Algoritmos de Ordenação: *Uma análise comparativa*

Pedro Henrique Di Francia Rosso
21201920808

28 de Novembro de 2019

SUMÁRIO

1 INTRODUÇÃO

Algoritmos de ordenação são objetos de estudo a muito tempo na área da Computação. Existem várias formas de se ordenar uma sequência, entre elas, as formas chamadas de Ordenação por Comparação, que consiste em ordenar uma sequência baseando-se na comparação entre os elementos.

Os algoritmos mais comuns nesse tipo de ordenação são: *Insertion Sort*, *Selection Sort*, *Bubble Sort*, *Merge Sort*, *Quick Sort* e *Heap Sort*.

Este trabalho tem como objetivo abordar cada um desses algoritmos, trazendo seu funcionamento, implementação (em C++) e uma análise comparativa do tempo desprendido por cada algoritmo em diferentes cenários de entrada. A ideia de analisar os tempos de cada algoritmo na pratica é ver o comportamento de cada algoritmo e comparar com o impacto das análise de complexidade estudadas em sala.

2 ALGORITMOS DE ORDENAÇÃO

Dentre os algoritmos de ordenação abordados, temos algoritmos de diversas complexidades que podem mudar de acordo com o tipo de entrada, conforme estudados em aula e dispostos na tabela abaixo:

Tabela 1: Complexidade dos algoritmos de ordenação com base no tipo de entrada.

Algoritmo	**Complexidade**		
	Melhor Caso	**Caso Médio**	**Pior Caso**
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

2.1 *Insertion Sort*

O algoritmo de ordenação por inserção está entre os mais simples e tem complexidade quadrática. O algoritmo consiste em percorrer a lista de números do início até o fim, e em cada iteração, o número atual é trocado um a um com seus antecessores enquanto o mesmo for menor que o antecessor comparado. Conforme o código do algoritmo 1:

Algorithm 1: *Insertion Sort*

```
Sendo N o tamanho do vetor numbers;
Result: Vetor de N números ordenados crescentemente.
int aux, i, j;
for i = 1; i < N; i = i + 1 do
    aux = numbers[i];
    j = i − 1;
    while (j >= 0) && (numbers[j] < aux) do
        numbers[j + 1] = numbers[j];
        j = j − 1;
    end
    numbers[j + 1] = aux;
end
```

2.2 *Selection Sort*

O algoritmo de ordenação por seleção também está entre os mais simples e tem complexidade quadrática. O algoritmo consiste em percorrer a lista de números do início até o fim, e para cada iteração, procura-se o menor número entre os seguintes, e se o índice do menor for diferente do índice atual, troca-se os dois números de lugar, dessa forma, a medida que as iterações vão se passando, a lista vai se ordenando, conforme o código do algoritmo 2:

Algorithm 2: *Selection Sort*

```
Sendo N o tamanho do vetor numbers;
Result: Vetor de N números ordenados crescentemente.
int cmin, aux, i, j;
for i = 0; i < N − 1; i = i + 1 do
    cmin = i;
    for j = i + 1; j < N; j = j + 1 do
        if numbers[j] < numbers[cmin] then
            cmin = j;
        end
    end
    if i ! = cmin then
        aux = numbers[cmin];
        numbers[cmin] = numbers[i];
        numbers[i] = aux;
    end
end
```

2.3 *Bubble Sort*

O algoritmo de ordenação por borbulhamento também figura entre os mais simples e tem complexidade quadrática. O algoritmo consiste em percorrer a lista de números do fim até o início, onde para cada iteração, percorre-se a lista do início até o índice atual, comparando o elemento do índice interno com o seguinte, se o seguinte for menor, troca-se os números de lugar, conforme o código do algoritmo 3:

Algorithm 3: *Bubble Sort*

```
Sendo N o tamanho do vetor numbers;
Result: Vetor de N números ordenados crescentemente.
int aux, i, j;
for i = N − 1; i >= 1; i = i − 1 do
    for j = 0; j < i; j = j + 1 do
        if numbers[j] > numbers[j + 1] then
            aux = numbers[j];
            numbers[j] = numbers[j + 1];
            numbers[j + 1] = aux;
        end
    end
end
```

2.4 *Merge Sort*

O algoritmo de ordenação *Merge Sort* é um algoritmo de divisão e conquista que está entre os mais rápidos com complexidade $O(n \log n)$. O algoritmo consiste de duas funções principais, a função recursiva de ordenação, que divide o vetor ao meio e chama novamente a mesma função para cada uma das metades do vetor e depois faz a intercalação, que consiste na outra função, que tem como objetivo, intercalar o resultado das duas metades do vetor (cada uma já ordenada) no vetor de maneira que o resultado seja um vetor do tamanho igual a soma dos tamanhos das duas metades contendo os elementos de cada uma de maneira ordenada. O algoritmo de intercalação é representado no algoritmo 4 e o algoritmo do *Merge Sort* é representado no algoritmo 5.

Algorithm 4: *Intercala*

```
Sendo W um vetor auxiliar, p o ínicio, r o fim e q a metade;
Result: Vetor com os elementos das metades ordenado.
int i, j, t, k;
k = 0;
i = p;
j = q + 1;
while (i <= q) && (j <= r) do
    if numbers[i] <= numbers[j] then
        W[k++] = numbers[i];
        i++;
    else
        W[k++] = numbers[j];
        j++;
    end
end
while (i <= q) do
    W[k++] = numbers[i];
    i++;
end
while (j <= r) do
    W[k++] = numbers[j];
    j++;
end
for t = 0; t < k; t = t + 1 do
    numbers[p + t] = W[t]
end
```

Algorithm 5: *MergeSort*

```
Sendo r o limite final do vetor e p o inicial;
Result: Vetor de N números ordenados crescentemente.
if (r − p) >= 1 then
    q = (r + p)/2;
    MergeSort(numbers, p, q);
    MergeSort(numbers, q + 1, r);
    Intercala(numbers, p, q, r);
end
```

2.5 *Quick Sort*

O algoritmo de ordenação *Quick Sort* também é um algoritmo de divisão e conquista que está entre os mais rápidos com complexidade $O(n \log n)$ sendo $O(n^2)$ no pior caso. O algoritmo consiste em definir um elemento pivô e rearranjar o vetor de maneira que todos os elementos anteriores ao vetor são menores que ele, e os posteriores são maiores, essa operação consiste na função de particionar do *Quick Sort*, com o vetor particionado, o algoritmo de ordenação consiste em chamadas recursivas para cada parte e as particionando até que sobre apenas um elemento, no final desses particionamentos recursivos, o algoritmo estará ordenado. O algoritmo de particionamento é representado no algoritmo 6 e o algoritmo do *Quick Sort* é representado no algoritmo 7.

Algorithm 6: *Particiona*

```
Sendo p o ínicio, r o fim e q a posição de particionamento;
Result: Vetor com os elementos das metades ordenado.
int x, j, i, aux;
i = p − 1;
j = r + 1;
x = numbers[p];
while (i < j) do
    while (numbers[i] < x) do
        i++;
    end
    while (numbers[j] > x) do
        j−−;
    end
    if i < j then
        aux = numbers[i];
        numbers[i] = numbers[j];
        numbers[j] = aux;
    end
end
return j;
```

Algorithm 7: *QuickSort*

```
Sendo r o limite final do vetor e p o inicial;
Result: Vetor de N números ordenados crescentemente.
if (r > p) then
    q = Particiona(numbers, p, r);
    QuickSort(numbers, p, q);
    QuickSort(numbers, q + 1, r);
end
```

2.6 *Heap Sort*

O algoritmo de ordenação *Heap Sort* é um outro algoritmo de ordenação com complexidade $O(n \log n)$ em qualquer caso, assim como o *Merge Sort*. O algoritmo se utiliza de uma estrutura de dados especial, o *Heap*, o Heap tem propriedades especiais, é uma árvore binária, com índices começando do topo e seguindo sempre da esquerda para a direita, nível após nível. Nessa árvore, cada nó pai, é maior do que seus filhos, ou seja, de acordo com a indexação da árvore (quando em um vetor) temos que o elemento do índice i é sempre maior ou igual aos elementos dos índices $i * 2$ e $i * 2 + 1$, o que caracteriza a regra 2 do *Heap*, por fim, os elementos de um *Heap* devem sempre ser adicionados com uma folha mais a esquerda possível do nível. O *Heap Sort* possui quatro operações, *RestauraHeap* que consiste em restaurar as condições de um *Heap* após uma mudança ou adição no mesmo (algoritmo 8), *ConstróiHeap* que consiste na transformação de uma lista em um *Heap*, para fazer isso, utiliza-se a restauração do *Heap* para a primeira metade dos elementos da lista (algoritmo 9), *HeapOrdena* que efetua a ordenação, começando do final da lista, a função troca o primeiro elemento (o maior do *Heap*) com o último e restaura o *Heap*, isso é feito enquanto se avalia os índices maiores que 1 (algoritmo 10) e por ultimo a função *HeapSort*, que é responsável por chamar as funções de construção e ordenação dos elementos (algoritmo 11).

Algorithm 8: *HeapRestaura*

```
Sendo N o tamanho do vetor numbers em avaliação e i o ponto do começo da
  restauralção;
Result: Devolve o vetor numbers restaurado.
int j = i, k, aux;
bool regra2 = false;
while (j <= (N/2)) && (regra2 == false) do
    if (2 * j < N) then
        if (numbers[2 * j] < numbers[2 * j + 1]) then
            k = 2 * j + 1;
        else
            k = 2 * j;
        end
    else
        k = 2 * j;
    end
    if (numbers[j] < numbers[k]) then
        aux = numbers[i];
        numbers[i] = numbers[j];
        numbers[j] = aux;
        j = k;
    else
        regra2 = true;
    end
end
```

Algorithm 9: *HeapConstroi*

```
Sendo N o tamanho do vetor numbers;
Result: Transforma o vetor em um Heap.
int i;
for i = N/2; i >= 1; i = i - 1 do
    HeapRestaura(numbers, i, N);
end
```

Algorithm 10: *HeapOrdena*

Sendo N o número de elementos de *numbers*;
Result: Ordena a lista de elementos.
int $u = N$;
while $(u > 1)$ **do**
    int $aux = numbers[1]$;
    $numbers[1] = numbers[u]$;
    $numbers[u] = aux$;
    $u = u - 1$;
    HeapRestaura$(numbers, 1, u)$;
end

Algorithm 11: *HeapSort*

Sendo N o número de elementos de *numbers*;
Result: Vetor de N números ordenados crescentemente.
HeapConstroi$(numbers, N)$;
HeapOrdena$(numbers, N)$;

3 COMPARAÇÃO ENTRE OS MÉTODOS

Para testar os algoritmos, foram feitos vários cenários de testes seguindo:

- Entradas de tamanho: 10000, 20000, 40000, 80000, 160000 e 320000.
- Para cada tamanho de entrada, um teste com vetor crescente, decrescente e aleatório.
- O tempo de medição foi apenas sobre o algoritmo de ordenação.
- Para verificar a corretude, ao final de cada ordenação, uma *flag* de corretude é dada como verdadeira, a partir do segundo elemento ordenado, compara-se este com seu anterior, se o anterior for maior a *flag* é alterada para falsa já que o vetor apresentado não está ordenado.

3.1 Bancada de testes

Sistema Operacional:Windows 10.
Memória RAM: 8GB DDR3 1600MHz.
Processador: Intel Core i7 3630qm 2.4GHz.

3.2 Resultados Obtidos

Tabela 2: Tempos (ms) obtidos para entrada de tamanho 10000.

Entrada	Insertion	Selection	Bubble	Merge	Quick	Heap
Aleatória	23	230	129	6	1	2
Crescente	0	211	33	4	45	1
Decrescente	48	208	82	4	51	1

Tabela 3: Tempos (ms) obtidos para entrada de tamanho 20000.

Entrada	Insertion	Selection	Bubble	Merge	Quick	Heap
Aleatória	94	810	575	12	2	3
Crescente	0	792	132	10	210	2
Decrescente	195	789	333	12	213	3

Tabela 4: Tempos (ms) obtidos para entrada de tamanho 40000.

Entrada	Insertion	Selection	Bubble	Merge	Quick	Heap
Aleatória	377	3272	2448	26	5	6
Crescente	0	3215	526	15	778	5
Decrescente	757	3430	1344	16	777	5

Tabela 5: Tempos (ms) obtidos para entrada de tamanho 80000.

Entrada	Insertion	Selection	Bubble	Merge	Quick	Heap
Aleatória	1523	9134	9973	46	12	18
Crescente	0	9116	2165	33	3287	11
Decrescente	3050	9219	5471	32	3314	12

Tabela 6: Tempos (ms) obtidos para entrada de tamanho 160000.

Entrada	Insertion	Selection	Bubble	Merge	Quick	Heap
Aleatória	6060	36450	40070	104	21	43
Crescente	0	40985	8800	91	13009	20
Decrescente	12227	51993	21630	80	13207	29

Tabela 7: Tempos (ms) obtidos para entrada de tamanho 320000.

Entrada	Insertion	Selection	Bubble	Merge	Quick	Heap
Aleatória	25656	168567	169213	215	47	111
Crescente	1	149000	34434	174	42744	47
Decrescente	59015	147604	121824	181	34150	51

A partir dos resultados obtidos e da Tabela 1 é possível ver a relação entre os tempos desprendidos em cada algoritmo para cada tipo de entrada com a complexidade analisada em aula. Pode-se dividir os algoritmos estudados em 2 grandes categorias, os de ordem quadrática (*Insertion, Selection e Bubble*) e os de ordem logarítmica (*Merge, Quick e Heap*).

Dentre os algoritmos de ordem quadrática é possível ver o comportamento dos mesmos a medida que aumentamos o tamanho da entrada, como a complexidade é de ordem quadrática, percebe-se a medida que dobramos o tamanho da entrada, um aumento de pelo menos 4 vezes no tempo utilizado pelo algoritmo. Vale salientar que dois desses algoritmos tem complexidade $O(n)$ no melhor caso, para entrada crescente, é possível ver que o tempo desprendido pelos algoritmos de *Inserção* e *Borbulhamento* é bem menor que o tempo desprendido por cada para os outros tipos de entradas, o que vai de encontro ao que foi descrito na Tabela 1.

Observado os algoritmos de ordem logarítmica é possível ver uma grande diferença em relação aos outros, esses algoritmos são (provados) os melhores algoritmos para ordenação. O *HeapSort* foi o algoritmo mais rápido dos testes, apesar de ser da mesma ordem de complexidade que o *MergeSort* (que também obteve ótimos resultados em todos os testes) ainda sim, se saiu melhor. O algoritmo *QuickSort*, como visto, no pior caso é de ordem quadrática, o que pode ser visto nos resultados obtidos para entrada crescente e decrescente.

3.3 Análise do QuickSort

QuickSort possui uma função de particionamento que tem um elemento chave chamado Pivô, nos testes acima, o Pivô foi sempre definido como o primeiro elemento do vetor avaliado. O índice final do Pivô, após o particionamento, desempenha o papel principal na definição da complexidade do *QuickSort*. Sabe-se que o algoritmo tem duas chamadas recursivas, uma do início até o pivô e outra do pivô até o final do vetor, para os casos de entrada crescente e decrescente, no resultado do particionamento, o pivô fica em uma das extremidades, isso faz com que a recorrência resultante seja:

$$T(n) = \begin{cases} 1, & n = 1 \\ T(1) + T(n-1) + Cn, & n > 1 \end{cases} \tag{1}$$

Conforme estudado, essa recorrência resulta em complexidade quadrática, devido ao particionamento desbalanceado para a entradas ordenadas ou inversamente ordenadas. Se alterarmos o pivô definindo como aleatório ou no meio do vetor nesses casos obteremos partições mais bem definidas que resultam na recorrência (para o caso do piso no ponto médio):

$$T(n) = \begin{cases} 1, & n = 1 \\ T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + Cn, & n > 1 \end{cases} \tag{2}$$

A recorrência da equação 2 resulta em complexidade da ordem de $O(n \log n)$. Para obter esse resultado, basta alterar a maneira que o pivô x é definido no algoritmo 6. A tabela a seguir traz os resultados para o *QuickSort* com pivô no ponto médio e pivô aleatório:

Tabela 8: Tempos (ms) obtidos para os algoritmos com pivô aleatório e pivô de ponto médio.

Entrada	QuickRandom			QuickPivoMédio		
	Aleat.	Cresc.	Decresc.	Aleat.	Cresc.	Decresc.
10000	2	0	0	0	0	0
20000	3	1	2	1	0	0
40000	5	3	3	3	0	0
80000	10	6	6	7	2	2
160000	20	12	12	14	3	3
320000	41	25	25	29	6	7

É possível ver que apenas a mudança na definição do pivô no algoritmo do *QuickSort* muda drasticamente o comportamento do mesmo no pior caso, dada as alterações, tanto o algoritmo com pivô aleatório quanto o com pivô no ponto médio mostraram-se melhores que todos os demais algoritmos avaliados nesse trabalho.

4 CONCLUSÃO

Algoritmos de ordenação tem sido muito estudados e possuem diversas aplicações. A análise de complexidade desses algoritmos é fundamental na definição de qual algoritmo utilizar para cada tipo de aplicação dentro das suas demandas de complexidade e tempo de execução.

Foram vistos algoritmos de ordem quadrática e logarítmica e pode-se perceber na prática a diferença de tempo entre os algoritmos. O pior algoritmo dentre os estudados foi o *Selection Sort* e o melhor foi a versão do algoritmo *Quick Sort* com pivô no ponto médio