Universidade de São Paulo

Instituto de Ciências Matemáticas e Computação

Eficiência de Algoritmos de Ordenação

Rafael Zimmer

5 de outubro de 2021

Resumo

Esse relatório descreve o funcionamento e propriedades de três algoritmos de ordenação: Bubble, Insertion e Merge Sort; buscando comparar as funções de eficiência que os descrevem, assim como analisar os cenários em que os tempos de execução para cada um deles é pior ou melhor. Ao final, será demonstrado visualmente o crescimento do tempo de execução de cada algoritmo em função do tamanho do vetor de entrada.

---

1 Introdução

O objetivo principal é a realização de um experimento a fim de analisar a eficiência dos três algoritmos de ordenação de listas, com o intento de demonstrar e ilustrar as diferentes funções de eficiência que caracterizam tais algoritmos, assim como comparar os tempos de execução de cada um deles em função de uma lista de tamanho definido.

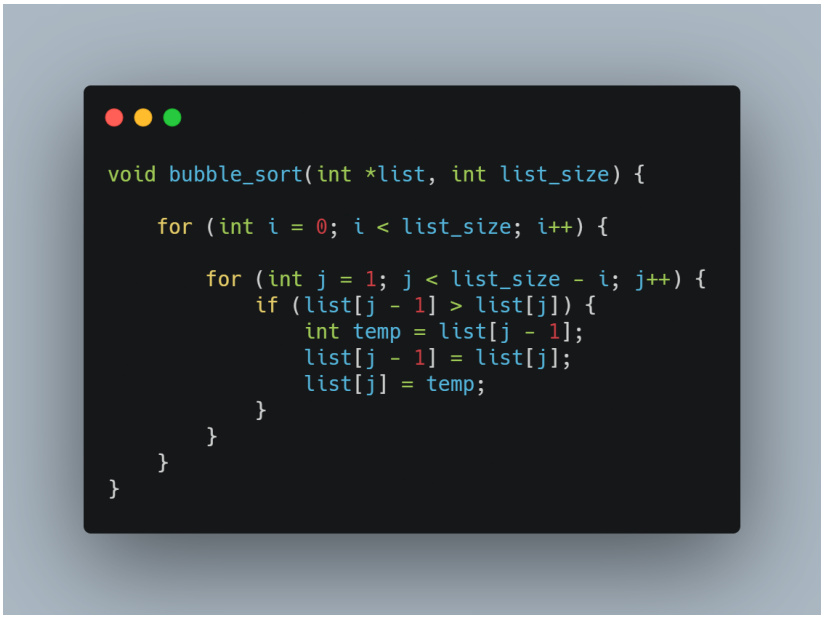
Para tal, será usado a linguagem de programação de alto nível 'C', assim como as implementações dos três algoritmos [1], das quais o código fonte será apresentado a seguir. Para as medições de tempo, será usado a biblioteca <time.h>, inclusa no compilador padrão da linguagem, que permitirá a medição do tempo do sistema antes e depois da execução do algoritmo de ordenação, sendo possível calcular assim o tempo de execução individual de cada algoritmo em cada situação de teste.

Cada medição de tempo será realizada em dois cenários diferentes: Com vetores de valores aleatórios, com um total de 25, 100, 1000 e 10000 elementos (5 casos diferentes); Com vetores de valores já ordenados, seja em ordem crescente e decrescente, de modo a visualizar os casos em que os algoritmos tem performance melhor e pior.

2 Metodologia e desenvolvimento

2.1 Desenvolvimento dos algoritmos de ordenação

Bubble Sort Primeiro, foi desenvolvido uma função para implementar o Bubble Sort, que irá iterar sobre todos os elementos da lista e para cada iteração percorre a lista a partir de sua posição inicial até um índice n, de modo que n seja o tamanho da lista menos a quantidade de iterações já feitas, verificando se dois elementos em sequência devem ser trocados. Ao final de todas as iterações a lista terá sido percorrida $\frac{tamanho \cdot (tamanho-1)}{2}$ vezes, sendo tamanho a quantidade de elementos na lista e sua complexidade assintótica é portanto $O(n^2)$.



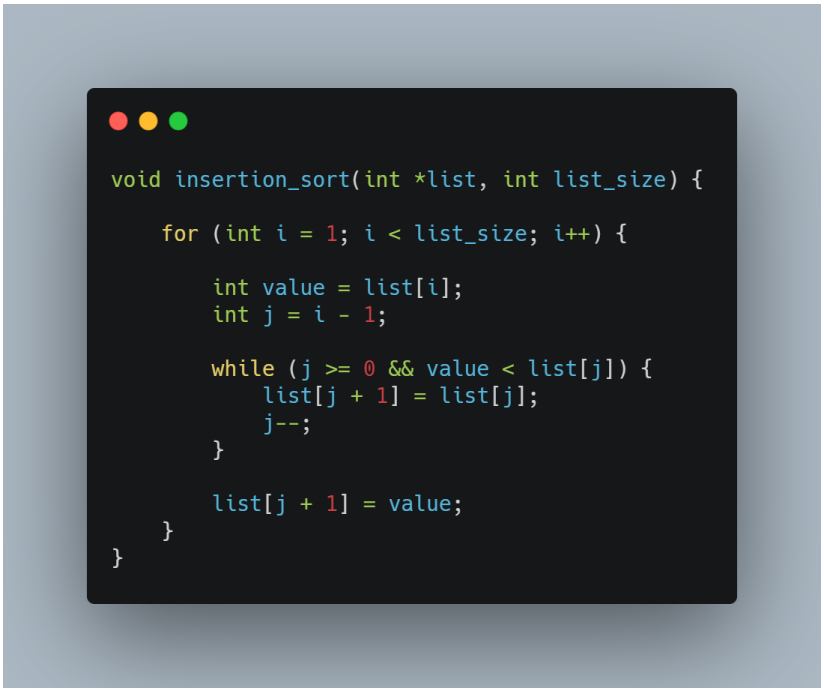
```c
void bubble_sort(int *list, int list_size) {

    for (int i = 0; i < list_size; i++) {

        for (int j = 1; j < list_size - i; j++) {
            if (list[j - 1] > list[j]) {
                int temp = list[j - 1];
                list[j - 1] = list[j];
                list[j] = temp;
            }
        }
    }
}
```

Figura 1: Implementação do Bubble Sort em 'C'

Insertion Sort Em seguida, foi desenvolvido a função para o Insertion Sort, que também itera sobre todos os elementos da lista, contudo, em vez de iterar sobre a lista novamente igual o Bubble Sort, agora, o elemento no índice atual é comparado com os anteriores até encontrar um elemento menor. Portanto, o Insertion Sort apresenta duas notações assintóticas diferentes: uma para o caso em que ocorrerá menos trocas (a lista já está ordenada), sendo a notação assintótica nesse cenário $\Omega(n)$ e caso seja necessário fazer também *tamanho* comparações, no caso da lista estar em ordem reversa a notação assintótica correspondente será $O(n^2)$.



```c
void insertion_sort(int *list, int list_size) {

    for (int i = 1; i < list_size; i++) {

        int value = list[i];
        int j = i - 1;

        while (j >= 0 && value < list[j]) {
            list[j + 1] = list[j];
            j--;
        }

        list[j + 1] = value;
    }
}
```

Figura 2: Implementação do Insertion Sort em 'C'

Merge Sort Por último, foi implementado a função recursiva do Merge Sort. O Merge Sort parte da intuição de Divisão e Conquista, em que a lista maior a ser ordenada é divida em duas listas menores com metade do tamanho da inicial, as quais são ordenadas e depois juntadas (Transforma a complexidade de ordenação de $O(n)$ para $O(\log_2 n)$. Chegando ao final da cadeia recursiva, o caso base é uma lista com um ou dois elementos (1 elemento está ordenada, 2 elementos basta uma comparação). Para cada retorno é necessário juntar duas listas de modo que a lista à ser retornada será preenchida em ordem crescente dos menores elementos de cada sub-lista, tendo complexidade $O(n)$, pois é necessário iterar sobre cada elemento das duas listas. Assim, a complexidade temporal do Merge Sort em todos os casos é $O(n \log n)$ e a de espaço é $O(n)$, pois há alocação de sub-listas de tamanho conjunto n.

```c
void merge_sort(int *list, int list_size) {

    if (list_size > 1) {
        int center = (int) list_size / 2;
        merge_sort(list, center);
        merge_sort(list + center, list_size - center);
        merge_arrays(list, center, list + center, list_size - center);
    }
}

void merge_arrays(int *first_array, int first_size, int *second_array, int second_size) {
    int temp[first_size + second_size];

    int i = 0, j = 0;

    for (int h = 0; h < (first_size + second_size); h++) {
        if (i >= first_size) {
            temp[h] = second_array[j++];
        } else if (j >= second_size) {
            temp[h] = first_array[i++];
        } else {
            if (first_array[i] < second_array[j]) {
                temp[h] = first_array[i++];
            } else {
                temp[h] = second_array[j++];
            }
        }
    }

    memcpy(first_array, temp, (first_size + second_size) * sizeof(int));
}
```

Figura 3: Implementação do Merge Sort em "C"

3 Resultados

3.1 Experimento Proposto

Para medir e comparar as eficiências de cada algoritmo de ordenação, foram propostos e experimentados dois cenários [2]: Ordenação de um vetor aleatório de 25, 100, 1000 e 10000 valores; Ordenação de dois vetores de 1000 elementos que descrevem o melhor e o pior caso de cada algoritmo. Cada situação foi executada 10 vezes por cada algoritmo, e a média das 10 iterações foi retornada no console.

Como o bubble sort sempre itera sobre a lista inteira n^2 vezes, não tem diferença no tempo do pior para o melhor caso. O insertion sort, como percorre uma parte da lista até achar um elemento menor têm seu pior caso quando o vetor está inversamente ordenado, e o melhor quando o vetor está inicialmente ordenado. O Merge Sort, semelhante ao Bubble Sort, tem complexidade igual para o melhor e pior caso também.

3.2 Resultados dos Casos Experimentais

Abaixo, a saida do programa no primeiro cenário:

```
./timing

-------------------------------------------

Vetor de 25 elementos aleatorios entre 0 e 25
Bubble Sort: 0.000003s
Insertion Sort: 0.000001s
Merge Sort: 0.000004s

-------------------------------------------

Vetor de 100 elementos aleatorios entre 0 e 100
Bubble Sort: 0.000018s
Insertion Sort: 0.000003s
Merge Sort: 0.000008s

-------------------------------------------

Vetor de 1000 elementos aleatorios entre 0 e 1000
Bubble Sort: 0.001179s
Insertion Sort: 0.000072s
Merge Sort: 0.000070s

-------------------------------------------

Vetor de 10000 elementos aleatorios entre 0 e 10000
Bubble Sort: 0.124363s
Insertion Sort: 0.007362s
Merge Sort: 0.000913s

-------------------------------------------
```

Em seguida, foram executados os algoritmos para vetores ordenados em ordem crescente e decrescente:

```
./timing_specific

-------------------------------------------

Vetor de 1000 elementos ordenados decrescentemente
Bubble Sort: 0.001165s
Insertion Sort: 0.000134s
Merge Sort: 0.000052s

-------------------------------------------

Vetor de 1000 elementos ordenados crescentemente
Bubble Sort: 0.001143s
```

```
Insertion Sort: 0.000004s
Merge Sort: 0.000061s
```

4 Conclusão

Para melhor análise da saída do contador de tempo de execução, foi desenhado um gráfico [3] pra cada algoritmo e seus respectivos tempos de execução para determinado tamanho de vetor de entrada:

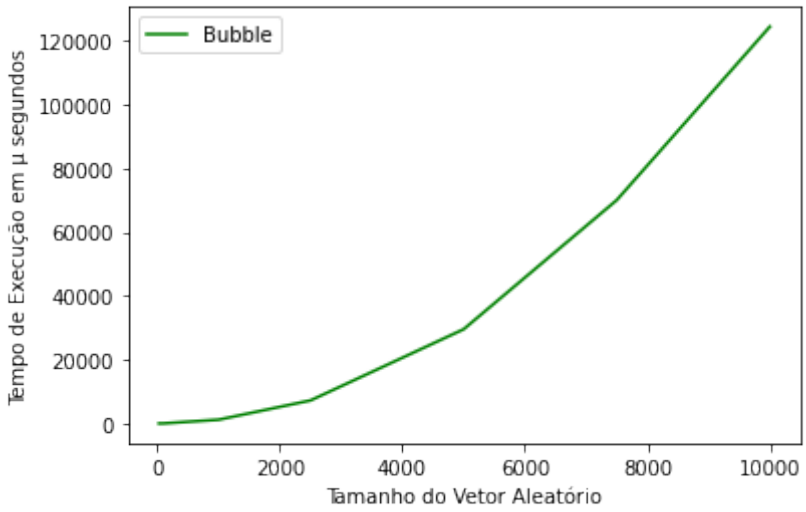


Figura 4: Implementação do Insertion Sort em "C"

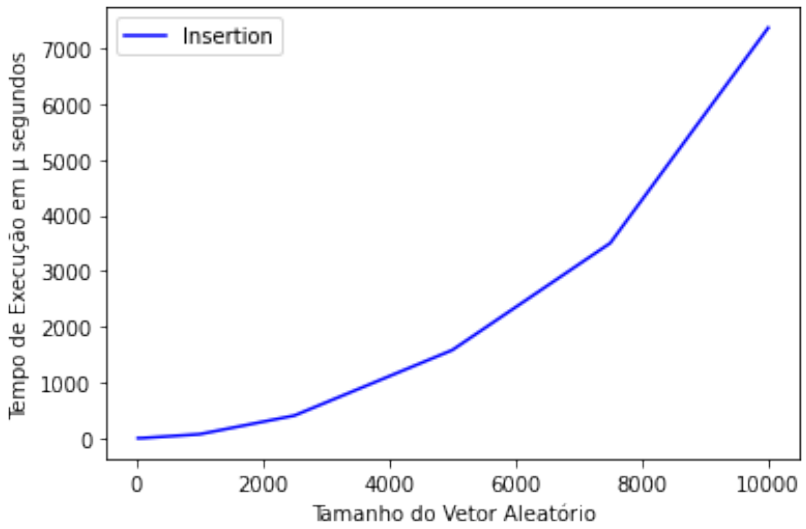


Figura 5: Implementação do Insertion Sort em "C"

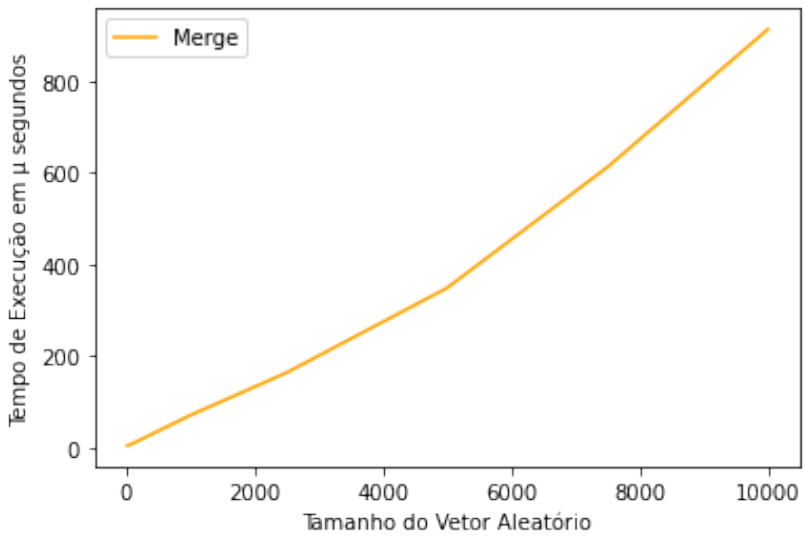


Figura 6: Implementação do Insertion Sort em 'C'

Com a visualização do tempo de execução dos algoritmos em gráficos é possível notar que, enquanto o crescimento do tempo de execução do Merge Sort se assemelha à uma função linear × uma função logarítmica, o crescimento do tempo para o Insertion e o Bubble Sort se assemelham ambos a uma função exponencial de segundo grau, confirmando assim a notação assintótica obtida para cada algoritmo durante o desenvolvimento.

Analisando a saída usando vetores já ordenados (crescentemente e inversamente ordenados), percebe-se que o tempo de execução do Bubble Sort permanece próximo em ambos os casos, assim como o do Merge Sort, confirmando o fato do melhor e pior caso ter a mesma notação assintótica para estes algoritmos. Contudo, o Insertion Sort tem tempo de execução para o pior caso maior que o tempo de execução para o melhor caso, demonstrando o fato do melhor caso do Insertion Sort ter função de eficiência linear e não quadrática.

Assim, conclui-se a análise das funções de eficiência dos três algoritmos de ordenação, sendo possível escolher o mais eficiente para situações específicas, de modo a obter sempre o menor tempo de execução para a ordenação de um vetor.

Referências

[1] A. Laaksonen, "Competitive programmer's handbook," 2018.

[2] F. P. d. S. *et al.*, "Orientação para produção de relatório: Módulo 1," p. 2, 2021.

[3] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.