Ordenação

Prof. Paulo Henrique Pisani

abril/2022

Tópicos

- Algoritmos de ordenação sem comparação entre elementos:
  - Counting sort
  - Radix sort

Counting sort

Counting sort

Ordenação por contagem

- Assume que cada um dos elementos de entrada é um inteiro na faixa [0; k];
- Para cada elemento *x*, conta o número de itens menores que *x*;
- Dessa forma, é possível posicionar cada elemento na posição correta no vetor de saída.

Counting sort

Ordenação por contagem

- Assume que cada um dos elementos de entrada é um inteiro na faixa [0; k];
- Para cada elemento *x*, conta o número de itens menores que *x*;
- Dessa forma, é possível posicionar cada elemento na posição correta no vetor de saída.

Algoritmo em que **não ocorre comparação entre elementos no arranjo.**

Frequência de cada elemento

Cada elemento é um inteiro na faixa [0; 10]

6	10	6	2	1	4	0	6	3	1

Frequência de cada elemento

Cada elemento é um inteiro na faixa [0; 10]

6	10	6	2	1	4	0	6	3	1

Contadores:

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

Frequência de cada elemento

Contadores:

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

Frequência de cada elemento

Contadores:

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

Frequência de cada elemento

Contadores:

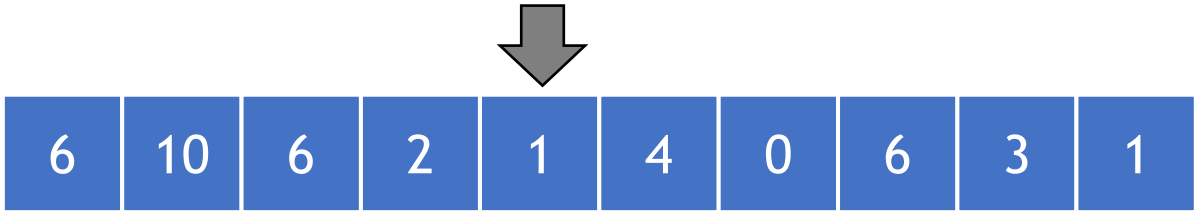
0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	2	0	0	0	1

Frequência de cada elemento

Contadores:

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

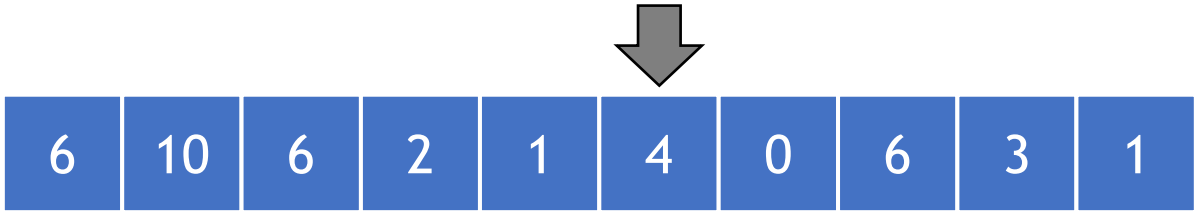
Frequência de cada elemento



Contadores:

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

Frequência de cada elemento



Contadores:

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

Frequência de cada elemento

Contadores:

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

Frequência de cada elemento

Contadores:

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

Frequência de cada elemento

Contadores:

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

Frequência de cada elemento

Contadores:

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

Frequência de cada elemento

6	10	6	2	1	4	0	6	3	1

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

Somar frequências

6	10	6	2	1	4	0	6	3	1

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

Somar frequências

6	10	6	2	1	4	0	6	3	1

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

Somar frequências

6	10	6	2	1	4	0	6	3	1

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

Somar frequências

6	10	6	2	1	4	0	6	3	1

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

Somar frequências

6	10	6	2	1	4	0	6	3	1

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

Somar frequências

6	10	6	2	1	4	0	6	3	1

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

Somar frequências

6	10	6	2	1	4	0	6	3	1

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

Somar frequências

6	10	6	2	1	4	0	6	3	1

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

Somar frequências

6	10	6	2	1	4	0	6	3	1

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

Somar frequências

6	10	6	2	1	4	0	6	3	1

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

Somar frequências

6	10	6	2	1	4	0	6	3	1

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

Somar frequências

6	10	6	2	1	4	0	6	3	1

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

Somar frequências

6	10	6	2	1	4	0	6	3	1

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

Somar frequências

6	10	6	2	1	4	0	6	3	1

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

Somar frequências

6	10	6	2	1	4	0	6	3	1

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

Somar frequências

6	10	6	2	1	4	0	6	3	1

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

Somar frequências

6	10	6	2	1	4	0	6	3	1

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

Somar frequências

6	10	6	2	1	4	0	6	3	1

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

Somar frequências

6	10	6	2	1	4	0	6	3	1

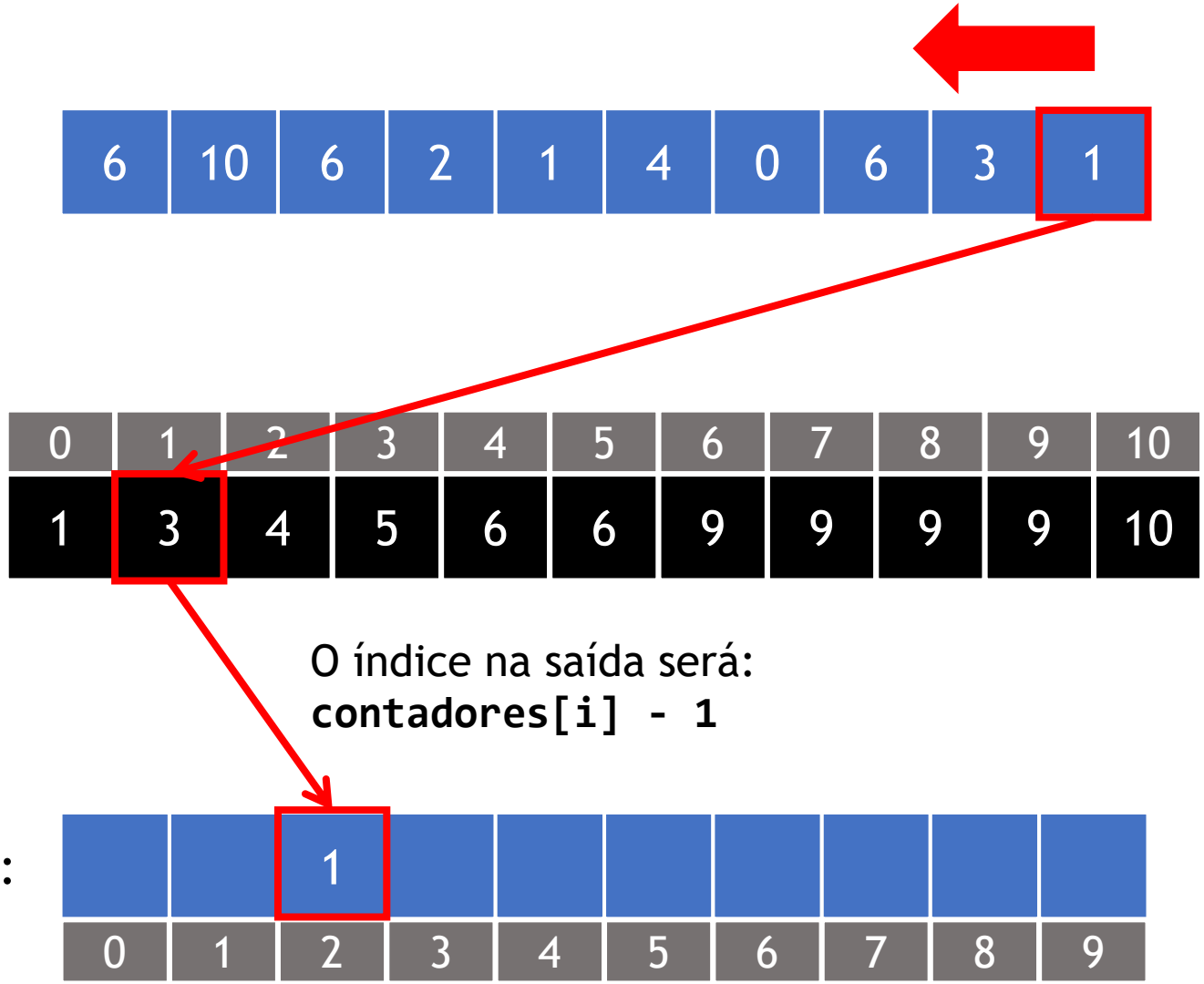
0	1	2	3	4	5	6	7	8	9	10
1	3	4	5	6	6	9	9	9	9	1

Somar frequências

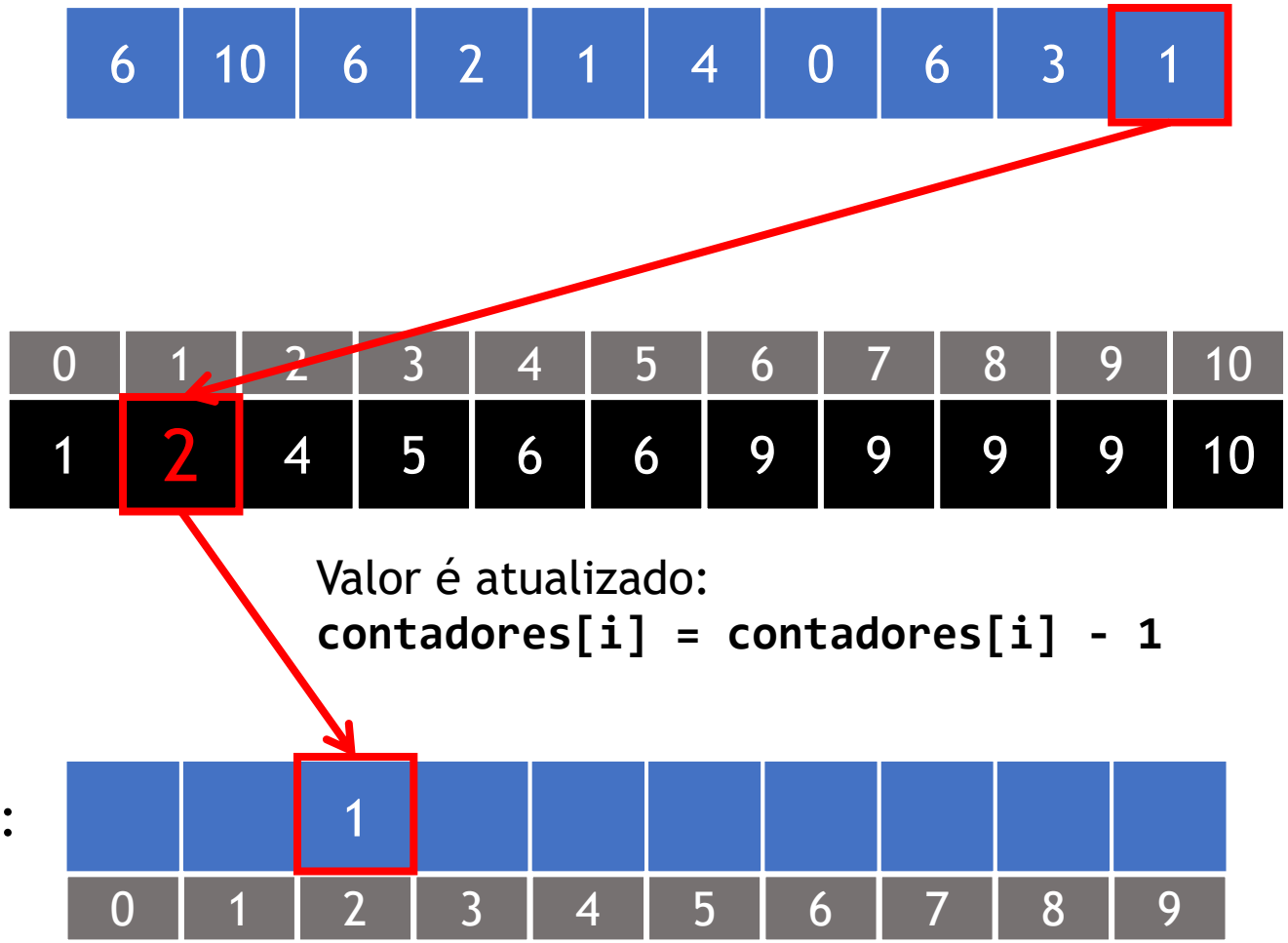
6	10	6	2	1	4	0	6	3	1

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

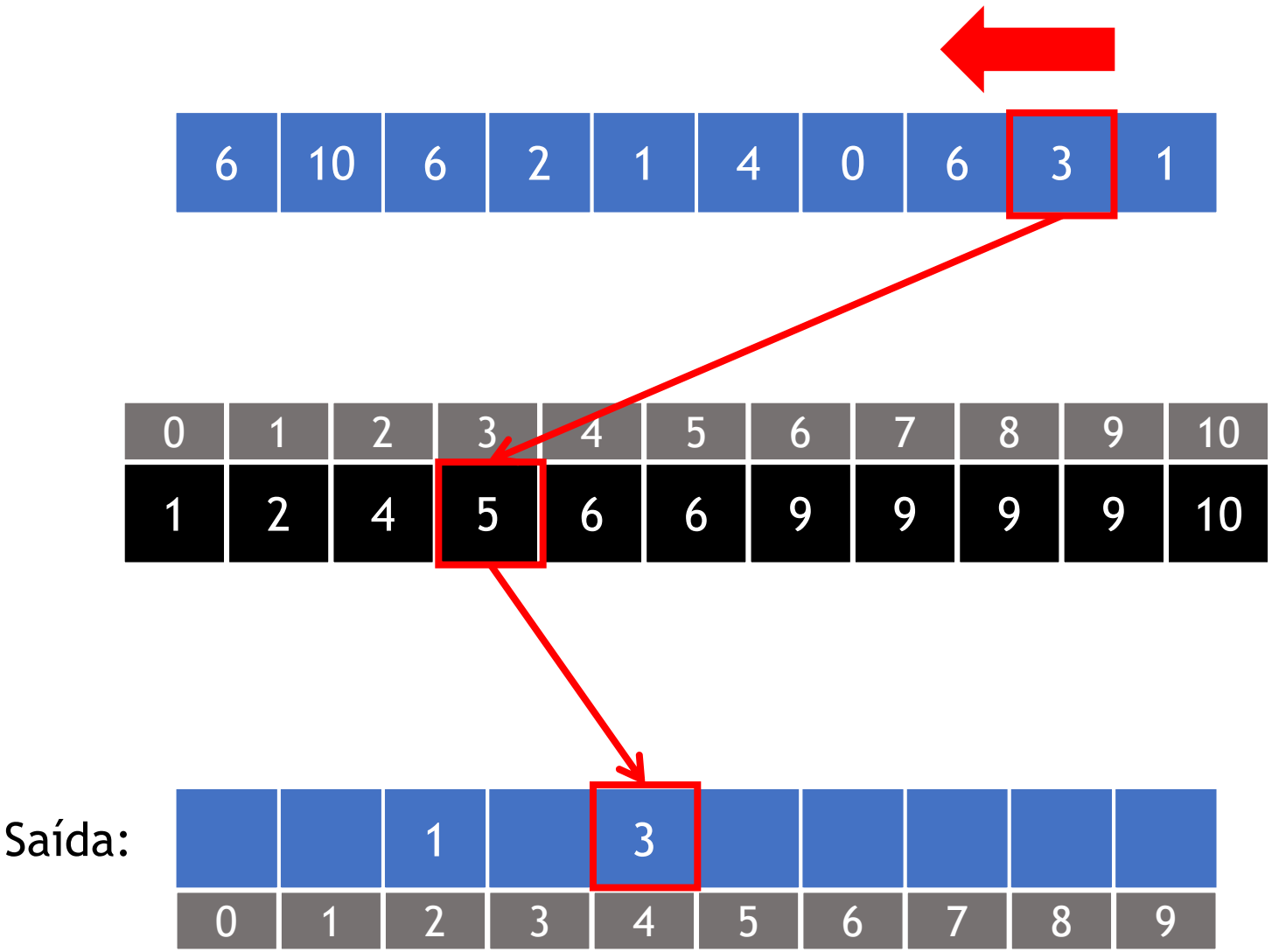
Preencher vetor de saída



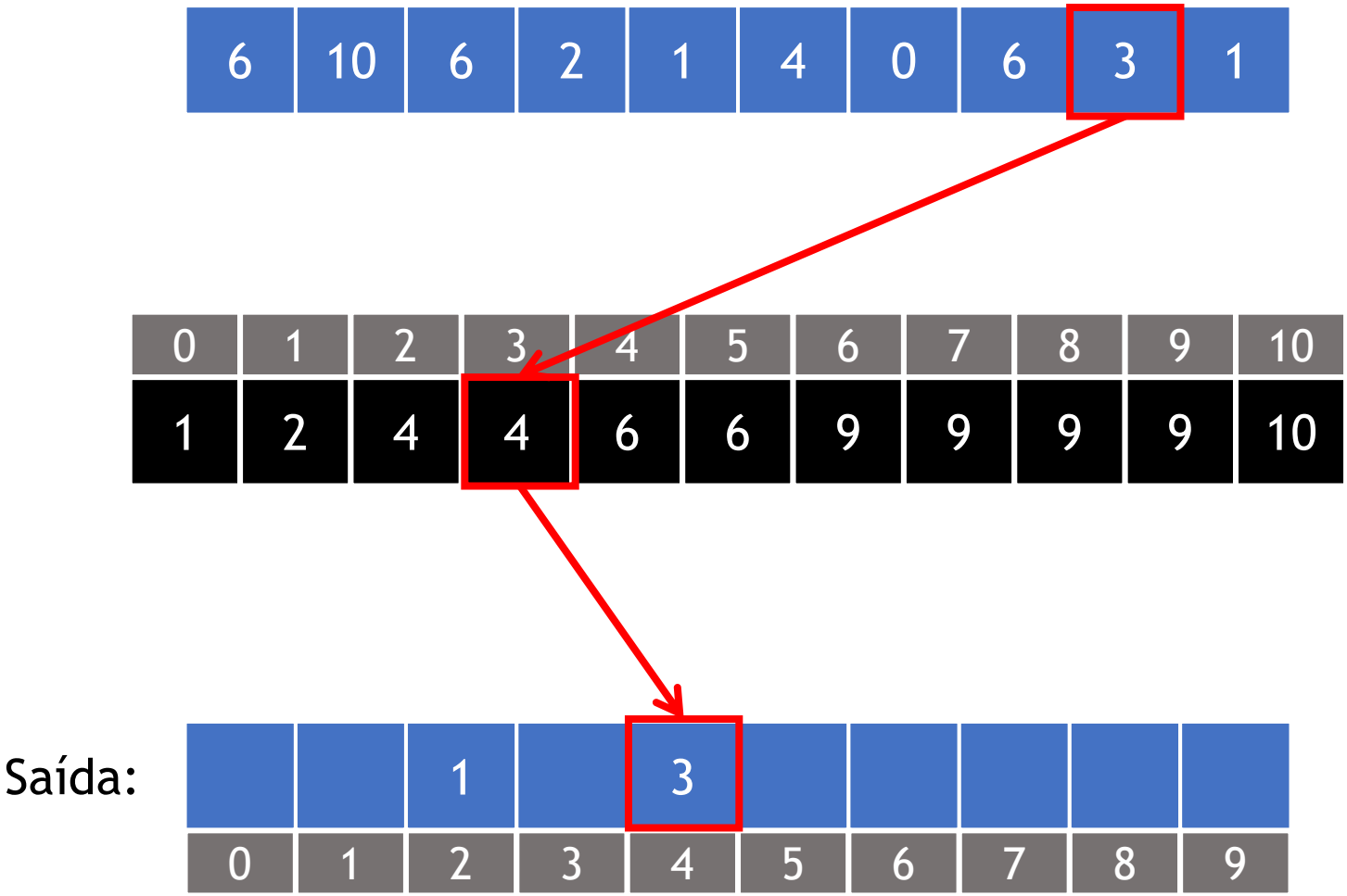
Preencher vetor de saída



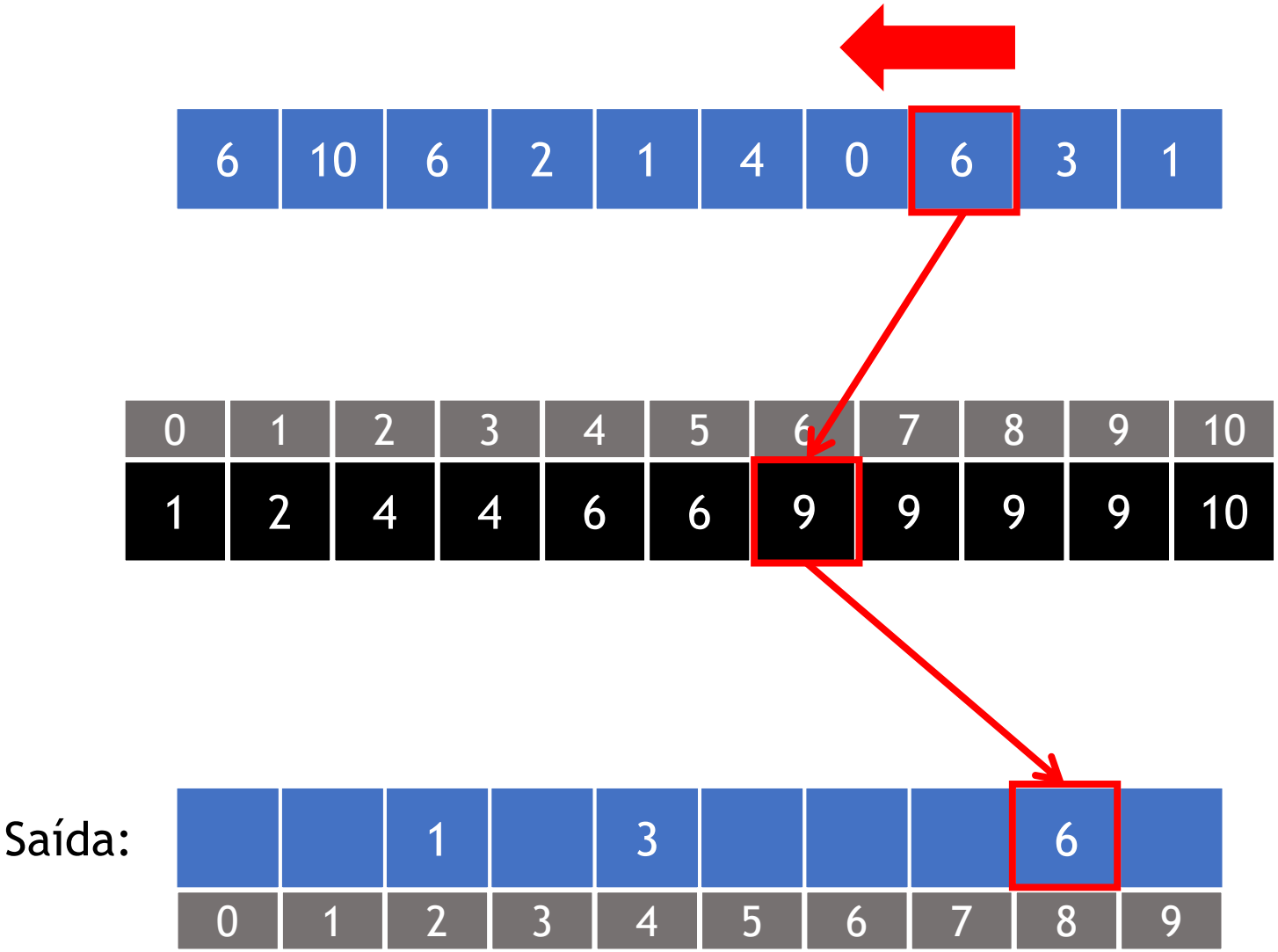
Preencher vetor de saída



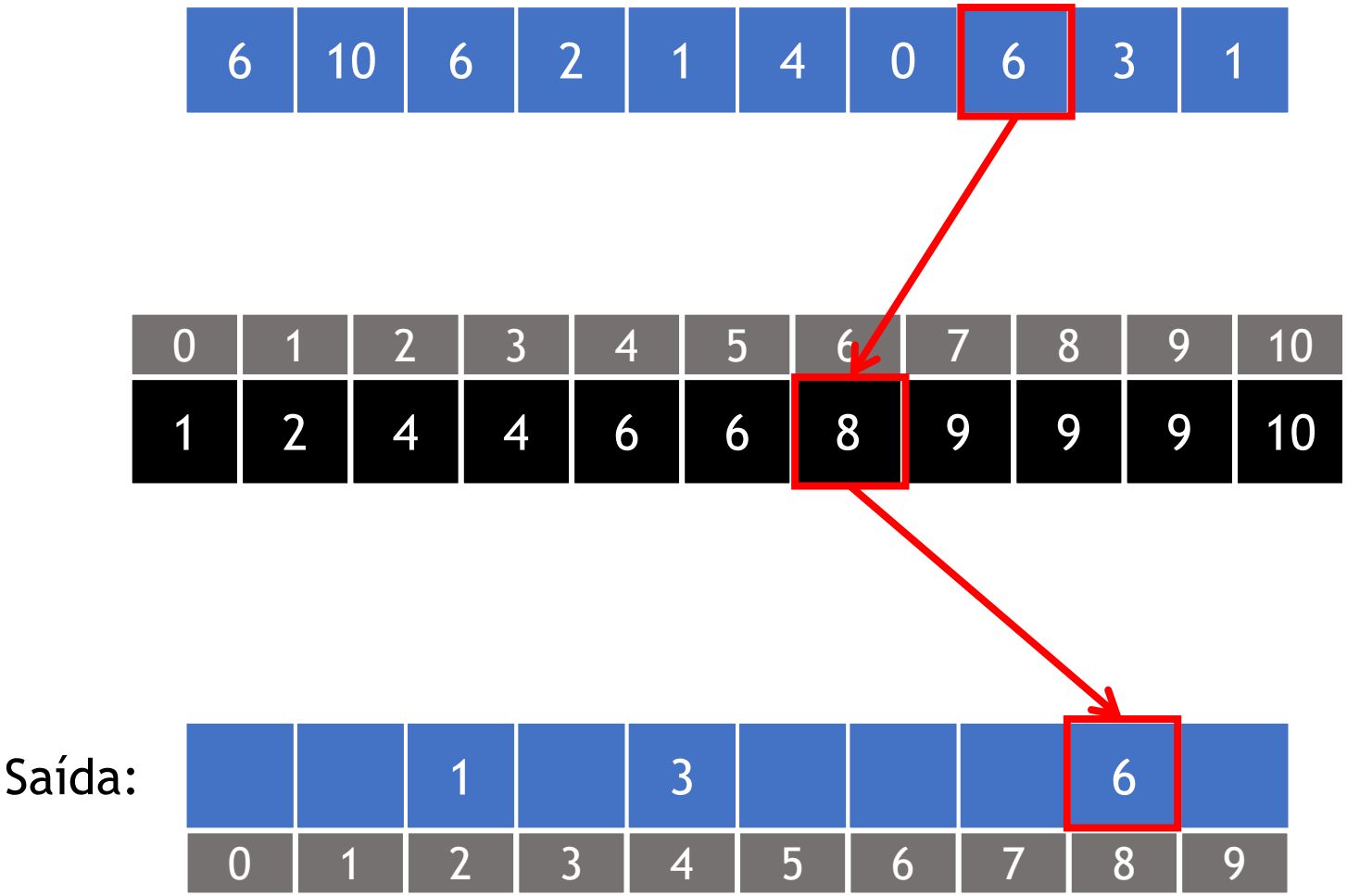
Preencher vetor de saída



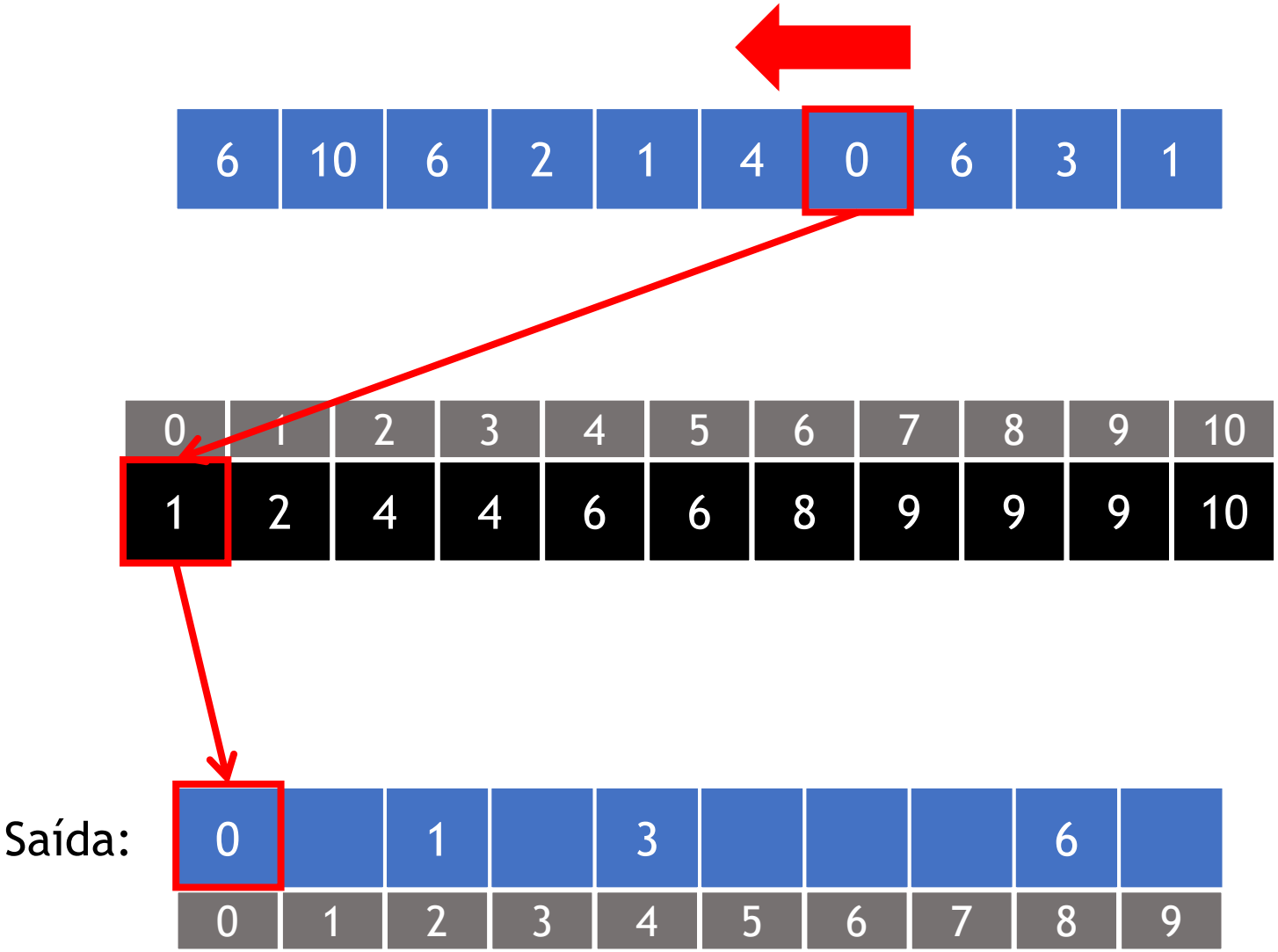
Preencher vetor de saída



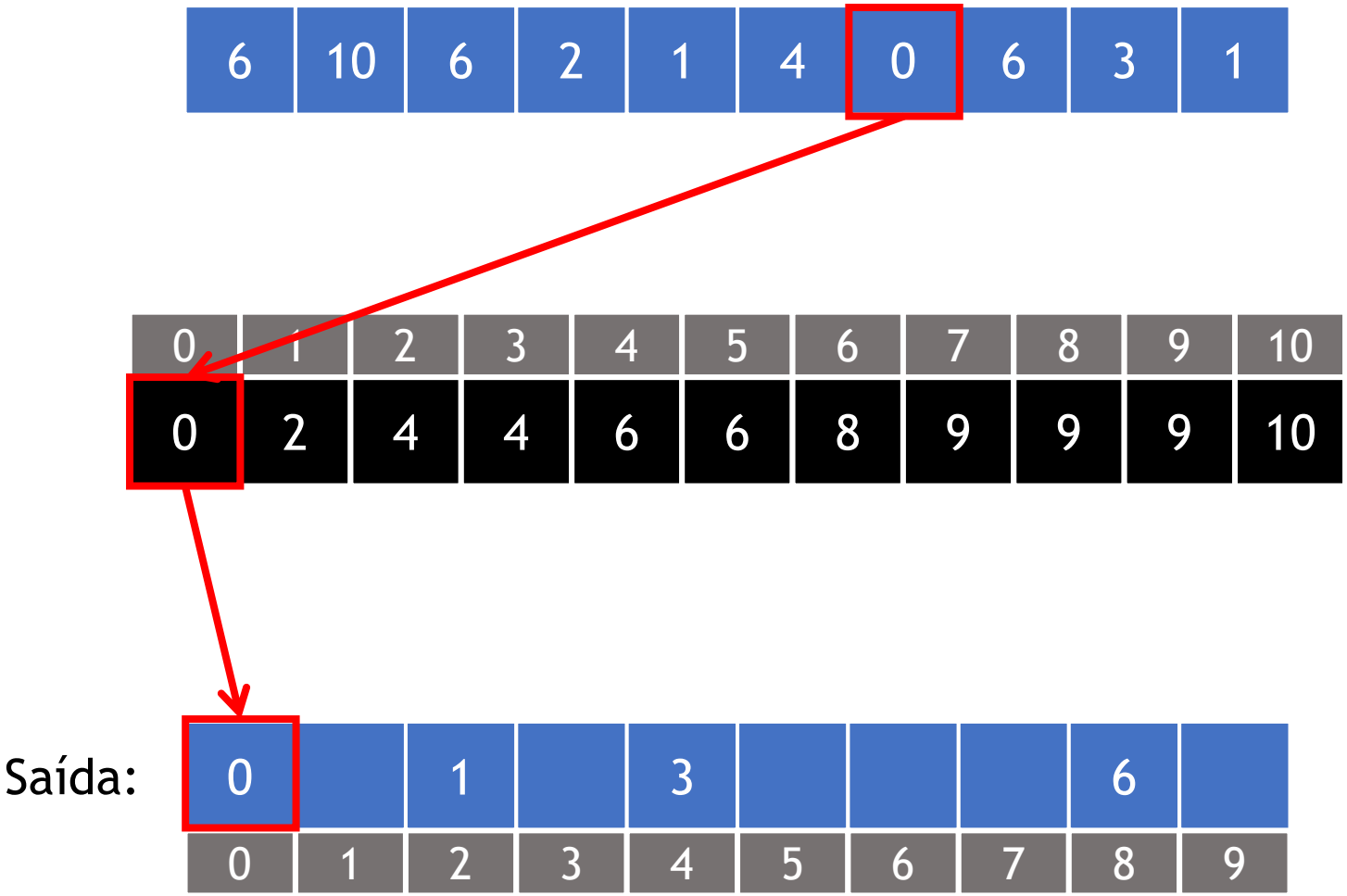
Preencher vetor de saída



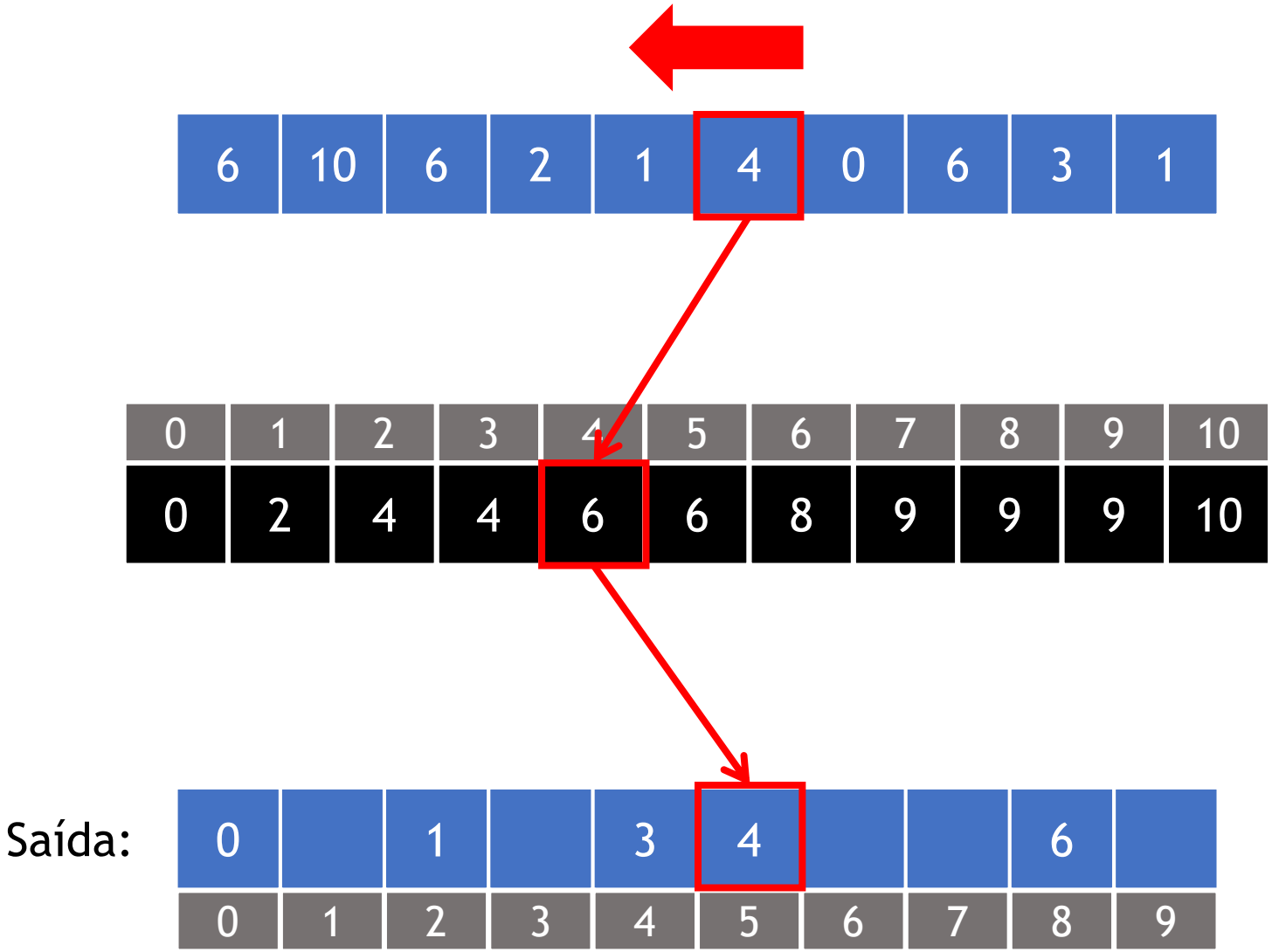
Preencher vetor de saída



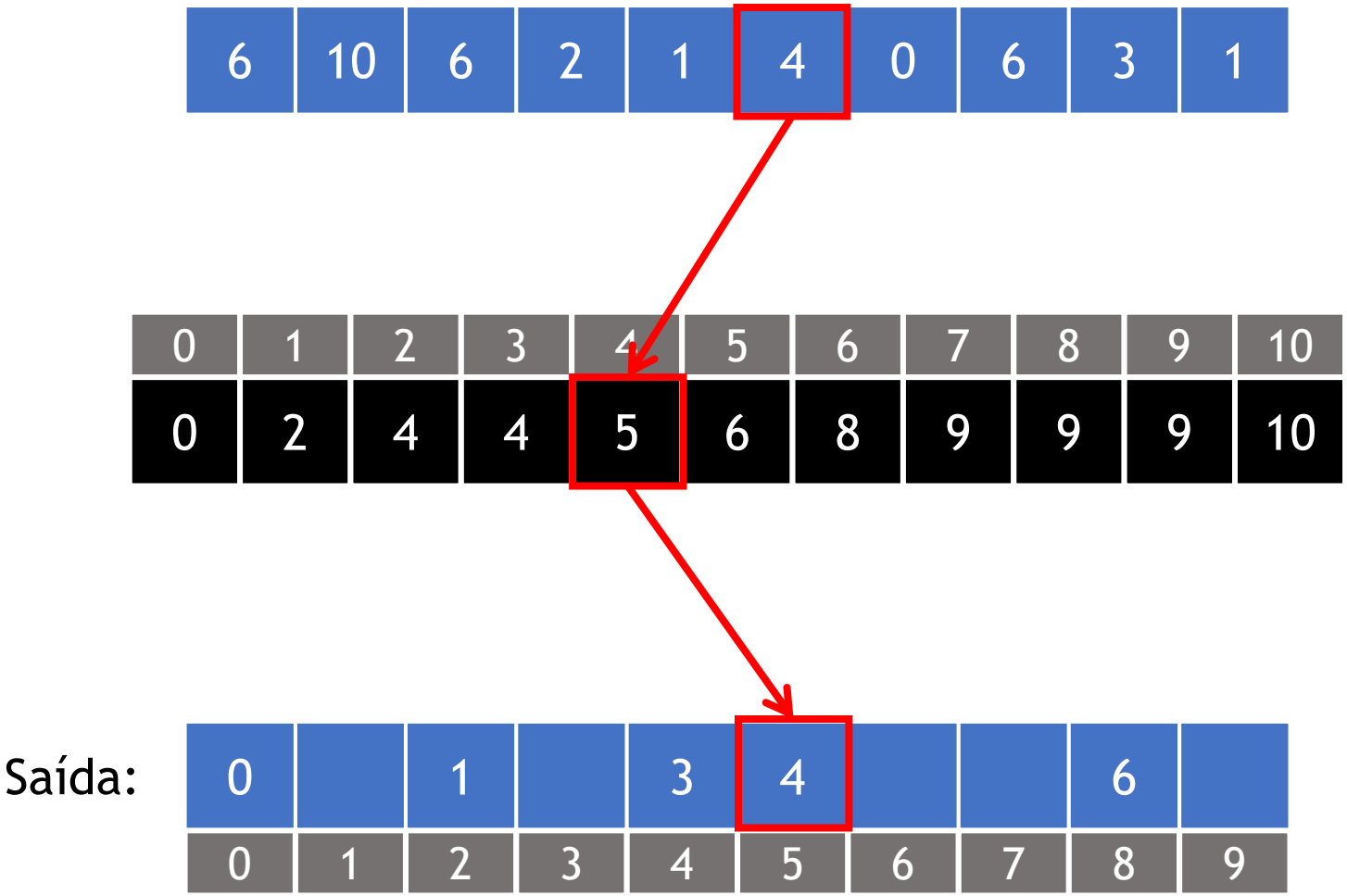
Preencher vetor de saída



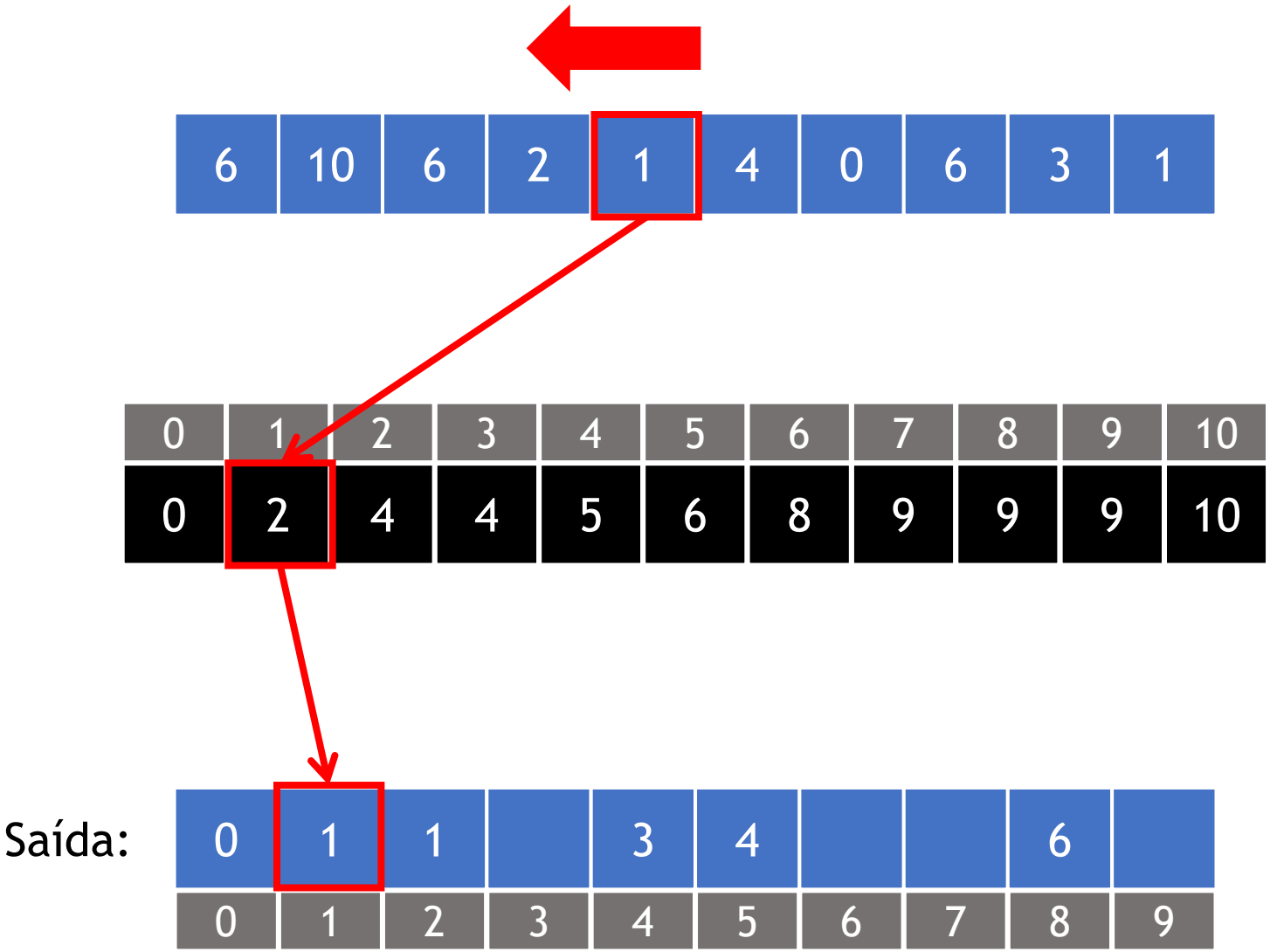
Preencher vetor de saída



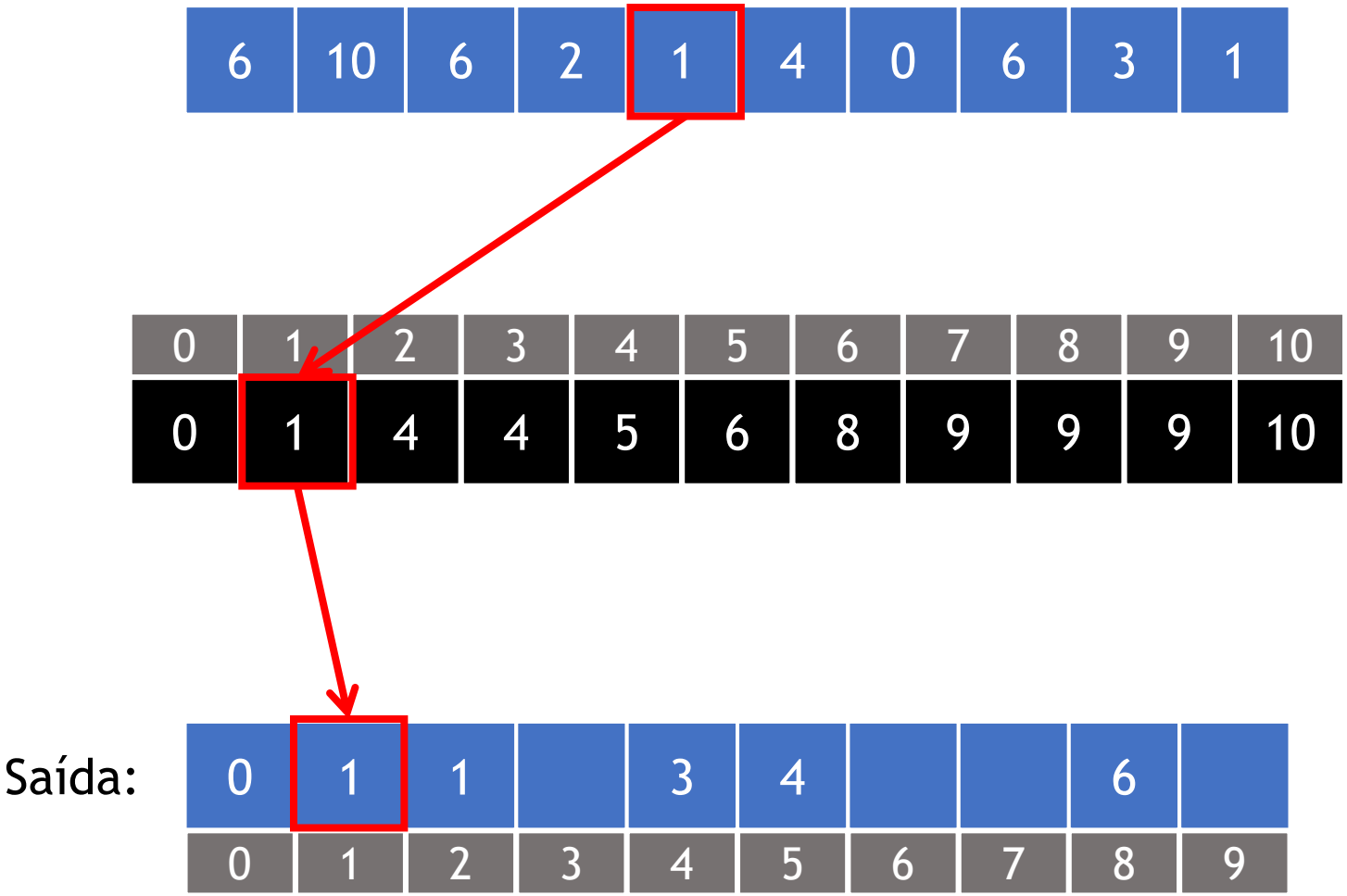
Preencher vetor de saída



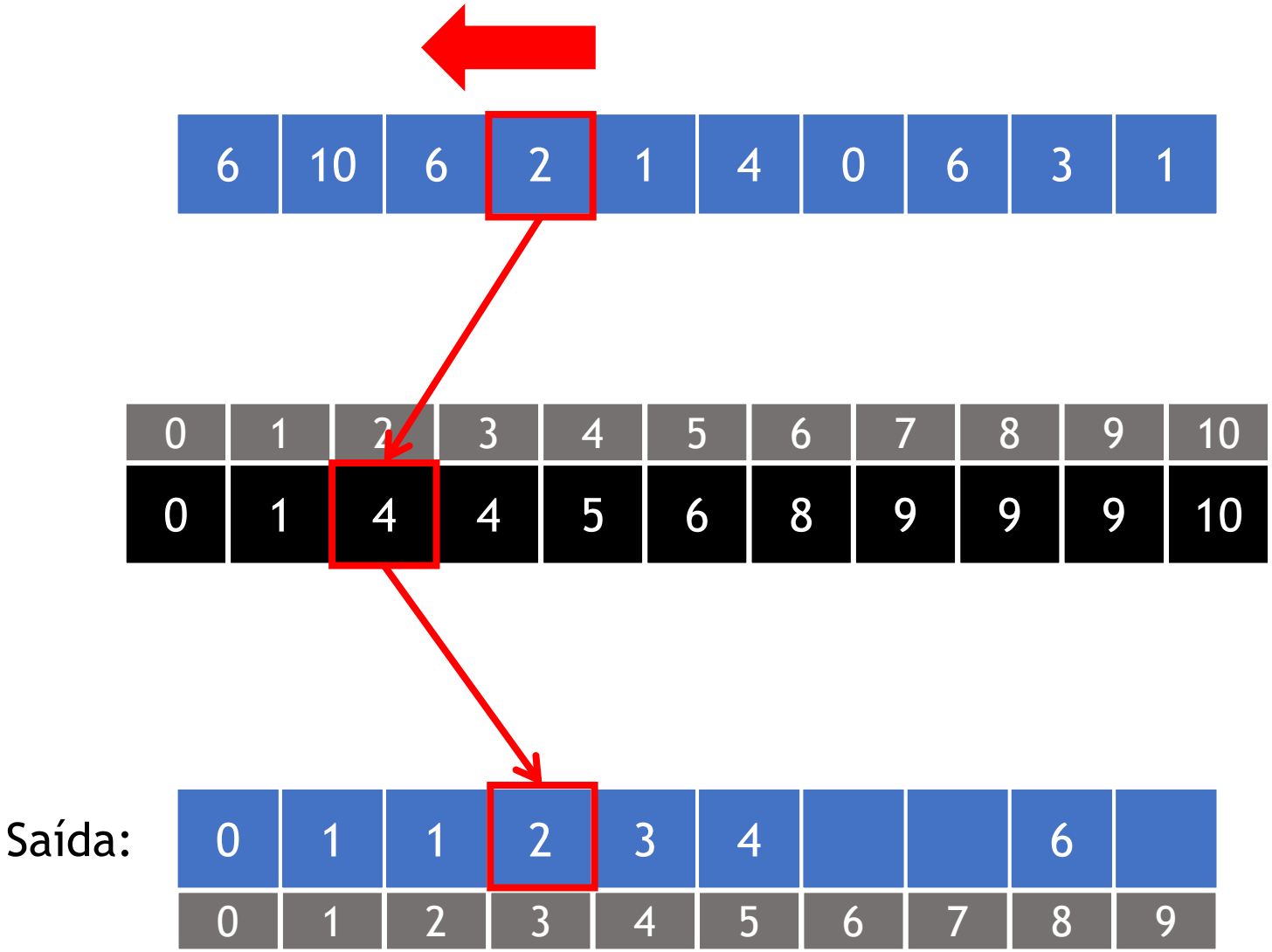
Preencher vetor de saída



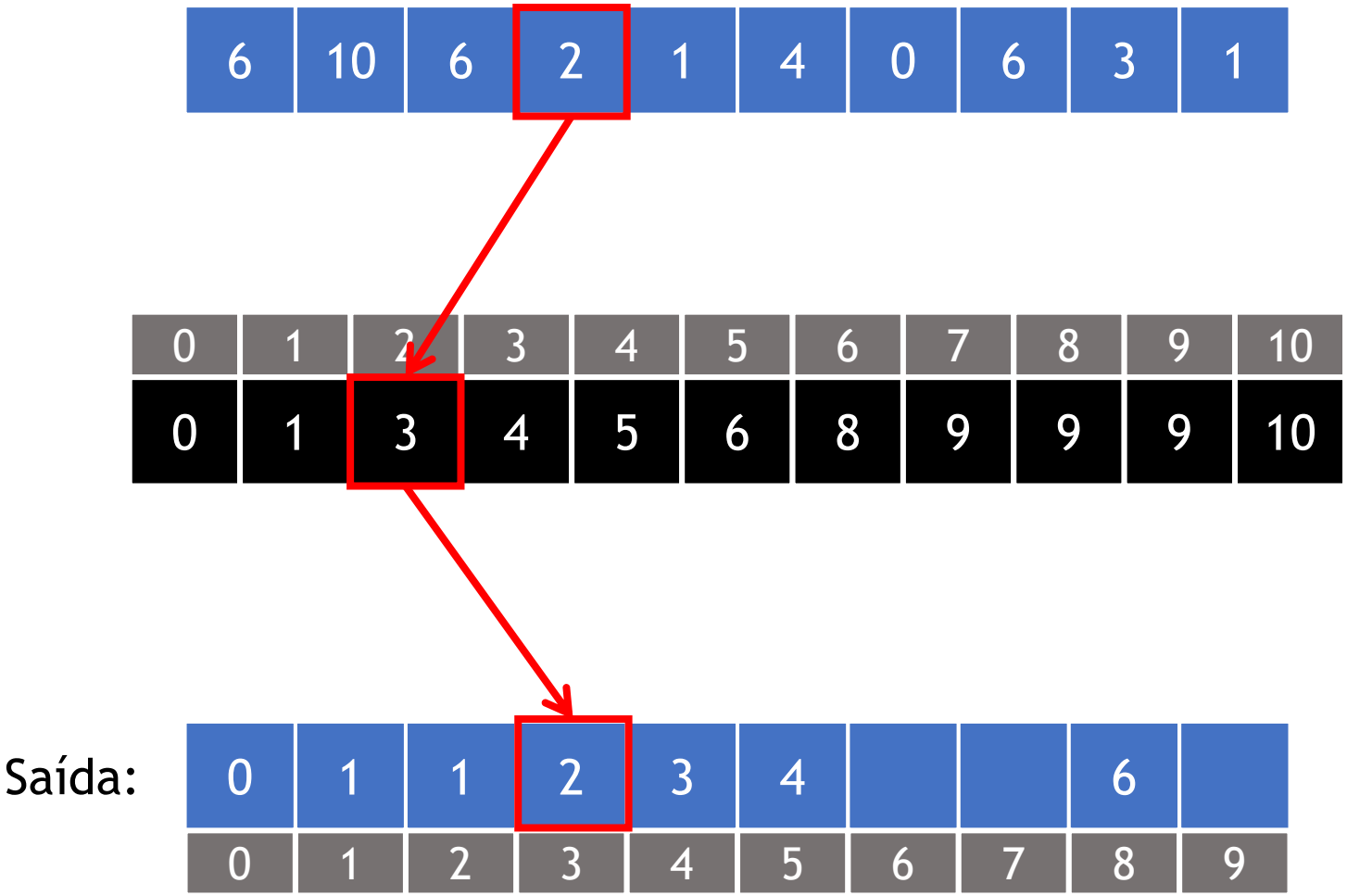
Preencher vetor de saída



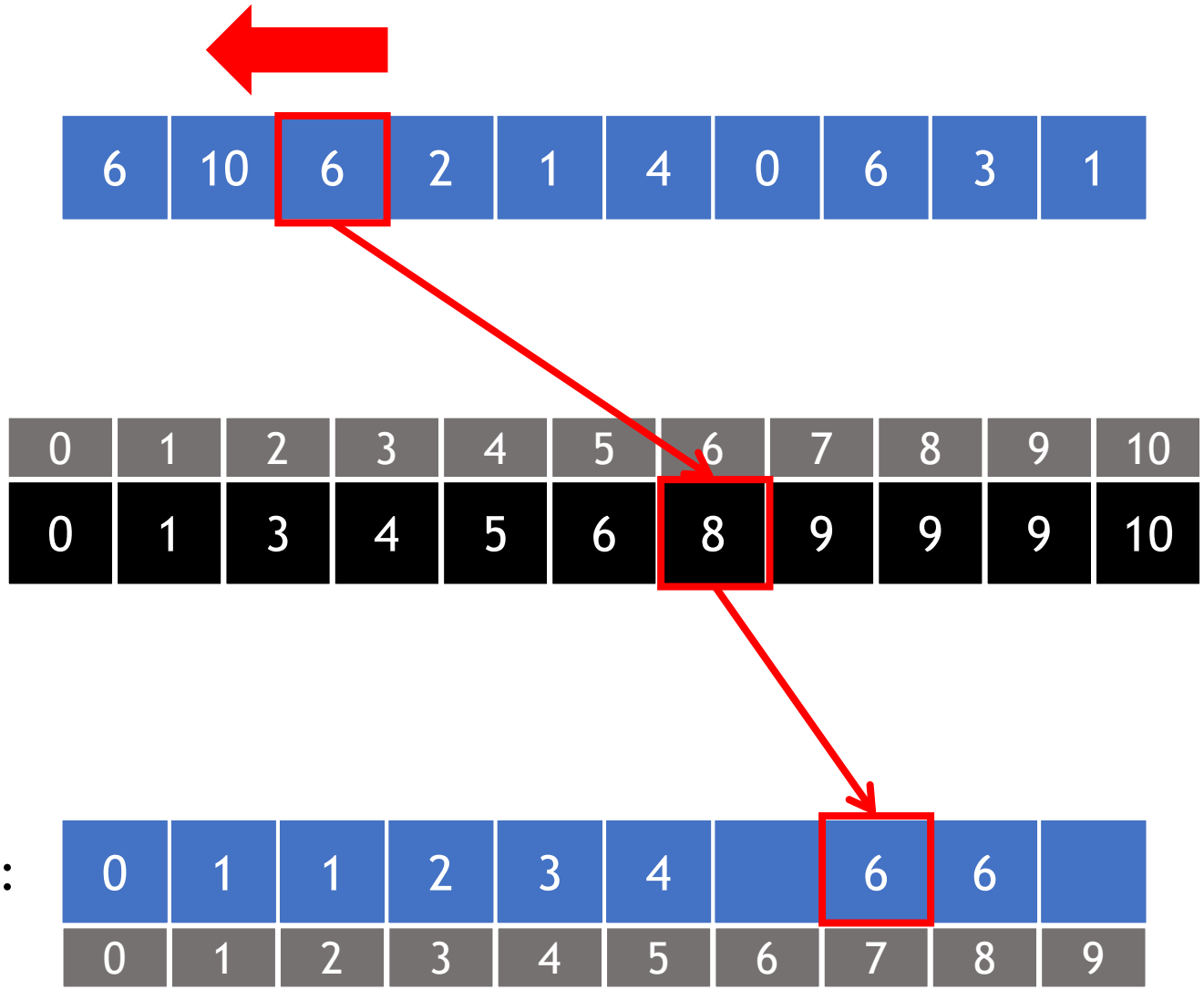
Preencher vetor de saída



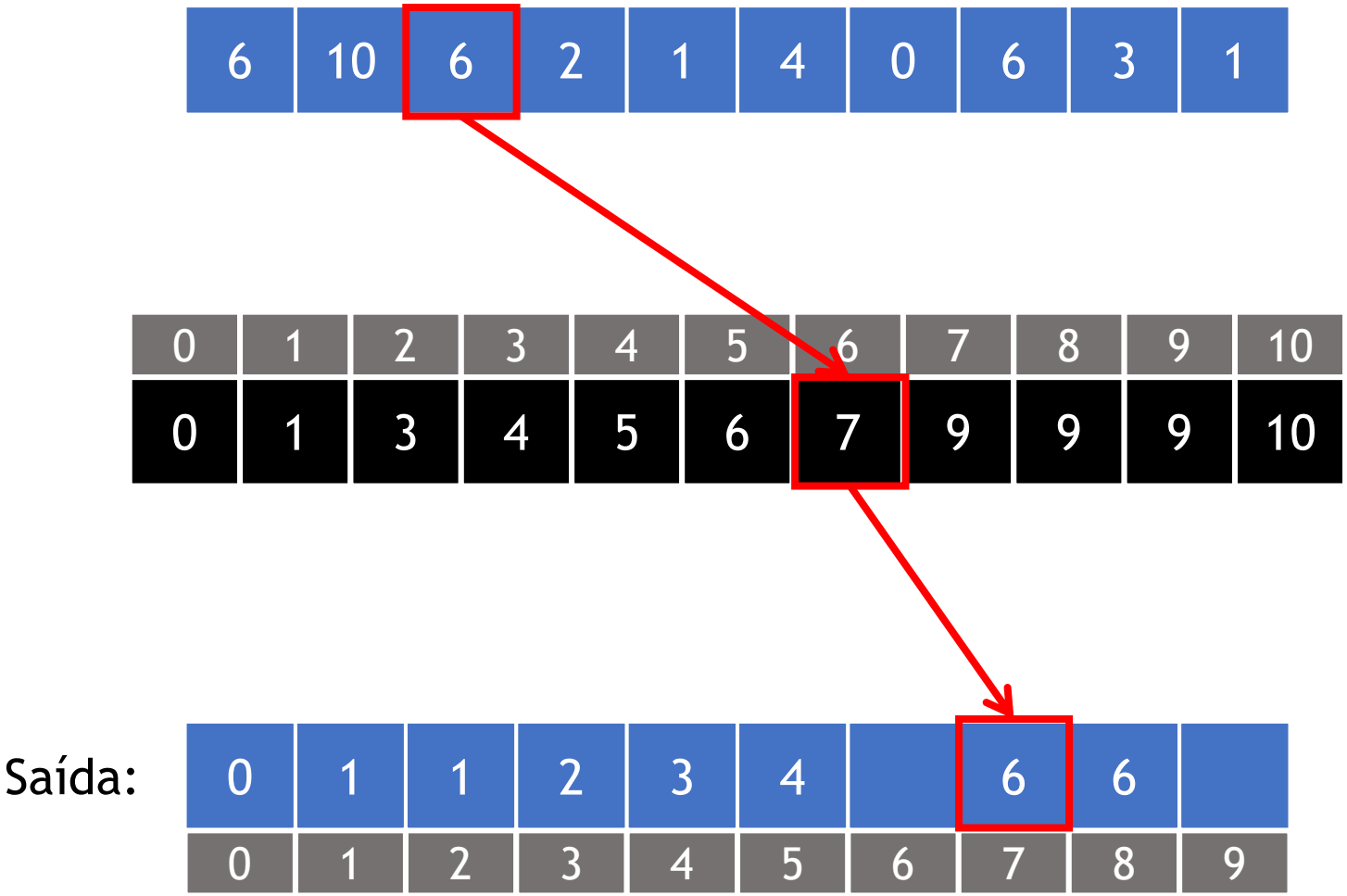
Preencher vetor de saída



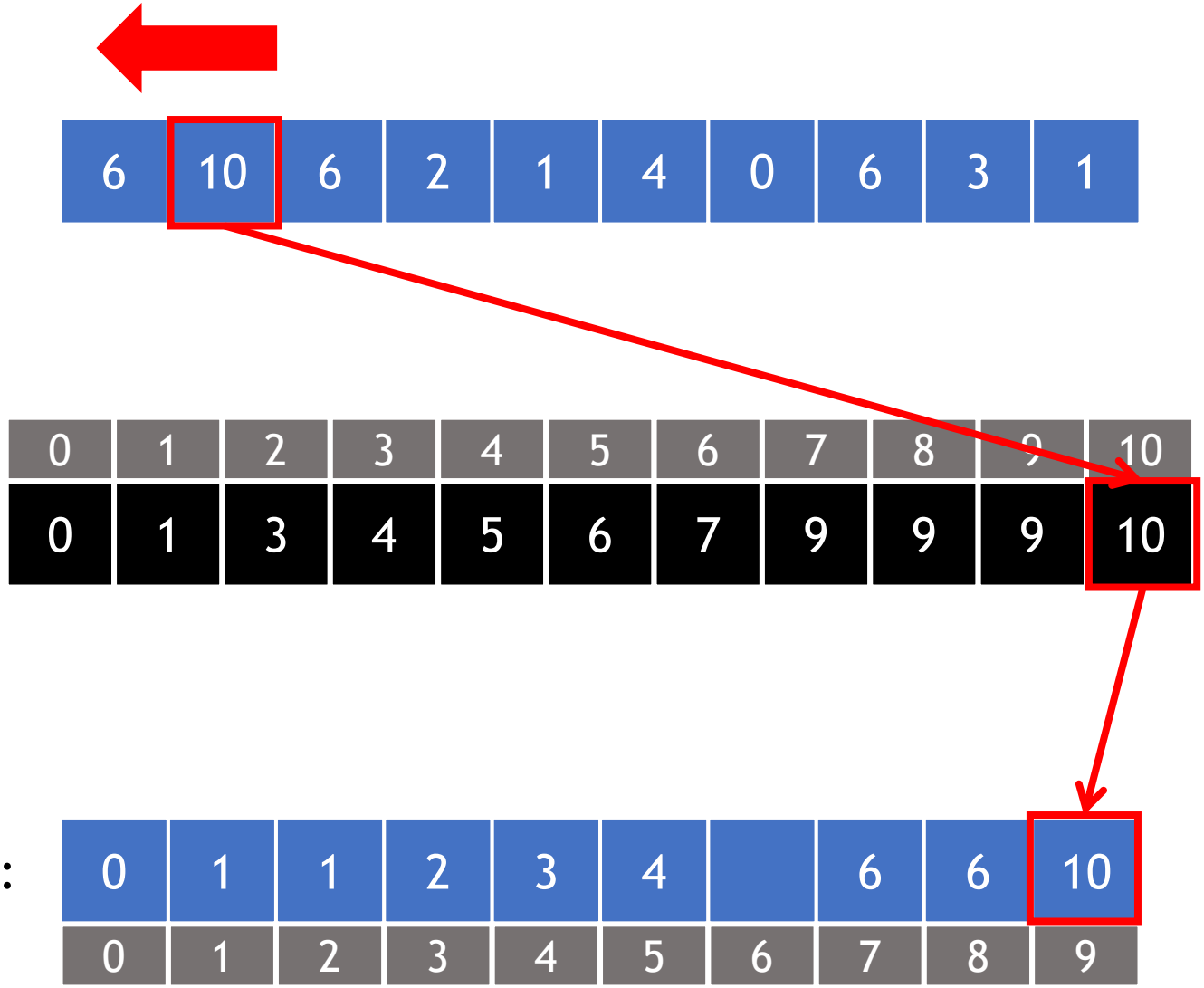
Preencher vetor de saída



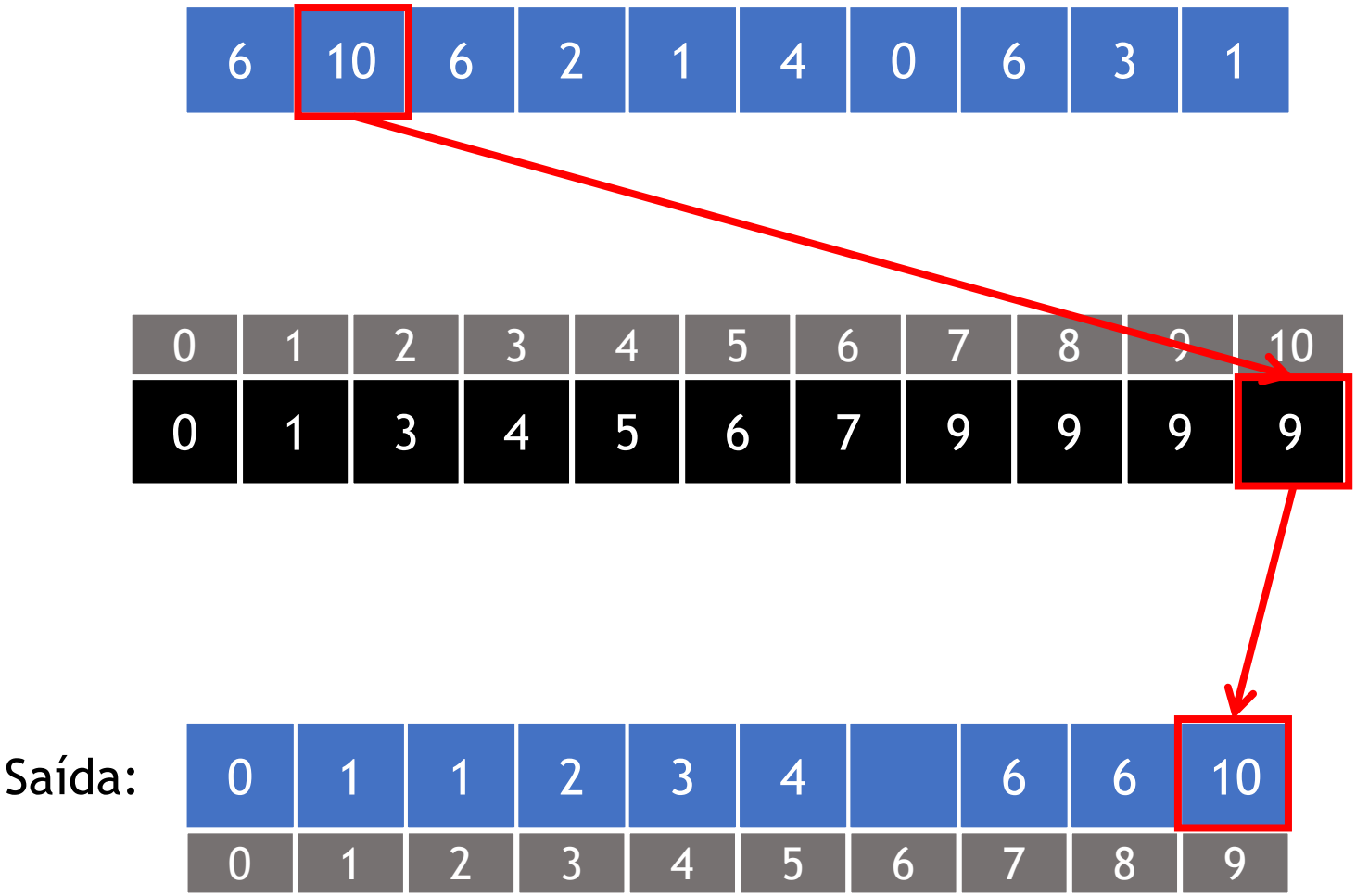
Preencher vetor de saída



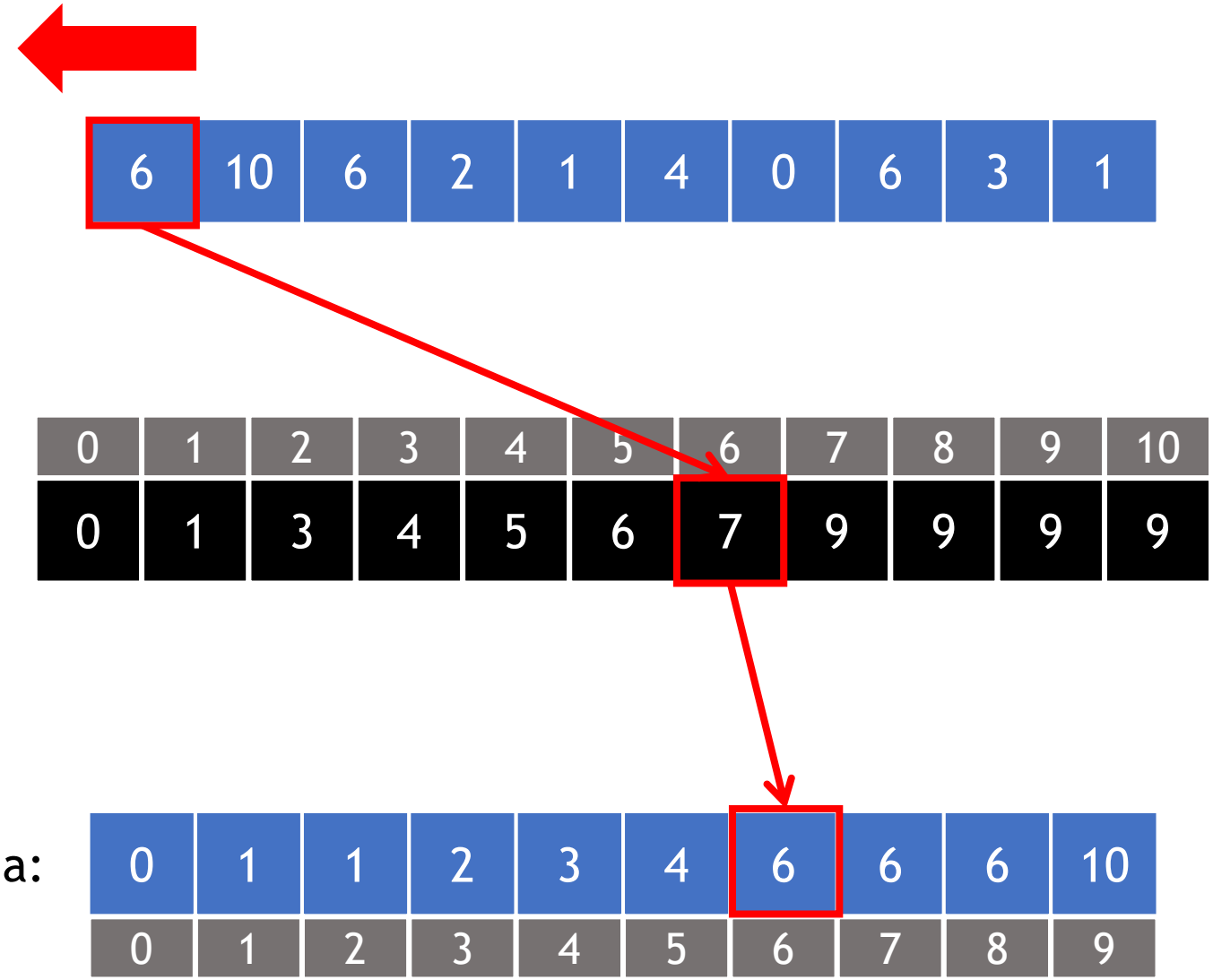
Preencher vetor de saída



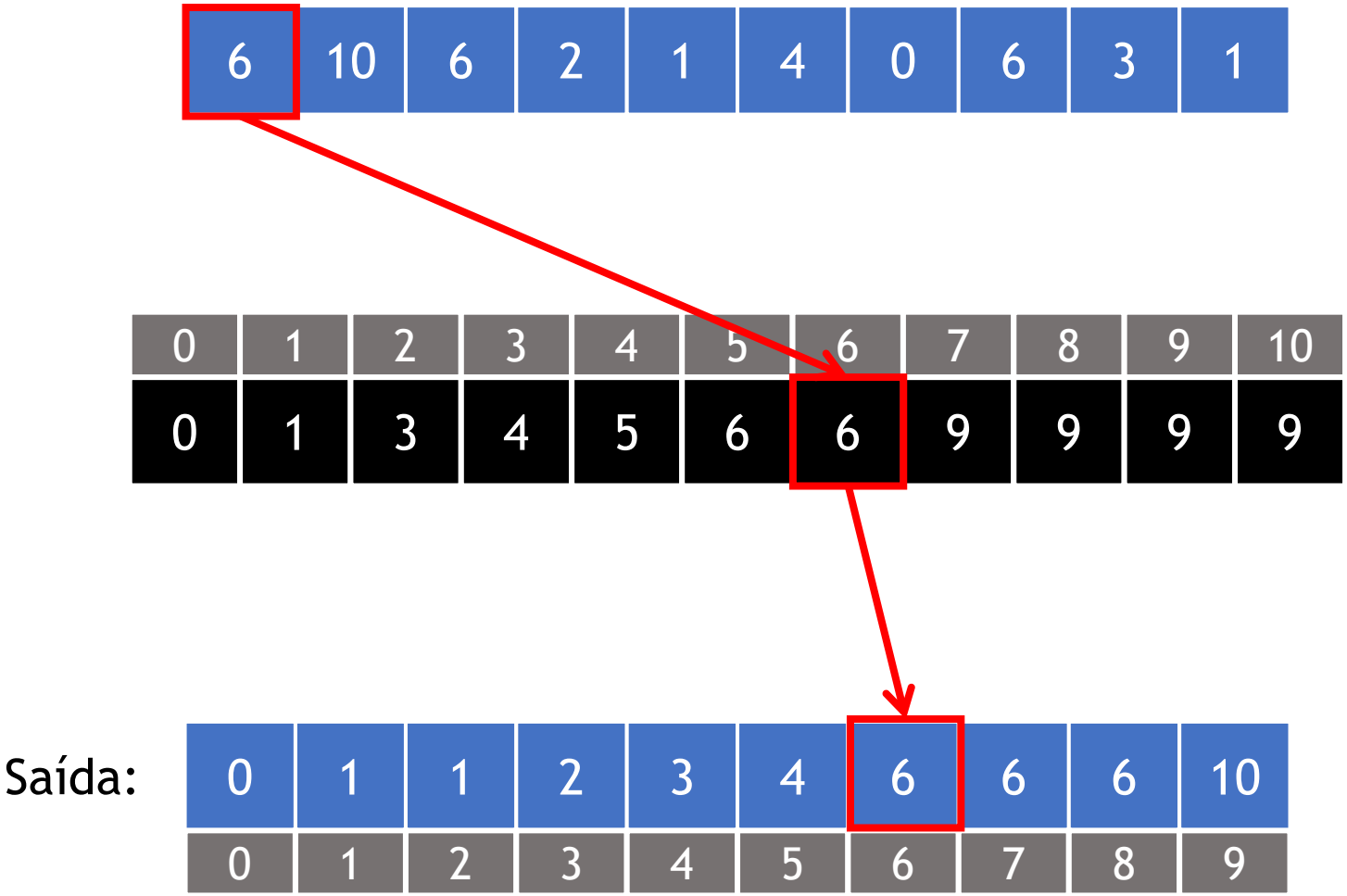
Preencher vetor de saída



Preencher vetor de saída



Preencher vetor de saída



Preencher vetor de saída

6	10	6	2	1	4	0	6	3	1

Saída:

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

Preencher vetor de saída

6	10	6	2	1	4	0	6	3	1

Na ordenação por contagem (counting sort), o vetor é percorrido da direita para a esquerda.

A ordenação seria realizada corretamente se o vetor for percorrido da esquerda para direita? Se sim, há alguma diferença?

Counting sort

- Implementação C

Chamada:

```
countingsort(vetor, n, k);
```

Implementação

```
void countingsort(int *v, int n, int k) {
    int c[k + 1];
    int i;
    for (i = 0; i < k + 1; i++)
        c[i] = 0;

    for (i = 0; i < n; i++)
        c[v[i]]++;

    for (i = 1; i < k + 1; i++)
        c[i] += c[i - 1];

    int saida[n];
    int atual;
    for (i = n - 1; i >= 0; i--) {
        atual = v[i];
        saida[c[atual] - 1] = atual;
        c[atual]--;
    }

    for (i = 0; i < n; i++)
        v[i] = saida[i];
}
```

Custo do algoritmo

```
void countingsort(int *v, int n, int k) {
    int c[k + 1];
    int i;
    for (i = 0; i < k + 1; i++)
        c[i] = 0;
```

→ Executa k+1 vezes.

```
    for (i = 0; i < n; i++)
        c[v[i]]++;
```

→ Executa n vezes.

```
    for (i = 1; i < k + 1; i++)
        c[i] += c[i - 1];
```

→ Executa k vezes.

```
    int saida[n];
    int atual;
    for (i = n - 1; i >= 0; i--) {
        atual = v[i];
        saida[c[atual] - 1] = atual;
        c[atual]--;
    }
```

→ Executa n vezes.

```
    for (i = 0; i < n; i++)
        v[i] = saida[i];
}
```

→ Executa n vezes.

Custo do algoritmo

Quando $k = O(n)$, o tempo de execução da ordenação por contagem é:

$$O(n)$$

```
void countingsort(int *v, int n, int k) {
    int c[k + 1];
    int i;
    for (i = 0; i < k + 1; i++)
        c[i] = 0;
```

→ Executa k+1 vezes.

```
    for (i = 0; i < n; i++)
        c[v[i]]++;
```

→ Executa n vezes.

```
    for (i = 1; i < k + 1; i++)
        c[i] += c[i - 1];
```

→ Executa k vezes.

```
    int saida[n];
    int atual;
    for (i = n - 1; i >= 0; i--) {
        atual = v[i];
        saida[c[atual] - 1] = atual;
        c[atual]--;
    }
```

→ Executa n vezes.

```
    for (i = 0; i < n; i++)
        v[i] = saida[i];
}
```

→ Executa n vezes.

Custo do algoritmo

```
void countingsort(int *v, int n) {
    int k = get_max(v, n);
    int c[k + 1];
    int i;
    for (i = 0; i < k + 1; i++)
        c[i] = 0;

    for (i = 0; i < n; i++)
        c[v[i]]++;

    for (i = 1; i < k + 1; i++)
        c[i] += c[i - 1];

    int saida[n];
    int atual;
    for (i = n - 1; i >= 0; i--) {
        atual = v[i];
        saida[c[atual] - 1] = atual;
        c[atual]--;
    }

    for (i = 0; i < n; i++)
        v[i] = saida[i];
}
```

Alternativa para evitar a necessidade de informar o valor de k.

Radix sort

Radix sort

Ordenação digital

- Ordena os números dígito por dígito:
  - Primeiro ordena os números pelo dígito menos significativo;
  - Depois ordena pelo segundo dígito menos significativo;
  - E assim a ordenação segue até chegar no dígito mais significativo.

Radix sort

Ordenação digital

- Ordena os números dígito por dígito:
  - Primeiro **ordena** os números pelo dígito menos significativo;
  - Depois **ordena** pelo segundo dígito menos significativo;
  - E assim a **ordenação** segue até chegar no dígito mais significativo.

Importante: a ordenação utilizada deve ser **estável**.
Será usado o algoritmo **Counting sort**.

Radix sort

383	886	777	915	793	335	386	492	649	421

Radix sort

Ordenação inicia pelo dígito menos significativo:

383	886	777	915	793	335	386	492	649	421

Radix sort

Ordenação inicia pelo dígito menos significativo:

383	886	777	915	793	335	386	492	649	421

Para base 10, cada dígito pode assumir 10 valores. Portanto, o vetor de contadores terá 10 posições.

Contadores:

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

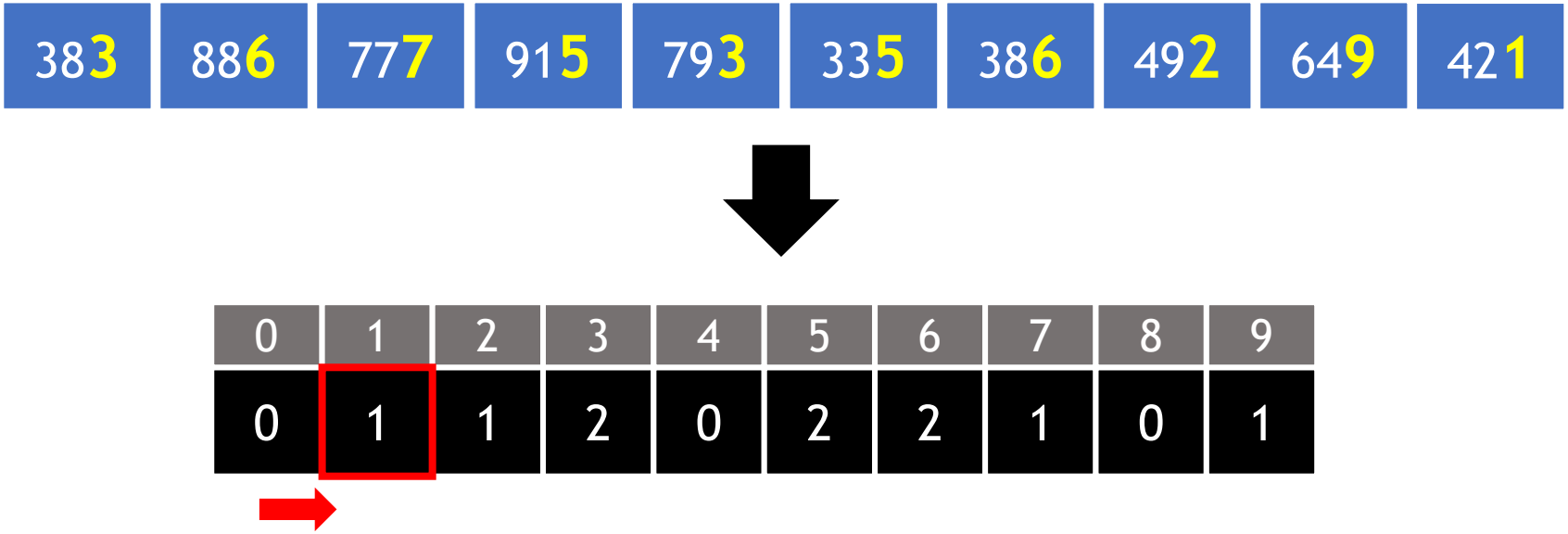
Radix sort

383	886	777	915	793	335	386	492	649	421

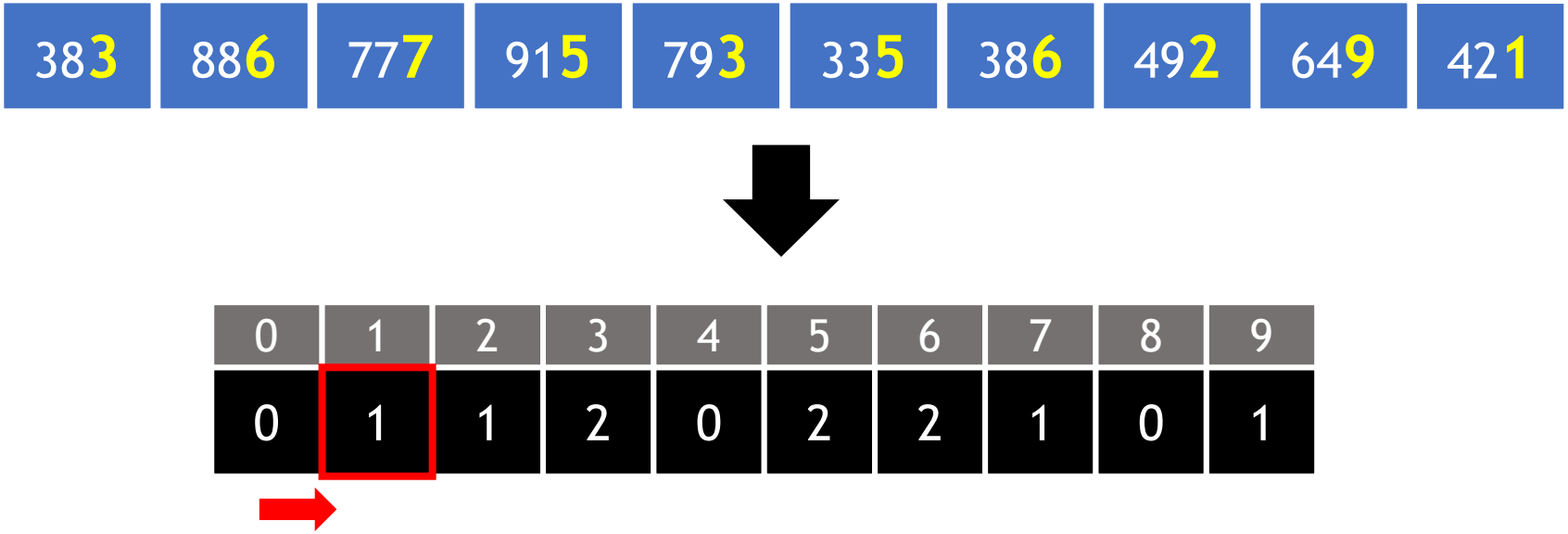
Contadores:

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

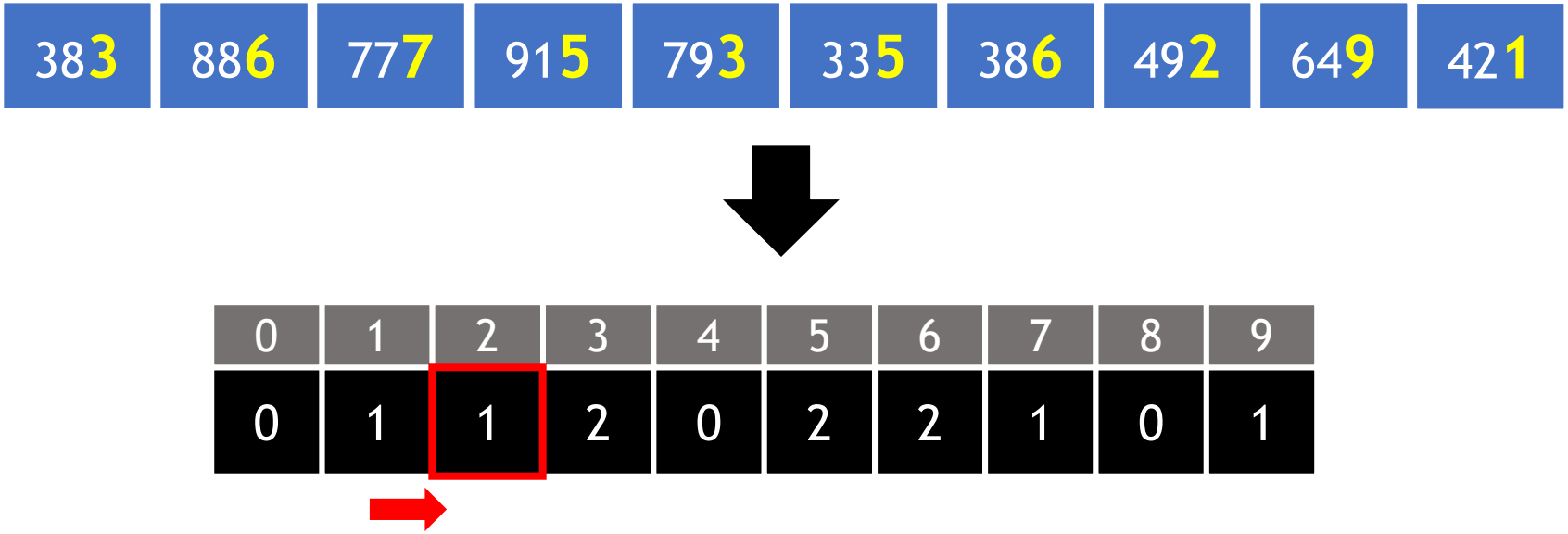
Radix sort


383
886
777
915
793
335
386
492
649
421
0 1 2 3 4 5 6 7 8 9
0 1 1 2 0 2 2 1 0 1

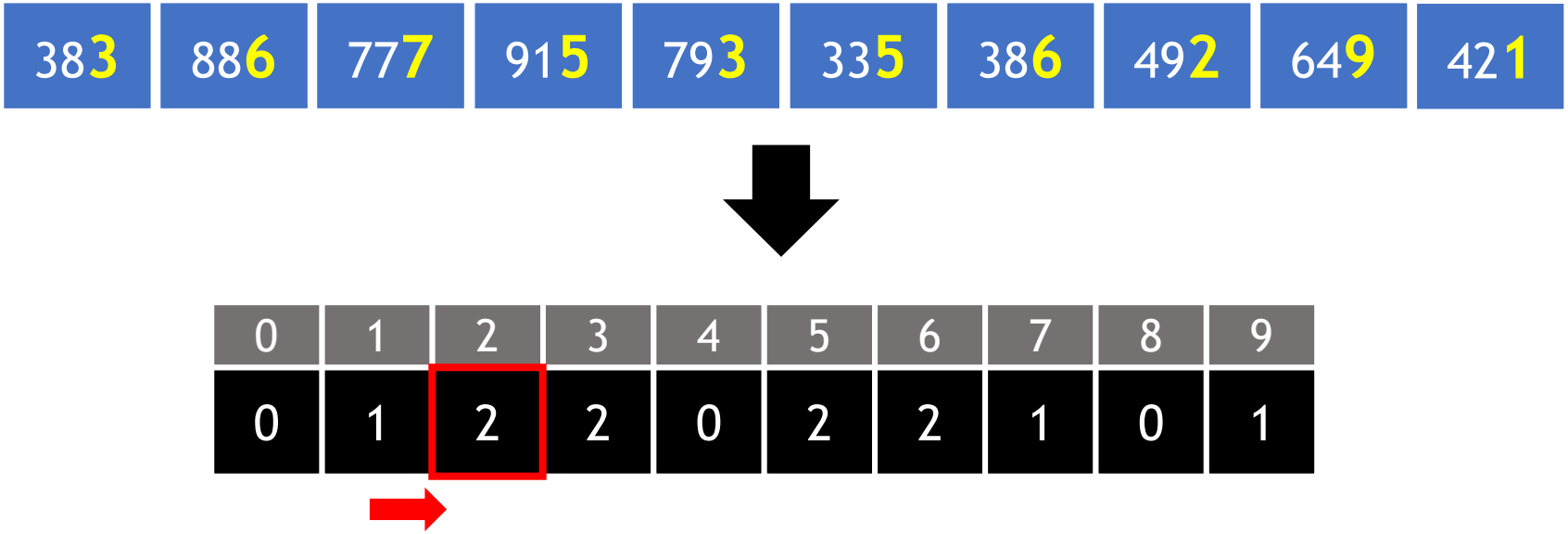
Radix sort


383
886
777
915
793
335
386
492
649
421
0 1 2 3 4 5 6 7 8 9
0 1 1 2 0 2 2 1 0 1

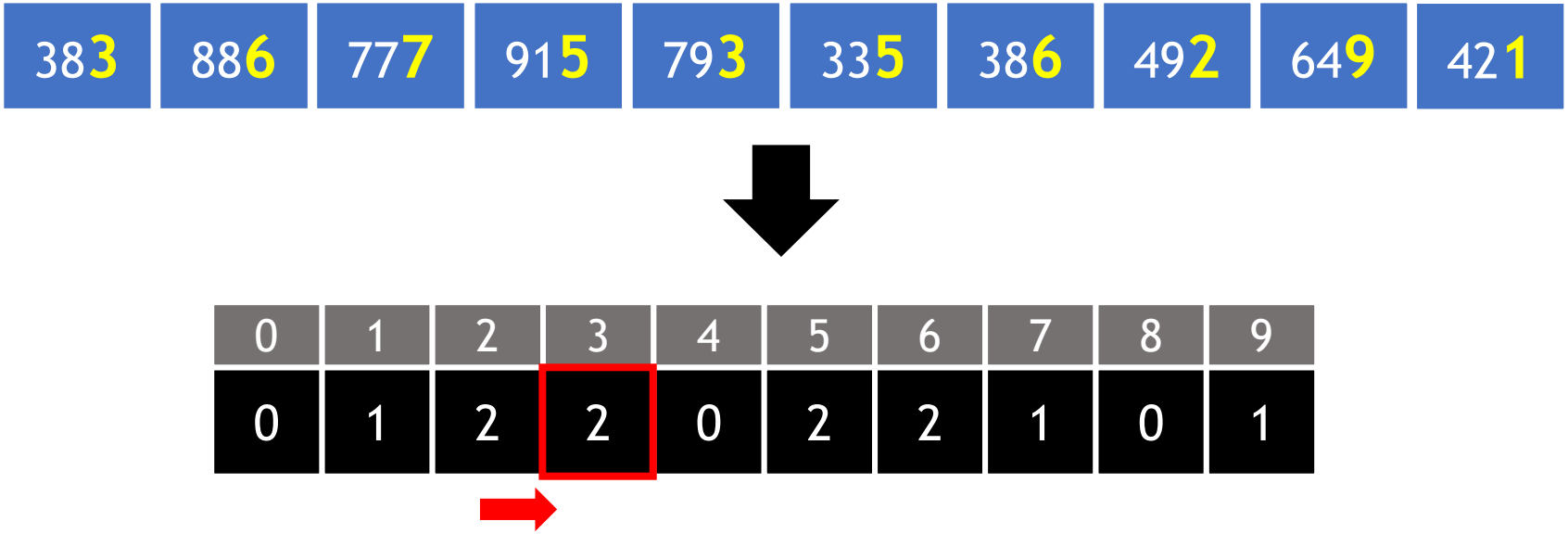
Radix sort


383
886
777
915
793
335
386
492
649
421
0 1 2 3 4 5 6 7 8 9
0 1 1 2 0 2 2 1 0 1

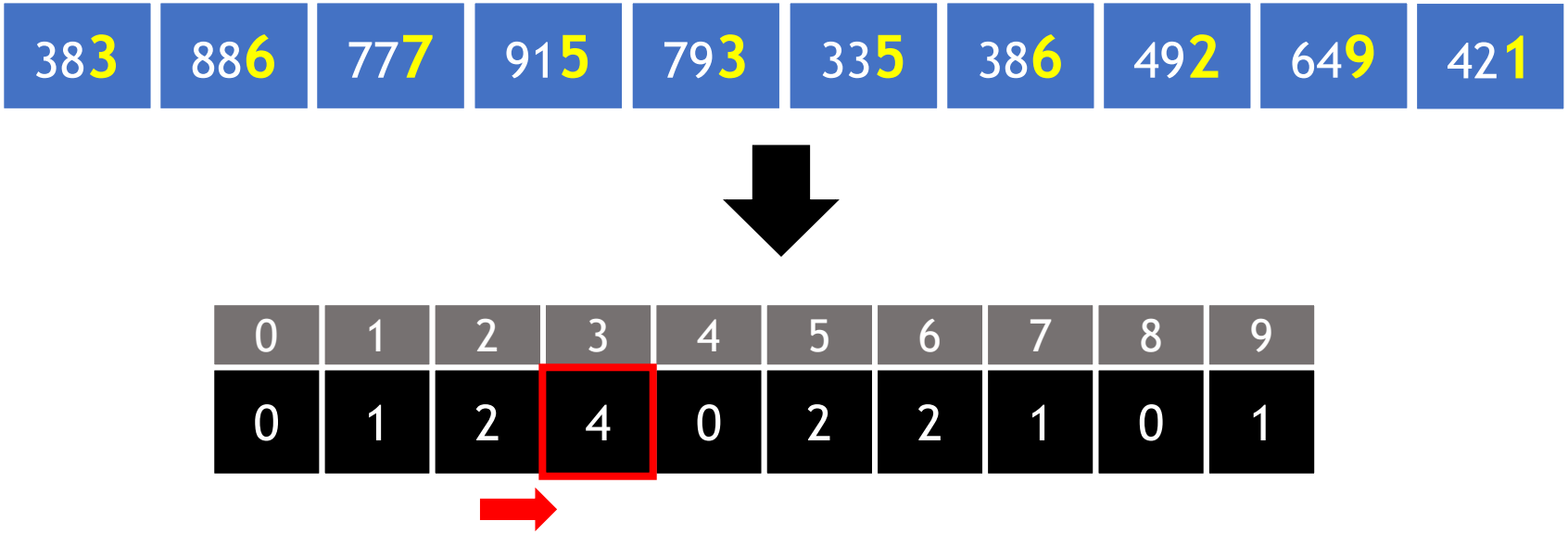
Radix sort


383
886
777
915
793
335
386
492
649
421
0 1 2 3 4 5 6 7 8 9
0 1 2 2 0 2 2 1 0 1

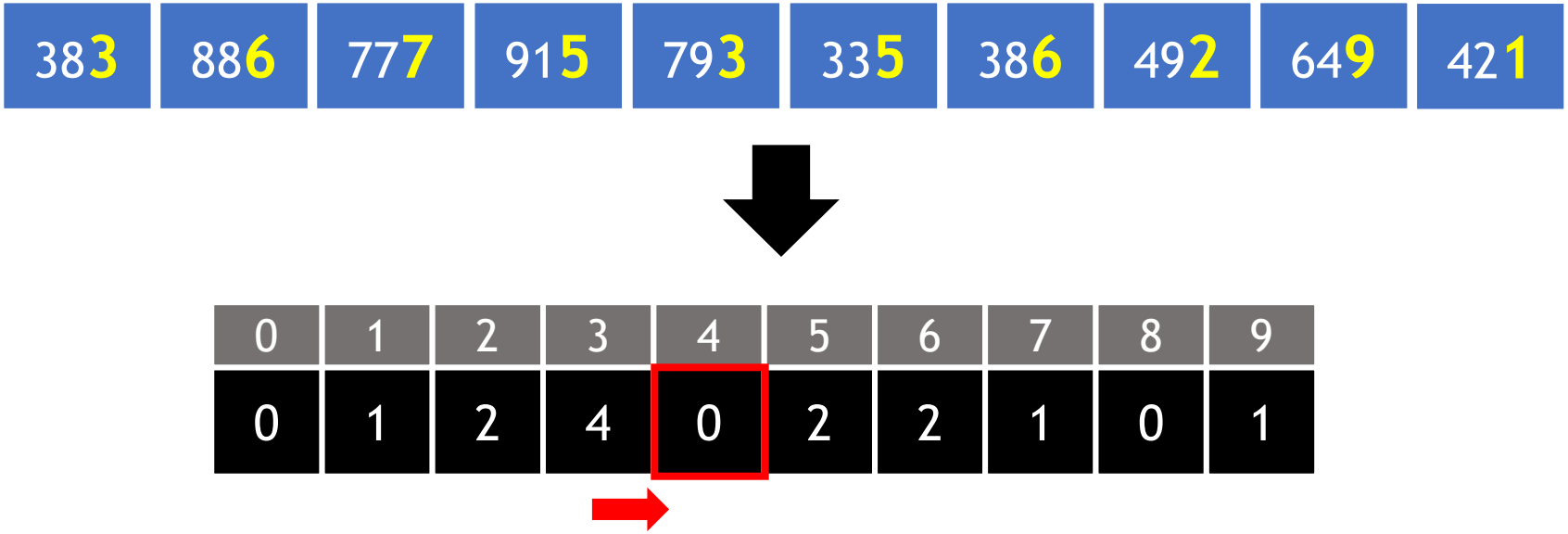
Radix sort


383
886
777
915
793
335
386
492
649
421
0 1 2 3 4 5 6 7 8 9
0 1 2 2 0 2 2 1 0 1

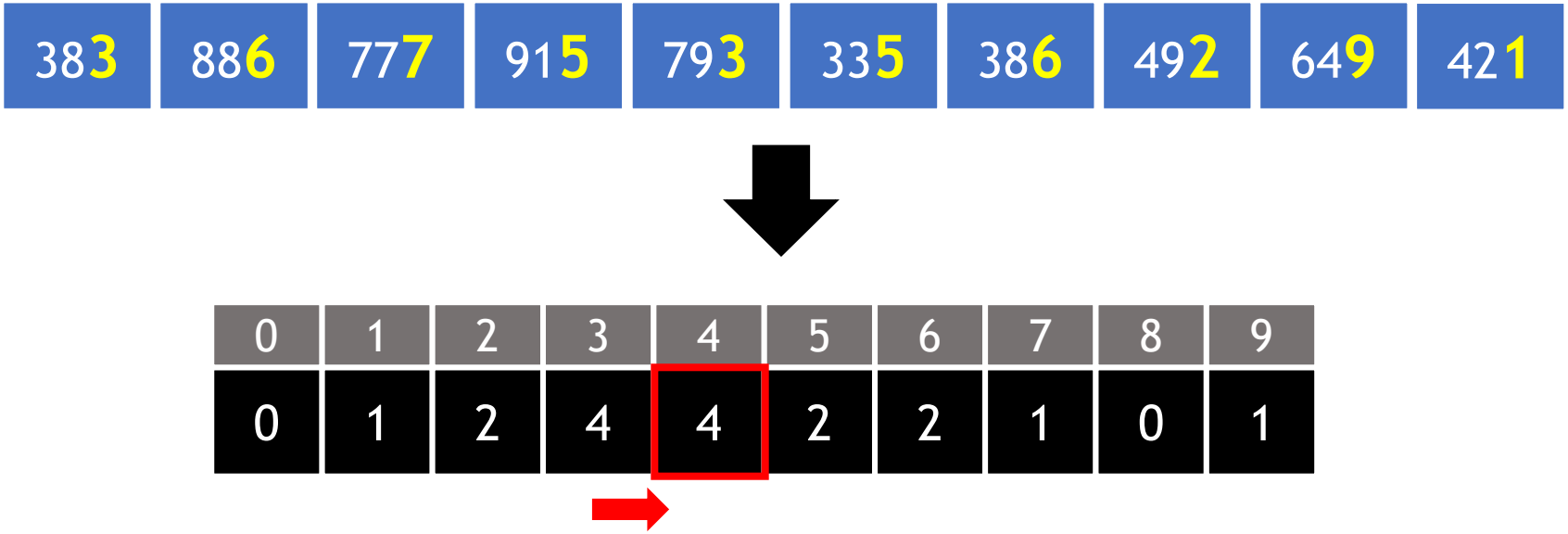
Radix sort


383
886
777
915
793
335
386
492
649
421
0 1 2 3 4 5 6 7 8 9
0 1 2 4 0 2 2 1 0 1

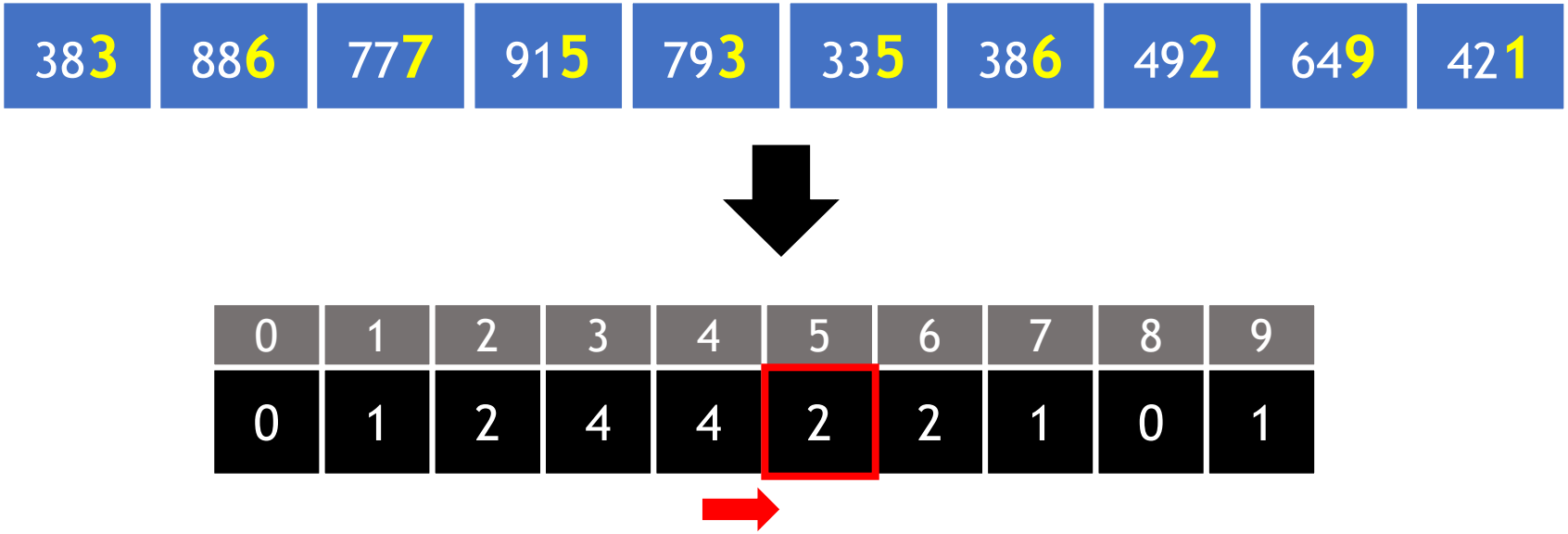
Radix sort


383
886
777
915
793
335
386
492
649
421
0 1 2 3 4 5 6 7 8 9
0 1 2 4 0 2 2 1 0 1

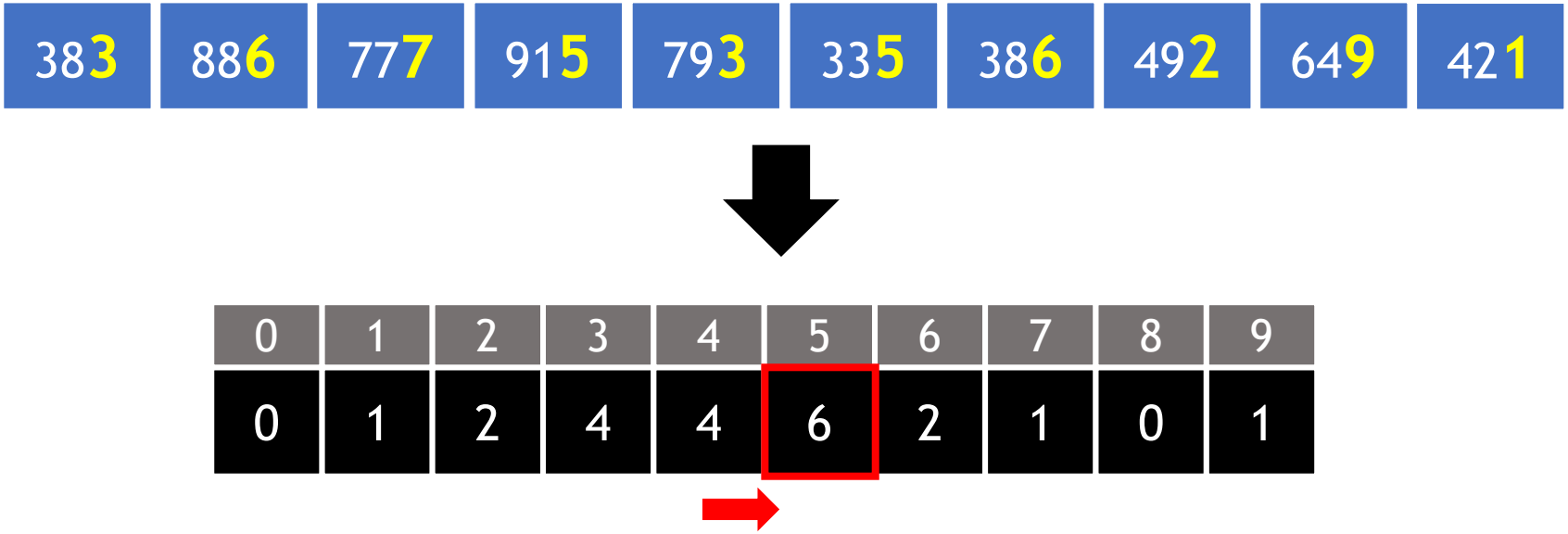
Radix sort


383
886
777
915
793
335
386
492
649
421
0 1 2 3 4 5 6 7 8 9
0 1 2 4 4 2 2 1 0 1

Radix sort


383
886
777
915
793
335
386
492
649
421
0 1 2 3 4 5 6 7 8 9
0 1 2 4 4 2 2 1 0 1

Radix sort


383
886
777
915
793
335
386
492
649
421
0 1 2 3 4 5 6 7 8 9
0 1 2 4 4 6 2 1 0 1

Radix sort

383	886	777	915	793	335	386	492	649	421

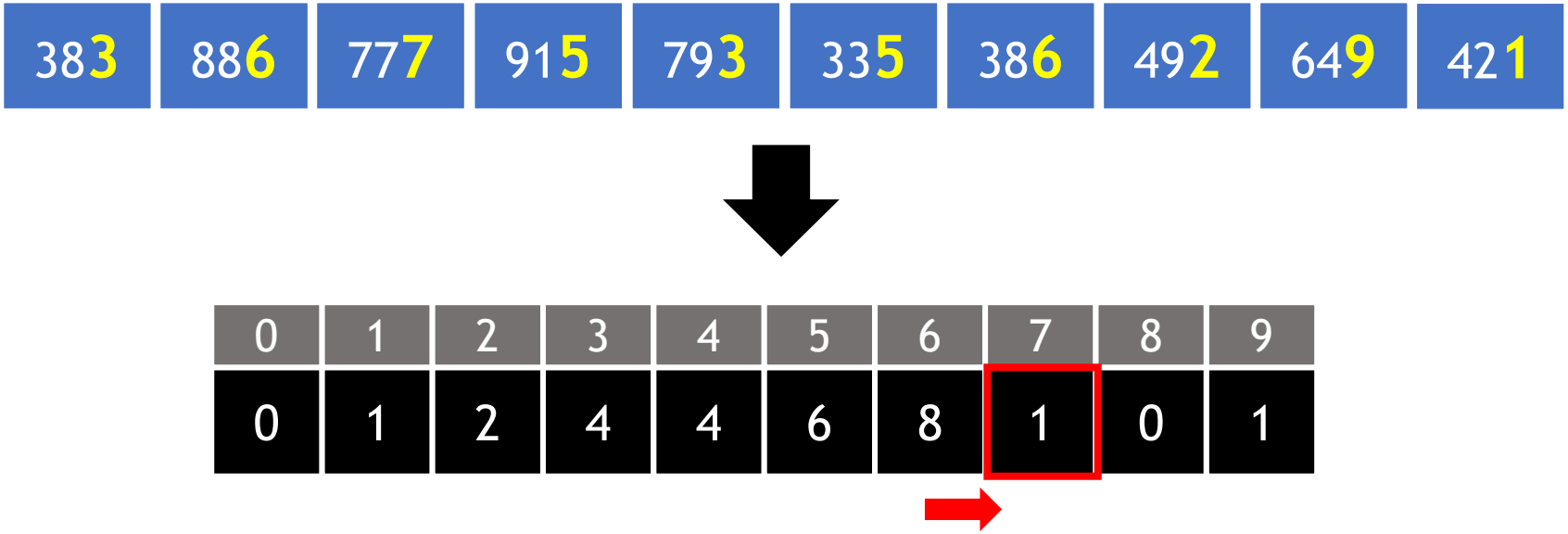
0	1	2	3	4	5	6	7	8	9
0	1	2	4	4	6	2	1	0	1

Radix sort

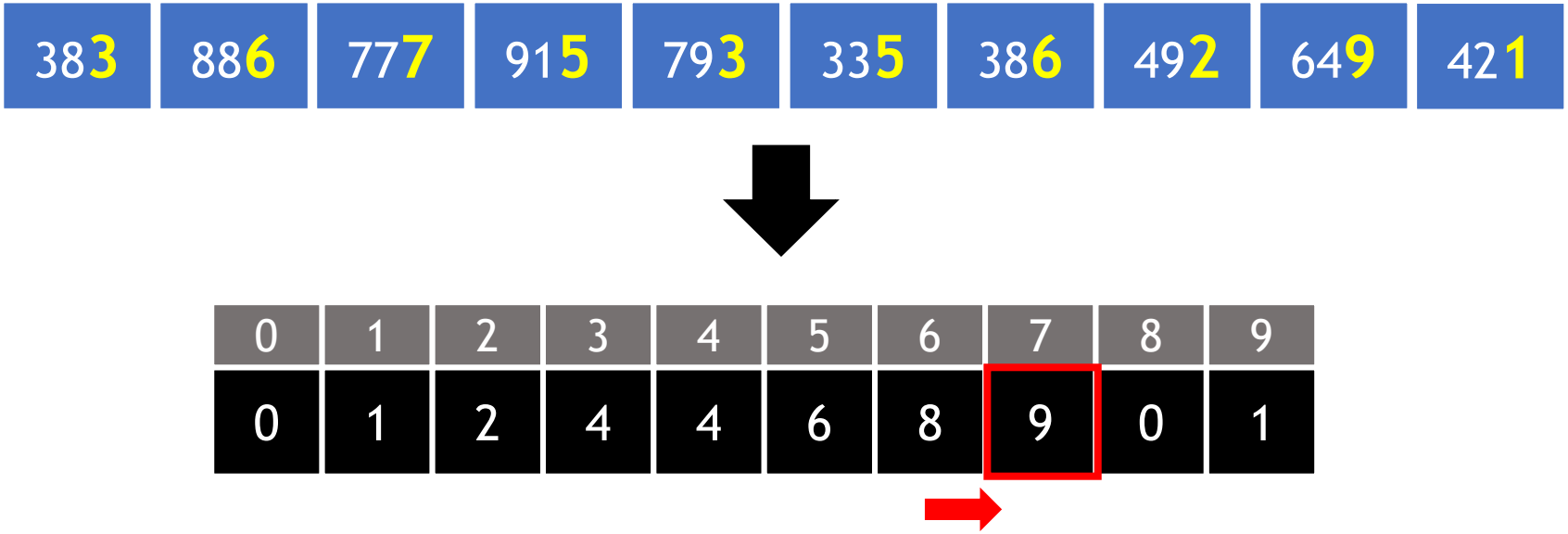
383	886	777	915	793	335	386	492	649	421

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

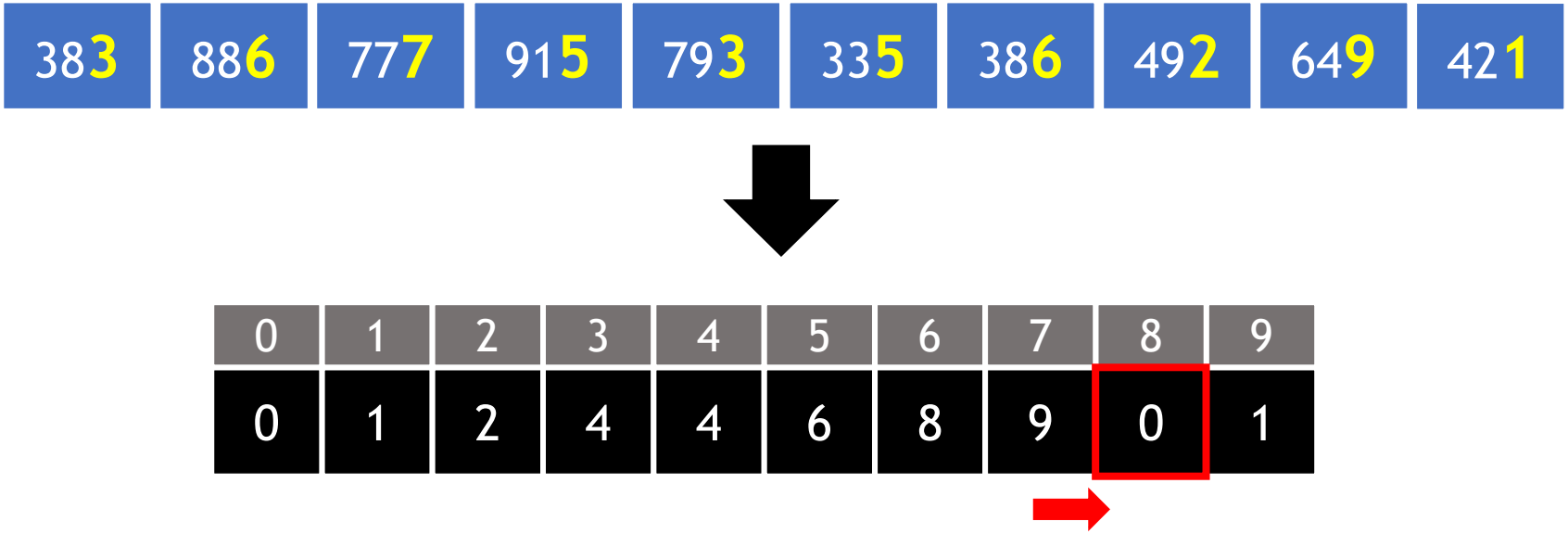
Radix sort


383
886
777
915
793
335
386
492
649
421
0 1 2 3 4 5 6 7 8 9
0 1 2 4 4 6 8 1 0 1

Radix sort


383
886
777
915
793
335
386
492
649
421
0 1 2 3 4 5 6 7 8 9
0 1 2 4 4 6 8 9 0 1

Radix sort


383
886
777
915
793
335
386
492
649
421
0 1 2 3 4 5 6 7 8 9
0 1 2 4 4 6 8 9 0 1

Radix sort

383	886	777	915	793	335	386	492	649	421

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

Radix sort

383	886	777	915	793	335	386	492	649	421

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

Radix sort

383	886	777	915	793	335	386	492	649	421

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

Radix sort

383	886	777	915	793	335	386	492	649	421

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

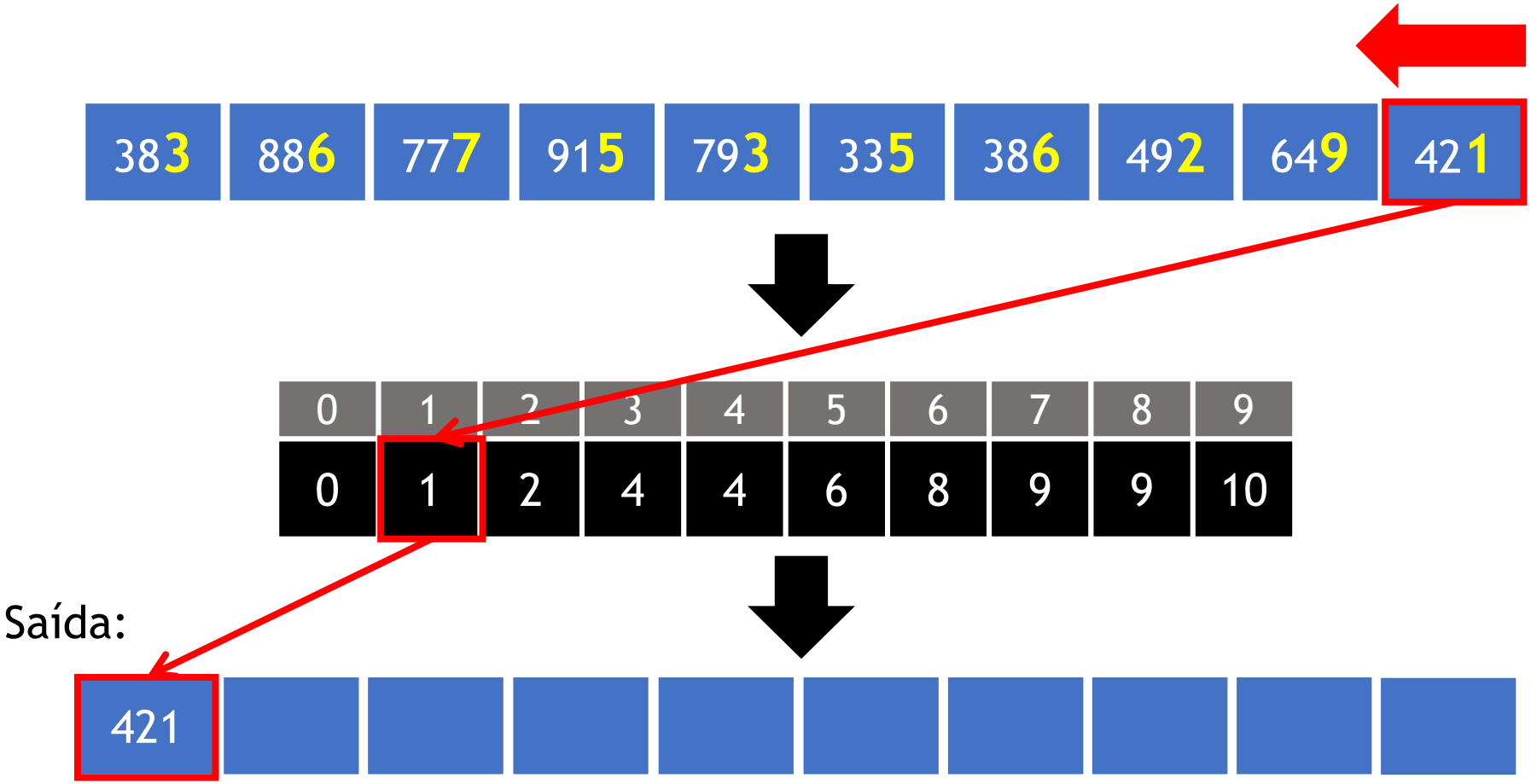
Radix sort

383	886	777	915	793	335	386	492	649	421

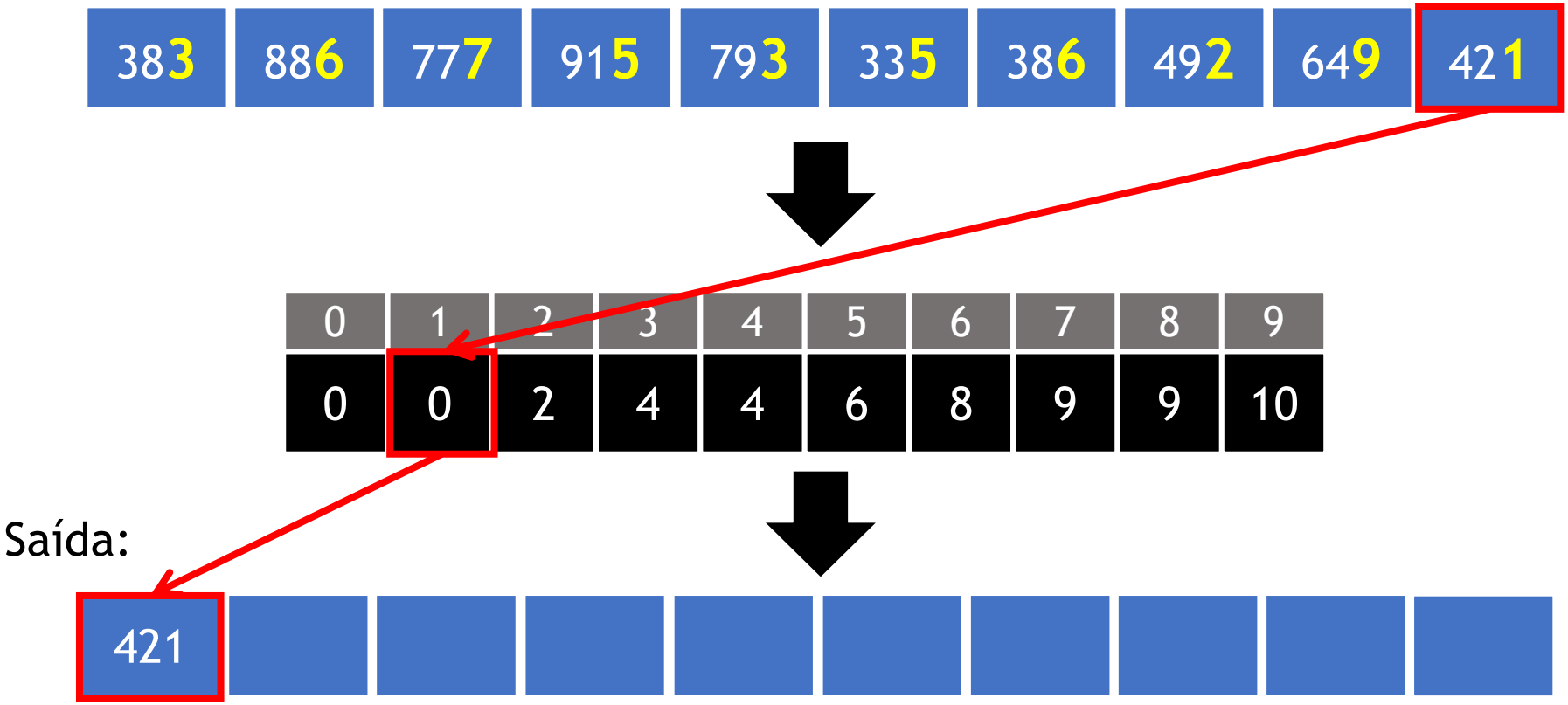
0	1	2	3	4	5	6	7	8	9
0	1	2	4	4	6	8	9	9	10

Saída:

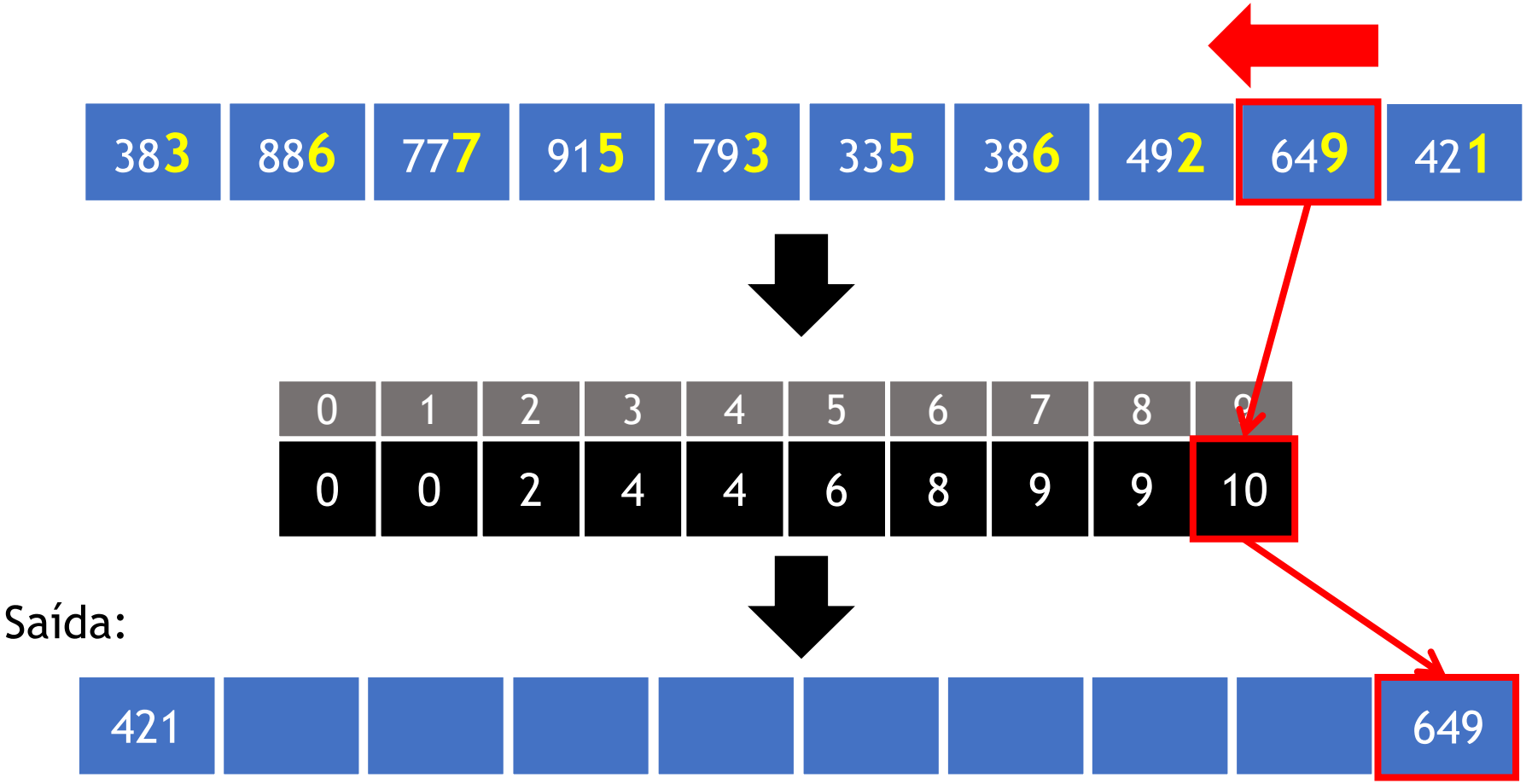
Radix sort


383
886
777
915
793
335
386
492
649
421
0 1 2 3 4 5 6 7 8 9
0 1 2 4 4 6 8 9 9 10
Saída:
421

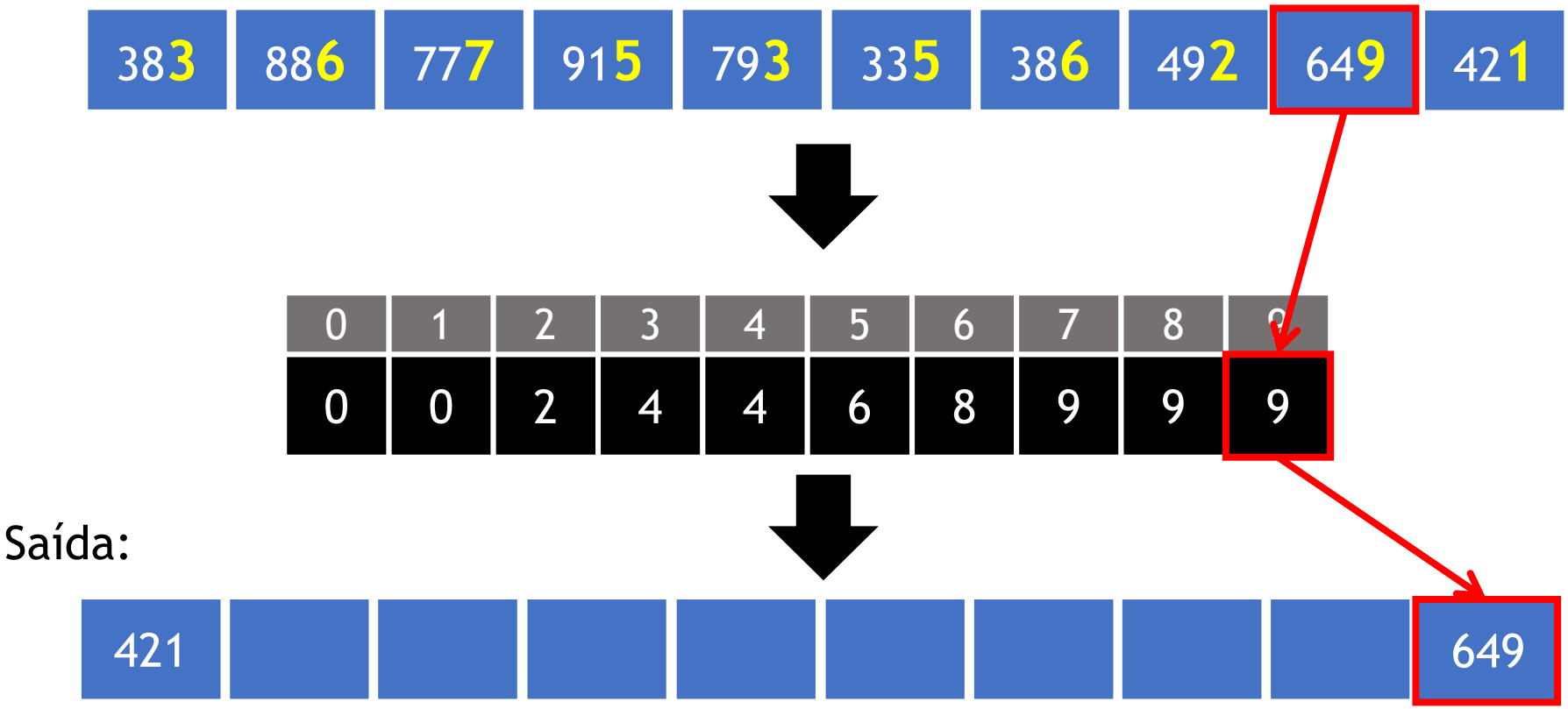
Radix sort


383
886
777
915
793
335
386
492
649
421
0 1 2 3 4 5 6 7 8 9
0 0 2 4 4 6 8 9 9 10
Saída:
421

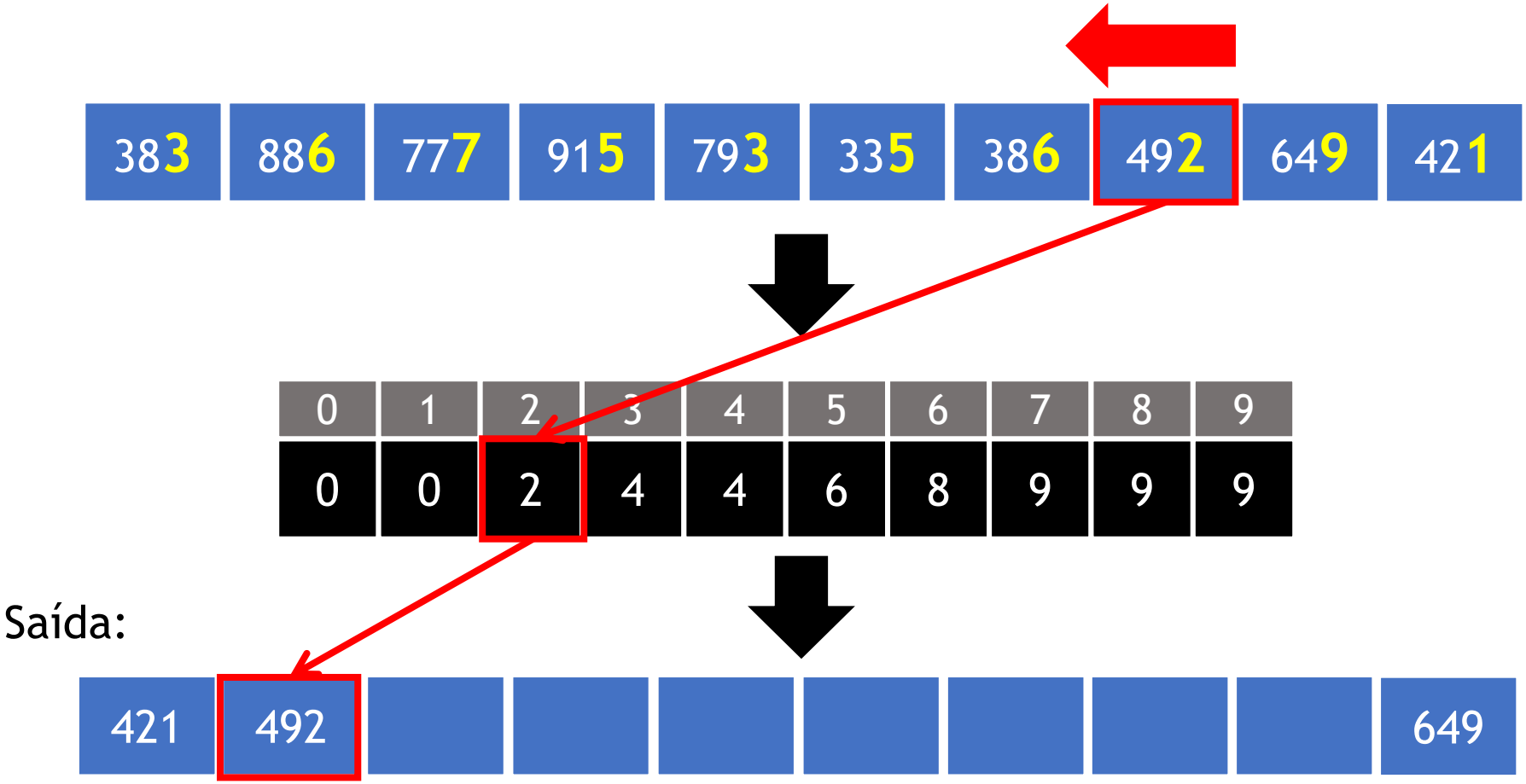
Radix sort


383
886
777
915
793
335
386
492
649
421
0 1 2 3 4 5 6 7 8 9
0 0 2 4 4 6 8 9 9 10
Saída:
421
649

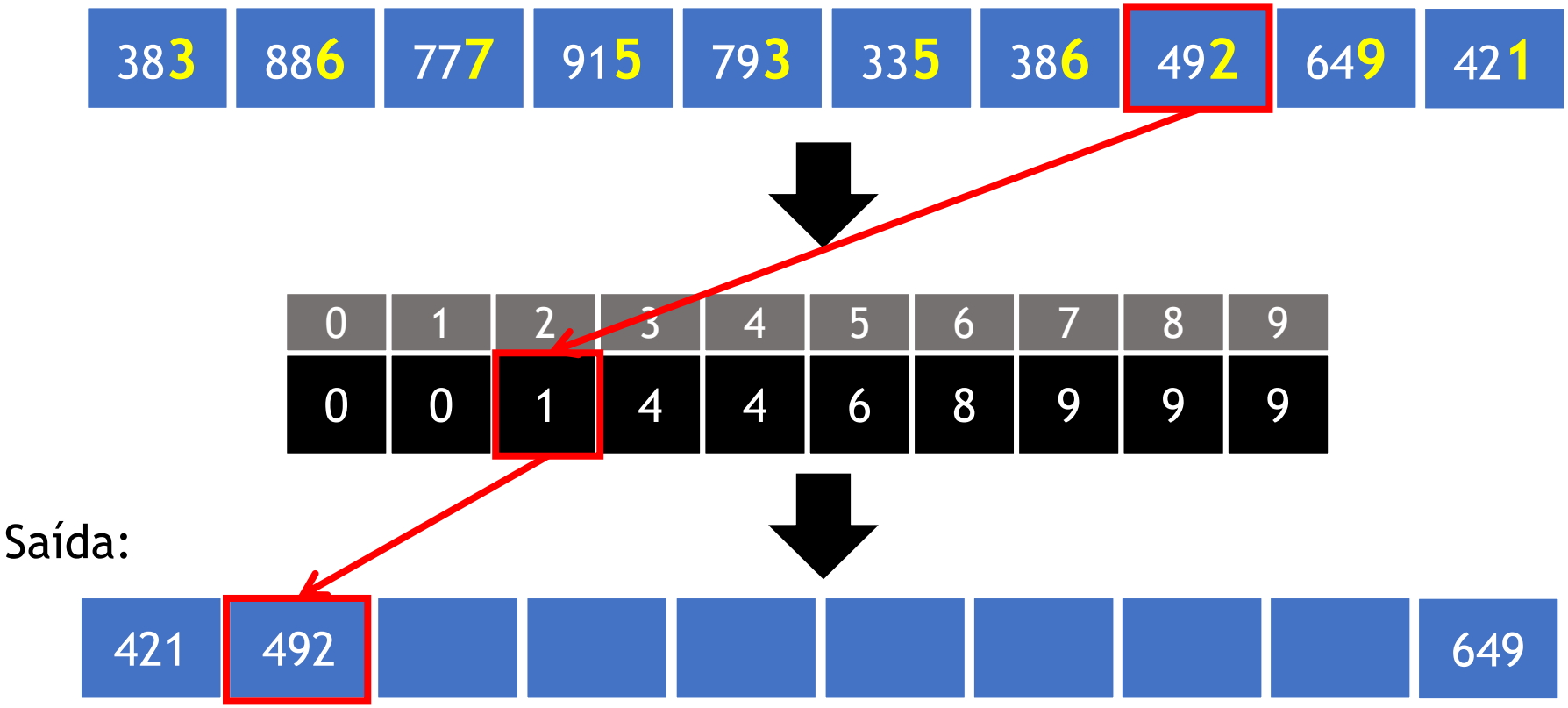
Radix sort


383
886
777
915
793
335
386
492
649
421
0 1 2 3 4 5 6 7 8 9
0 0 2 4 4 6 8 9 9 9
Saída:
421
649

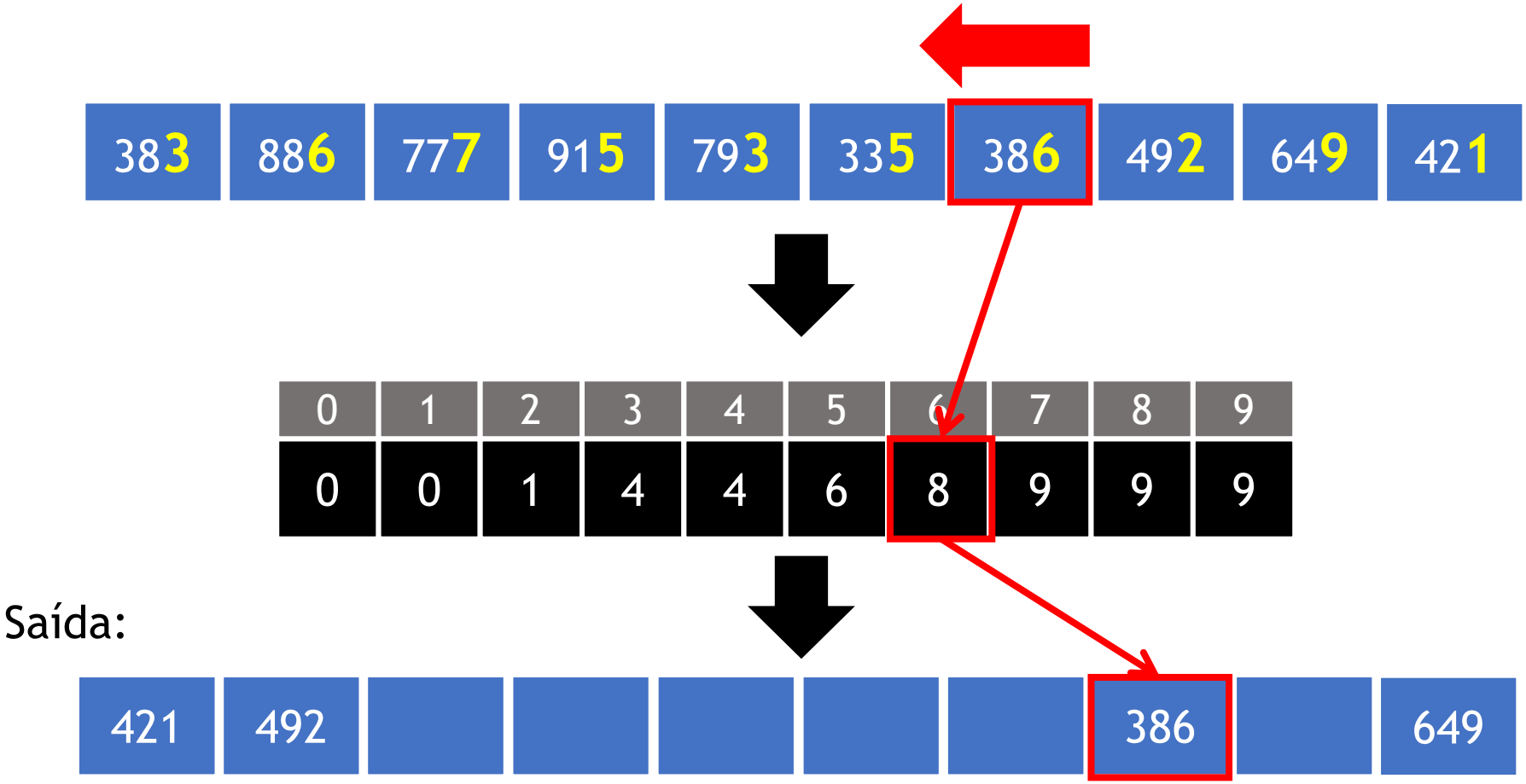
Radix sort


383
886
777
915
793
335
386
492
649
421
0 1 2 3 4 5 6 7 8 9
0 0 2 4 4 6 8 9 9 9
Saída:
421
492
649

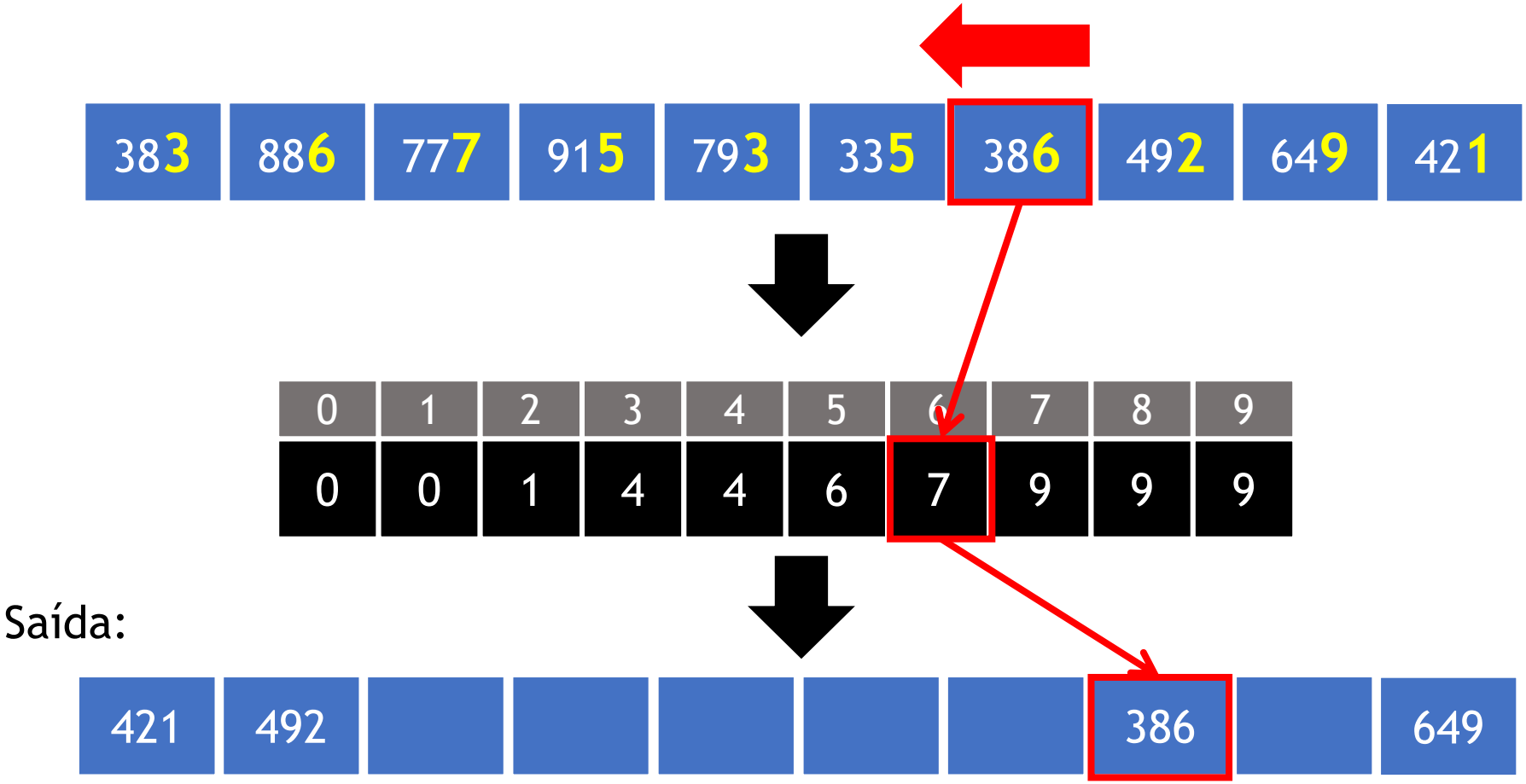
Radix sort


383
886
777
915
793
335
386
492
649
421
0 1 2 3 4 5 6 7 8 9
0 0 1 4 4 6 8 9 9 9
Saída:
421
492
649

Radix sort


383
886
777
915
793
335
386
492
649
421
0 1 2 3 4 5 6 7 8 9
0 0 1 4 4 6 8 9 9 9
Saída:
421
492
386
649

Radix sort


383
886
777
915
793
335
386
492
649
421
0 1 2 3 4 5 6 7 8 9
0 0 1 4 4 6 7 9 9 9
Saída:
421
492
386
649

Radix sort

383	886	777	915	793	335	386	492	649	421

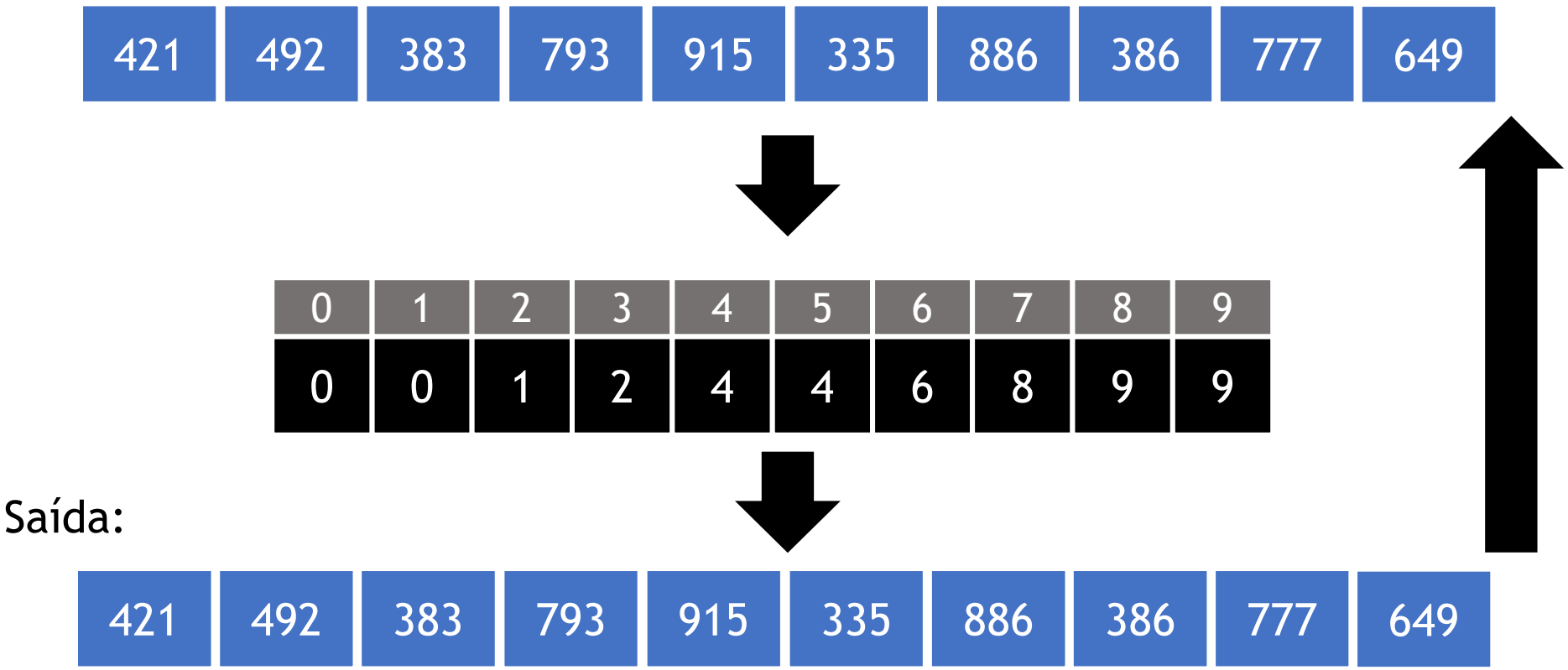
0	1	2	3	4	5	6	7	8	9
0	0	1	2	4	4	6	8	9	9

Saída:

421	492	383	793	915	335	886	386	777	649

Radix sort

O vetor de saída é copiado para o vetor de entrada.



Radix sort

Após ordenar pelo dígito menos significativo, a ordenação segue para o segundo dígito menos significativo.

421	492	383	793	915	335	886	386	777	649

Radix sort

Ordenação pelo segundo dígito menos significativo:

421	492	383	793	915	335	886	386	777	649

Contadores:

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

Radix sort

Ordenação pelo segundo dígito menos significativo:

421	492	383	793	915	335	886	386	777	649

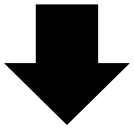
Contadores:

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

Radix sort

Ordenação pelo segundo dígito menos significativo:

421	492	383	793	915	335	886	386	777	649



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

Radix sort

Ordenação pelo segundo dígito menos significativo:

421	492	383	793	915	335	886	386	777	649

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

Saída:

915	421	335	649	777	383	886	386	492	793

Radix sort

915	421	335	649	777	383	886	386	492	793

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

Saída:

915	421	335	649	777	383	886	386	492	793

Radix sort

Após ordenar pelo segundo dígito menos significativo, a ordenação segue para o terceiro dígito menos significativo.

915	421	335	649	777	383	886	386	492	793

Radix sort

915	421	335	649	777	383	886	386	492	793

Contadores:

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

Radix sort

915	421	335	649	777	383	886	386	492	793

Contadores:

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

Radix sort

915	421	335	649	777	383	886	386	492	793

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

Radix sort

915	421	335	649	777	383	886	386	492	793

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

Saída:

335	383	386	421	492	649	777	793	886	915

Radix sort

A ordenação é encerrada após ordenar por todos os dígitos:

335	383	386	421	492	649	777	793	886	915

Radix sort

Na ordenação digital (radix sort), são ordenados os dígitos menos significativos até os mais significativos.

A ordenação seria realizada corretamente se primeiro ordenar os dígitos mais significativos?

Radix sort

- Implementação C

Chamada:

```
radixsort(vetor, n, n_digitos);
```

Implementação

```
void r_counting_sort(int *v, int n, int base, int potencia) {
    int c[base];
    int i;
    for (i = 0; i < base; i++)
        c[i] = 0;

    int d;
    for (i = 0; i < n; i++) {
        d = (v[i] / potencia) % base;
        c[d]++;
    }

    for (i = 1; i < base; i++)
        c[i] += c[i - 1];

    int saida[n];
    for (i = n - 1; i >= 0; i--) {
        d = (v[i] / potencia) % base;
        saida[c[d] - 1] = v[i];
        c[d]--;
    }

    for (i = 0; i < n; i++)
        v[i] = saida[i];
}
```

Implementação

```
void radixsort(int *v, int n, int n_digitos) {
     int i, potencia = 1;
     for (i = 0; i < n_digitos; i++) {
          r_counting_sort(v, n, 10, potencia);
          potencia *= 10;
     }
}
```

Custo do algoritmo

```
void radixsort(int *v, int n, int n_digitos) {
     int i, potencia = 1;
     for (i = 0; i < n_digitos; i++) {
          r_counting_sort(v, n, 10, potencia);
          potencia *= 10;
     }
}
```

Executa *n_digitos* vezes.

Neste algoritmo, k é a base - 1 (no caso, foi considerada base 10).

Counting sort:
Quando $k = O(n)$, o tempo de execução da ordenação por contagem é:

$$O(n)$$

Radix sort:
Quando n_digitos é constante e $base = O(n)$, o tempo de execução da ordenação digital é:

$$O(n)$$

Radix sort 256

- Utiliza base 256;
- Isso permite otimizar o cálculo para obter um dígito de um número:

```
d = (v[i] / potencia) % base;
```

```
s = 8 * expoente;
d = (v[i] >> s) & 0xff;
```

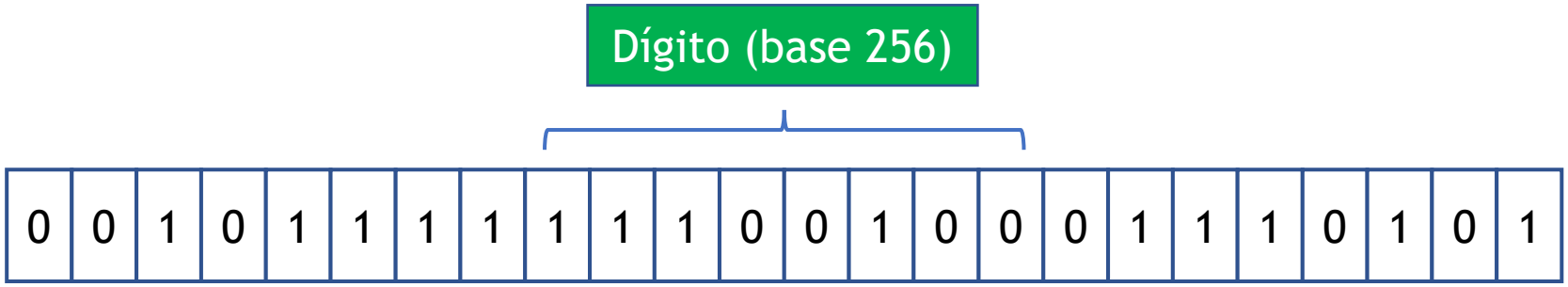
Obter dígito (base 256)

```
s = 8 * expoente;
d = (v[i] >> s) & 0xff;
```

Obter dígito (base 256)

```
s = 8 * expoente;
d = (v[i] >> s) & 0xff;
```

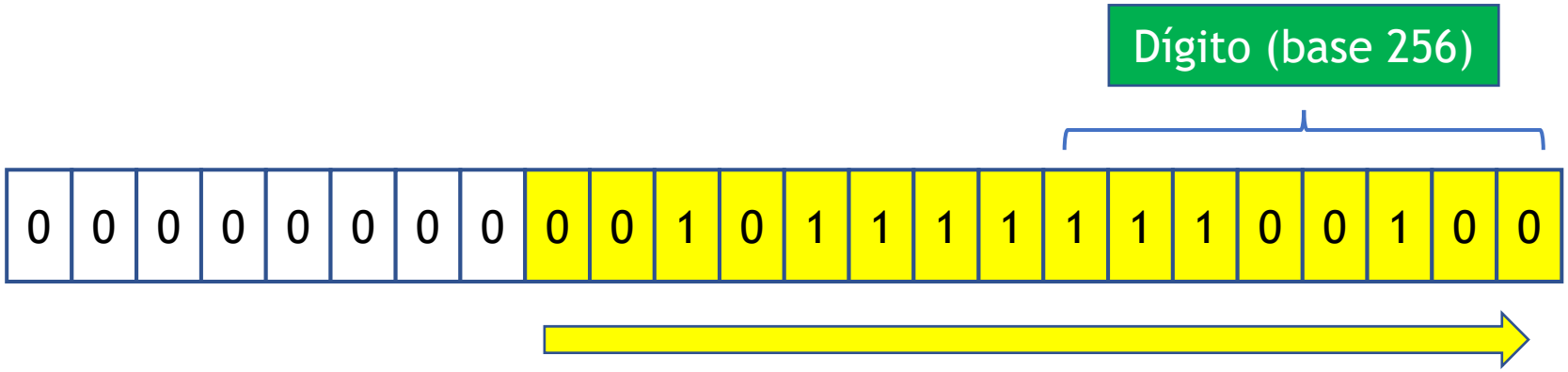
Exemplo: obter segundo dígito
expoente = 1



Obter dígito (base 256)

```
s = 8 * expoente;
d = (v[i] >> s) & 0xff;
```

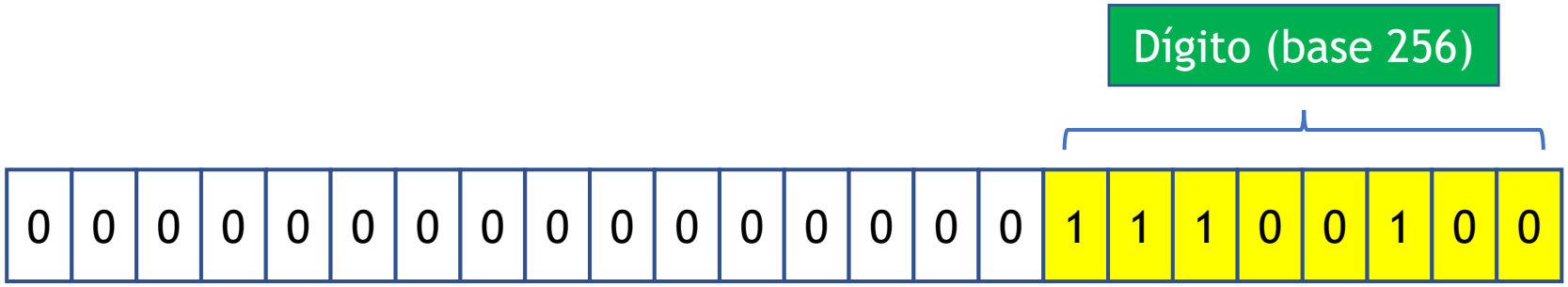
Exemplo: obter segundo dígito
expoente = 1



Obter dígito (base 256)

```
s = 8 * expoente;
d = (v[i] >> s) & 0xff;
```

Exemplo: obter segundo dígito
expoente = 1



Implementação

```
void r_counting_sort256(int *v, int n, int base, int s) {
     int c[base];
     int i;
     for (i = 0; i < base; i++)
          c[i] = 0;

     int d;
     for (i = 0; i < n; i++) {
          d = (v[i] >> s) & 0xff;
          c[d]++;
     }

     for (i = 1; i < base; i++)
          c[i] += c[i - 1];

     int saida[n];
     for (i = n - 1; i >= 0; i--) {
          d = (v[i] >> s) & 0xff;
          saida[c[d] - 1] = v[i];
          c[d]--;
     }

     for (i = 0; i < n; i++)
          v[i] = saida[i];
}
```

É possível calcular o resultado de 8 * potencia a cada iteração do Radix sort.

Implementação

```
void radixsort256(int *v, int n, int n_digitos) {
    int i, s = 0;
    for (i = 0; i < n_digitos; i++) {
        r_counting_sort256(v, n, 256, s);
        s += 8;
    }
}
```

Referências

- Nivio Ziviani. Projeto de Algoritmos: com implementações em Pascal e C. Cengage Learning, 2015.
- Jayme L. Szwarcfiter, Lilian Markenzon. Estruturas de Dados e Seus Algoritmos. 3ª edição. LTC, 2012.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Algoritmos: Teoria e Prática. Elsevier, 2012.
- Robert Sedgewick, Kevin Wayne. Algorithms 4ª edição. Pearson, 2011.

Bibliografia básica

- CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. Algoritmos: teoria e prática. 2. ed. Rio de Janeiro, RJ: Campus, 2002.
- KNUTH, D. E. The art of computer programming. Upper Saddle River, USA: Addison-Wesley, 2005.
- SEDGEWICK, R. Algorithms in C: parts 1-4 (fundamental algorithms, data structures, sorting, searching). Reading, USA: Addison-Wesley, 1998.

Bibliografia complementar

- DROZDEK, A. Estrutura de dados e algoritmos em C++. São Paulo, SP: Thomson Learning, 2002.
- RODRIGUES, P.; PEREIRA, P.; SOUSA, M. Programação em C++: conceitos básicos e algoritmos. Lisboa, PRT: FCA de Informática, 2000.
- SZWARCFITER, J. L.; MARKENZON, L. Estruturas de dados e seus algoritmos. 3. ed. Rio de Janeiro, RJ: LTC, 2010.
- TENENBAUM, A. M.; LANGSAM Y.; AUGENSTEIN M. J. Estruturas de dados usando C. São Paulo, SP: Pearson Makron Books, 1995.
- ZIVIANI, N. Projeto de algoritmos com implementação em Java e C++. São Paulo, SP: Thomson Learning, 2007.