

Universidade de São Paulo  
Instituto de Matemática e Estatística  
Departamento de Ciência da Computação  
Bacharelado em Ciência da Computação  
MAC110 - Introdução à Computação

# **Análise de diferentes algoritmos de ordenação**

Aluno(a): João Viktor Souza Almeida

Professor: Roberto Hirata Junior



## Resumo

O relatório a seguir visa dissertar acerca algoritmos de ordenação clássicos, tais como bolha, contagem, inserção e seleção, bem como investigar exemplos, apresentar a implementação em Python e elucidar como o algoritmo se comporta ao ordenar uma lista. Além disso, é verificado como suas eficiências mudam dadas diferentes condições iniciais, tais como porcentagem de ordenação e composição da lista a ser ordenada.

## Metodologia

Na criação do relatório, os testes dos algoritmos foram realizados utilizando a linguagem de programação Python, com a versão 3.10.11, no sistema operacional Windows. Durante a execução dos algoritmos, foi utilizada uma máquina com as seguintes configurações:

- Processador (CPU): Ryzen 5 3350G
- Memória (RAM): 16GB DDR4 @ 3200MHz
- Armazenamento: 256GB SSD
- Placa de Vídeo (GPU): Radeon Vega 11
- Sistema Operacional: Windows 11 Pro

Adicionalmente, durante a execução dos algoritmos, foram desabilitados o máximo de programas possíveis a fim de minimizar as interferências nos resultados dos testes.

Para computar a média de execução, foi invocada uma função *timeMe*, na qual eram armazenados os resultados das 10 itinações do mesmo algoritmo e, no final, retornava a média e o desvio padrão dos resultados.

Para fins simplifcatórios, foram ignorados possíveis margens de erros da função que retorna a lista embaralhada. Contudo, essa margem de erro torna-se desprezível durante os testes devido ao tamanho das listas utilizadas.

## Testes

Para a realização do relatório, foram realizados dois testes.

O primeiro focou em observar os impactos causados pelo tamanho da lista a ser ordenada, ou seja, como o desvio padrão e a média mudam ao aumentar ou diminuir o tamanho da lista a ser ordenada por cada algoritmo. Para isso, foi

criada uma malha de repetição e computados as médias e os desvios-padrões de cada algoritmo com listas de tamanhos 1000, 5000, 10000, 50000 e 100000.

O segundo, por outro lado, visou elucidar acerca das mudanças causadas pela taxa de ordenação de uma lista. Para isso, foi estabelecida uma segunda malha de repetição, na qual também foi computada as médias e os desvios-padrões de cada algoritmo, mas com uma lista de 100000 elementos e com porcentagens de ordenação de 1%, 3%, 5%, 10% e 50%.

Para computar a média, foi criada uma função cujos parâmetros são uma lista e o tamanho desta, respectivamente. O desvio padrão, sob o mesmo ponto de vista, foi calculado por meio de uma função que recebe os mesmos parâmetros que a função outrora citada.

**Palavras-chave:** insertion, bubble, counting, selection, algoritmos, ordenação, análise;

# Conteúdo

<b>1</b>	<b>Algoritmo de seleção</b>	<b>4</b>
1.1	Exemplo . . . . .	4
1.2	Implementação . . . . .	5
1.3	Quantidade de comparações . . . . .	5
<b>2</b>	<b>Algoritmo de bolha</b>	<b>6</b>
2.1	Exemplo . . . . .	6
2.2	Implementação . . . . .	7
2.3	Quantidade de comparações . . . . .	7
<b>3</b>	<b>Algoritmo de inserção</b>	<b>8</b>
3.1	Exemplo . . . . .	8
3.2	Implementação . . . . .	9
3.3	Quantidade de comparações . . . . .	9
<b>4</b>	<b>Algoritmo de contagem</b>	<b>10</b>
4.1	Exemplo . . . . .	10
4.2	Implementação . . . . .	11
4.3	Quantidade de comparações . . . . .	11
<b>5</b>	<b>Resultados</b>	<b>12</b>
5.1	Variação na quantidade de elementos . . . . .	12
5.2	Ordenação prévia . . . . .	13
<b>6</b>	<b>Conclusão</b>	<b>14</b>

# 1 Algoritmo de seleção

O algoritmo de seleção é um algoritmo de ordenação no qual, a cada iteração, o menor elemento da lista é garantido estar na posição correta. Ou seja, na primeira iteração, assegura-se que o menor elemento ficará na primeira posição da lista, na segunda iteração, o segundo elemento, e assim por diante. Uma observação pertinente é que a quantidade de comparações a ser feita independe da porcentagem de ordenação da lista.

## 1.1 Exemplo

Seja  $M$  uma lista `[4,3,2,1,0]`. Na primeira passagem, o algoritmo detectará o menor elemento da lista e, logo após, irá permutá-lo com o primeiro elemento da lista, ou seja, o elemento 4 e 0 serão trocados de lugar. Na segunda passagem, começando pelo segundo elemento, a segunda malha de repetição irá detectar o segundo menor elemento e, similarmente, irá permutá-lo com o segundo elemento da lista. Nas próximas execuções, a forma é análoga.

Na  $n$ -ésima passagem, onde  $n$  é o tamanho da lista, todos os elementos estarão em ordem crescente e, portanto, ordenados.

Assim, seguem as impressões da lista a cada modificação.

```
1 [4, 3, 2, 1, 0]
2 [0, 3, 2, 1, 4]
3 [0, 1, 2, 3, 4]
4 [0, 1, 2, 3, 4]
5 [0, 1, 2, 3, 4]
6 [0, 1, 2, 3, 4]
```

Percebe-se que, mesmo a lista já estando ordenada, o algoritmo continuará realizando os comandos, ou seja, o número de passagens a ser feita não é mudado devido às condições iniciais de ordenação.

## 1.2 Implementação

A seguir, segue a implementação do código em Python.

```
1 def selection(V, n):
2     for i in range(0, n):
3         smallest_num_index = i
4         for j in range(i, n):
5             if V[j] < V[smallest_num_index]:
6                 smallest_num_index = j
7         V[i], V[smallest_num_index] = V[
            smallest_num_index], V[i]
```

## 1.3 Quantidade de comparações

Nesta subseção, irá ser debatida a quantidade de comparações feitas pelo algoritmo em questão. Como a quantidade de comparações é "constante" neste algoritmo, será "fácil" de analisá-la; nos outros algoritmos, contudo, a análise é mais complexa devido à oscilação na quantidade de comparações.

Suponha-se que o algoritmo recebeu uma lista com  $n$  elementos. Assim, como não há um dispositivo que pare o código antes, tem-se que a primeira malha de repetição será executada  $n$  vezes, em que o índice inicial  $i = 1$ , na segunda, quando o índice  $i = 2$  e assim por diante. Por consequência, em cada uma dessas execuções, a segunda malha (que está dentro da primeira) será executada  $n - i$  vezes, ou seja, na primeira execução, o código contido na segunda malha de repetição será executado  $n$  vezes, na segunda,  $n - 1$ , e assim por diante. Dessa forma, o código será executado  $n + (n - 1) + (n - 2) + \dots + (1)$ , ou seja,  $\sum_{i=1}^n i = \frac{n(n+1)}{2}$

Notavelmente, não há variações na quantidade de comparações, ou seja, independente do estado inicial da lista, a forma como o código será executado é a mesma.

## 2 Algoritmo de bolha

O algoritmo de bolha é um algoritmo cuja complexidade é, assim como o anterior, quadrática. Sua principal característica é que, na  $n$ -ésima passagem pela lista, o algoritmo assegura estarem ordenados os  $n$  últimos elementos da lista. Um fato interessante é que o nome do algoritmo faz referência às bolhas em bebidas, uma vez que as bolhas maiores sobem mais rapidamente do que as menores.

### 2.1 Exemplo

Seja  $M$  uma lista  $[4,3,2,1,0]$ . Na primeira passagem da primeira malha de repetição, será detectado que os elementos 4 e 3 estão em posições erradas e, assim serão trocados, deixando a lista como  $[3,4,2,1,0]$ . Posteriormente, será detectado que os elementos 4 e 2 estão trocados e, assim como no primeiro passo, ambos serão permutados e, conseqüentemente, a lista tornar-se-á  $[3,2,4,1,0]$ . O algoritmo continuará realizando estes passos, isto é, trocar o elemento  $j$  com o  $j + 1$  se o primeiro for maior que o segundo, até que, na 5ª passagem, (pois a lista possui 5 elementos), a lista estará totalmente ordenada.

Destarte, seguem as impressões da lista a cada modificação realizada.

```
1 [4, 3, 2, 1, 0]
2 [3, 4, 2, 1, 0]
3 [3, 2, 4, 1, 0]
4 [3, 2, 1, 4, 0]
5 [3, 2, 1, 0, 4]
6 [2, 3, 1, 0, 4]
7 [2, 1, 3, 0, 4]
8 [2, 1, 0, 3, 4]
9 [1, 2, 0, 3, 4]
10 [1, 0, 2, 3, 4]
11 [0, 1, 2, 3, 4]
```

Perceptivelmente, o maior elemento é, a cada modificação, levado uma posição para a direita até estar posição correta.



## 2.2 Implementação

Segue, a seguir, o código utilizado na implementação do algoritmo em questão em Python:

```
1 def bubble(V, n):
2     lim = n - 1
3     while lim >= 0:
4         isIncreasing = True
5         for j in range(lim):
6             if V[j] > V[j + 1]:
7                 isIncreasing = False
8                 V[j], V[j + 1] = V[j + 1], V[j]
9         if (isIncreasing == True):
10            break
11        lim -= 1
```

Caso a lista já esteja ordenada, o `isIncreasing` continuará, na primeira execução da malha de repetição, com o valor `True` e, por isso, o comando `break` será executado, parando, assim, a ordenação.

Constata-se, dessa forma, que, ao contrário do algoritmo anterior, este não realizará todas as etapas caso a lista já esteja ordenada, pois a variável `isIncreasing` funciona como um verificador.

## 2.3 Quantidade de comparações

Como a quantidade de comparações realizadas pelo bolha depende da lista, analisar-se-ão os casos.

Caso a lista com  $n$  elementos já esteja 100% ordenada, o algoritmo detectará a ordenação na primeira passagem, pois a variável `isIncrease` continuará com o seu valor booleano verdadeiro após a malha de repetição mostrada na implementação, ou seja, serão realizadas  $n$  comparações.

Por outro lado, na hipótese da lista está totalmente desordenada, haverá  $\frac{n(n+1)}{2}$  comparações, pois, pelo mesmo motivo do inserção, serão feitas  $n$  comparações na primeira iteração,  $n - 1$  na segunda, e assim sucessivamente até a última, na qual será feita 1 comparação.

Finalmente, nos outros casos, a análise é mais complexa e depende de outros fatores que não estão no escopo do relatório. Contudo, em casos nos quais a lista não está nem ordenada, nem totalmente desordenada, o algoritmo possui um comportamento quadrático[1].

### 3 Algoritmo de inserção

O algoritmo de inserção possui, como principal característica, a asseguaração de que, a cada iteração  $n$ , os  $n$ -ésimos primeiros elementos estarão ordenados. Além disso, assim como o de bolha, este varia com a porcentagem de ordenação da lista recebida, ou seja, caso a lista já esteja ordenada, será detectado logo na primeira passagem e, assim, não haverá nada a ser feito.

#### 3.1 Exemplo

Seja, para fins de exemplificação,  $V$  a lista  $[4, 3, 2, 1, 0]$ .

O algoritmo em questão funcionará da seguinte forma: a lista começa dada como ordenada até que se ache um  $j$  tal que  $V[j] > V[j + 1]$ , onde  $j$  é um inteiro maior ou igual a zero e menor que o tamanho da lista menos um. Caso isso ocorra, o algoritmo trocará os dois valores e comparará, da mesma forma,  $V[j - 1]$  e  $V[j]$ . Quando o valor da antiga posição  $j$  for menor do que a posição sucessora, garante-se, então, que a lista está ordenada de 0 até  $j + 1$ . Contudo, caso não exista um  $j$ , então o algoritmo indica que a lista já está ordenada e, assim, evita mais comparações.

Logo, tomando a lista  $M$ , caso esta fosse impressa a cada modificação, os resultados seriam:

```
1 [4, 3, 2, 1, 0]
2 [3, 4, 2, 1, 0]
3 [3, 2, 4, 1, 0]
4 [2, 3, 4, 1, 0]
5 [2, 3, 1, 4, 0]
6 [2, 1, 3, 4, 0]
7 [1, 2, 3, 4, 0]
8 [1, 2, 3, 0, 4]
9 [1, 2, 0, 3, 4]
10 [1, 0, 2, 3, 4]
11 [0, 1, 2, 3, 4]
```

Visivelmente, observa-se que, sempre que é encontrado um elemento  $e$  menor que o anterior, ambos são trocados e, em seguida, o algoritmo começa a verificar e mover  $e$  para trás até o elemento em questão fique em sua posição correta.

## 3.2 Implementação

Como se pode perceber, quando é encontrado um elemento  $j$  menor do que o anterior, este é levado para trás até que esteja na posição correta

```
1 def insertion(V, n):
2     last_index = 0
3     for i in range(last_index, n - 1):
4         if V[i] > V[i + 1]:
5             j = i
6             while V[j] > V[j + 1]:
7                 V[j + 1], V[j] = V[j], V[j + 1]
8                 if j <= 0:
9                     break
10            j -= 1
```

## 3.3 Quantidade de comparações

Tendo em vista que as comparações realizadas depende da lista, os casos serão analisados.

Caso a lista com  $n$  elementos esteja em ordem, o algoritmo detectará na primeira iteração devido à permanência do valor verdadeiro da variável `isIncreasing`, ou seja, serão realizadas  $n$  comparações.

Por outro lado, na hipótese da lista está totalmente desordenada, haverá  $\frac{n(n+1)}{2}$ . A prova é análoga à do bolha, uma vez que haverá  $n$  comparações na primeira execução,  $n-1$  na segunda, e assim por diante, ou seja, serão realizadas  $n + (n-1) + (n-2) + \dots + 1 = \frac{n(n+1)}{2}$  comparações.

Finalmente, nos outros casos, a análise é, assim como no algoritmo anterior, complexa. Contudo, em cenários nos quais a lista não se enquadra nos casos supracitados, o algoritmo possui um comportamento quadrático[2].

## 4 Algoritmo de contagem

Diferentemente dos algoritmos outrora apresentados, o algoritmo em tópico possui uma complexidade linear, isto é, o tempo levado para ordenar uma lista, ao contrário dos apresentados, não cresce de maneira quadrática.

Em relação ao uso de memória, contudo, há uma grande desvantagem no algoritmo, uma vez que a lista auxiliar utilizada possuirá  $\max lista - \min lista$  elementos; ou seja, caso o maior elemento seja 5000 e o menor 1000, a lista auxiliar possuirá de 4000 elementos.

Dessa forma, o algoritmo possui um melhor proveito se utilizado para listas com pouca variação de tamanho entre os seus elementos.

### 4.1 Exemplo

Seja  $M$  a lista  $[4, 2, 3, 1, 4, 2, 2, 0]$ . Como a diferença entre o maior e o menor é elemento é 4, a lista auxiliar  $Aux$  será de tamanho 4. Assim, a lista auxiliar será da seguinte forma (assumindo que o primeiro elemento é  $Aux[0]$ ): o primeiro elemento terá o valor igual à quantidade de zeros na lista original, que é 1, o segundo elemento também será 1 por motivo análogo, o terceiro elemento de  $Aux$ , por outro lado, receberá o valor 3, uma vez que há três elementos 2 na lista original. Essa lógica seguirá até chegar no último elemento. Assim, de início, a lista  $Aux$  será  $[1, 1, 3, 1, 2]$ .

Em seguida, começará a modificação da lista original: serão adicionados à lista original  $n$  vezes o elemento referente ao índice da lista auxiliar, ou seja, como  $Aux[0]$  é igual a 1, será adicionado um zero a  $M$  (adicionando pela direita e um ao lado do outro). Em  $Aux[2]$ , por exemplo, o elemento é 3, assim, será adicionado o elemento 2 três vezes seguidas à lista  $M$ .

Seguem os valores de `Aux` a cada modificação e:

```
1 [0, 0, 0, 0, 0]
2 [1, 0, 0, 0, 0]
3 [1, 1, 0, 0, 0]
4 [1, 1, 3, 0, 0]
5 [1, 1, 3, 1, 0]
6 [1, 1, 3, 1, 2]
```

Para fins elucidativos, serão substituídos por "0" os elementos já existentes em  $M$  a fim de mostrar os valores adicionados posteriormente, ou seja, supor-se-á que a lista  $M$  teve todos os seus valores trocados para 0.

```
1 [0, 0, 0, 0, 0, 0, 0, 0]
2 [0, 0, 0, 0, 0, 0, 0, 0]
3 [0, 1, 0, 0, 0, 0, 0, 0]
4 [0, 1, 2, 0, 0, 0, 0, 0]
5 [0, 1, 2, 2, 0, 0, 0, 0]
6 [0, 1, 2, 2, 2, 0, 0, 0]
7 [0, 1, 2, 2, 2, 3, 0, 0]
8 [0, 1, 2, 2, 2, 3, 4, 0]
9 [0, 1, 2, 2, 2, 3, 4, 4]
```

Percebe-se, portanto, que a lista foi ordenada realizando-se nenhuma comparação.

## 4.2 Implementação

Segue a implementação do algoritmo em questão em Python.

```
1 def counting(V, n):
2     max_element = max(V)
3     hist_list = [0 for _ in range(max_element + 1)]
4     for i in range(max_element + 1):
5         hist_list[i] = count_element_in_array(i, V)
6     index = 0
7     for i in range(max_element + 1):
8         for _ in range(hist_list[i]):
9             V[index] = i
10            index += 1
```

Como se pode perceber, o algoritmo de contagem, assim como os outros, possui duas malhas de repetição. Contudo, em vez de as malhas estarem aninhada, uma ocorre após a outra. Apesar disso, uma singularidade desse algoritmo é que, ao contrário dos outros, a quantidade de iterações na primeira malha de repetição depende do tamanho do maior elemento (pensando apenas na implementação com números positivos), tornando-o eficaz para listas grandes e com elementos menores.

## 4.3 Quantidade de comparações

Diferentemente de todos os algoritmos outrora citados, este não realiza qualquer comparação.

## 5 Resultados

Nesta seção, será debatido acerca dos resultados obtidos por meio dos dois testes realizados.

O primeiro visou analisar o tempo médio de cada algoritmo, bem como a sua variância.

No segundo teste, por outro lado, buscou-se avaliar o comportamento dos algoritmos bolha e inserção quando submetidos a listas com diferentes taxas de ordenação, sendo elas 1%, 3%, 5%, 10% e, por fim, 50%.

Logo abaixo, têm-se os resultados em gráficos.

### 5.1 Variação na quantidade de elementos

Como comentado, os resultados comprovam o comportamento quadrático do bolha, seleção e inserção. Contudo, é necessário destacar que, caso houvesse uma maior variância nos números da lista a ser ordenada, isto é, caso a diferença entre o maior e o menor valor fosse um valor muito grande, o algoritmo de contagem apresentaria tempo para ordenar maior, podendo ultrapassar o bolha, por exemplo.

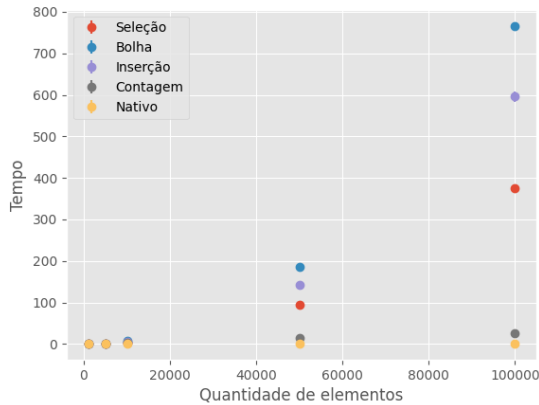


Figura 1: Gráfico elucidando o comportamento de cada algoritmo em tópico. No eixo x, há a quantidade de elementos; no eixo y, o tempo médio gasto em segundos

## 5.2 Ordenação prévia

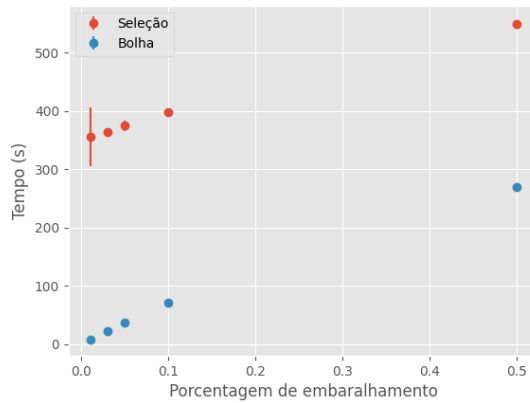


Figura 2: Gráfico ilustrando o tempo médio 'gasto' dos algoritmos bolha e inserção em relação à taxa de ordenação inicial dos vetores.

O bolha, apesar de variar dependendo da taxa de ordenação, possui o maior tempo necessário para ordenar. Em seguida, o algoritmo de inserção detém o segundo maior tempo para ordenar, e isso se dá pelo fato de que o inserção possui uma complexidade quadrática independente da posição inicial dos elementos.

O algoritmo de seleção, como esperado, contém o melhor tempo de resposta dentre os algoritmos de comparação, uma vez que, além de ser mais eficiente em listas com alguns elementos ordenados, realiza menos permutações que o bolha.

Por último, o contagem apresentou o melhor desempenho dentre os implementados. Contudo, é necessário destacar que o contagem obteve este desempenho devido à faixa de números aleatórios escolhida (de 0 a 9999), pois, como analisado, o algoritmo em questão varia com os números recebidos.

Assim, percebe-se que, para listas com muitos elementos, o algoritmo contagem é o mais eficiente entre os implementados, enquanto o bolha apresentou o resultado menos satisfatório para o teste.

## 6 Conclusão

Desta maneira, apesar de possuírem diferentes desempenhos dependendo da ordenação da lista, os algoritmos em questão, possuem um baixo proveito com listas muito grandes. Portanto, a fim de ordenar uma lista com maior rapidez, é notável que a função built-in do Python deve, pelo menos para listas grandes, ser usada em relação aos outros métodos outrora discutidos.

Adicionalmente, o algoritmo de contagem, ainda que possua um comportamento linear em relação ao tamanho das listas, pode sofrer variações em razão da diferença entre o maior e o menor elemento da lista, isto é, ainda que a lista seja relativamente pequena, o tempo de realização do algoritmo pode ser muito alto caso dependendo da variação entre os números. Assim, é aconselhável a utilização do algoritmo em pauta apenas com listas que possuem uma exígua variação.

Em outra perspectiva, o bolha foi o algoritmo que apresentou o menor rendimento quando submetido a listas com diferentes tamanhos; o contagem, por outro lado, garantiu, apesar da faixa de valores maior, o melhor desempenho, principalmente nas listas com mais elementos. Dessa forma, o bolha é recomendado apenas para fins didáticos.



## Referências

- [1] Bubble Sort Time Complexity and Algorithm Explained, builtin, 2023. Disponível em: [https://builtin.com/data-science/bubble-sort-time-complexity#:~:text=The%20bubble%20sort%20algorithm%27s%20average,complexity%3A%20O\(n%C2%B2\).](https://builtin.com/data-science/bubble-sort-time-complexity#:~:text=The%20bubble%20sort%20algorithm%27s%20average,complexity%3A%20O(n%C2%B2).) Acesso em: 08 de jun. de 2024.
- [2] Insertion Sort Explained—A Data Scientists Algorithm Guide, 2021. Disponível em: [https://developer.nvidia.com/blog/insertion-sort-explained-a-data-scientists-algorithm-guide/#:~:text=The%20worst%2Dcase%20\(and%20average,O\(n\)%20time%20complexity..](https://developer.nvidia.com/blog/insertion-sort-explained-a-data-scientists-algorithm-guide/#:~:text=The%20worst%2Dcase%20(and%20average,O(n)%20time%20complexity..) Acesso em: 08 de jun. de 2024.