Universidade Federal de Viçosa
Campus **Rio Paranaíba**

Filipe Brener - 5952

ANÁLISE DE ALGORITMOS DE ORDENAÇÃO

Rio Paranaíba - MG
2022

Universidade Federal de Viçosa
Campus **Rio Paranaíba**

Filipe Brener Ferreira Santos - 5952

ANÁLISE DE ALGORITMOS DE ORDENAÇÃO

Trabalho apresentado para obtenção de créditos na disciplina SIN 213 - Projeto de Algoritmo da Universidade Federal de Viçosa - Campus de Rio Paranaíba, ministrada pelo Professor Pedro Moisés de Souza.

Rio Paranaíba - MG
2022

RESUMO

O acúmulo de dados tem se tornado cada vez mais um problema urgente a ser resolvido. Então é necessário usar um algoritmo de ordenação, que nada mais é que um processo lógico de organizar algum tipo de estrutura linear. Este estudo tem como objetivo avaliar experimentalmente os dados gerados usando o algoritmo de ordenação por inserção. Seis vetores de diferentes tamanhos foram testados em três cenários e os dados coletados foram comparados.

Sumário

1 INTRODUÇÃO

Ao iniciar com um vetor de capacidade N, são avaliados os tempos de execução de:

1. Uma lista ordenada em ordem crescente
2. Uma lista ordenada em ordem decrescente
3. Uma lista desordenada com números aleatórios entre 1 e N

Esse procedimento é realizado com vetores de tamanhos (valores de N) 10, 100, 1.000, 10.000, 100.000 e 1.000.000. Com os resultados espera - se apontar as vantagens e desvantagens ao se fazer uso de um algoritmo.

O estudo se dispõe da seguinte forma: na próxima seção (2 Algoritmos de Ordenação) será apresentado de forma sucinta o algoritmo de ordenação cujo se faz presente como objeto de estudo deste projeto, os algoritmos são Insertion Sort, Bubble Sort, Selection Sort e Shell Sort, e seus respectivos códigos implementados na linguagem C; na terceira seção (3 Resultados) os dados coletados nos testes realizados serão apresentados; e finalmente na quarta seção (4 Conclusão) serão apresentadas as conclusões sobre à análise dos resultados obtidos na seção anterior.

Pode ser observado nas figuras que ilustram a implementação dos algoritmos, uma função implementada em Linguagem C que, recebe uma estrutura por parâmetro e ordena sua lista de entrada (s->input_list) e calcula seu tempo de execução.

2 ALGORITMOS DE ORDENAÇÃO

Os algoritmos de ordenação são algoritmos que direcionam a ordenação e rearranjo dos valores apresentados em uma determinada sequência, para que os dados possam então ser acessados com mais eficiência. Um dos principais objetivos desse tipo de algoritmo é a ordenação de vetores, pois uma mesma variável pode ter várias colocações, dependendo do tamanho do vetor declarado. Por exemplo, organizar uma lista de frequência escolar para que a lista seja organizada em ordem alfabética.

2.1 INSERTION SORT

O Insertion Sort é de fácil implementação e seu funcionamento se dá por comparação e inserção direta. A medida que o algoritmo percorre a lista, o mesmo os organiza, um a um, em sua posição mais correta. De forma resumida é como se basicamente o algoritmo procurasse o elemento de menor valor e o inserisse em uma nova lista, ao final de sua execução a nova lista estará ordenada, por isso o nome Insertion Sort, ordenação por inserção.

Figura 1. implementação do algoritmo insertion sort

```
void insertion_sort(structure *s){
    int current_number;
    int current_index;
    s->execution_time = 0.0;
    clock_t begin = clock();
    for(int i = 1; i < s->input_size; i++){
        current_number = s->input_list[i];
        current_index = i - 1;
        while(current_index >= 0 && s->input_list[current_index] > current_number){
            s->input_list[current_index + 1] = s->input_list[current_index];
            current_index--;
        }
        s->input_list[current_index + 1] = current_number;
    }
    clock_t end = clock();
    s->execution_time += (double)(end - begin) / CLOCKS_PER_SEC;
}
```

2.2 BUBBLE SORT

Esse algoritmo é um dos mais simples e funciona comparando dois elementos e trocando-os de forma que o elemento de maior valor permaneça à direita do segundo após passar pelo primeiro vetor completo. Podemos garantir que o maior objeto será movido para o último ponto do vetor. Este movimento é repetido até que o vetor esteja completamente ordenado.

Figura 2. implementação do algoritmo bubble sort

```
void bubble_sort(structure *s){
    s->execution_time = 0.0;
    int aux;
    clock_t begin = clock();
    for(int i = 0; i < s->input_size; i++){
        for(int j = 0; j < s->input_size - i; j ++){
            if(s->input_list[j] > s->input_list[j + 1]){
                aux = s->input_list[j];
                s->input_list[j] = s->input_list[j + 1];
                s->input_list[j + 1] = aux;
            }
        }
    }
    clock_t end = clock();
    s->execution_time += (double)(end - begin) / CLOCKS_PER_SEC;
}
```

2.3 SELECTION SORT

Um dos propósitos básicos mais simples e mais usados da ordenação por seleção é mover o menor valor para a primeira posição, o segundo menor para a segunda posição, e assim por diante para n valores da posição n, onde o valor da esquerda . é sempre menor que o valor correto (valor esquerdo < valor direito).

Figura 3. implementação do algoritmo selection sort

```
void selection_sort(structure *s){
    s->execution_time = 0.0;
    int current_lower_index;
    int aux;
    clock_t begin = clock();
    for(int i = 0; i < s->input_size; i++){
        current_lower_index = i;
        for(int j = i + 1; j < s->input_size; j ++){
            if(s->input_list[current_lower_index] < s->input_list[j]){
                current_lower_index = j;
            }
        }
        aux = s->input_list[i];
        s->input_list[i] = s->input_list[current_lower_index];
        s->input_list[current_lower_index] = aux;
    }
    clock_t end = clock();
    s->execution_time += (double)(end - begin) / CLOCKS_PER_SEC;
}
```

2.4 SHELL SORT

A ordenação de shell baseada na ordenação por mesclagem de forma avançada é um algoritmo que, utilizando um valor intermediário denominado gap, reorganiza os elementos da estrutura por mesclagem direta. O espaço de valor de distância superior nada mais é que um valor que define a distância na qual os elementos são comparados, ou seja. quando no tipo de inserção, cada elemento é comparado com o elemento imediatamente seguinte e menor. o valor adicionado à posição à esquerda do segundo, a distância especificada pelo programador em ShellSort faz com que os elementos comparados, não mais tão próximos. Essa pequena diferença pode parecer pequena, mas aplicada, acelera o processo, pois devido à distância de referência, meio que organizamos não uma, mas duas estruturas ao mesmo tempo.

Figura 4. implementação do algoritmo shell sort

```
void shell_sort(structure *s){
    int j, current_value;
    clock_t begin = clock();
    int h = 1;
    while(h < s->input_size) {
        h = 3*h+1;
    }
    while (h > 0) {
        for(int i = h; i < s->input_size; i++) {
            current_value = s->input_list[i];
            j = i;
            while (j > h-1 && current_value <= s->input_list[j - h]) {
                s->input_list[j] = s->input_list[j - h];
                j = j - h;
            }
            s->input_list[j] = current_value;
        }
        h = h/3;
    }
    clock_t end = clock();
    s->execution_time += (double)(end - begin) / CLOCKS_PER_SEC;
}
```

3 RESULTADOS

Foi definido como meio de codificação a linguagem C juntamente com o editor Visual Studio Code como ambiente de desenvolvimento, para a realização dos testes que serão apresentados a seguir. Em relação ao hardware utilizado, a máquina em que foram executados as simulações possui as seguintes especificações:

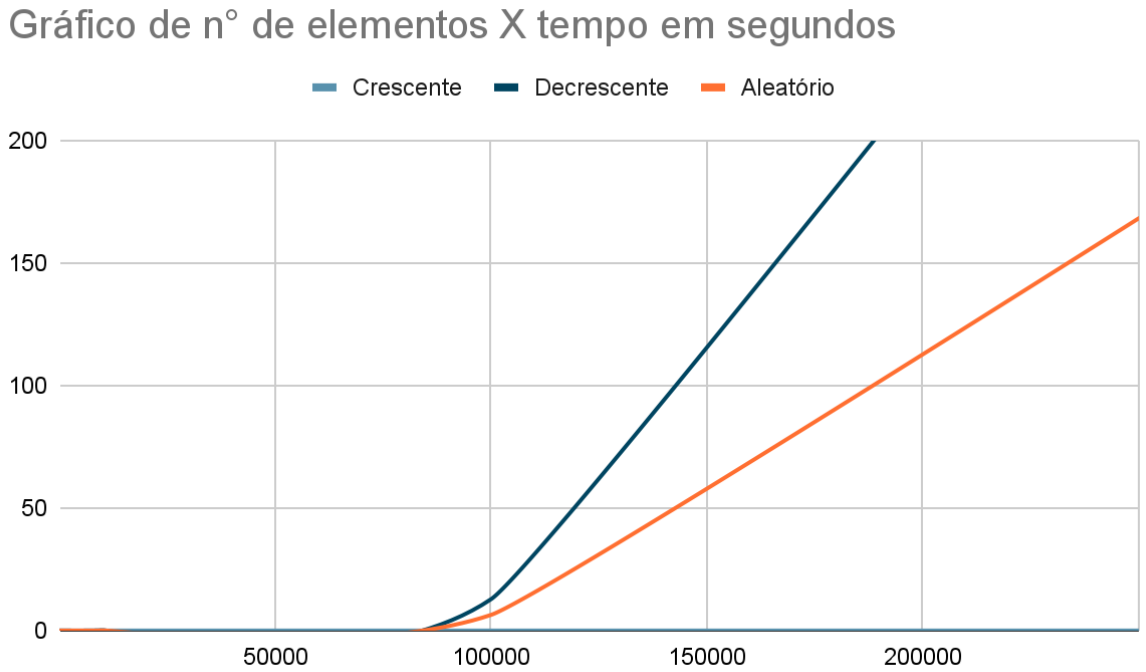
- Processador: AMD Ryzen 5 3600;
- Frequência: 4000 GHz;
- Memória RAM: 16GB (DDR4 3.200MHz);
- Sistema Operacional: Ubuntu (Linux)

3.1 INSERTION SORT

Tabela 1. Tempo de execução por tamanho do vetor - INSERTION SORT

	10	100	1.000	10.000	100.000	1.000.000
Crescente	0.000001	0.000001	0.000004	0.000041	0.000300	0.002898
Decrescente	0.000000	0.000014	0.001278	0.128645	12.834253	2028.224182
Aleatório	0.000001	0.000008	0.000771	0.066064	6.417593	1016.672662

Gráfico 1



3.2 BUBBLE SORT

Tabela 2. Tempo de execução por tamanho do vetor - BUBBLE SORT

	10	100	1.000	10.000	100.000	1.000.000
Crescente	0.000001	0.000011	0.000896	0.086877	8.724042	981.562551
Decrescente	0.000001	0.000014	0.000911	0.087020	8.741907	1006.888385
Aleatório	0.000001	0.000016	0.000947	0.102624	16.538666	1702.165045

Gráfico 2

3.3 SELECTION SORT

Tabela 3. Tempo de execução por tamanho do vetor - SELECTION SORT

	10	100	1.000	10.000	100.000	1.000.000
Crescente	0.000000	0.000013	0.000904	0.090412	8.681950	1004.677974
Decrescente	0.000001	0.000010	0.000862	0.086747	8.420777	949.812305
Aleatório	0.000001	0.000013	0.000900	0.087969	8.432594	941.533959

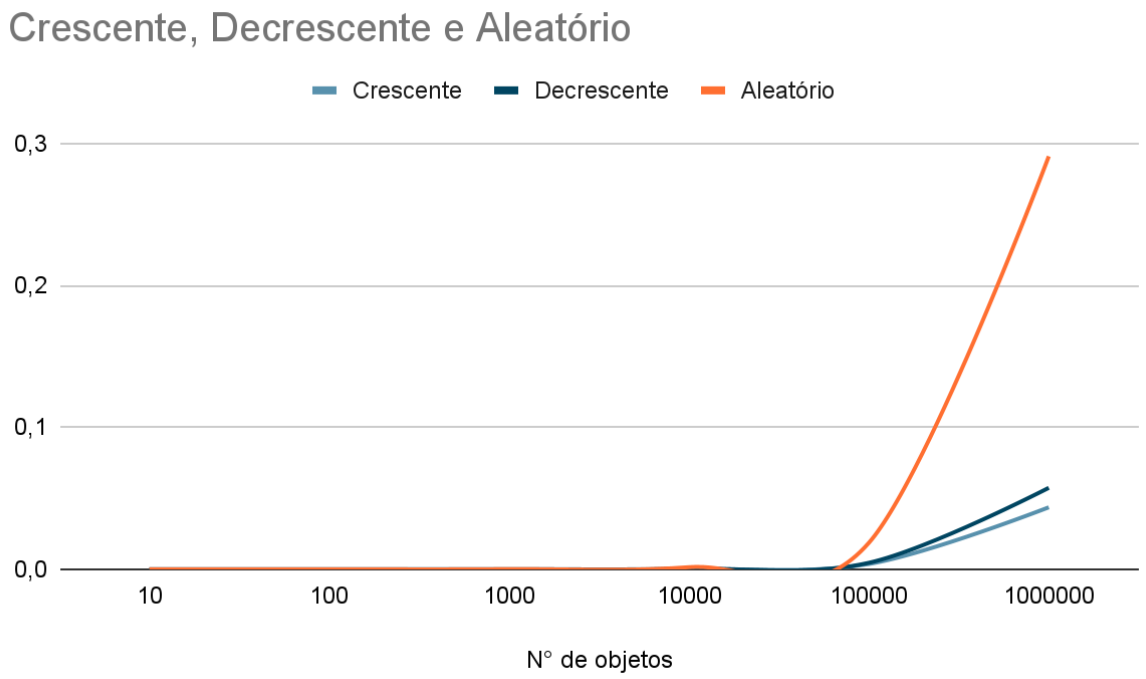
Gráfico 3

3.4 SHELL SORT

Tabela 4. Tempo de execução por tamanho do vetor - SHELL SORT

	10	100	1.000	10.000	100.000	1.000.000
Crescente	0.000001	0.000003	0.000020	0.000238	0.003778	0.043668
Decrescente	0.000000	0.000003	0.000029	0.000378	0.004602	0.057391
Aleatório	0.000000	0.000006	0.000092	0.001499	0.018713	0.290828

Gráfico 4



4 CONCLUSÃO

4.1 INSERTION SORT

O Insertion Sort, por sua vez, é útil para estruturas lineares pequenas, pode ser observado no Gráfico 1 que, até perto da ordem de 100.000 objetos a diferença de tempo de execução entre as ordens é praticamente nula, e após esse intervalo, a diferença entre os cenários(crescente, decrescente e aleatório), já cresce exponencialmente. Em seu melhor caso o algoritmo tem como, em notação BIG O, O(n) que é linear em relação ao número de objetos da lista. Já no seu pior caso tem como notação O(n^2), quadrática, crescendo exponencialmente em relação à quantidade de ítens da lista.

4.2 BUBBLE SORT

O algoritmo bubble sort, embora seja o mais fácil de implementar, não fornece resultados satisfatórios, principalmente em termos de número de comparações. A ineficiência desse algoritmo pode se traduzir em consumo de processamento significativo, o que resulta em velocidades lentas e longa latência para máquinas com poucos (ou limitados) recursos computacionais. Na opinião do autor, sua aplicação é apenas para fins educacionais, pois os projetos que a utilizam podem ser considerados ineficientes e/ou fracos.

4.3 SELECTION SORT

A ordenação por seleção torna-se útil em estruturas lineares semelhantes à ordenação por inserção, no entanto, como há muito menos trocas do que comparações, o número de elementos é muito maior, consumindo mais tempo de leitura e menos tempo de gravação.

4.4 SHELL SORT

Com base nos dados deste trabalho, o Shell Sort é o que apresenta os resultados mais satisfatórios, em sua maioria grandes e desorganizados. Por ser considerado uma melhoria na ordenação por seleção, o Shell Sort, quando utilizado para a mesma finalidade de seu antecessor - exigindo menos funcionalidades escritas - apresentará melhor desempenho, prolongando assim a vida útil dos recursos.

5 REFERÊNCIAS

Szwarcfiter, J. L. and Markezon, L. (2015). “Estruturas de Dados e Seus Algoritmos.” 3ª edição. Rio de Janeiro. LTC.

SOUZA, Jackson EG; RICARTE, João Victor G.; DE ALMEIDA LIMA, Náthalee Cavalcanti. Algoritmos de Ordenação: Um estudo comparativo. **Anais do Encontro de Computação do Oeste Potiguar ECOP/UFERSA (ISSN 2526-7574)**, n. 1, 2017.