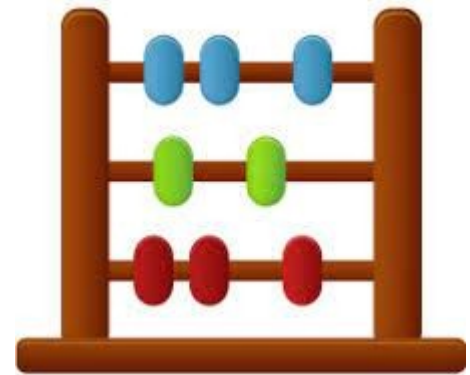


# Pesquisa e Ordenação de Dados

Unidade 2.7:

## Counting Sort



# Ordenação por Comparação de Chaves

- Todos os algoritmos vistos até o momento utilizam comparações entre chaves
- Estes algoritmos possuem um limite inferior de complexidade de tempo  $\Omega(n \log n)$ 
  - algoritmos com pior caso igual a  $O(n \log n)$ , como Merge Sort e Heap Sort, são ditos ótimos, pois assintoticamente não se pode usar menos comparações
- No entanto, podemos obter o tempo de  $O(n)$  sob certas circunstâncias, as quais tornam possível ordenar sem o uso de comparações

# Counting Sort

- Proposto por Harold H. Seward em 1954
- Ordenação por contagem
  - ao invés de comparar as chaves entre si, conta o número de ocorrências de cada chave
  - depois, distribui os elementos no vetor ordenado considerando a frequência das chaves
- As chaves devem ser números naturais, pois servirão como índices do vetor de contagem
  - ideal que estejam em um intervalo pequeno

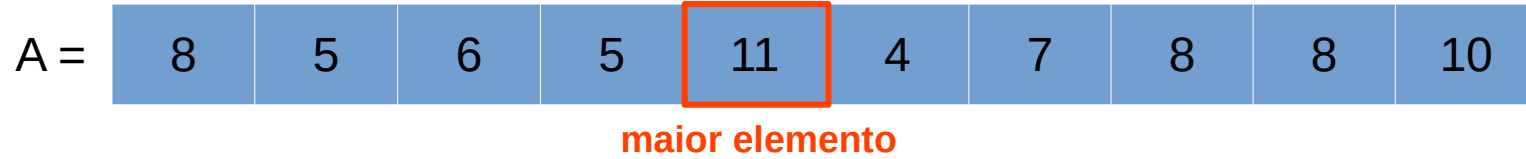
# Counting Sort

## Funcionamento

- Supondo que temos um vetor **A** de **n** elementos a serem ordenados e que o maior elemento do vetor é **k**:
  - 1) Criar um array de contagem **count** com **k+1** posições (ou seja, com intervalo de 0 até k), todas inicializadas com 0;

# Counting Sort

## Exemplo (1)



# Counting Sort

## Exemplo (2)

A =

8	5	6	5	11	4	7	8	8	10
---	---	---	---	----	---	---	---	---	----

maior elemento

count =

0	0	0	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11

array de [0..11]

# Counting Sort

## Funcionamento

- Supondo que temos um vetor **A** de **n** elementos a serem ordenados e que o maior elemento do vetor é **k**:
  - 1) Criar um array de contagem **count** com **k+1** posições (ou seja, 0..k), todas inicializadas com 0;
  - 2) Iterar sobre **A** e, a cada ocorrência de uma chave, incrementar em **count** o valor na posição cujo índice seja o próprio valor da chave **A[i]**;

# Counting Sort

## Exemplo (3)

A =

8	5	6	5	11	4	7	8	8	10
---	---	---	---	----	---	---	---	---	----

contando ocorrências

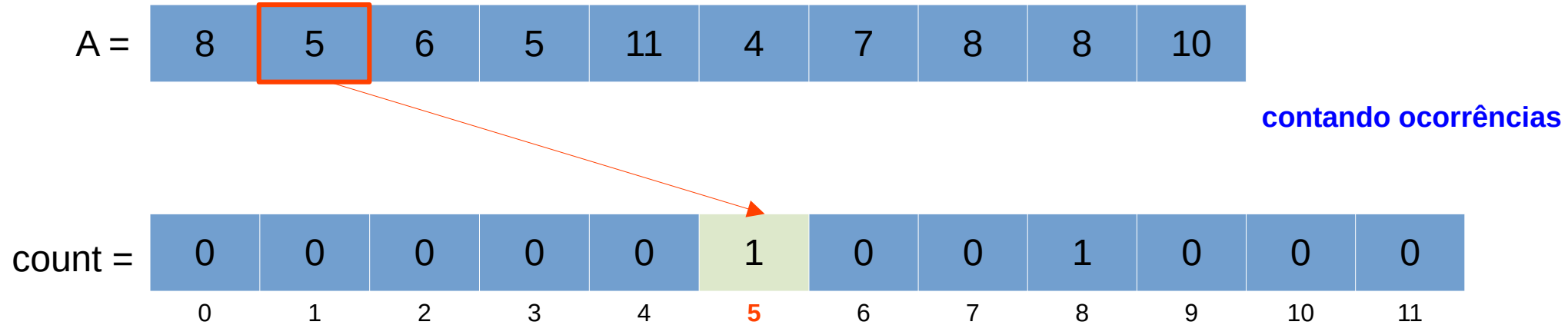
count =

0	0	0	0	0	0	0	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11



# Counting Sort

## Exemplo (4)



# Counting Sort

## Exemplo (5)

A =

8	5	6	5	11	4	7	8	8	10
---	---	---	---	----	---	---	---	---	----

contando ocorrências

count =

0	0	0	0	0	1	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11

# Counting Sort

## Exemplo (6)



contando ocorrências



# Counting Sort

## Exemplo (7)

A =

8	5	6	5	11	4	7	8	8	10
---	---	---	---	----	---	---	---	---	----

contando ocorrências

count =

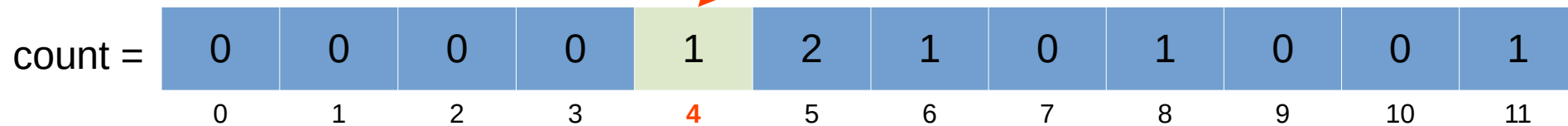
0	0	0	0	0	2	1	0	1	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11

# Counting Sort

## Exemplo (8)



contando ocorrências

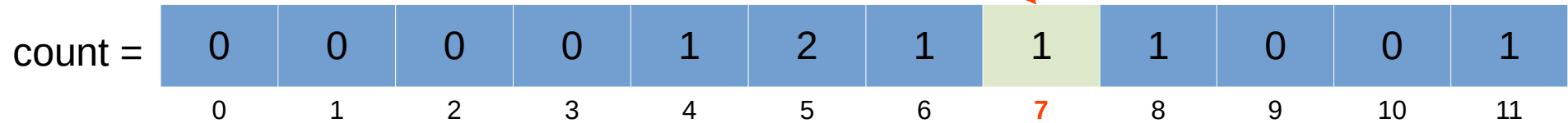


# Counting Sort

## Exemplo (9)



contando ocorrências

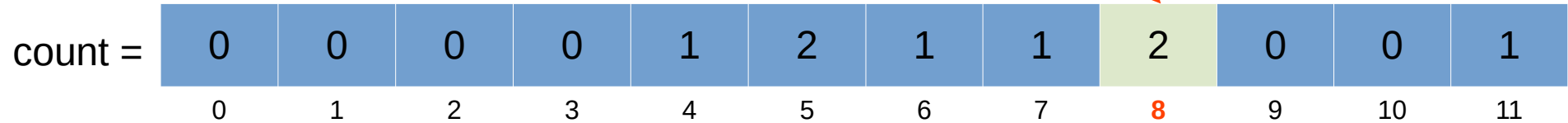


# Counting Sort

## Exemplo (10)

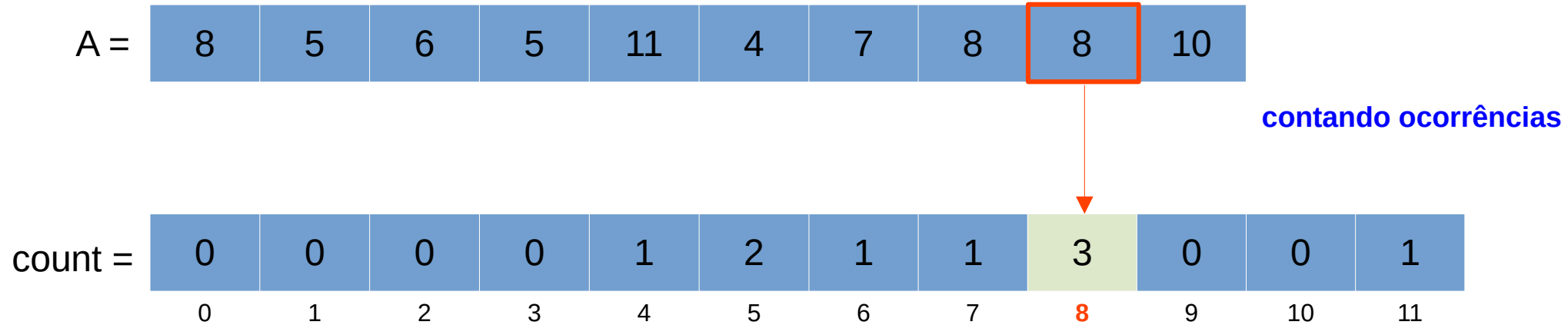


contando ocorrências



# Counting Sort

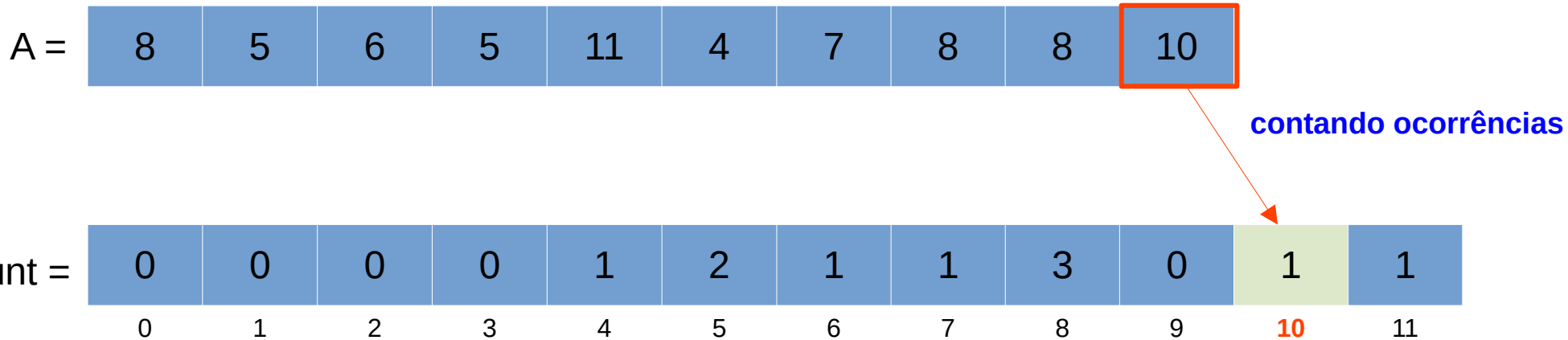
## Exemplo (11)





# Counting Sort

## Exemplo (12)



# Counting Sort

## Exemplo (12)

A =

8	5	6	5	11	4	7	8	8	10
---	---	---	---	----	---	---	---	---	----

contando ocorrências

count =

0	0	0	0	1	2	1	1	3	0	1	1
0	1	2	3	4	5	6	7	8	9	10	11

- Resumo:
  - não ocorrem: 0, 1, 2, 3 e 9
  - ocorrem 1 vez: 4, 6, 7, 10 e 11
  - ocorre 2 vezes: 5
  - ocorre 3 vezes: 8

# Counting Sort

## Exemplo (12)

A =

8	5	6	5	11	4	7	8	8	10
---	---	---	---	----	---	---	---	---	----

count =

0	0	0	0	1	2	1	1	3	0	1	1
0	1	2	3	4	5	6	7	8	9	10	11

Se o conjunto de entrada é uma lista simples de números, neste ponto, basta iterar sobre count: todo o índice cujo valor for diferente de 0 é copiado para A o número de vezes armazenado naquele índice.

4	5	5	6	7	8	8	8	10	11
0	1	2	3	4	5	6	7	8	9

Porém, se o conjunto de entrada é uma lista que contém mais campos além da chave numérica? Neste caso, seguimos os próximos passos...

# Counting Sort

## Funcionamento

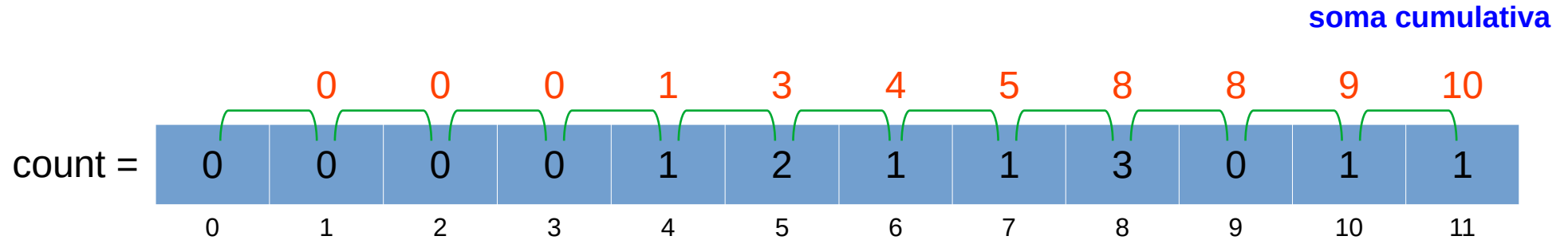
- Supondo que temos um vetor **A** de **n** elementos a serem ordenados e que o maior elemento do vetor é **k**:
  - 1) Criar um array de contagem **count** com **k+1** posições (ou seja, 0..k), todas inicializadas com 0;
  - 2) Iterar sobre **A** e, a cada ocorrência de uma chave, incrementar em **count** o valor na posição cujo índice seja o próprio valor da chave **A[i]**;
  - 3) Realizar a soma cumulativa em **count**: cada posição conterá a soma de todas as posições anteriores;

# Counting Sort

## Exemplo (13)

A =

8	5	6	5	11	4	7	8	8	10
---	---	---	---	----	---	---	---	---	----



# Counting Sort

## Exemplo (14)

A =

8	5	6	5	11	4	7	8	8	10
---	---	---	---	----	---	---	---	---	----

count atualizado com  
a soma cumulativa

count =

0	0	0	0	1	3	4	5	8	8	9	10
0	1	2	3	4	5	6	7	8	9	10	11

A soma cumulativa armazenada em uma posição representa quantos valores são menores do que o valor representado pelo índice desta posição.

Com isso, sabemos em que posição cada chave deve estar na ordenação final.

Ex: a chave 6 estará na 4ª posição, pois há 3 chaves menores que ela; a chave 4 estará na 1ª posição.

# Counting Sort

## Funcionamento

- Supondo que temos um vetor **A** de **n** elementos a serem ordenados e que o maior elemento do vetor é **k**:
  - 1) Criar um array de contagem **count** com **k+1** posições (ou seja, 0..k), todas inicializadas com 0;
  - 2) Iterar sobre **A** e, a cada ocorrência de uma chave, incrementar em **count** o valor na posição cujo índice seja o próprio valor da chave **A[i]**;
  - 3) Realizar a soma cumulativa em **count**: cada posição conterá a soma de todas as posições anteriores;
  - 4) Criar um array auxiliar **aux** com o mesmo tamanho de **A**. Para cada chave em **A**, começando pelo final, decrementamos a contagem correspondente em **count** e inserimos a chave em **aux** na posição da contagem;

# Counting Sort

## Exemplo (15)

A =

8	5	6	5	11	4	7	8	8	10
---	---	---	---	----	---	---	---	---	----

count =

0	0	0	0	1	3	4	5	8	8	9	10
0	1	2	3	4	5	6	7	8	9	10	11

aux =

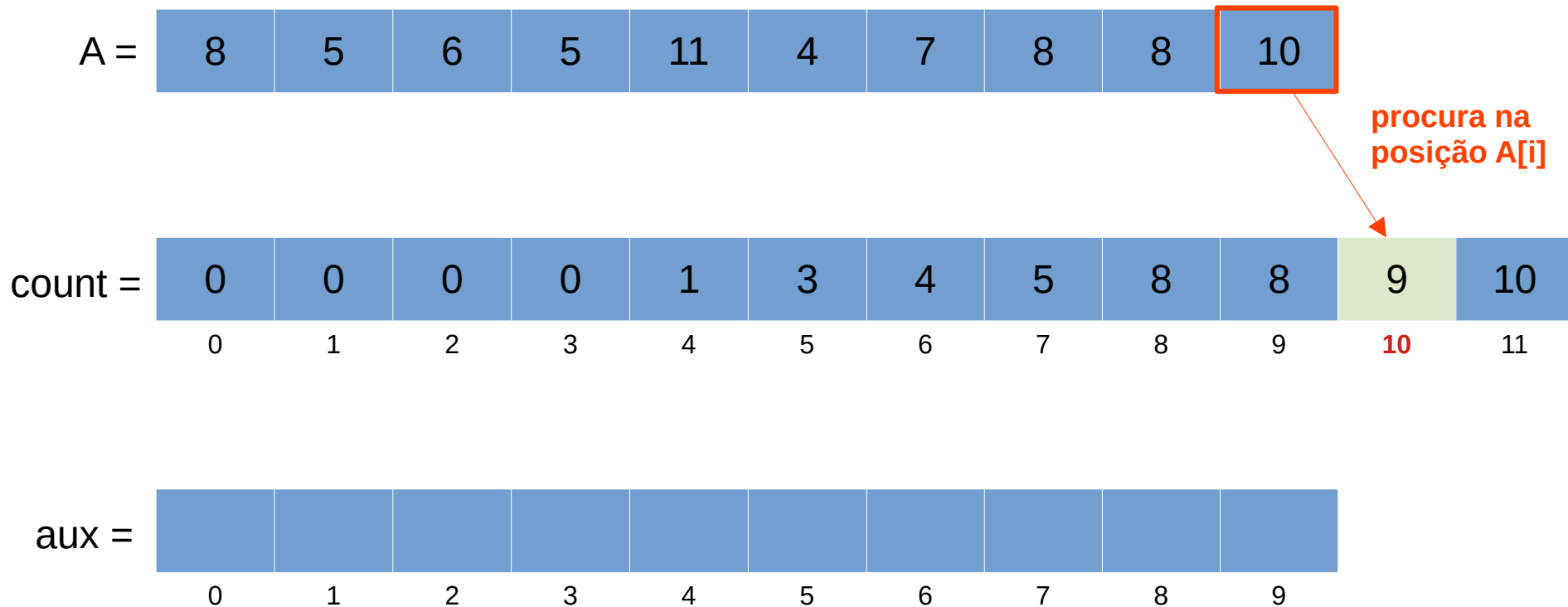
0	1	2	3	4	5	6	7	8	9

vetor auxiliar com  
mesmo tamanho de A



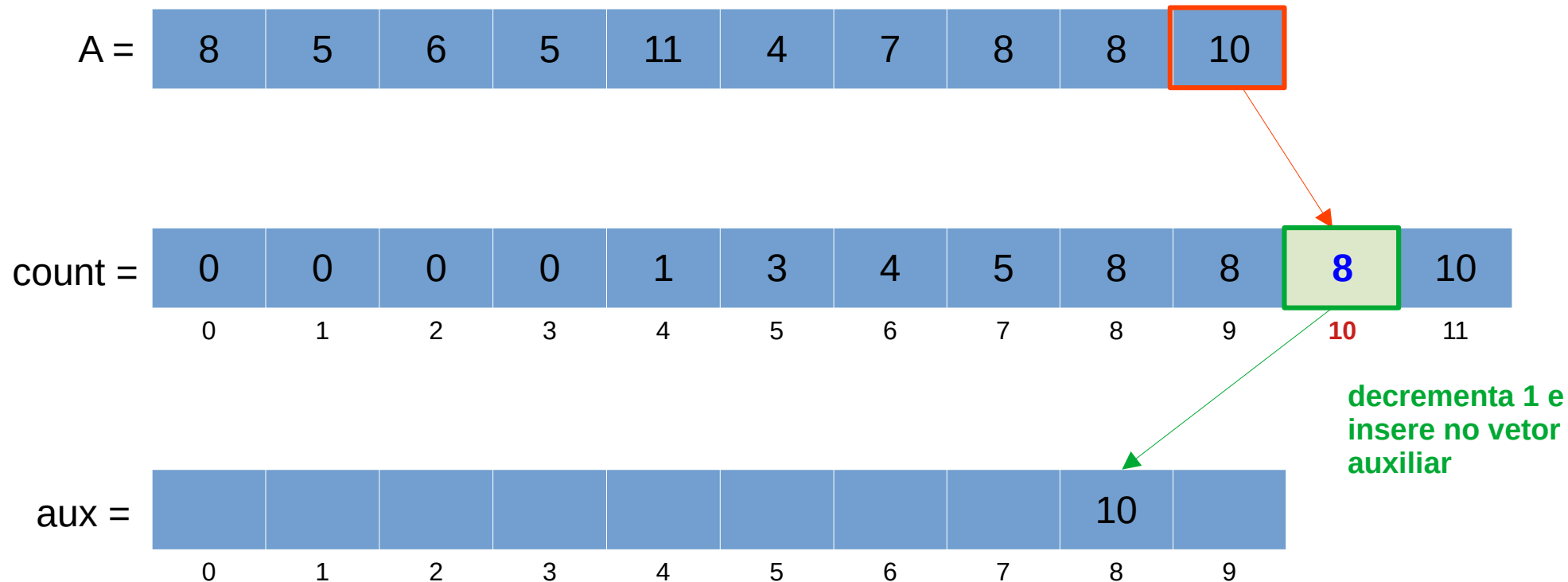
# Counting Sort

## Exemplo (16)



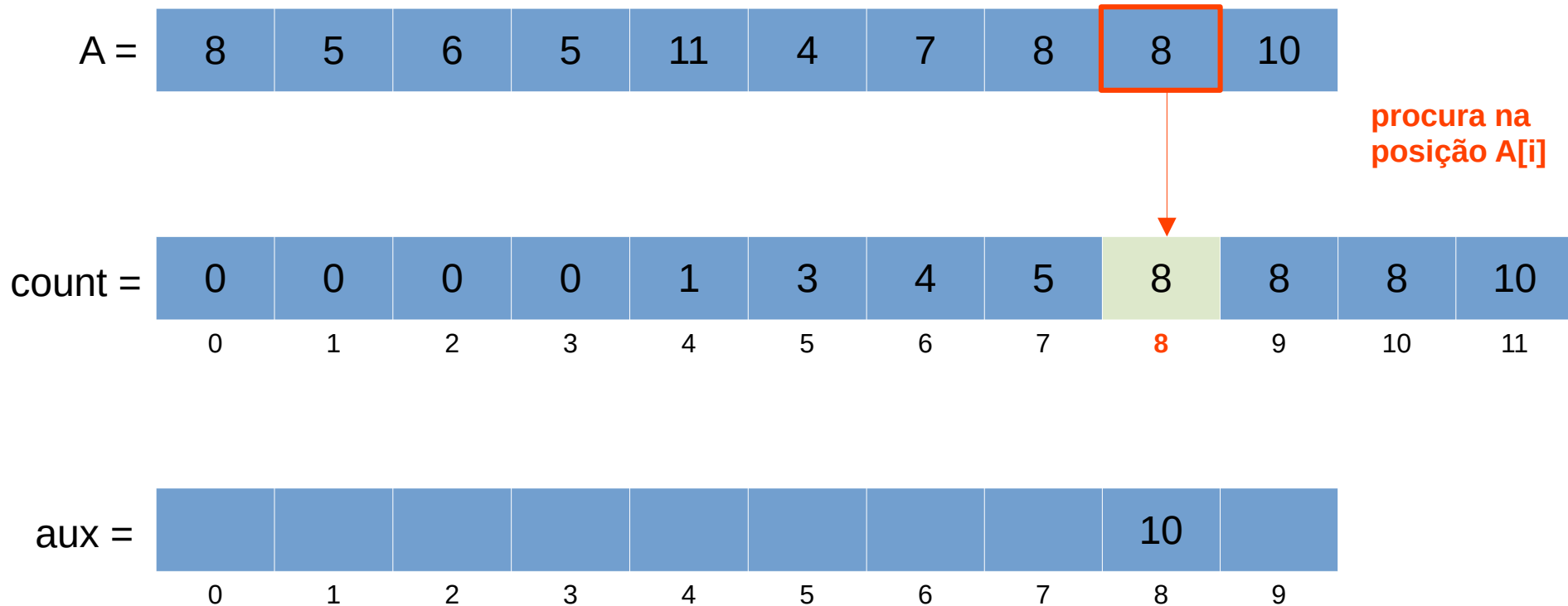
# Counting Sort

## Exemplo (17)



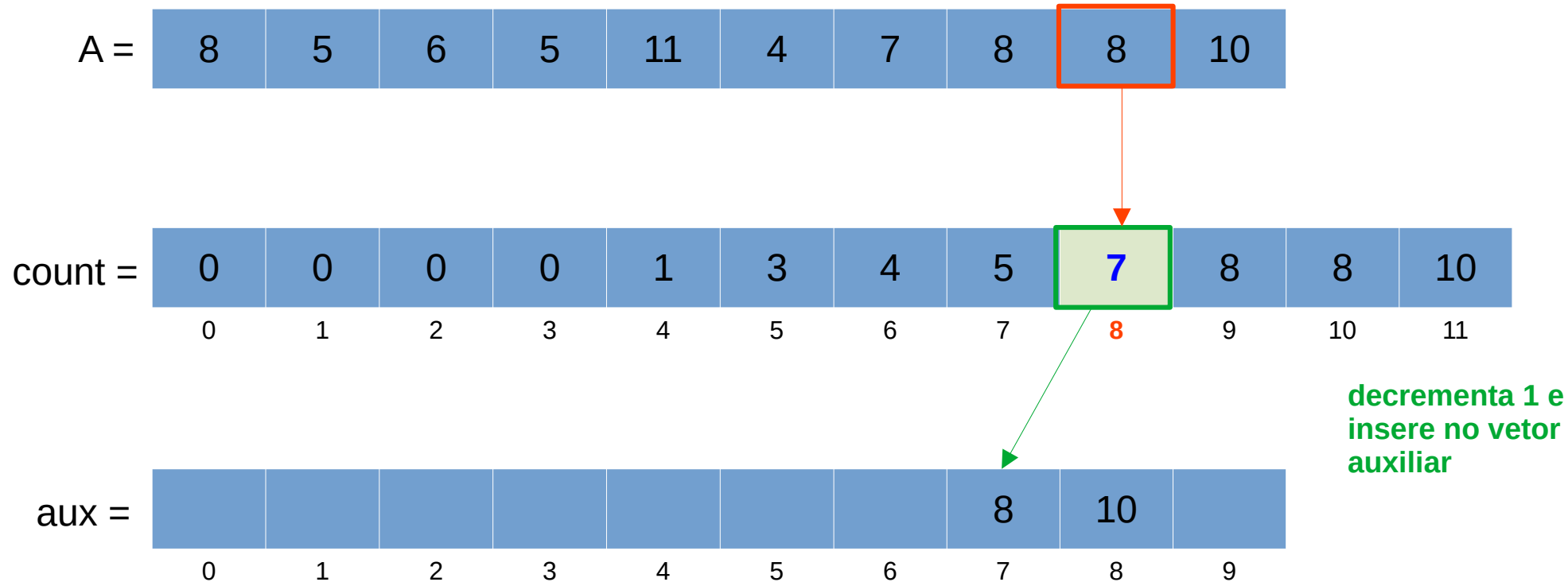
# Counting Sort

## Exemplo (18)



# Counting Sort

## Exemplo (19)

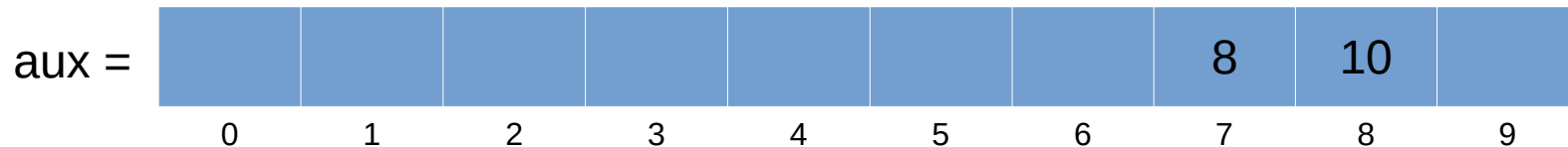
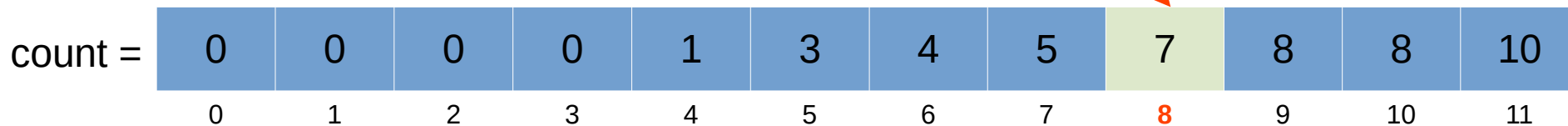


# Counting Sort

## Exemplo (20)

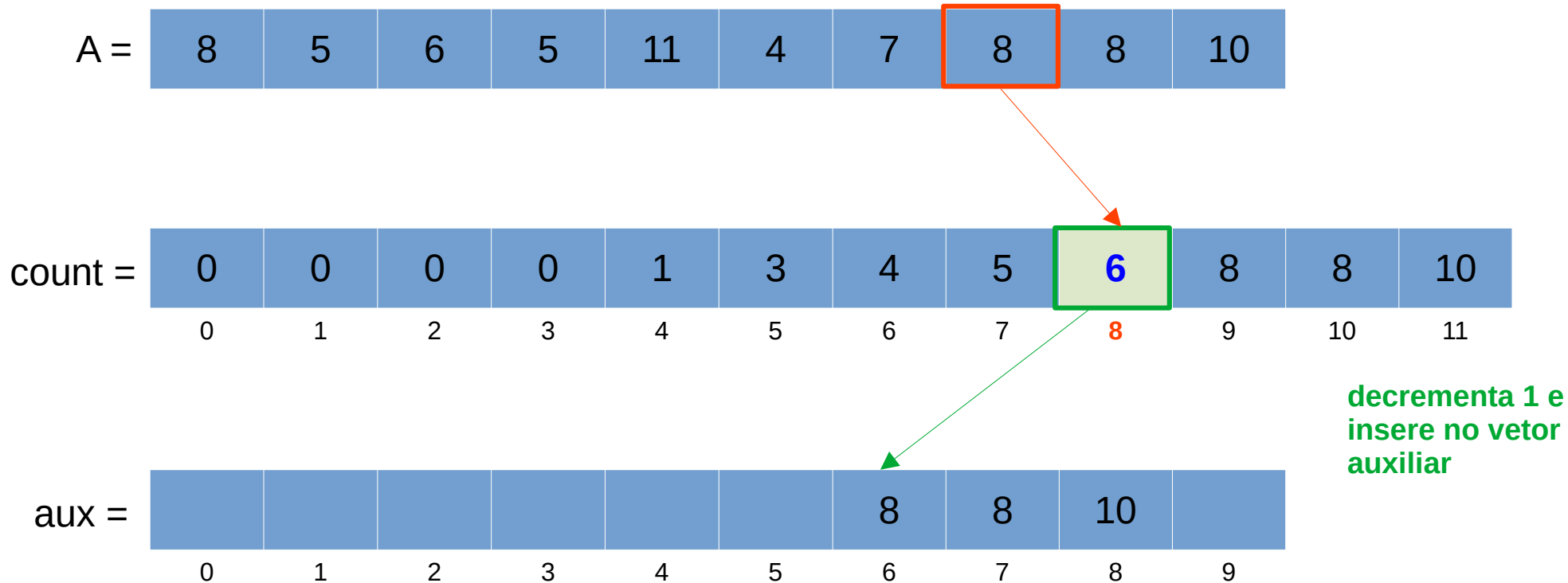


procura na  
posição A[i]



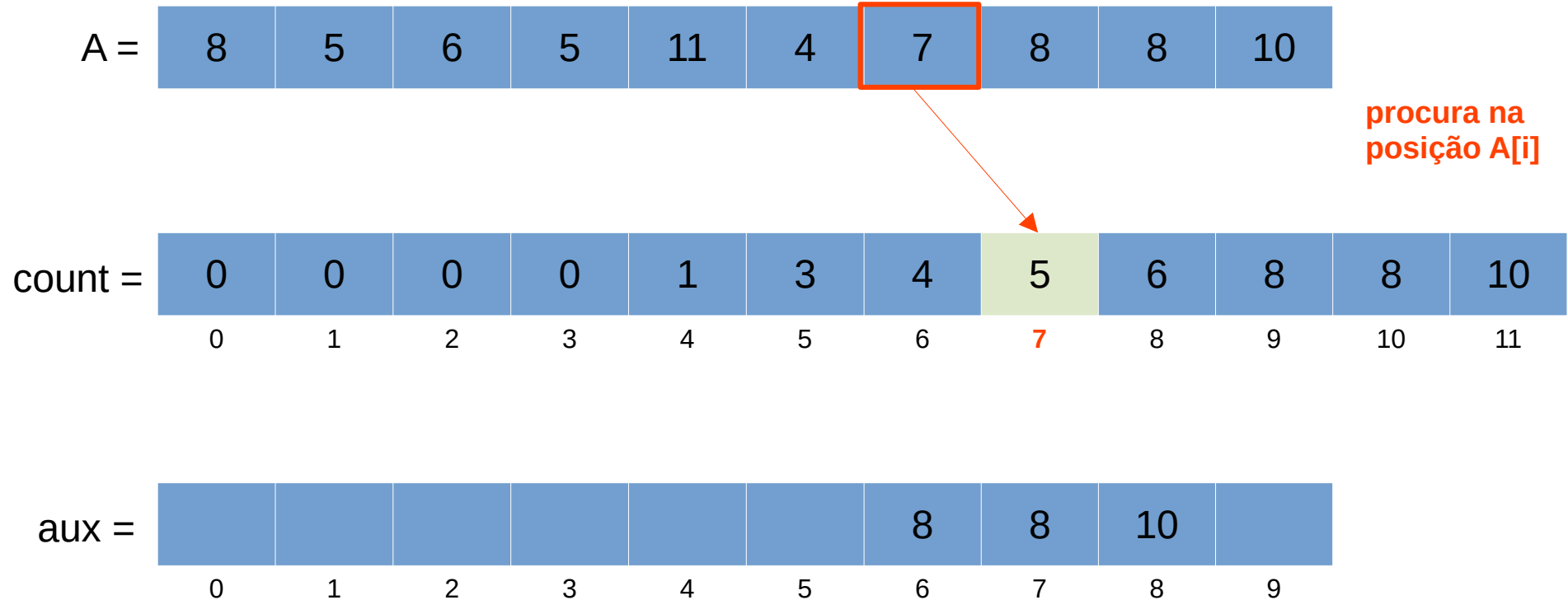
# Counting Sort

## Exemplo (21)



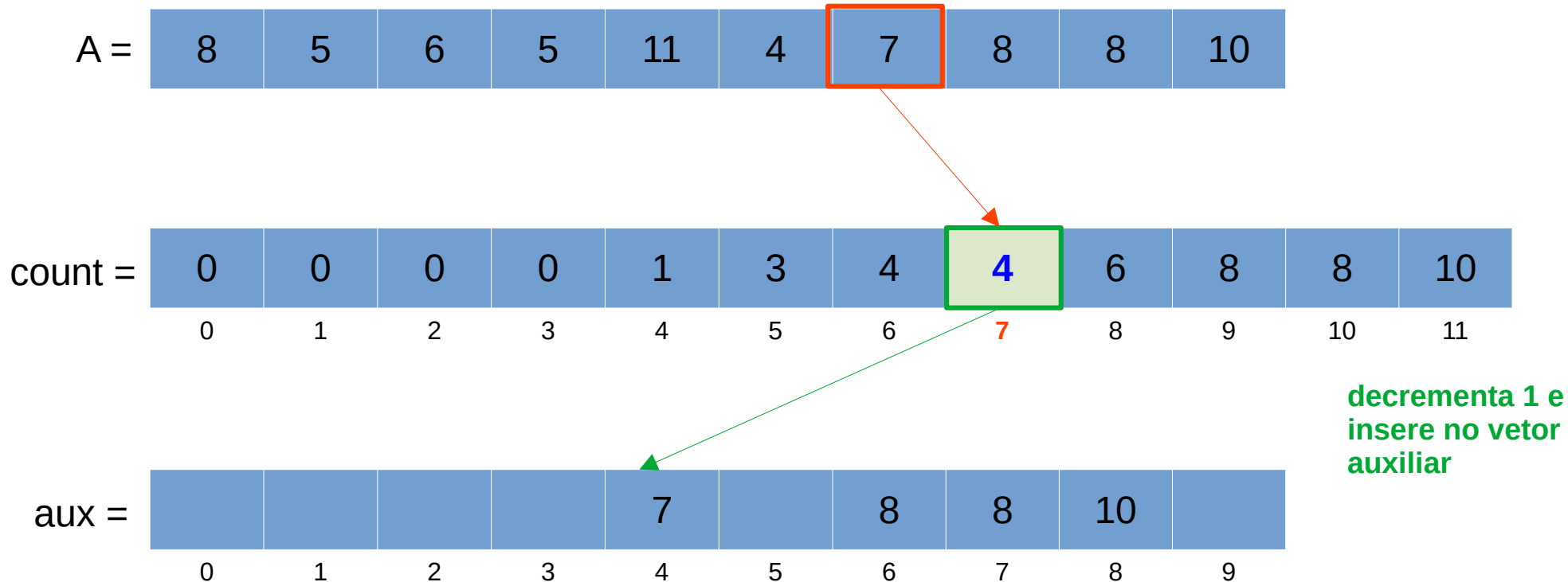
# Counting Sort

## Exemplo (22)



# Counting Sort

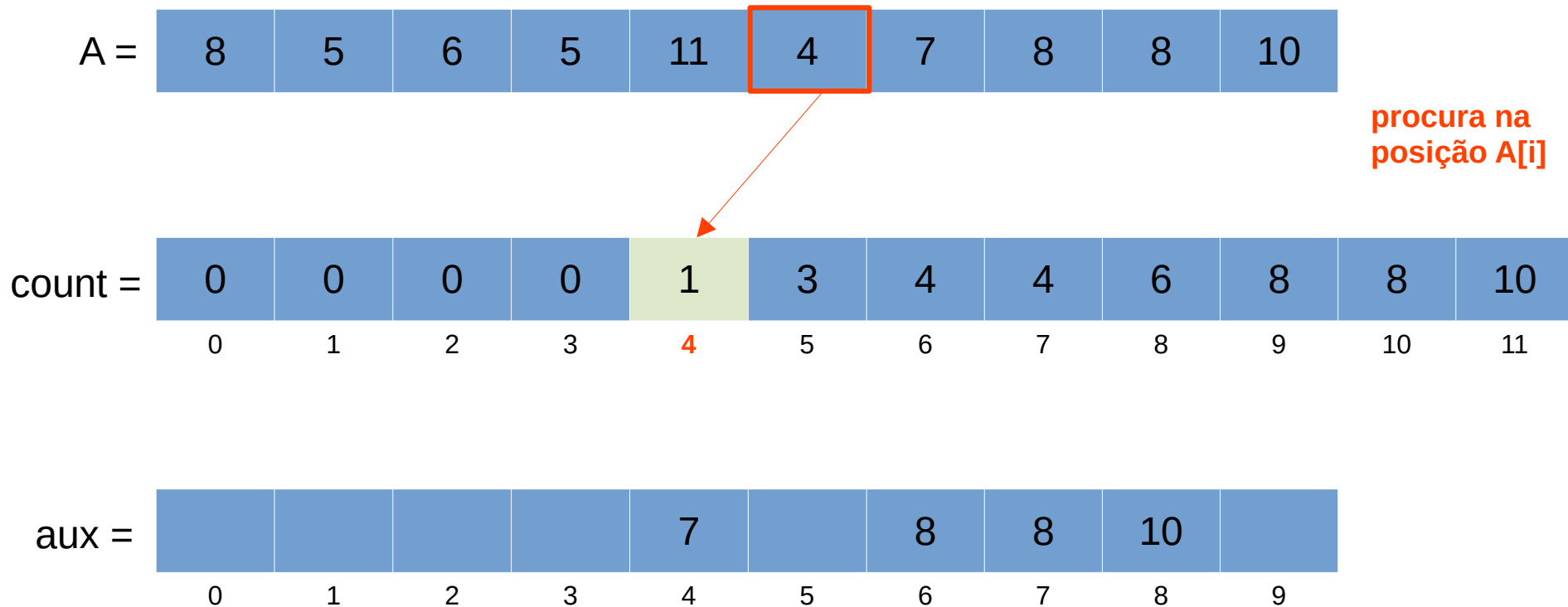
## Exemplo (23)





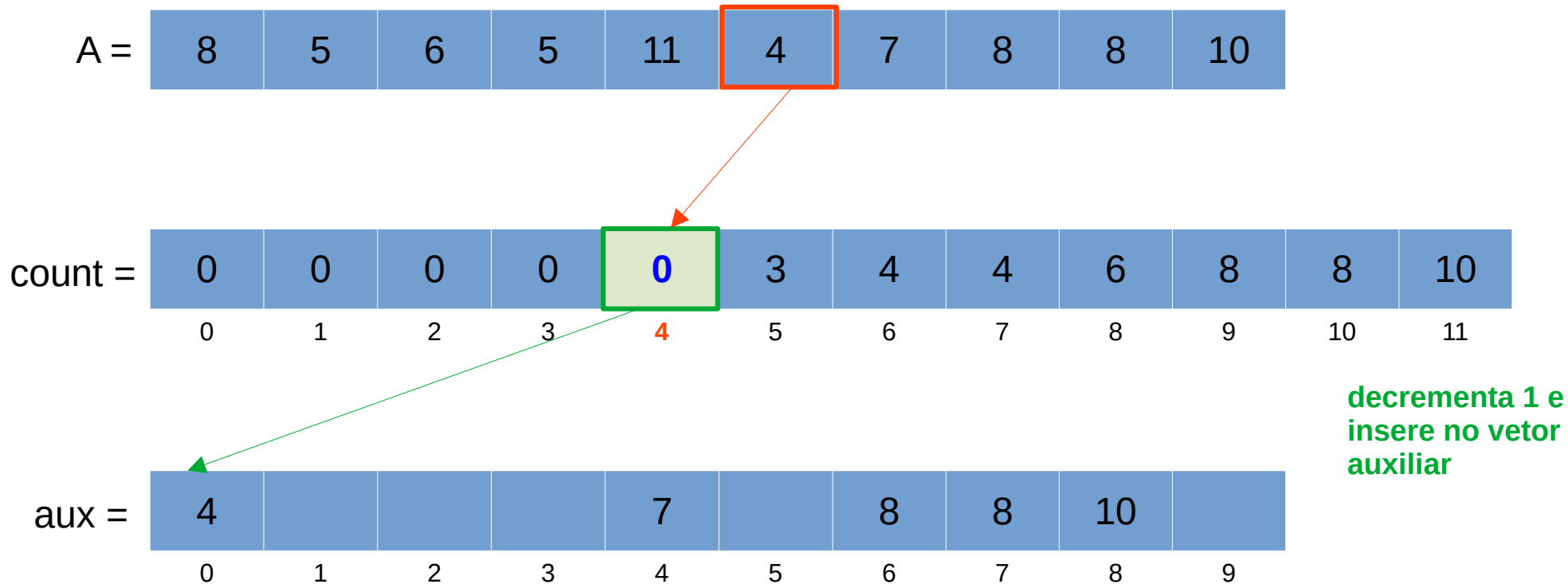
# Counting Sort

## Exemplo (24)



# Counting Sort

## Exemplo (25)



# Counting Sort

## Exemplo (26)

A =

8	5	6	5	11	4	7	8	8	10
---	---	---	---	----	---	---	---	---	----

procura na  
posição A[i]

count =

0	0	0	0	0	3	4	4	6	8	8	10
0	1	2	3	4	5	6	7	8	9	10	11

aux =

4				7		8	8	10	
0	1	2	3	4	5	6	7	8	9

# Counting Sort

## Exemplo (27)

A =

8	5	6	5	11	4	7	8	8	10
---	---	---	---	----	---	---	---	---	----

count =

0	0	0	0	0	3	4	4	6	8	8	9
0	1	2	3	4	5	6	7	8	9	10	11

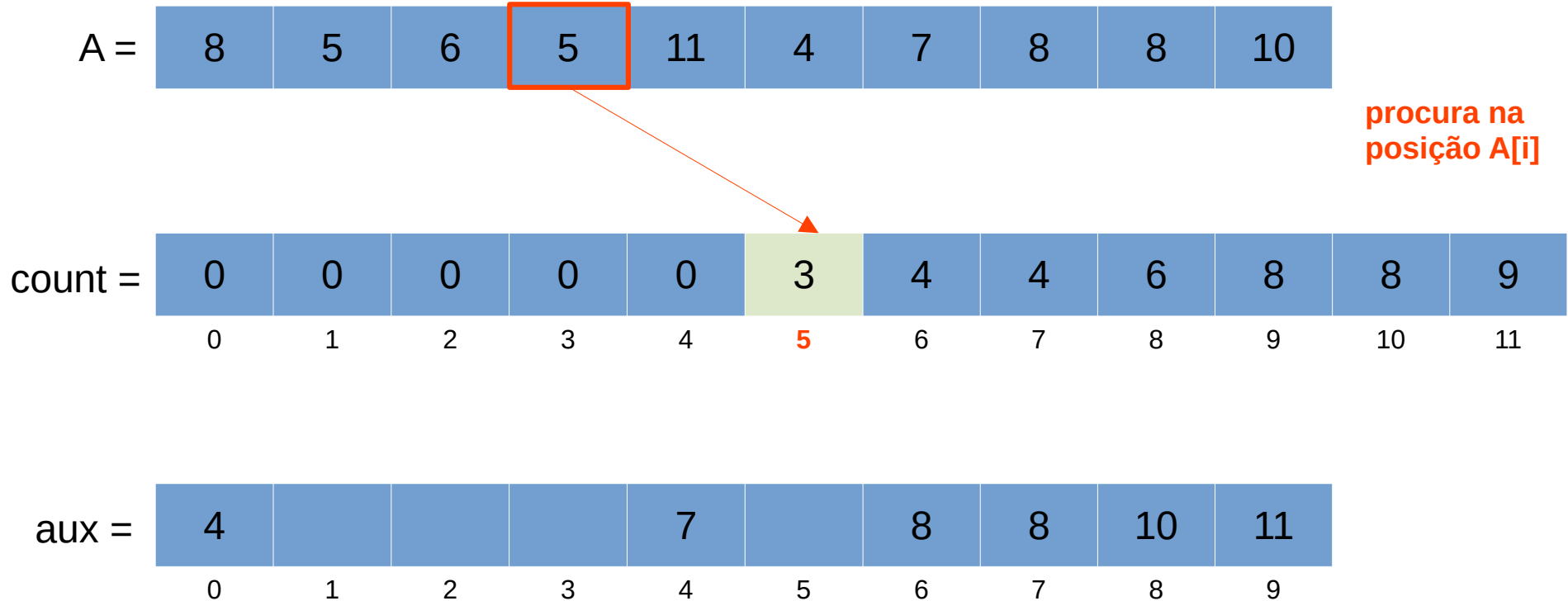
aux =

4				7		8	8	10	11
0	1	2	3	4	5	6	7	8	9

decrementa 1 e  
insere no vetor  
auxiliar

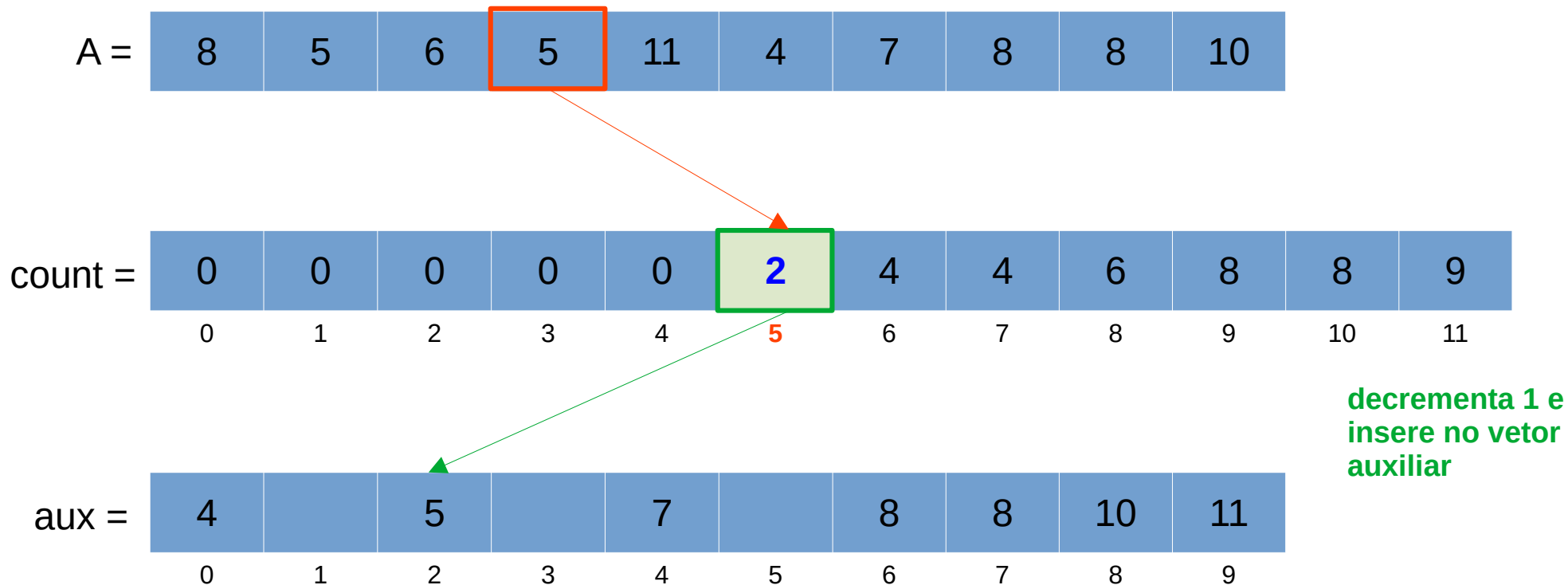
# Counting Sort

## Exemplo (28)



# Counting Sort

## Exemplo (29)



# Counting Sort

## Exemplo (30)

A =

8	5	6	5	11	4	7	8	8	10
---	---	---	---	----	---	---	---	---	----

procura na  
posição A[i]

count =

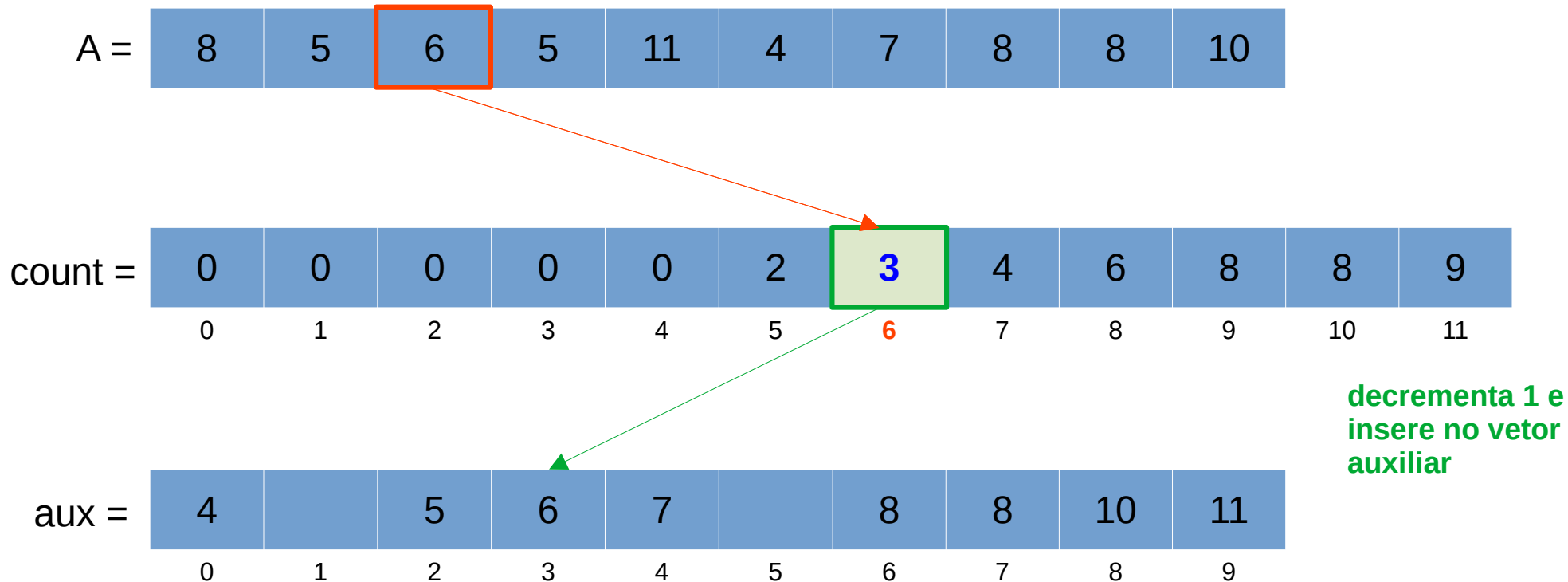
0	0	0	0	0	2	4	4	6	8	8	9
0	1	2	3	4	5	6	7	8	9	10	11

aux =

4		5		7		8	8	10	11
0	1	2	3	4	5	6	7	8	9

# Counting Sort

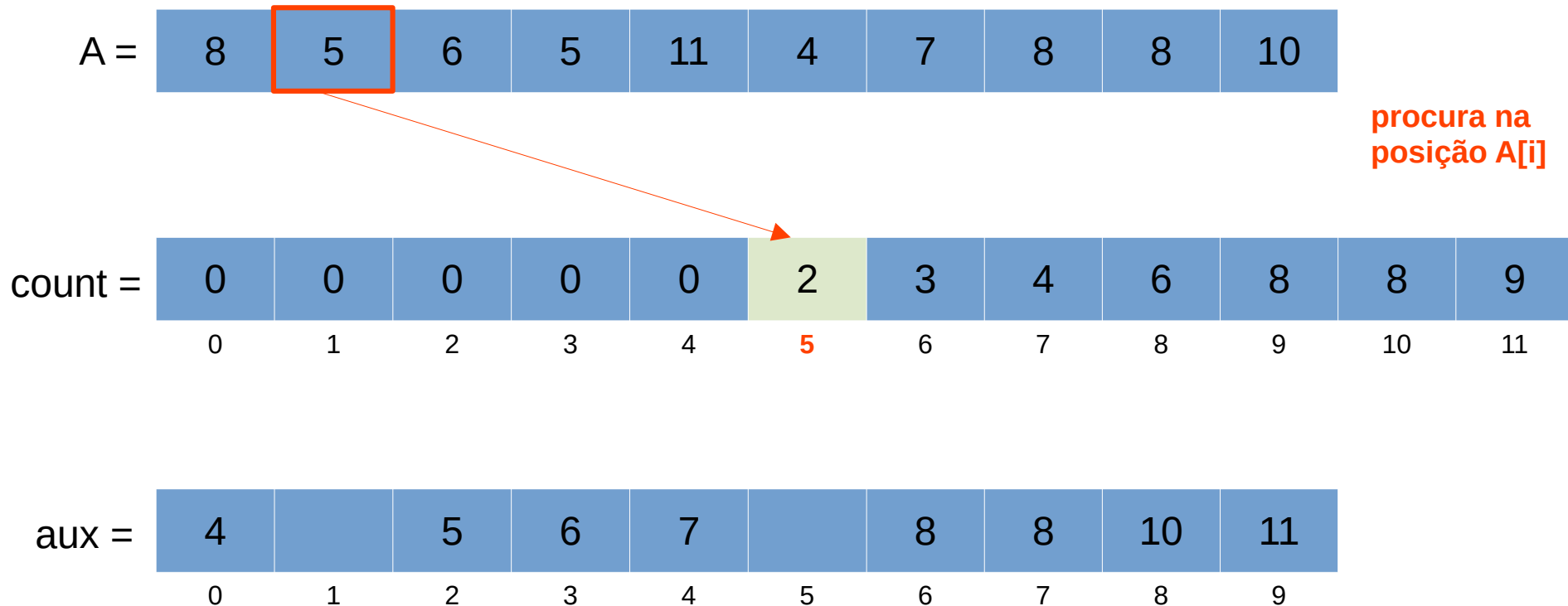
## Exemplo (31)





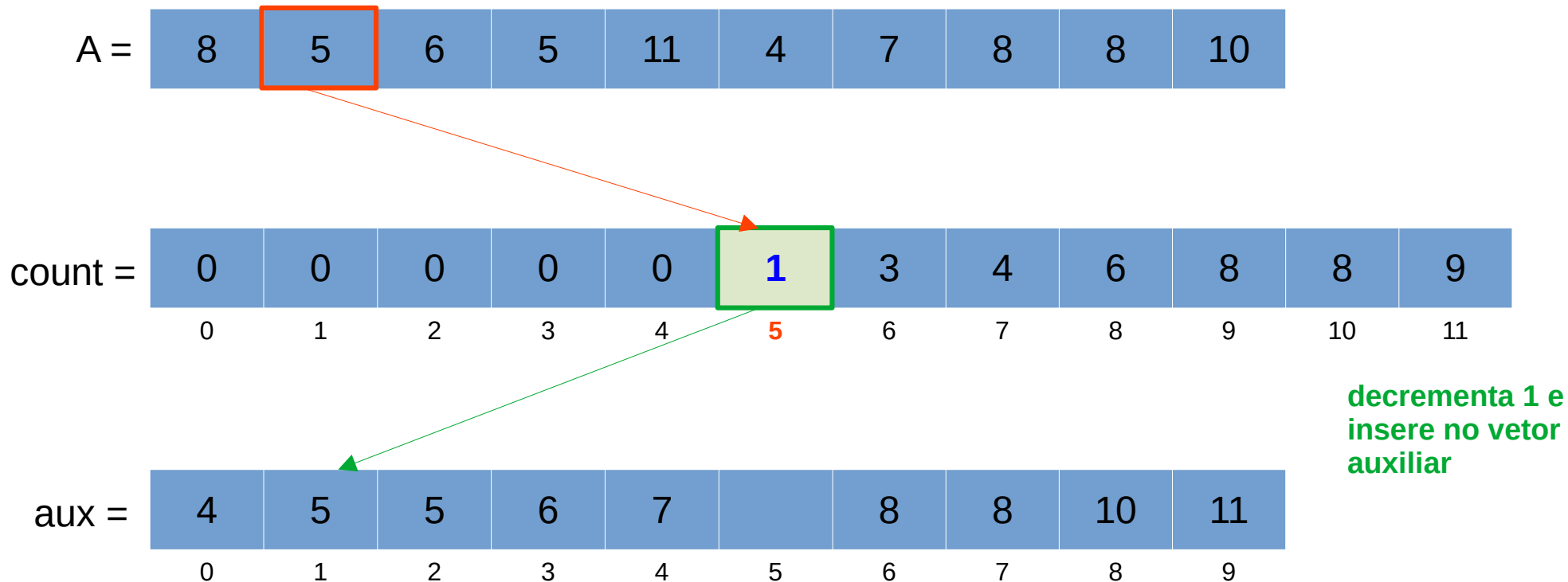
# Counting Sort

## Exemplo (32)



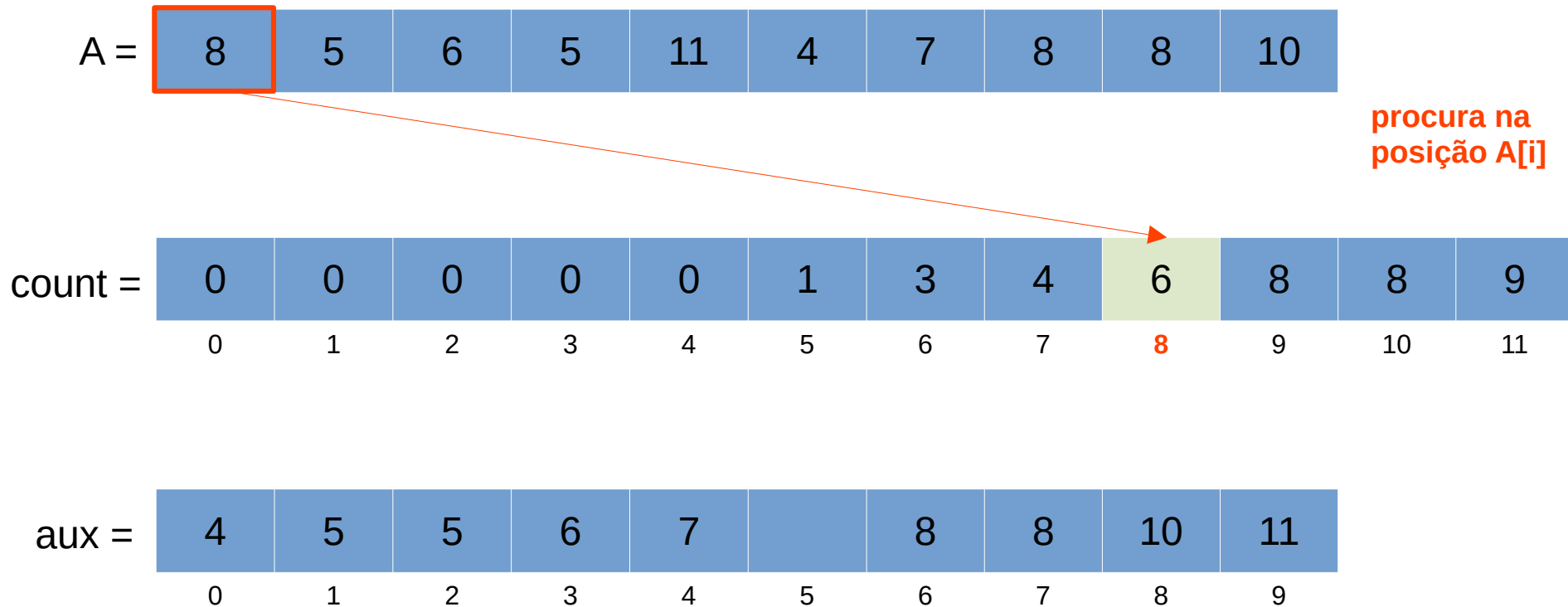
# Counting Sort

## Exemplo (33)



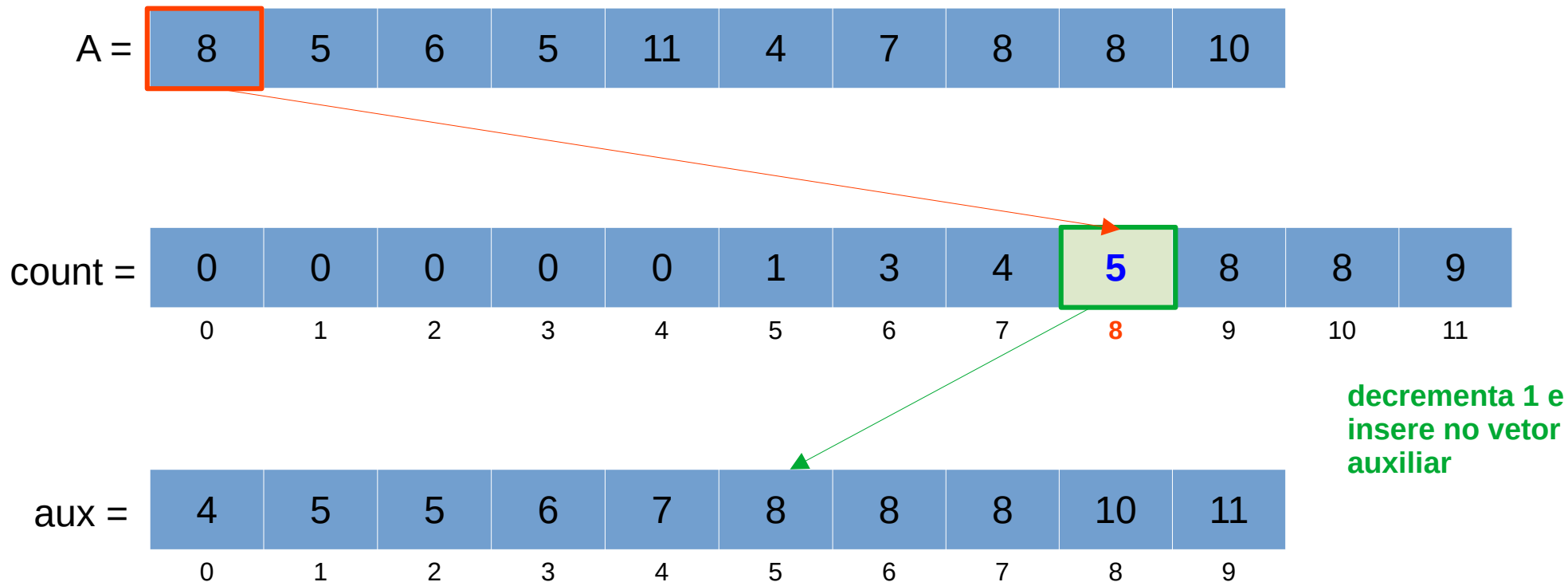
# Counting Sort

## Exemplo (34)



# Counting Sort

## Exemplo (35)



# Counting Sort

## Funcionamento

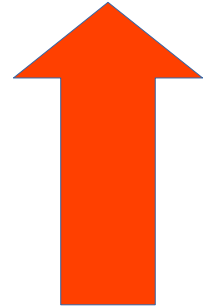
- Supondo que temos um vetor **A** de **n** elementos a serem ordenados e que o maior elemento do vetor é **k**:
  - 1) Criar um array de contagem **count** com **k+1** posições (ou seja, 0..k), todas inicializadas com 0;
  - 2) Iterar sobre **A** e, a cada ocorrência de uma chave, incrementar em **count** o valor na posição cujo índice seja o próprio valor da chave **A[i]**;
  - 3) Realizar a soma cumulativa em **count**: cada posição conterà a soma de todas as posições anteriores;
  - 4) Criar um array auxiliar **aux** com o mesmo tamanho de **A**. Para cada chave em **A**, começando pelo final, decrementamos a contagem correspondente em **count** e inserimos a chave em **aux** na posição da contagem;
  - 5) Copiar os dados de **aux** para **A**.

# Counting Sort

## Exemplo (37)

A =

8	5	6	5	11	4	7	8	8	10
---	---	---	---	----	---	---	---	---	----



copia dados de  
volta para o vetor  
original A

aux =

4	5	5	6	7	8	8	8	10	11
0	1	2	3	4	5	6	7	8	9

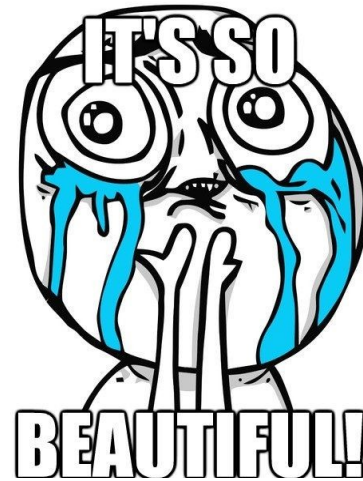
# Counting Sort

## Exemplo (38)

A =

4	5	5	6	7	8	8	8	10	11
0	1	2	3	4	5	6	7	8	9

FIM!



# Counting Sort

## Pseudocódigo

Algoritmo CountingSort

Início

$k \leftarrow$  maior elemento de A

declara vetor count com  $k+1$  posições

declara vetor aux com n posições

para i de 0,  $i \leq k$  faça

    count[i] = 0 /\* inicializa contagem com zeros \*/

fimPara

para i de 0,  $i < n$  /\* para cada elemento de A \*/

    count[A[i]]++ /\* incrementa contador \*/

fimPara

para i de 1,  $i \leq k$  faça

    count[i] += count[i-1] /\* soma acumulada \*/

fimPara

para i de  $n-1$ ,  $i \geq 0$  (passo -1) faça

    count[A[i]] = count[A[i]] - 1

    aux[count[A[i]]] = A[i] /\* insere em aux \*/

fimPara

para i de 0,  $i < n$  faça

    A[i] = aux[i] /\* copia para vetor original \*/

fimPara

Fim



# Counting Sort

## Análise

- $O(n + k)$  (tempo e espaço), para todos os casos
  - equivale a  $O(n)$  se  $k$  for pequeno (no máximo igual a  $n$ )
- Recomendável apenas quando  $k$  não é muito maior que  $n$  e quando os valores são densos, ou seja, sem grandes intervalos entre eles
- Estável, pois elementos com mesma chave são movidos na ordem em que aparecem
- Como utiliza memória auxiliar, não é *in place*