Universidade de São Paulo

Instituto de Ciências Matemáticas e Computação

Análise de algoritmos de ordenação: Bubble Sort, Insertion Sort e Merge Sort

Ana Lívia Ruegger Saldanha

2 de novembro de 2021

Resumo

Este relatório apresenta uma análise teórica e experimental acerca da eficiência de três algoritmos de ordenação: bubble sort, insertion sort e merge sort. Com base na bibliografia, cada algoritmo é brevemente debatido a partir de seu funcionamento e implementação em linguagem C, identificando-se suas respectivas funções de eficiência e complexidades assintóticas (notações Big-O e Big-Omega). Essa análise é complementada com os dados obtidos experimentalmente através de medições temporais feitas também em linguagem C.

---

1 Introdução

O problema da ordenação apresentado por Cormen no primeiro capítulo de *Algoritmos: teoria e prática* [4] é frequentemente apresentado para introduzir o conceito de algoritmos e eficiência. Pode ser definido formalmente da seguinte maneira:

- **Entrada:** Uma sequência de n números $< a_1, a_2, ..., a_n >$.
- **Saída:** Uma permutação (reordenação) $< a'_1, a'_2, ..., a'_n >$ da sequência de entrada, tal que $a'_1 \leq a'_2 \leq ... \leq a'_n$.

Para solucioná-lo, podemos utilizar diversos algoritmos, mas, mesmo que todos sejam corretos e resolvam o problema dado, esses algoritmos podem ser muito diferentes em termos de eficiência. Neste relatório, temos por objetivo debater essa questão através da análise e comparação de três algoritmos de ordenação: bubble sort (ordenação por flutuação), insertion sort (ordenação por inserção) e merge sort (ordenação por intercalação).

Primeiramente, buscamos contextualizar cada um dos três algoritmos abordados, apresentando seu funcionamento básico e implementações em linguagem C, assim como as funções obtidas através da contagem de operações de cada algoritmo e respectivas análises utilizando o conjunto de notações assintóticas Big-O ($\mathcal{O}$) e Big-Omega (Ω).

Em seguida, apresentamos os resultados de um experimento realizado com os três algoritmos através de medições temporais. O experimento tem por objetivo comprovar as discussões levantadas anteriormente para que possamos, por fim, realizar uma análise abrangente desses algoritmos em diferentes cenários.

2 Metodologia e desenvolvimento

Abaixo, discutiremos mais a fundo cada um dos algoritmos estudados.

2.1 Bubble Sort

A ordenação por flutuação ou *bubblesort* é um algoritmo bastante popular e, devido à sua lógica simples, de fácil implementação; porém, trata-se de uma solução ineficiente. Seu funcionamento consiste em permutar repetidamente elementos adjacentes que estão fora de ordem (flutuação de valores), ou seja, ao longo de sua execução os maiores valores vão sendo posicionados no final do vetor e os menores, no início. A seguir, apresentamos sua implementação em linguagem C.[1]

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

Podemos notar, pela lógica mostrada no código acima, que o algoritmo será sempre executado n^2 vezes, mas pode ser mais ou menos custoso a depender do número de trocas que precisarão ser realizadas. Dessa forma, para o *bubblesort* o pior caso ocorre quando o vetor está inversamente ordenado, pois é quando será executado maior número de trocas; já o melhor caso ocorre quando o vetor já está ordenado e nenhuma troca precisa ser realizada.

O número de operações de comparação feitas pelo *bubblesort* em função do tamanho n da entrada é dado por:

$$\sum_{k=1}^{n-1} n - k \quad \Longrightarrow \quad f(n) = \frac{n^2 - n}{2} \tag{1}$$

Portanto, podemos afirmar que $f(n) \in \mathcal{O}(n^2)$ e $f(n) \in \Omega(n^2)$.

2.2 Insertion Sort

O *insertion sort*, ou ordenação por inserção, é um algoritmo eficiente para ordenar um número pequeno de elementos; seu funcionamento lembra a inserção ordenada de cartas em um baralho. Trata-se de um algoritmo estável (preserva a ordem de registros de chaves iguais) e adaptativo (tira vantagem da ordenação existente na entrada).

[1] As implementações utilizadas no experimento e reproduzidas neste relatório estão de acordo com o material apresentado em aula [3]. Já as equações e análises assintóticas apresentadas foram retiradas da bibliografia, especialmente do livro *Algoritmos: teoria e prática* de Thomas Cormen [4].

Abaixo, reproduzimos a implementação do *insertion sort* em linguagem C:

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

O número de operações de comparação feitas pelo *insertion sort* em função do tamanho n da entrada é dado, no pior caso, por:

$$\sum_{k=1}^{n-1} n - k \quad \Longrightarrow \quad f(n) = \frac{n^2 - n}{2} \tag{2}$$

No melhor caso, temos:

$$\sum_{k=1}^{n-1} 1 \quad \Longrightarrow \quad f(n) = n - 1 \tag{3}$$

Portanto, podemos afirmar que $f(n) \in \mathcal{O}(n^2)$ e $f(n) \in \Omega(n)$.

2.3 Merge Sort

O *mergesort* utiliza a lógica da divisão e conquista para executar a ordenação dos valores. É um algoritmo altamente eficiente, mesmo para entradas de grande tamanho, porém apresenta a desvantagem de necessitar de memória auxiliar.

A implementação do *mergesort* em linguagem C apresenta a seguinte função recursiva:

```c
void mergeSort(int *vetor, int inicio, int fim)
{
    if (fim <= inicio)
        return;

    int centro = (int)((inicio + fim) / 2.0);
    mergeSort(vetor, inicio, centro);
    mergeSort(vetor, centro + 1, fim);

    intercala(vetor, inicio, centro, fim);
}
```

Ainda, é necessário o uso de uma outra função no processo de conquista: a função geralmente chamada *merge* (intercala). Abaixo reproduzimos a sua implementação, também em linguagem C.

```c
void intercala(int *vetor, int inicio, int centro, int fim)
{
    int *vetorAux = (int *)malloc(sizeof(int) * (fim - inicio) + 1);

    int i = inicio;
    int j = centro + 1;
    int k = 0;

    while (i <= centro && j <= fim)
    {
        if (vetor[i] <= vetor[j])
        {
            vetorAux[k] = vetor[i];
            i++;
        }
        else
        {
            vetorAux[k] = vetor[j];
            j++;
        }
        k++;
    }

    while (i <= centro)
    {
        vetorAux[k] = vetor[i];
        i++;
        k++;
    }

    while (j <= fim)
    {
        vetorAux[k] = vetor[j];
        j++;
        k++;
    }

    for (i = inicio, k = 0; i <= fim; i++, k++)
        vetor[i] = vetorAux[k];

```

```
41      free(vetorAux);
42  }
```

Observando a implementação do *mergesort*, podemos notar que seu funcionamento não depende da ordem inicial dos valores da entrada, ou seja, não há melhor ou pior caso para este algoritmo.

O número de operações de comparação feitas pelo *mergesort* em função do tamanho n da entrada é dado por:

$$T(n) = \begin{cases} 1, \ se \ n = 1 \\ 2T(\frac{n}{2}) + n, \ se \ n > 1 \end{cases} \tag{4}$$

Para a k-ésima chamada, temos:

$$T(n) = 2^k.T(\frac{n}{2^k}) + n.k \tag{5}$$

Chegamos, então, à forma fechada da função:

$$f(n) = n.log_2 n + n \tag{6}$$

Portanto, podemos afirmar que $f(n) \in \mathcal{O}(n \ logn)$ e $f(n) \in \Omega(n \ logn)$.

3 Resultados

A seguir, apresentamos os resultados dos dois experimentos realizados a fim de estudar, na prática, a eficiência dos algoritmos apresentados no item anterior.

3.1 Vetores gerados aleatoriamente

O primeiro experimento realizado[2] teve como objetivo analisar o desempenho dos algoritmos com vetores de entrada cada vez maiores. Destacamos algumas escolhas que fizemos na implementação do experimento:

- Foram gerados vetores com valores aleatórios entre 0 e $2n$, sendo n o tamanho da entrada.
- Para cada tamanho de vetor, realizamos 10 iterações/medições; os valores apresentados a seguir, tanto na Tabela 1 quanto no gráfico (Figura 1), referem-se à média do tempo de execução nessas 10 iterações.
- Os testes dos três algoritmos foram realizados em uma única execução do programa, com exatamente os mesmos vetores, gerando um arquivo .csv para cada algoritmo. Os dados contidos nesses arquivos foram utilizados para plotar o gráfico reproduzido a seguir (Figura 1).

[2]Este teste foi executado no meu computador pessoal.

Tamanho da entrada	Tempo de execução médio [s] **Bubble Sort**	**Insertion Sort**	**Merge Sort**
25	0.000003	0.000002	0.000005
100	0.000038	0.000013	0.000015
500	0.000756	0.000249	0.000081
1000	0.002265	0.000772	0.000151
2500	0.014865	0.005047	0.000432
5000	0.066596	0.020300	0.000853
7500	0.164222	0.046901	0.001350
10000	0.296579	0.078389	0.001787
15000	0.698452	0.176235	0.002885
20000	1.245493	0.318745	0.003837
25000	2.000412	0.486748	0.004968
30000	2.894625	0.702058	0.005932
35000	3.907826	0.939350	0.006980
40000	5.158371	1.231537	0.007958
45000	6.510746	1.563532	0.009022
50000	8.098092	1.920094	0.010178

Tabela 1: Resultado das medições temporais para cada algoritmo; a média de tempo de execução foi tirada a partir de 10 iterações para cada tamanho de vetor.

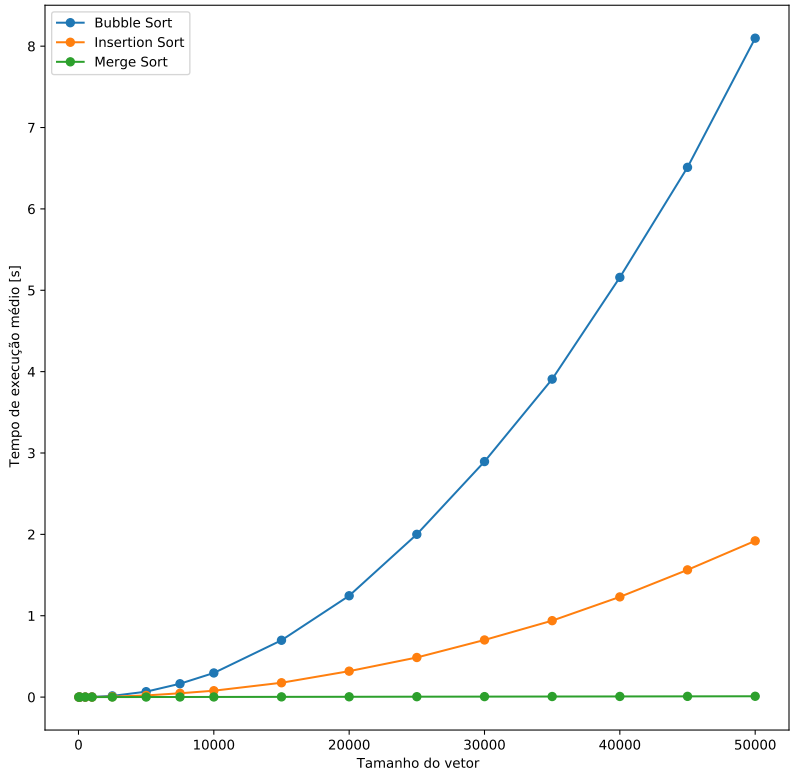

Figura 1: Resultado das medições temporais para cada algoritmo; a média de tempo de execução foi tirada a partir de 10 iterações para cada tamanho de vetor.

3.2 Melhor e pior caso

Em um segundo experimento[3], realizamos testes com vetores de mesmo tamanho ($n = 1000$), porém de diferentes tipos: gerados aleatoriamente, ordenados e inversamente ordenados. De forma complementar ao experimento anterior, este teste visou confirmar o que foi anteriormente apontado a respeito dos melhores e piores casos para cada algoritmo.

Tipo de vetor (tamanho $n = 1000$)	Tempo de execução médio [s]		
	Bubble Sort	**Insertion Sort**	**Merge Sort**
Gerado aleatoriamente	0.003206	0.000924	0.000182
Ordenado	0.001963	0.000006	0.000121
Inversamente ordenado	0.003650	0.001849	0.000121

Tabela 2: Resultado das medições temporais para ordenação de diferentes tipos de vetor, todos com o mesmo tamanho; a média de tempo de execução foi tirada a partir de 10 iterações para cada tipo de vetor.

Pelos dados apresentados na tabela acima, podemos confirmar que, de fato, o caso do vetor inversamente ordenado é o pior caso tanto para o *bubblesort* quanto para o *insertion sort*. Por outro lado, o vetor ordenado é o melhor caso também para ambos, ainda que, como esperado, a diferença entre o melhor e o pior caso seja muito mais expressiva no *insertion sort*. No caso do *mergesort*, podemos observar que não há diferença considerável entre os tempos de execução para diferentes tipos de vetor[4].

4 Conclusão

Através dos resultados dos experimentos, pudemos confirmar as análises apresentadas no Item 2 deste relatório. Em suma:

- O *bubblesort* é o menos eficiente dos três algoritmos em termos de tempo de execução e número de operações executadas; quanto à memória utilizada, apresenta vantagem em relação ao *mergesort* por não utilizar memória auxiliar (e iguala-se ao *insertion sort* nesse aspecto).
- Em termos de tempo de execução e número de operações executadas, o *insertion sort* mostrou-se melhor que o *bubblesort* e pior que o *mergesort*; quanto ao uso de memória, apresenta vantagem em relação ao *mergesort* por não necessitar de memória auxiliar, da mesma forma que o *bubblesort*. Também observa-se que este é o algoritmo mais eficiente para entradas suficientemente pequenas ($n \leq 100$).
- O *mergesort* mostrou-se o mais eficiente dos três algoritmos quanto ao tempo de execução; quanto ao uso de memória, apresenta a desvantagem de necessitar de memória auxiliar.

Dessa forma, podemos concluir que, para todas as situações em que o sistema possui memória de sobra, recomenda-se o uso do *mergesort*. Para os casos em que há escassez de memória, ou quando o tamanho da entrada é reduzido ($n \leq 100$), recomenda-se o uso do *insertion sort*.

[3] Diferentemente do anterior, este teste foi realizado no GDB Online para evitar as otimizações de branch prediction do meu computador (que melhoram o significativamente o desempenho da ordenação no pior caso).

[4] Apesar de o caso do vetor gerado aleatoriamente ter apresentado maior tempo médio de execução, considero que os dados estejam ainda consistentes com o esperado, pois a diferença entre as médias nos três casos é muito pequena.

Referências

[1] Sorting algorithm. Disponível em: <https://en.wikipedia.org/wiki/Sorting_algorithm>. Acesso em: 29 de outubro de 2021.

[2] Fernando Pereira dos Santos. Material da disciplina Introdução à Ciência da Computação II (SCC0201). ICMC-USP, 2º semestre de 2021.

[3] Fernando Pereira dos Santos. Material da disciplina Laboratório de Introdução à Ciência da Computação II (SCC0220). ICMC-USP, 2º semestre de 2021.

[4] Thomas H. Cormen et al. *Algoritmos: teoria e prática*. Rio de janeiro: Elsevier, 2012.

[5] Moacir Antonelli Ponti. Material da disciplina Introdução à Ciência da Computação II (SCC0201). ICMC-USP, 2º semestre de 2021.