


Ordenação e busca em arrays

3.1 DEFINIÇÃO


A ordenação nada mais é do que o ato de colocar um conjunto de dados em determinada ordem predefinida, como mostra o exemplo a seguir:

5, 2, 1, 3, 4: FORA DE ORDEM.
1, 2, 3, 4, 5: ORDENADO.



A ordenação permite que o acesso aos dados seja feita de forma mais eficiente.

A ordenação de um conjunto de dados é feita utilizando como base uma chave específica.




A chave de ordenação é o “campo” do item utilizado para comparação. É por meio dele que sabemos se determinado elemento está à frente ou não de outros no conjunto ordenado.

Para realizar a ordenação, podemos usar qualquer tipo de chave, desde que exista uma regra de ordenação bem definida. Existem vários tipos possíveis de ordenação. Os tipos de ordenação mais comuns são:

Numérica: 1, 2, 3, 4, 5.
Lexicográfica (ordem alfabética): Ana, André, Bianca, Ricardo.

Além disso, independente do tipo, a ordenação pode ser:

- Crescente:
 - 1, 2, 3, 4, 5.
 - Ana, André, Bianca, Ricardo.
- Decrescente:
 - 5, 4, 3, 2, 1.
 - Ricardo, Bianca, André, Ana.




Um algoritmo de ordenação é aquele que coloca os elementos de dada sequência em certa ordem predefinida.

Existem vários algoritmos para realizar a ordenação dos dados. Eles podem ser classificados como de **ordenação interna (in-place)** ou **externa**:

- **Ordenação interna:** o conjunto de dados a ser ordenado cabe todo na memória principal. Qualquer elemento pode ser imediatamente acessado.

- **Ordenação externa:** o conjunto de dados a ser ordenado não cabe na memória principal (está armazenado em memória secundária, por exemplo, em um arquivo). Os elementos são acessados sequencialmente ou em grandes blocos.

Além disso, um algoritmo de ordenação pode ser considerado estável ou não.

	Um algoritmo de ordenação é considerado estável se a ordem dos elementos com chaves iguais não muda durante a ordenação.
---	--

Imagine um conjunto de dados não ordenado com dois valores iguais, no caso, 5a e 5b:

5a, 2, 5b, 3, 4, 1: **dados não ordenados.**

Um algoritmo de ordenação será considerado **estável** se o valor **5a** vier antes do valor **5b** quando esse conjunto de dados for ordenado de forma crescente, ou seja, o algoritmo preserva a ordem relativa original dos valores:

1, 2, 3, 4, **5a, 5b**: **ordenação estável.**


1, 2, 3, 4, **5b, 5a**: **ordenação não estável.**

Nas próximas seções, iremos abordar alguns dos principais algoritmos de ordenação de dados armazenados em arrays (ordenação interna).

3.2 ALGORITMOS BÁSICOS DE ORDENAÇÃO

3.2.1 Ordenação por “bolha”: bubble sort

O algoritmo bubble sort, também conhecido como ordenação por “bolha”, é um dos algoritmos de ordenação mais conhecidos que existem. Ele tem esse nome por remeter à ideia de bolhas flutuando em um tanque de água em direção ao topo, até encontrarem o seu próprio nível (ordenação crescente).



O algoritmo bubble sort trabalha de forma a movimentar, uma posição por vez, o maior valor existente na porção não ordenada de um array para a sua respectiva posição no array ordenado. Isso é repetido até que todos os elementos estejam nas suas posições correspondentes.

A Figura 3.1 mostra a implementação do algoritmo bubble sort. O princípio de funcionamento deste algoritmo é a troca de valores em posições consecutivas de array para que, desse modo, fiquem na ordem desejada. Para entender esse processo, imagine um seguinte conjunto de valores não ordenados, como mostrado na Figura 3.2. Ao se comparar os dois primeiros valores, percebe-se que eles não estão ordenados (ordem crescente). Então, o algoritmo os troca de lugar. O processo é repetido para cada par de valores em posições consecutivas do array (linhas 5-12). Ao final do processo, teremos o maior valor na última posição do array. Falta ordenar o restante do array. Isso é feito diminuindo o valor da variável **fim** (linha 2) que está associada à última posição do array (linha 13). Em seguida, executamos todo o processo de levar o maior valor até o final do array com um comando **do-while** (linhas 3-14).

Método bubble sort

```
01 void bubbleSort(int *V , int N){
02     int i, continua, aux, fim = N;
03     do{
04         continua = 0;
05         for(i = 0; i < fim-1; i++){
06             if(V[i] > V[i+1]){//anterior > próximo? Mudar!
07                 aux = V[i];
08                 V[i] = V[i+1];
09                 V[i+1] = aux;
10                 continua = i;
11             }
12         }
13         fim--;
14     }while(continua != 0);
15 }
```

FIGURA 3.1

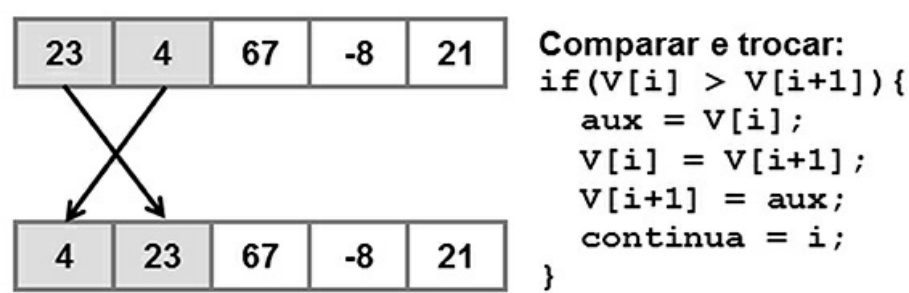


FIGURA 3.2



Sempre que uma troca de valores ocorrer, a variável continua será modificada (linha 10) em relação ao seu valor original (linha 4). Desse modo, se nenhuma troca de valores ocorrer, o algoritmo poderá ser finalizado mais cedo.

A Figura 3.3 mostra um exemplo de ordenação completa de um array em ordem crescente. Os itens coloridos são os valores comparados, enquanto os hachurados representam a porção já ordenada do array. Percebe-se que:

- Primeira iteração do comando **do-while**: encontra-se o maior valor e o movimenta até a última posição.
- Segunda iteração do comando **do-while**: encontra-se o segundo maior valor e o movimenta até a penúltima posição.
- Esse processo continua até que todo o array esteja ordenado.

Sem Ordenar

23	4	67	-8	21
----	---	----	----	----

1º Iteração do-while

i=0

23	4	67	-8	21
----	---	----	----	----

 $V[i] > V[i+1]$: Trocar

i=1

4	23	67	-8	21
---	----	----	----	----

 $V[i] < V[i+1]$: Manter

i=2

4	23	67	-8	21
---	----	----	----	----

 $V[i] > V[i+1]$: Trocar

i=3

4	23	-8	67	21
---	----	----	----	----

 $V[i] > V[i+1]$: Trocar

Final

4	23	-8	21	67
---	----	----	----	----

2º Iteração do-while

i=0

4	23	-8	21	67
---	----	----	----	----

 $V[i] < V[i+1]$: Manter

i=1

4	23	-8	21	67
---	----	----	----	----

 $V[i] > V[i+1]$: Trocar

i=2

4	-8	23	21	67
---	----	----	----	----

 $V[i] > V[i+1]$: Trocar

Final

4	-8	21	23	67
---	----	----	----	----

3º Iteração do-while

i=0

4	-8	21	23	67
---	----	----	----	----

 $V[i] > V[i+1]$: Trocar

i=1

-8	4	21	23	67
----	---	----	----	----

 $V[i] < V[i+1]$: Manter

Final

-8	4	21	23	67
----	---	----	----	----

4º Iteração do-while

i=0

-8	4	21	23	67
----	---	----	----	----

 $V[i] < V[i+1]$: Manter

Não houve mudanças: ordenação concluída

Ordenado

-8	4	21	23	67
----	---	----	----	----



O bubble sort é um algoritmo simples e de fácil entendimento e implementação. Além disso, está entre os mais difundidos métodos de ordenação existentes. Infelizmente, não é um algoritmo eficiente, sendo estudado apenas para fins de desenvolvimento de raciocínio.

Isso ocorre porque sua eficiência diminui drasticamente à medida que o número de elementos no array aumenta. Ou seja, ele não é recomendado para aplicações que envolvam grandes quantidades de dados ou que precisem de velocidade. Considerando um array com N elementos, o tempo de execução do bubble sort é:

- $O(N)$, melhor caso: os elementos já estão ordenados.
- $O(N^2)$, pior caso: os elementos estão ordenados na ordem inversa.
- $O(N^2)$, caso médio.

3.2.2 Ordenação por seleção: selection sort

O algoritmo selection sort, também conhecido como ordenação por “seleção”, é outro algoritmo de ordenação bastante simples. Ele tem esse nome, pois a cada passo “seleciona” o melhor elemento (maior ou menor, dependendo do tipo de ordenação) para ocupar aquela posição do array. Na prática, este algoritmo possui um desempenho quase sempre superior quando comparado com o bubble sort.



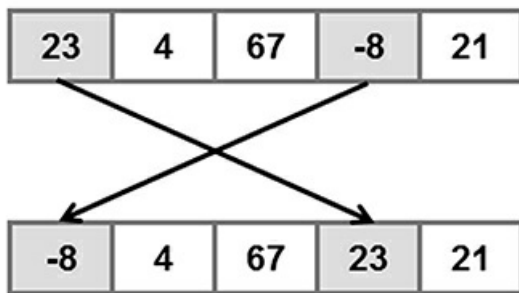
O algoritmo selection sort divide o array em duas partes: a parte ordenada, à esquerda do elemento analisado, e a parte que ainda não foi ordenada, à direita do elemento. Para cada elemento do array, começando do primeiro, o algoritmo procura na parte não ordenada (direita) o menor valor (ordenação crescente) e troca os dois valores de lugar. Em seguida, o algoritmo avança para a próxima posição do array e esse processo é feito até que todo o array esteja ordenado.

A Figura 3.4 mostra a implementação do algoritmo selection sort. O princípio de funcionamento deste algoritmo é a seleção do melhor elemento para ocupar aquela posição do array. Para entender esse processo, imagine um conjunto de valores não ordenados, como mostrado na Figura 3.5. Perceba que o valor da primeira posição do array (23) não está na sua posição correta, pois existem valores menores do que ele na porção não ordenada do array (direita). Então, o algoritmo percorre os elementos da direita procurando o índice do **menor** valor dentre todos (linhas 4-8). Ao chegar ao final do array, o algoritmo troca os valores do elemento atual, índice **i**, com o menor valor encontrado, índice **menor** (linhas 9-13). O processo de comparar o valor de uma posição do array com seus sucessores é repetido para cada posição do array (linhas 3-14).

Método selection sort

```
01 void selectionSort(int *V, int N){
02     int i, j, menor, troca;
03     for(i = 0; i < N-1; i++){
04         menor = i;
05         for(j = i+1; j < N; j++){
06             if(V[j] < V[menor])
07                 menor = j;
08         }
09         if(i != menor){
10             troca = V[i];
11             V[i] = V[menor];
12             V[menor] = troca;
13         }
14     }
15 }
```

FIGURA 3.4



Procura o menor valor a direita:

```
menor = i;
for(j = i+1; j < N; j++){
    if(V[j] < V[menor])
        menor = j;
}
```

Troca os valores de lugar:

```
if(i != menor){
    troca = V[i];
    V[i] = V[menor];
    V[menor] = troca;
}
```

FIGURA 3.5

A Figura 3.6 mostra um exemplo de ordenação completa de um array em ordem crescente. Para cada valor da variável *i* temos o array antes e depois de ser selecionado o menor valor para ocupar aquela posição. Os itens coloridos representam o valor da posição de índice *i* e a posição com o menor valor encontrado na porção não ordenada do array (à direita de *i*).

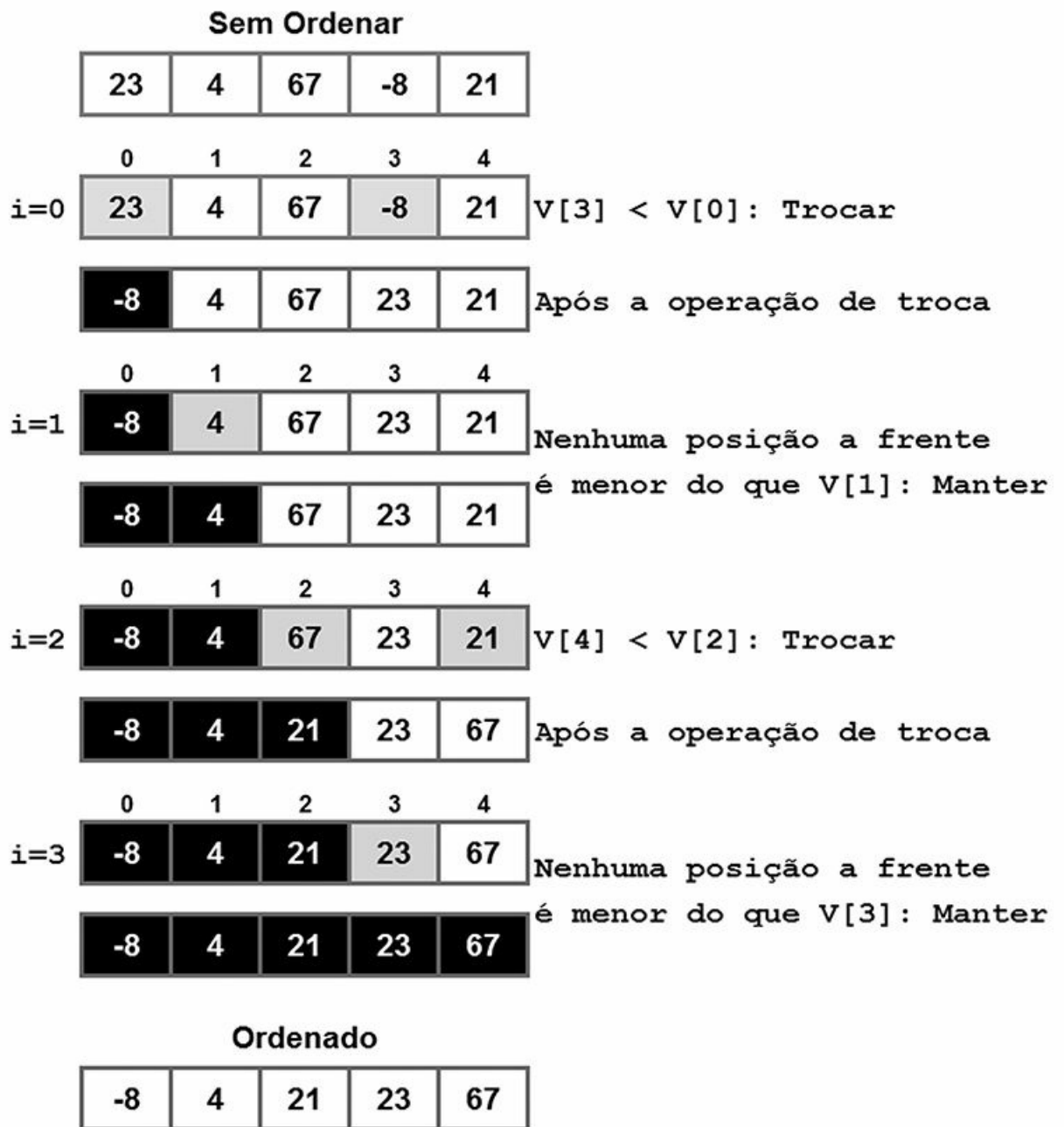


FIGURA 3.6

O selection sort, assim como o bubble sort, não é um algoritmo eficiente. Sua eficiência diminui drasticamente à medida que o número de elementos no array aumenta, não sendo recomendado para aplicações que envolvam grandes quantidades de dados ou que precisem de velocidade. Considerando um array com N elementos, o tempo de execução do selection sort é sempre de ordem $O(N^2)$. Como se pode notar, a eficiência do selection sort não depende da ordem inicial dos elementos.

Apesar de possuírem a mesma complexidade no caso médio, na prática, o selection sort quase sempre supera o desempenho do bubble sort, pois envolve um número menor de comparações.

3.2.3 Ordenação por inserção: insertion sort

O algoritmo insertion sort, também conhecido como ordenação por “inserção”, é outro algoritmo de ordenação bastante simples. Ele tem esse nome, pois se assemelha ao processo de ordenação de um conjunto de cartas de baralhos com as mãos: pega uma carta de cada vez e a “insere” em seu devido lugar, sempre deixando as cartas da mão em ordem. Na prática, este algoritmo possui um desempenho superior quando comparado com outros algoritmos como o bubble sort e o selection sort.



O algoritmo insertion sort percorre um array e, para cada posição **X**, verifica se o seu valor está na posição correta. Isso é feito andando para o começo do array, a partir da posição **X**, e movimentando uma posição para frente os valores que são maiores do que o valor da posição **X**. Desse modo, teremos uma posição livre para inserir o valor da posição **X** em seu devido lugar.

A Figura 3.7 mostra a implementação do algoritmo insertion sort. O princípio de funcionamento deste algoritmo é a inserção de um elemento do array na sua posição correta. Para entender esse processo, imagine um conjunto de valores não ordenados, como mostrado na Figura 3.8. Perceba que o valor da quarta posição do array (-8) não está na sua posição correta em relação aos seus antecessores. Então, o algoritmo movimenta os seus antecessores uma posição para frente e insere esse valor no início do array. O processo de comparar o valor de uma posição do array com seus antecessores é repetido para cada posição do array (linhas 3-8). Para cada posição, movimentam-se os valores maiores que o valor **atual** (linha 4) uma posição para frente no array (linhas 5-6). Ao final do processo, copia-se o valor atual para a sua posição correta, que é a posição do último valor movimentado (linha 7).

Método insertion sort

```
01 void insertionSort(int *V, int N){
02     int i, j, atual;
03     for(i = 1; i < N; i++){
04         atual = V[i];
05         for(j = i; (j > 0) && (atual < V[j - 1]); j--){
06             V[j] = V[j - 1];
07             V[j] = atual;
08         }
09     }
```

FIGURA 3.7

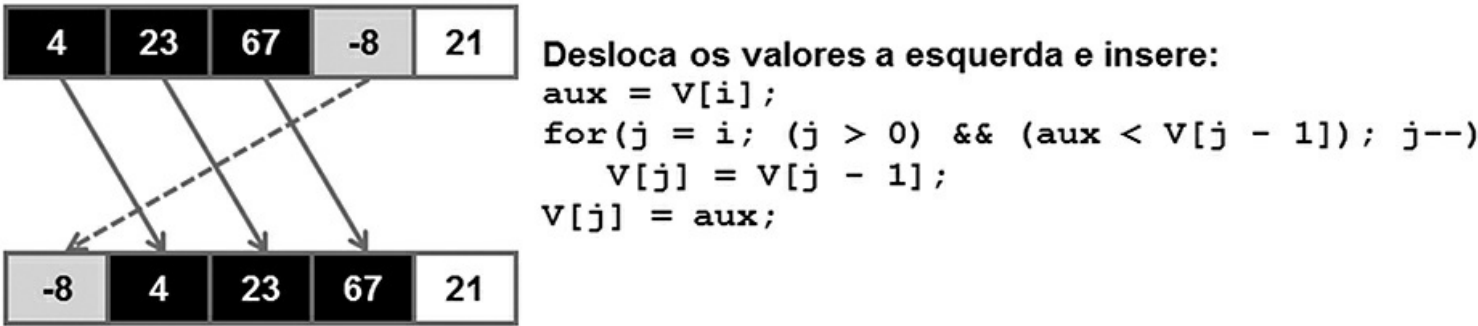
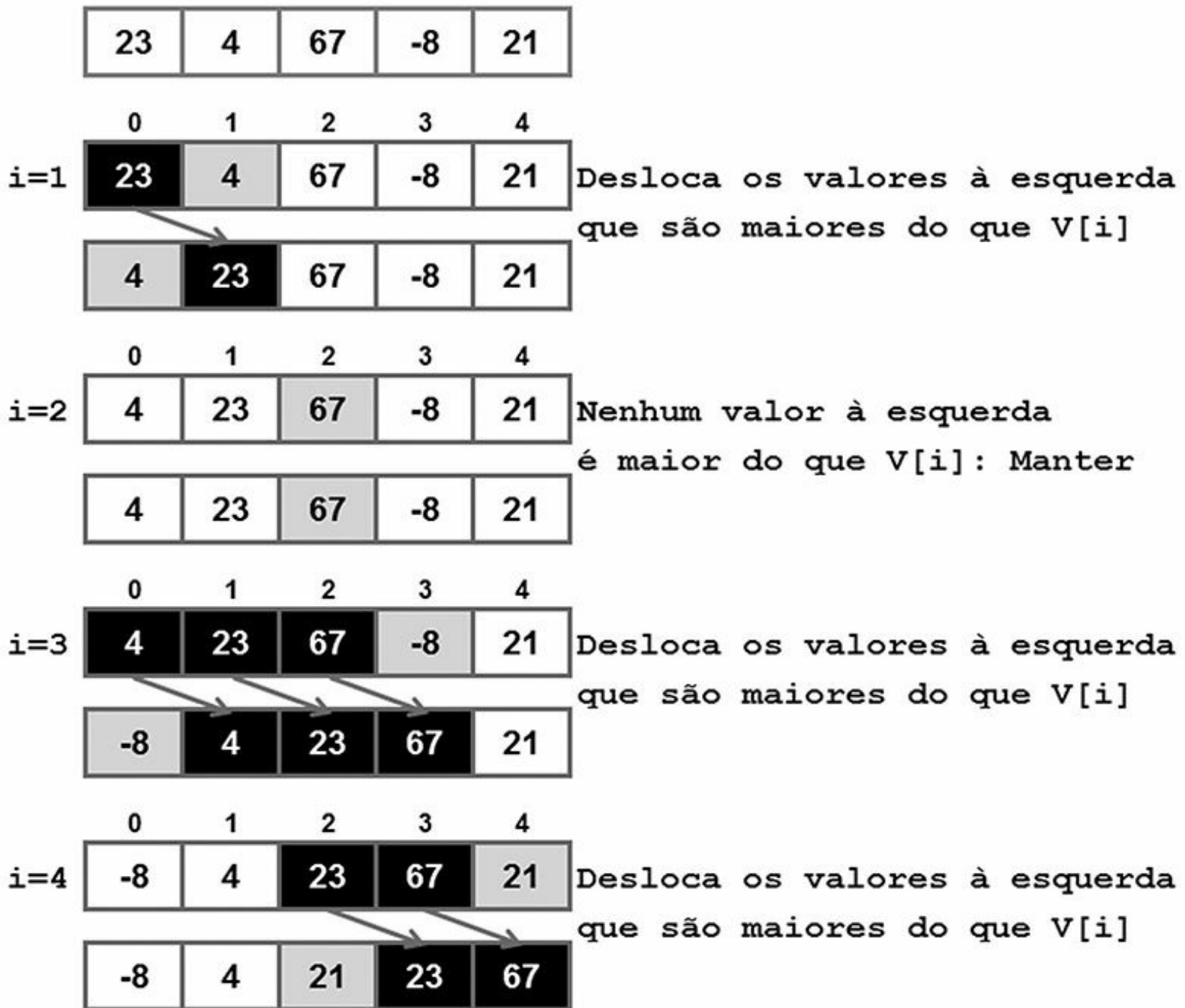


FIGURA 3.8

A Figura 3.9 mostra um exemplo de ordenação completa de um array em ordem crescente. Para cada valor da variável **i**, temos o array antes de ser ordenado e depois de ser ordenada aquela posição. Os itens coloridos são os valores verificados se estão na posição correta, enquanto os hachurados representam a porção do array movimentada durante a inserção do valor na sua posição correta.

Sem Ordenar



Ordenado

-8	4	21	23	67
----	---	----	----	----

FIGURA 3.9

Considerando um array com N elementos, o tempo de execução do insertion sort é:

- $O(N)$, melhor caso: os elementos já estão ordenados.
- $O(N^2)$, pior caso: os elementos estão ordenados na ordem inversa.
- $O(N^2)$, caso médio.



Na prática, o insertion sort é mais eficiente que a maioria dos algoritmos de ordem quadrática, como o selection sort e o bubble sort. De fato, trata-se de um dos mais rápidos algoritmos de ordenação para conjuntos pequenos de dados, superando inclusive o quick sort.

Além de ser um algoritmo de fácil implementação, o insertion sort tem a vantagem de ser:

- **Estável:** a ordem dos elementos iguais não muda durante a ordenação.
- **On-line:** pode ordenar elementos na medida em que os recebe, ou seja, não precisa ter todo o conjunto de dados para colocá-los em ordem.