunto de operações que executam determinada operação e possui um nome e parâmetros próprios.

Os dados utilizados para a ordenação correspondem a sequências de valores (vetores) inteiros maiores que zero, gerados de 3 modos, aleatoriamente, ordenado de forma crescente e ordenado de forma decrescente, a partir da função *rand()* (semente *srand(0)*) da biblioteca *<stdio.h>*. Esses vetores possuem tamanhos diferentes, que são: 25, 100, 1000, 1500, 2000, 2500, 3000, 3500, 4000, 4500, 5000, 5500, 6000, 6500, 7000, 7500, 8000, 8500, 9000, 9500 e 10000. A partir de cada n, os vetores foram gerados 20 vezes, para evitar influência de erros durante alguma execução, e submetidos aos algoritmos de ordenação, que os ordenou de forma crescente.

Para obter o tempo de execução dos testes foi usado a função *clock()*, da biblioteca *<time.h>*. Após a obtenção de todos os resultados, foi calculado a média simples dos tempos referentes a cada n, com o intuito de gerar gráficos que demonstrem o crescimento da função, comprovando as análises assintóticas que serão apresentadas.
Os algoritmos e funções citados abaixo foram baseados no livro Algoritmos: Teoria e Prática [1] (exceto o Bubble Sort - aula do professor Moacir Antonelli Ponti [3]).
As variáveis c_i indicam o tempo de execução de determinadas instruções no código, mas para efeito de análise, podem ser generalizadas à 1 e desenvolvimento das funções estão comentadas nos anexos de cada código.

2.1 Bubble Sort

O algoritmo utilizado para realizar os testes está apresentado no Anexo A. O Bubble realiza sua ordenação, a partir do início do vetor, inserindo o valor em sua posição ideal a partir de trocas, posição a posição.
O algoritmo possui uma complexidade de tempo representada pela fórmula:

$$T(n) = \frac{n^2+n}{2}$$

A partir dessa função, podemos obter a notação assintótica do Bubble Sort, $\mathcal{O}(n^2)$, sendo a ordenação inversa, o seu pior caso, e $\Omega(n)$, onde o vetor já ordenado representa o melhor caso. Esses valores indicam que o tempo de execução será relativamente alto, algo que não é desejável para um algoritmo.

2.2 Insertion Sort

O código utilizado para o algoritmo Insertion Sort está apresentado no Anexo B. O Insertion, como o próprio nome diz, realiza uma ordenação por inserção, para isso, subdivide, virtualmente, o vetor em uma parte ordenada (O) e outra desordenada (D), a partir disso, cada valor da lista D é inserida em sua posição ideal na parte O. Esse movimento do valor pelo vetor também ocorre a partir de trocas posição a posição, como no Bubble Sort.
O algoritmo possui uma complexidade de tempo representada pela fórmula:
Para o pior caso

$$T(n) = c_1 * n^2 + c_2 * n + c_3$$

e para o melhor caso

$$T(n) = c_4 * n + c_5$$

A partir dessa função, podemos obter a notação assintótica do Insertion Sort. Será $\mathcal{O}(n^2)$, para vetores ordenados aleatoriamente, sendo o pior dos casos quando o vetor está ordenado inversamente, e $\Omega(n)$, com o melhor caso quando os dados de entrada já estão ordenados.

2.3 Merge Sort

O código utilizado para o algoritmo Merge Sort está apresentado no Anexo C. O Merge, utiliza a abordagem de divisão e conquista. Essa método consiste em subdividir determinado problema em problemas menores (divisão) e resolve tais subproblemas recursivamente (conquista). No contexto da ordenação, o algoritmo irá, de forma recursiva, dividir o vetor em duas partes (uma recursão para cada metade) até atingir um subvetor de tamanho unitário, disso em diante,

irá ordenar os vetores menores e combinar os dois vetores que foram criados em uma recursão, até atingir o nível recursivo inicial.
O algoritmo possui uma complexidade de tempo representada pela fórmula:

$$T(n) = c_6 * n * log(n) + c_7 * n$$

A partir dessa função, podemos obter a notação assintótica do Merge Sort, $\mathcal{O}(log(n) * n)$ e $\Omega(log(n) * n)$.
É esperado então, que em relação à complexidade de tempo, a comparação dos limites das funções permita concluir que o Merge Sort terá o melhor desempenho entre os três, seguido do Insertion Sort e, por fim, o Bubble Sort.
Em relação a complexidade de espaço, apenas o Merge Sort utiliza espaço extra para realizar a ordenação do vetor, desse modo, sua utilização também depende de certa quantidade de memória disponível, a variar de acordo com n.

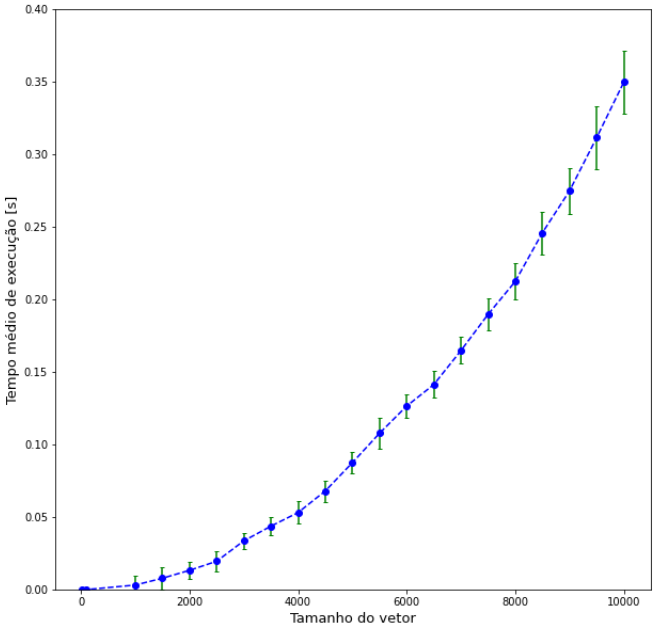
3 Resultados

As barras verdes nos pontos dos gráficos representam o desvio padrão nos tempos de execução para cada n.

3.1 Bubble Sort

Após gerar as médias de tempo para cada n, foi gerado o gráfico a seguir:

Figura 1: Gráfico Bubble Sort



Elaborado pelo autor

Como esperado, o gráfico se assemelha a uma parábola, o que causa um aumento de tempo cada vez maior para variações cada vez menores de n. Esse comportamento é gerado por conta do alto número de trocas entre os valores e, principalmente, pela alta quantidade de comparações (linha 6 do código), que, no algoritmo, é a operação que mais contribui para o aumento do tempo de execução.
Para comparar a eficiência do algoritmo entre os 3 diferentes tipos de entrada, foi gerado a tabela abaixo.

Tabela 1: Tempo de execução do Bubble Sort para diferentes modos de ordenação prévia

n	Crescente [s]	Aleatório [s]	Decrescente [s]
6500	0.06641	0.14375	0.13281
7000	0.07656	0.16563	0.15234
7500	0.08906	0.19141	0.17734
8000	0.10079	0.21484	0.20234
8500	0.11406	0.24219	0.22813
9000	0.12891	0.27344	0.25469
9500	0.14219	0.30547	0.28516
10000	0.15547	0.33828	0.31328

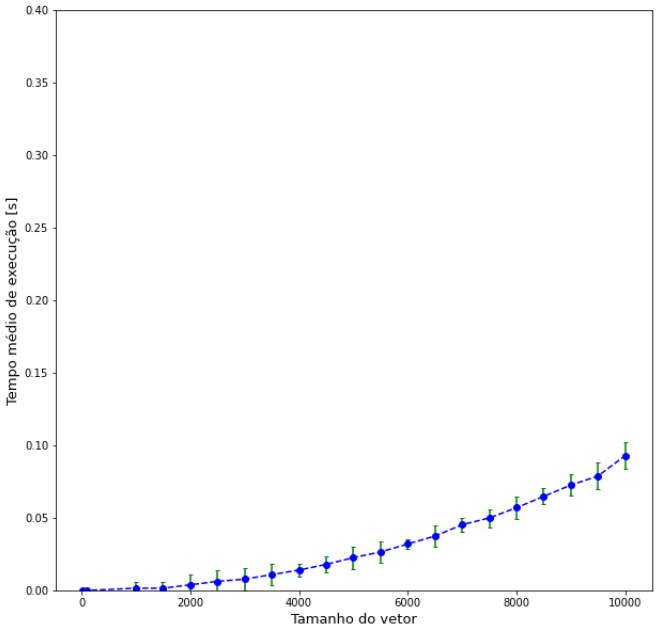
Elaborado pelo autor

Ao comparar os tempos e as análises assintóticas apresentadas, nota-se que há um possível erro nos dados produzidos. A ordenação com o vetor invertido (ordenação decrescente), deveria apresentar um tempo de execução superior aos dos vetores ordenados e aleatórios, porém, por conta de um conjunto de funcionalidades do hardware e do software, chamado *branch prediction*, isso não ocorre. Para compreendê-lo melhor, acesse [2].

3.2 Insertion Sort

Após gerar as médias de tempo para cada n, foi gerado o gráfico a seguir:

Figura 2: Gráfico Insertion Sort



Elaborado pelo autor

Mesmo possuindo uma complexidade $\mathcal{O}(n^2)$, a velocidade de crescimento não é igual ao do Bubble Sort, por possuir um método mais inteligente para ordenação. Por isso, o Insertion Sort é mais recomendado para ordenações de vetores pequenos e médios. Porém, caso n seja muito grande, o algoritmo ainda demorará um tempo alto para completar seu objetivo.

Vale lembrar que os vetores não tinham um padrão de ordenação prévio, com isso, caso o vetor gerado fosse ordenado na ordem inversa, o tempo aumentaria consideravelmente, como demonstrado a seguir.

Tabela 2: Tempo de execução do Bubble Sort para diferentes modos de ordenação prévia

n	Aleatório [s]	Decrescente [s]
5000	0.02266	0.04375
5500	0,02657	0,05391
6000	0,03204	0,06016
6500	0,03750	0,07188
7000	0,04532	0,08359
7500	0,05000	0,09531
8000	0,05702	0,10703
8500	0,06485	0,12266
9000	0,07266	0,14141
9500	0,07891	0,15469
10000	0,09297	0,17031

Elaborado pelo autor

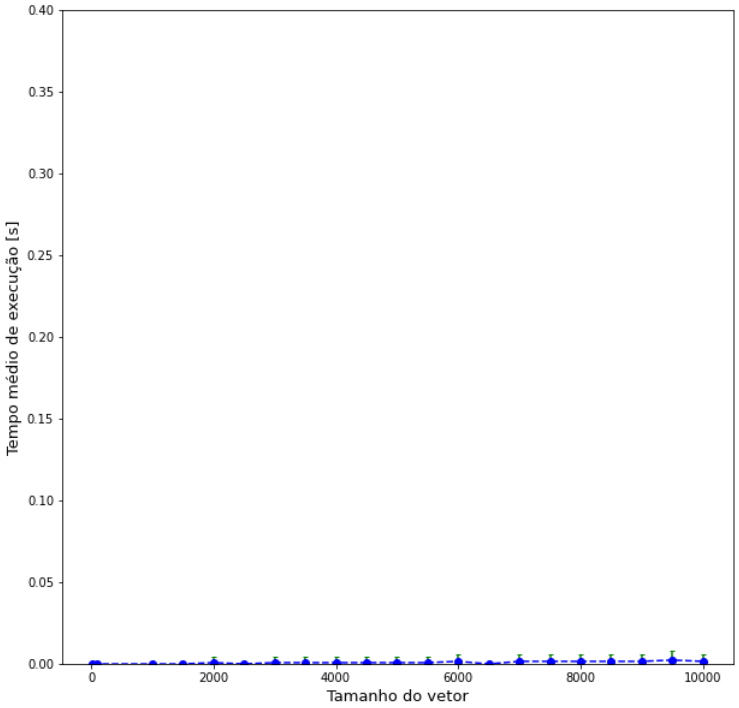
Para vetores já ordenados na ordem correta (crescente), todos os tempo de média foram iguais a 0.0 segundos.

Ao analisar a tabela apresentada, é notório o aumento significativo do tempo de execução para o vetor ordenado decrescentemente, pois é o momento em que a curva da função $T(n)$ estará mais próxima do limite superior $\mathcal{O}(n^2)$. Esse aumento se dá por conta do maior número de comparações realizadas (linha 6 do código), já que o maior valor, que está ocupando a posição 0 do vetor, terá que percorrer $(n-1)$ posições até atingir a sua posição ideal e assim em diante.

3.3 Merge Sort

Após gerar as médias de tempo para cada tamanho de entrada, foi gerado o gráfico a seguir:

Figura 3: Gráfico Merge Sort



Elaborado pelo autor

Utilizando a mesma escala e os mesmos tamanhos de vetores para o teste dos gráficos anteriores, é difícil perceber a verdadeira curva que o gráfico do Merge Sort deveria possuir, uma vez que a função *clock()*, utilizada para obter os tempos de execução, não consegue calcular os tempos com precisão tão alta para todos os três modelos de entrada, o que comprova a grande eficiência desse algoritmo.

4 Conclusão

Observando os dados demonstrados, é possível identificar qual o algoritmo de ordenação mais eficiente para casos específicos. Tais resultados foram previstos ao analisar as funções de eficiência, concomitantemente às análises assintóticas. A partir disso, percebe-se que o Merge Sort é o algoritmo mais eficiente dentre os três por apresentar o menor tempo de execução para qualquer um dos 3 modelos de entrada, porém, necessita de espaço de memória livre para sua execução, o que dificulta sua utilização em determinados ambientes. Caso a ordenação que precise ser feita possua vetores com uma quantidade de valores mediano e não haja espaço extra suficiente para a utilização do Merge Sort, o Insertion Sort é a melhor escolha. Por fim, o Bubble Sort apresentou o pior tempo de execução em todos os casos, o que faz dele a pior opção entre os 3 algoritmos.

Referências

[1] Cormen T. H. et al. *Algoritmos: Teoria e Prática*. Tradução da 3ª edição americana. Rio de Janeiro: Elsevier Editora Ltda, 2012.

[2] Computer Hope. *Branch prediction*. Último acesso em 28 de outubro de 2021. 2017. URL: `https://www.computerhope.com/jargon/b/branch-prediction.htm`.

[3] Moacir Antonelli Ponti. *ICC2 (1) 16 - Bubblesort - contagem de operações: parte 1*. Último acesso em 28 de outubro de 2021. 2020. URL: `https://www.youtube.com/watch?v=w7xdjTmVqog`.

ANEXO A

Esse anexo contém o código do algoritmo Bubble Sort em uma versão otimizada.

```c
void bubble_sort(int* vetor, int n){
    for(int i = 0; i < n - 1; i++){
        int troca = 0;

        for(int j = 0; j < n - i - 1; j++){
            if(vetor[j] > vetor[j + 1]){
                int auxiliar = vetor[j];
                vetor[j]     = vetor[i];
                vetor[i]     = auxiliar;
            }
            troca = 1;
        }

        if(troca == 0) break;
    }
}
```

Desenvolvendo a análise da complexidade de tempo

	BUBBLESORT(V, n)	custo	vezes
1	**for** $i = 0$ **to** n - 1	c1	n - 1
2	—$swap = 0$	c2	n - 1
3	—**for** $j = 0$ **to** i - 1	c3	$\sum_{i=0}^{n-2}(n-i-1)$
4	——**if** V[j] > V[$j + 1$]	c4	$\sum_{i=0}^{n-2}(n-i-1)$
5	———**then** trocar V[j] com V[$j + 1$]	c5	$\sum_{i=0}^{n-2}(n-i-1)$
6	—**if** $swap = 0$	c6	n - 1
7	——**then** FIM		

Elaborado pelo autor

A análise completa está presente no vídeo do professor Moacir [3].

ANEXO B

Esse anexo contém o código do algoritmo Insertion Sort.

```
1  void insertion_sort(int* vetor, int n) {
2      int i = 1;
3      while(i < n) {
4          int elemento = vetor[i];
5          int j = i - 1;
6          while(j >= 0 && elemento < vetor[j]){
7              vetor[j+1] = vetor[j];
8              j--;
9          }
10         vetor[j+1] = elemento;
11         i++;
12     }
13 }
```

Desenvolvendo a análise da complexidade de tempo

Figura 4: Análise da complexidade de tempo do Insertion Sort

INSERTION-SORT(A)	*custo*	*vezes*
1 **for** $j = 2$ **to** A·*comprimento*	c_1	n
2 $chave = A[j]$	c_2	$n-1$
3 //Inserir $A[j]$ na sequência ordenada $A[1..j-1]$.	0	$n-1$
4 $i = j-1$	c_4	$n-1$
5 **while** $i > 0$ e $A[i] > chave$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7 $i = i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8 $A[i+1] = chave$	c_8	$n-1$

Imagem retirada do Cormen [1]

ANEXO C

Esse anexo contém o código do algoritmo Merge Sort.

```
1  void mergesort(int* vetor, int inicio, int fim) {
2      if (fim <= inicio) return;
3
4      int central = (int) (fim+inicio)/2.0;
5
6      mergesort(vetor, inicio, central);
7      mergesort(vetor, central+1, fim);
8
9      int* aux = (int*) malloc(sizeof(int) * (fim-inicio+1));
10     int i = inicio, j = central + 1, k = 0;
11
12     while(i <= central && j <= fim) {
13         if (vetor[i] <= vetor[j]) {
14             aux[k] = vetor[i];
15             i++;
16         } else {
17             aux[k] = vetor[j];
18             j++;
19         }
20         k++;
21     }
22     while(i <= central) {
23         aux[k] = vetor[i];
24         i++; k++;
25     }
26     while(j <= fim) {
27         aux[k] = vetor[j];
28         j++; k++;
29     }
30
31     for(i = inicio, k = 0; i <= fim; i++, k++)
32         vetor[i] = aux[k];
33
34     free(aux);
35 }
```

A análise para a obtenção da função de complexidade de tempo do Merge Sort é muito longa, por isso não será apresentada nesse relatório, porém, pode ser encontrada na íntegra no livro base [1].