

# Algoritmos de ordenação

Vinicius A. Matias

May 18, 2021

## 1 Introdução

Ordernar elementos de uma estrutura de dados é uma das tarefas mais curiosas e da computação. Diferentes implementações podem ser realizadas, das mais triviais às mais eficientes, assumindo heurísticas dos algoritmos ou não, e cada um dos algoritmos tem seu valor para um melhor entendimento da complexidade computacional.

## 2 Insertion Sort

O método de ordenação por inserção é um dos mais simples. Ele consiste em verificar de dois em dois elementos se o valor corrente é maior que o valor antecessor no arranjo, e se for, trocar e repetir o processo até o arranjo ficar ordenado.

Esse algoritmo pode ser pensado no caso de segurarmos um conjunto de cartas em uma mão e queiramos ordená-las. Aplicando o algoritmo da ordenação por inserção, verificamos a segunda carta, vemos se a primeira é maior que ela e se sim, trocamos as cartas. Na terceira carta vemos se a segunda é maior, e se for, verificamos se ela também é maior que a primeira. Esse processo é repetido para todas as cartas, passando por cada carta da esquerda para a direita e comparando da direita para a esquerda.

### 2.1 Implementação iterativa

A implementação do algoritmo iterativo pode ser vista na Listing 1.

Como a operação de interesse desse algoritmo é a comparação entre elementos do arranjo e o valor corrente, devemos identificar que existem dois laços envolvidos para computar essa operação. O laço mais externo roda entre  $i = 0$  e  $i < \text{len}(A)$ , ou seja,  $n - 1$  vezes. No melhor dos casos não haverá necessidade de trocar os elementos pois o arranjo já está ordenado, e nesse

caso só será realizada uma verificação por volta do laço, levando à  $n - 1$  iterações no melhor caso.

Listing 1: Insertion Sort iterativo

```
1 def insertion_sort(A):
2     end = len(A)
3     i = 0
4     j = 0
5
6     while i < end:
7         value = A[i]
8         j = i
9         while j > 0 and A[j-1] > value:
10             A[j] = A[j-1]
11             j = j-1
12         A[j] = value
13         i += 1
14
15     return A
```

No pior caso, para cada uma das  $n - 1$  voltas do laço será necessário verificar todos os elementos anteriores à posição atual (segundo laço). Isso implica que a primeira execução fará uma verificação, a segunda duas, a terceira três até a  $n$ -ésima, realizando  $n$  comparações. Somando o número de comparações teremos algo como  $1 + 2 + 3 + \dots + n - 1$ , que pode ser vista como uma soma de Progressão Aritmética. A soma desta PA que cresce de 1 em um pode ser definida como  $\frac{n(0+n-1)}{2} = \frac{n^2-n}{2}$ , isto é, o algoritmo de ordenação por inserção iterativo tem crescimento assintótico  $\mathcal{O}(n^2)$ .

### 2.2 Demonstração de crescimento assintótico $\mathcal{O}(n^2)$

$f(n) = \frac{n^2-n}{2} \in \mathcal{O}(n^2)$  se existem constantes  $n_0 \geq 0$  e  $c \geq 0$  que satisfazem a inequação:

$$0 \leq \frac{n^2-n}{2} \leq cn^2$$

Resolvendo  $0 \leq \frac{n^2-n}{2}$  notamos que a inequação é verdadeira para qualquer  $n \in \mathcal{R}$

Resolvendo  $\frac{n^2-n}{2} \leq cn^2$ :

$$\begin{aligned}\frac{n(n-1)}{2} &\leq cn^2 \\ \frac{n-1}{2} &\leq cn \\ \frac{-1}{2} &\leq cn - \frac{n}{2} \\ \frac{-1}{2} &\leq n(c - \frac{1}{2})\end{aligned}$$

Para a inequação ser verdadeira,  $c - \frac{1}{2} \geq 0$  deve ser verdade, logo,  $c \geq \frac{1}{2}$ .

Assim, a inequação  $\frac{-1}{2} \leq n(c - \frac{1}{2})$  é verdadeira para qualquer valor  $n \geq 0$  e  $c \geq \frac{1}{2}$ , como exemplo:

$$n_0 = 1 \text{ e } c = 1$$

$$\text{Isso prova que } \frac{n^2-n}{2} \in \mathcal{O}(n^2)$$

### 2.3 Implementação recursiva

A listing 2 apresenta uma implementação recursiva do Insertion Sort.

O algoritmo é baseado na indução fraca, garantindo que sabe-se ordenar um arranjo com um elemento, pois ele já está ordenado (caso base). Cada chamada recursiva um dos  $n - 1$  sub arranjos possíveis.

Listing 2: Insertion Sort recursivo

```

1  def insertion_sort_rec(A, n):
2      if n == 1: return
3
4      insertion_sort_rec(A, n-1)
5      i = n - 1
6      aux = 0
7
8      while i > 0 and A[i-1] > A[i]:
9          aux = A[i]
10         A[i] = A[i-1]
11         A[i-1] = aux
12         i -= 1
13
14     return A

```

Considerando que a operação de interesse é a comparação entre elementos do arranjo  $A[i - 1] > A[i]$ , podemos definir duas equações de recorrência, uma para o melhor caso e outra para o pior caso.

**Melhor caso:** O arranjo está ordenado, portanto serão feitas  $n - 1$  chamadas recursivas para um arranjo de tamanho  $n$ , e uma comparação em cada uma dessas chamadas. Para o caso base não é feita nenhuma comparação entre elementos do arranjo.

$$T(n) = \begin{cases} 0, & n = 1 \\ T(n-1) + 1, & n > 1 \end{cases}$$

A resolução dessa equação de recorrência diz que  $T(n) = n - 1$

**Pior caso:** São feitas  $n - 1$  comparação para cada  $n$  passado na recorrência.

$$T(n) = \begin{cases} 0, & n = 1 \\ T(n-1) + n - 1, & n > 1 \end{cases}$$

E a resolução dessa equação de recorrência resulta em  $T(n) = (n^2 - n)/2$ , e que  $T(n) \in \mathcal{O}(n^2)$

### 2.4 Equação de recorrência para o pior caso

Demonstraremos que o resultado da equação de recorrência abaixo é  $T(n) \in \mathcal{O}(n^2)$

$$T(n) = \begin{cases} 0, & n = 1 \\ T(n-1) + n - 1, & n > 1 \end{cases}$$

Notando que  $T(n)$  varia as chamadas recursivas de 1 em 1, calcularemos as equações de recorrência para  $n - 1$ ,  $n - 2$  e  $n - 3$

$$T(n) = T(n-1) + n - 1$$

$$T(n-1) = T(n-1-1) + n-1-1 = T(n-2) + n-2$$

$$T(n-2) = T(n-2-1) + n-2-1 = T(n-3) + n-3$$

$$T(n-3) = T(n-3-1) + n-3-1 = T(n-4) + n-4$$

Podemos aplicar esses valores na equação de recorrência  $T(n)$ :

$$T(n) = T(n-1) + n - 1$$

$$T(n) = T(n-2) + n - 2 + n - 1$$

$$T(n) = T(n-2) + 2n - 2 - 1$$

$$T(n) = T(n-3) + n - 3 + 2n - 2 - 1$$

$$T(n) = T(n-3) + 3n - 3 - 2 - 1$$

$$T(n) = T(n-4) + n - 4 + 3n - 3 - 2 - 1$$

$$T(n) = T(n-4) + 4n - 4 - 3 - 2 - 1$$

[...]

$$T(n) = T(n-i) + in + \sum_{j=1}^i -j$$

A operação  $\sum_{j=1}^i -j$  é uma soma de Progressão Aritmética, podendo ser reescrita como  $\frac{i*(-1-i)}{2}$

$$\text{Assim, } T(n) = T(n-i) + in + \frac{i*(-1-i)}{2}$$

Quando  $i = n$ :

$$T(n) = T(n-i) + in + \frac{i*(-1-i)}{2}$$

$$T(n) = T(n-n) + n^2 + \frac{n*(-1-n)}{2}$$

$$T(n) = T(0) + n^2 + \frac{(-n-n^2)}{2}$$

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

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

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

E como foi demonstrado no item 2.2,  $\frac{(n^2-n)}{2} \in \mathcal{O}(n^2)$ , portanto:

$$T(n) \in \mathcal{O}(n^2)$$

### 3 Selection Sort

A ordenação por seleção parte do último elemento de um arranjo e compara com todos os anteriores para verificar se há um elemento maior que ele e, caso exista, capturar o maior de todo o subarranjo. O índice do maior valor do subarranjo é capturado e é comparado com o índice elemento que se partiu a ordenação (da direita para a esquerda, então, o último, penúltimo etc) para verificar se são iguais, caso forem iguais não há necessidade de trocar de posições pois o maior elemento do subarranjo já está mais à direita do arranjo. Caso sejam diferentes, o algoritmo troca a posição do então último elemento do arranjo pelo maior elemento encontrado.

#### 3.1 Implementação iterativa

Uma implementação em Python do algoritmo de seleção pode ser vista na Listing 3.

Listing 3: Selection Sort iterativo

```

1 def selection_sort(A):
2     n = len(A)
3     fim = n-1
4
5     while fim > 0:
6         max = fim
7         for j in range(fim):
8             if A[j] > A[max]:
9                 max = j
10        if fim != max:
11            temp = A[fim]
12            A[fim] = A[max]
13            A[max] = temp
14        fim -= 1
15
16    return A
```

A operação de interesse aqui é a comparação entre cada elemento de um subarranjo com o máximo encontrado até então. Note que tanto o loop mais externo quanto o mais interno (que compreende a comparação  $A[j] > A[max]$ ) são

executados sempre a mesma quantidade de vezes para um mesmo  $n$ , implicando que o melhor e o pior caso sejam iguais.

O loop externo é executado  $n - 1$  vezes e a quantidade de iterações do loop interno segue uma progressão aritmética ( $n - 1, n - 2, \dots, 2, 1$ ). A soma dessas iterações é dada por  $\frac{(n-1)*(n-1+1)}{2}$ , logo,  $\frac{n^2-n}{2}$  comparações para qualquer caso. Isso implica que esta implementação  $\in \Theta(n^2)$

#### 3.2 Implementação recursiva

Uma implementação recursiva do algoritmo de ordenação por seleção é exibido na Listing 4.

Listing 4: Selection Sort recursivo

```

1 def selection_sort_rec(A, n):
2     if n == 1: return A
3
4     max = n-1
5     for i in range(n):
6         if A[i] > A[max]:
7             max = i
8
9     if max != n-1:
10        temp = A[max]
11        A[max] = A[n-1]
12        A[n-1] = temp
13
14    return selection_sort_rec(A, n-1)
```

Partindo do caso base, sabe-se ordenar um arranjo de apenas um elemento (é o próprio arranjo). Para se ordenar para mais um elemento devem ser seguidas as diretrizes do algoritmo, como foi exibido acima. A equação de recorrência para esse ordenador (para o número de comparações entre elementos do arranjo) pode ser definida como:

$$T(n) = \begin{cases} 0, & n = 1 \\ T(n-1) + n - 1, & n > 1 \end{cases}$$

Levando à  $T(n) = \frac{n^2-n}{2}$  e, tanto no melhor quanto no pior caso, à uma implementação  $\in \Theta(n^2)$ .

### 4 Bubble Sort

O Bubble Sort é possivelmente o método de ordenação mais simples dos aqui estudados, contudo também o método que tem pior desempenho em aplicações reais. O algoritmo consiste

em passar por todos os possíveis pares de elementos e comparar se um é maior que o outro.

## 4.1 Implementação iterativa

O método bolha consiste apenas em uma troca de elementos em pares, uma das implementações iterativas possíveis está na Listing 5.

Listing 5: Bubble Sort iterativo

---

```

1  def bubble_sort(A, n):
2      i = n-1
3
4      while i > 0:
5          j = 1
6          while j <= i:
7              if A[j-1] > A[j]:
8                  temp = A[j-1]
9                  A[j-1] = A[j]
10                 A[j] = temp
11                 j += 1
12             i -= 1
13
14     return A

```

---

Sendo a comparação de interesse destacada em  $A[j-1] > A[j]$ , assim como no selection sort essa operação será realizada  $n-1, n-2, \dots, 2, 1$  vezes, mudando agora quais pares de elementos são comparados. A complexidade assintótica se mantém como  $\Theta(n^2)$ , pois a quantidade de operações segue a mesma soma de PA que resulta em  $\frac{(n^2-n)}{2}$ .

## 4.2 Implementação recursiva

Listing 6: Bubble Sort recursivo

---

```

1  def bubble_sort_rec(A, n):
2      i = 1
3      while i < n:
4          if A[i] < A[i-1]:
5              temp = A[i]
6              A[i] = A[i-1]
7              A[i-1] = temp
8              i += 1
9      return bubble_sort_rec(A, n-1)

```

---

Ainda que não seja comum, o bubble sort pode ser implementado recursivamente, trocando o laço mais externo por um chamada à própria função para  $n-1$ . A equação de recorrência é a mesma do selection sort recursivo, levando à uma complexidade  $\mathcal{O}(n^2)$ .

# 5 Merge Sort

O Merge Sort é um método de ordenação que utiliza a técnica de Divisão e Conquista também chamado de ordenação por intercalação. Ele parte da hipótese de indução forte de que sabemos ordenar um conjunto de elementos no intervalo  $[1, n]$ . A ordenação é feita dividindo o arranjo no meio para gerar dois subarranjos (esquerdo e direito). Para cada subarranjo será feito o mesmo processo de dividir entre esquerdo e direito até uma chamada para um arranjo de apenas um elemento, neste caso armazenaremos o valor e poderemos partir para a chamada do lado direito. Após algumas chamadas recursivas, teremos um elemento do arranjo esquerdo e um elemento do arranjo direito. Estes dois serão passados para a função merge, que será responsável por comparar os dois elementos e ordená-los.

Esse processo cresce conforme a pilha é desempilhada, havendo comparações dos elementos do arranjo esquerdo com o direito até um limite de  $n/2$  comparações, que é o caso da primeira chamada recursiva realizada. O Merge Sort permite que ordenemos arranjos "da esquerda" e "da direita", e isso permite que na próxima comparação, os dois subarranjos passados para o merge estejam ordenados, cabendo à cada chamada do merge comparar o novo esquerdo e direito e ordená-los.

O Merge Sort normalmente é implementado em duas funções: uma para a ordenação de dois arranjos diferentes e gerar um único (*merge()*) e uma para realizar as chamadas recursivas para dividir os arranjos (*merge\_sort()*).

Listing 7: Função merge sort

---

```

1  def merge_sort(A):
2      if len(A) > 1:
3          meio = len(A) // 2
4          esq = merge_sort(A[:meio])
5          dir = merge_sort(A[meio:])
6          A = merge(esq, dir)
7      return A

```

---

Diferentes implementações podem ser obtidas para retornar o mesmo resultado. Muitos materiais utilizam variáveis de início e fim para gerir o meio do arranjo, e sendo passadas nas chamadas recursivas. Dependendo das estruturas que a linguagem fornece, a implementação pode ser mais sucinta, como é o caso de Python. O pseudocódigo do livro Algoritmos (Cormen et al.), já mencionados em outro relatório fornece

um pseudocódigo do Merge Sort que tipicamente rege as implementações em Java e C. Em Python podemos utilizar os recursos da linguagem para pegar os índices que separam o arranjo esquerdo e o arranjo direito quando temos o meio calculado.

Listing 8: Função merge

---

```

1  def merge(esq, dir):
2      i, j = 0, 0
3      ordenados = list()
4
5      while i < len(esq) and j < len(dir):
6          if esq[i] <= dir[j]:
7              ordenados.append(esq[i])
8              i += 1
9          else:
10             ordenados.append(dir[j])
11             j += 1
12
13     if i < len(esq):
14         ordenados += esq[i:]
15
16     elif j < len(dir):
17         ordenados += dir[j:]
18
19     return ordenados

```

---

O merge funciona verificando o arranjo esquerdo e o direito para identificar quais valores são menores (do arranjo esquerdo ou do direito) e até quando. Os valores são armazenados em uma lista que será retornada ao no método *merge.sort()*. Também são verificados se houveram valores que não foram inseridos na comparação, ou seja, os valores do arranjo esquerdo ou direito que eram maiores que o maior valor do outro subarranjo.

Para analisar a complexidade do merge sort precisamos notar que ele realiza duas chamadas recursivas para  $n/2$  e uma chamada ao merge para cada  $n$ . Considerando a operação de interesse a comparação  $esq[i] \leq dir[j]$  do laço da função merge, teremos um custo para essa função de  $n - 1$  no pior caso e  $n/2$  no melhor caso. A equação de recorrência para o pior caso é:

$$T(n) = \begin{cases} 0, & n \leq 1 \\ 2T(n/2) + n - 1, & n > 1 \end{cases}$$

E calculando essa equação de divisão e conquista pelo Teorema Mestre (cláusula 2) descobrimos que a complexidade assintótica do merge sort para todos os casos  $\in \Theta(n * \log n)$ . Vale lembrar que para a generalização é necessário calcular a equação de recorrência para o melhor

caso também, que terá a mesma complexidade que o pior caso.

É importante observar que ainda que a complexidade do Merge Sort seja menor que o dos outros estudados até então, esse algoritmo tem um custo de criar novos arranjos para manipular os elementos durante a execução do método.