

## Ordenação por contagem (countingSort)

Este método é especializado na ordenação de vetores de inteiros pequenos,

- e não é baseado na comparação entre elementos do vetor.
- Por isso, vence o limitante inferior  $\Omega(n \lg n)$  para ordenação.

$\Omega(n \lg n)$

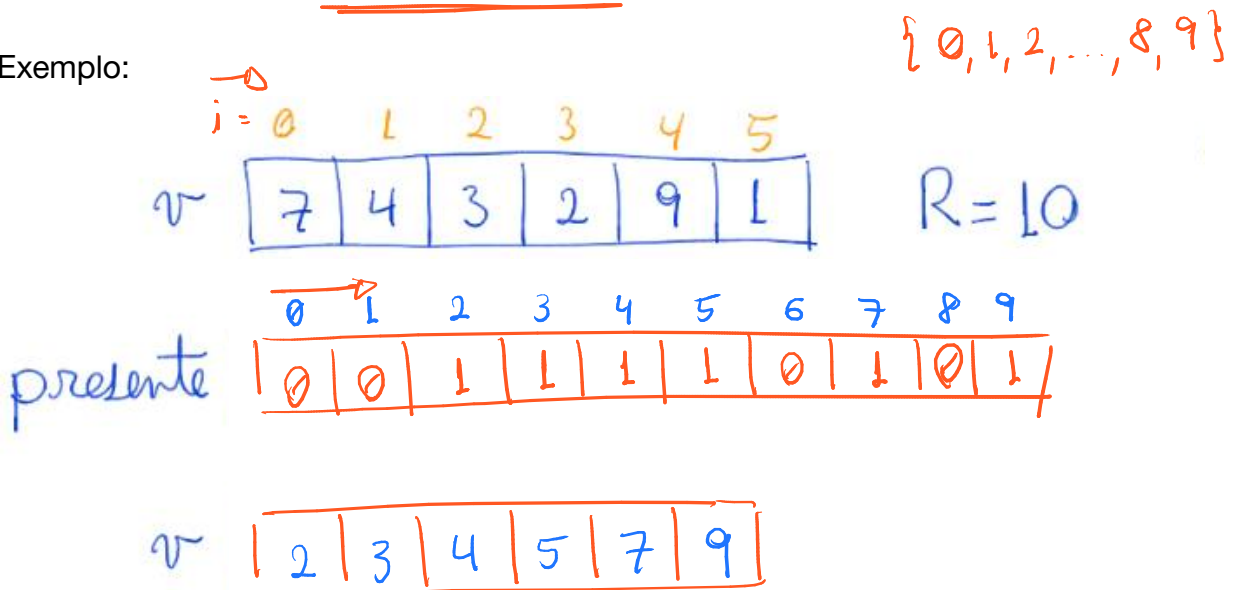
Para desenvolver a ideia do algoritmo vamos supor que no vetor  $v$  de tamanho  $n$

- só existem inteiros entre 0 e  $R - 1$ .

Ideia 1: para simplificar, primeiro supomos que não existem elementos repetidos.

- Neste caso, podemos alocar um vetor auxiliar presente
  - e inicializar presente com 0.
- Percorrer  $v$  com um índice  $i$ , marcando presente[v[i]] = 1
- Percorrer presente da esquerda para a direita com um índice valor,
  - colocando valor na próxima posição livre de  $v$ 
    - se presente[valor] = 1.

Exemplo:



Código:

// ordena um vetor  $v[0 .. n-1]$  de inteiros em  $[0, R)$   $\in \{0, 1, 2, \dots, R-1\}$   
 // desde que não existam elementos repetidos

```
void countingSortErrado1(int v[], int n, int R) {
```

```
    int *presente, valor, i;
```

```
    presente = malloc(R * sizeof(int));
```

```
    for (valor = 0; valor < R; valor++)
```

```
        presente[valor] = 0;
```

```
    for (i = 0; i < n; i++)
```

```
        presente[v[i]] = 1;
```

```
    i = 0;
```

```
    for (valor = 0; valor < R; valor++)
```

```
        if (presente[valor] == 1) v[i++] = valor;
```

```
    free(presente);
```

```
}
```

percorrer o vetor presente

Agora vamos considerar que podem existir elementos repetidos.

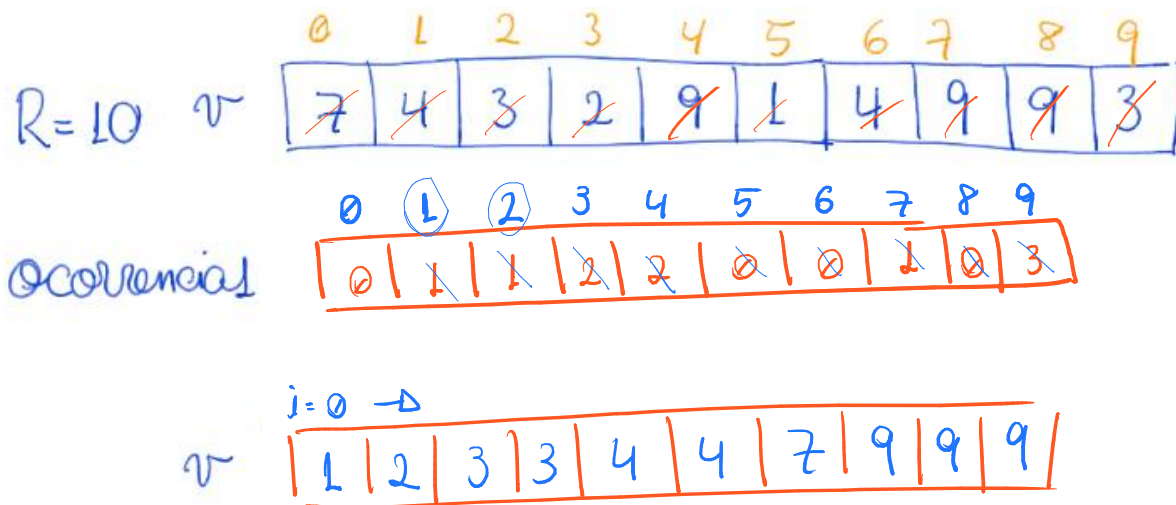
- Para tanto, vamos usar o número de ocorrências de um elemento.

Nesta nova abordagem, vamos alocar um vetor auxiliar *ocorrencias*.

$e \{0, 1, 2, \dots, R-1\}$

- Inicializar *ocorrencias* com 0.
- Percorrer *v* com um índice *i*,
  - fazendo  $ocorrencias[v[i]] += 1$
- Assim, para cada *valor* em  $[0, R)$ , ao final do laço
  - $ocorrencias[valor]$  possuirá o número de ocorrências de *valor*.
- Percorrer *ocorrencias* da esquerda para a direita com um índice *valor*,
  - colocando  $ocorrencias[valor]$  cópias de *valor*
    - nas próximas posições livres de *v*.

Exemplo:



Código:

```
// ordena um vetor v[0 .. n-1] de inteiros em [0, R)
// copia ao invés de rearranjar
void countingSortErrado2(int v[], int n, int R) {
    int *ocorrencias, valor, i, repet;
    ocorrencias = malloc(R * sizeof(int));
    for (valor = 0; valor < R; valor++)
        ocorrencias[valor] = 0;
    for (i = 0; i < n; ++i)
        ocorrencias[v[i]] += 1;
    i = 0;
    for (valor = 0; valor < R; valor++)
        for (repet = 0; repet < ocorrencias[valor]; repet++)
            v[i++] = valor;
    free(ocorrencias);
}
```

Apesar de aparentarem ser corretos, tanto esse último algoritmo quanto o primeiro,

- apresentam um erro fundamental. Qual?

Apesar de aparentarem ser corretos, esses últimos algoritmos tem um erro. Qual?

- Eles não estão ordenando os elementos originais,
  - mas apenas criando cópias das chaves destes.
- Esse é um problema grave quando as chaves sendo ordenadas
  - são parte de elementos que possuem outras informações,
    - como registros ou ponteiros, por exemplo.
- Ou ainda, quando são partes de uma chave maior, como veremos
  - na aplicação do **countingSort** como subrotina do LSD **radixSort**.

Para resolver esse problema, temos que percorrer  $v$  movendo os elementos

- para suas respectivas posições ordenadas. Para tanto, é preciso saber
  - o # de elementos **ocorr\_pred[val]** que ocorre antes de cada chave **val**.
- Assim, vamos calcular o número de ocorrências dos predecessores
  - usando o número de ocorrências **ocorrs[val]** de cada chave **val**.

- O número de ocorrências dos predecessores de  $val$  é

$$- \text{ocorr\_pred}[val] = \text{ocorrs}[0] + \text{ocorrs}[1] + \dots + \text{ocorrs}[val-1]$$

- Podemos usar uma definição recursiva:

- se  $val > 0$  então  $\text{ocorr\_pred}[val] = \text{ocorr\_pred}[val-1] + \text{ocorrs}[val-1]$
- se  $val = 0$  temos  $\text{ocorr\_pred}[0] = 0$

- Essa definição deriva da seguinte observação

$$\text{ocorr\_pred}[val] = (\text{ocorrs}[0] + \dots + \text{ocorrs}[val-2]) + \text{ocorrs}[val-1]$$

$$\text{ocorr\_pred}[val-1] = \text{ocorrs}[0] + \dots + \text{ocorrs}[val-2]$$

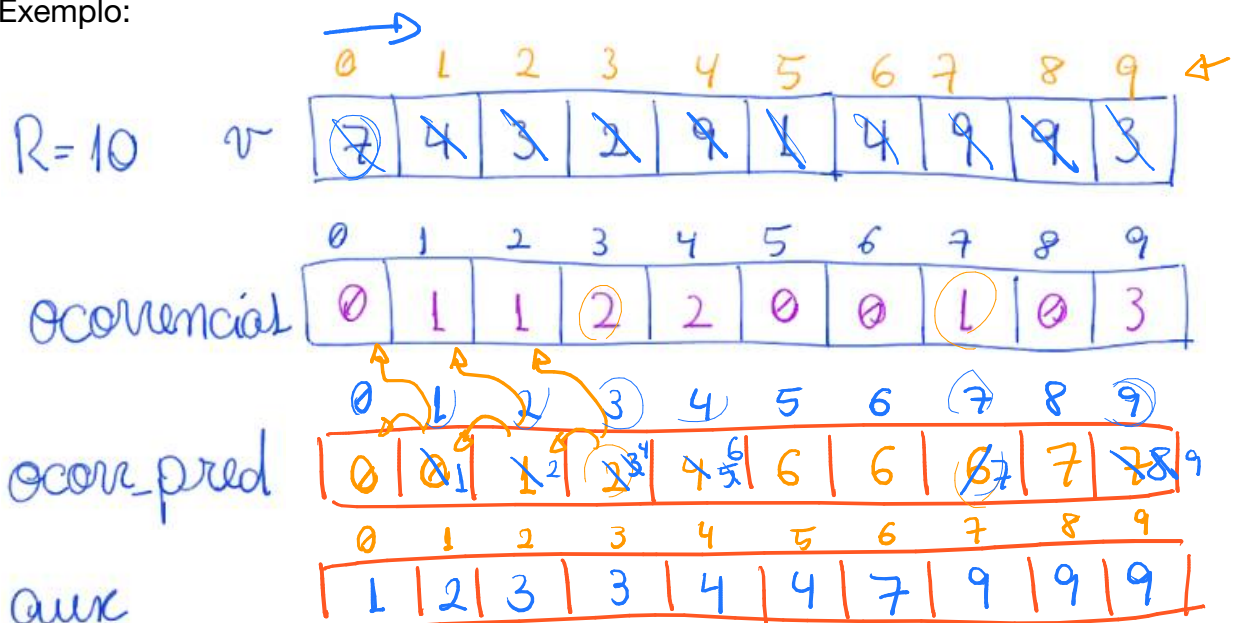
- Portanto,  $\text{ocorr\_pred}[val] = \text{ocorr\_pred}[val-1] + \text{ocorrs}[val-1]$

- Note que, **ocorre\_pred[val]** é a posição em que os elementos de chave  $val$ 
  - devem começar a ser colocados no vetor ordenado.

Também precisaremos de um vetor auxiliar **aux[0 .. n - 1]**

- para copiar um elemento de  $v$  para uma posição diferente em  $aux$ 
  - sem corromper elementos ainda não copiados de  $v$ .

Exemplo:



Código:

```
// ordena um vetor v[0 .. n-1] de inteiros em [0, R)
void countingSort(int v[], int n, int R) {
    int val, i;
    int *ocorrencias, *ocorr_pred, *aux;
    ocorrencias = malloc(R * sizeof(int));
    occurr_pred = malloc(R * sizeof(int));
    aux = malloc(n * sizeof(int));

    for (val = 0; val < R; val++) ocorrencias[val] = 0;
    for (i = 0; i < n; i++) ocorrencias[v[i]] += 1;

    occurr_pred[0] = 0;
    for (val = 1; val < R; val++)
        occurr_pred[val] = occurr_pred[val-1] + ocorrencias[val-1];
    // Elementos iguais a val começam no índice occurr_pred[val]
    for (i = 0; i < n; i++) {
        val = v[i];
        aux[occurr_pred[val]] = v[i];
        occurr_pred[val]++; // atualiza o número de predecessores
    }
    // aux[0 .. n-1] está em ordem crescente
    for (i = 0; i < n; ++i) v[i] = aux[i];

    free(ocorrencias);
    free(occurr_pred);
    free(aux);
}
```

Esta última versão do countingSort está correta.

- No entanto, ela desperdiça memória
  - por alocar espaço para *ocorrencias* e para *occurr\_pred*.
- Observe que, só usamos *ocorrencias* para calcular os valores de *occurr\_pred*.
  - Será que conseguimos economizar a memória de um desses vetores?
- Uma tentativa envolve alocar um único vetor *occurr\_pred*,
  - usá-lo, inicialmente, para armazenar
    - o número de ocorrências das chaves,
  - e reaproveitá-lo para armazenar
    - o número de ocorrências dos predecessores.
- Isso é possível,
  - mas exigirá algumas mudanças sutis.

Vamos usar um único vetor `ocorr_pred` para, inicialmente,

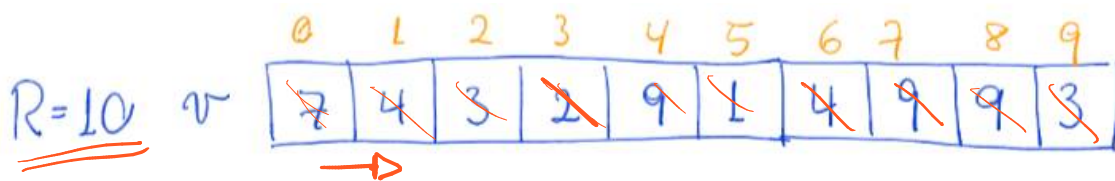
- armazenar o número de ocorrências das chaves,
- e, depois, reaproveitá-lo para armazenar
  - o número de ocorrências dos predecessores.

$O(R)$

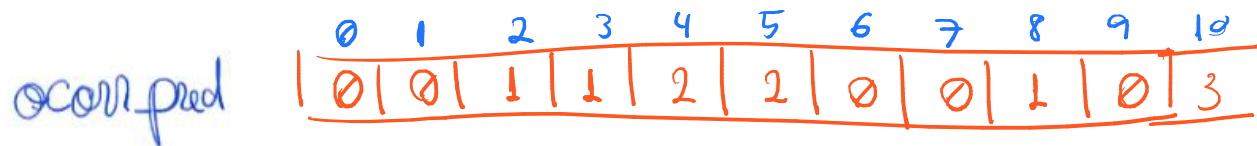
Isso exigirá algumas mudanças sutis. Em particular, vamos

- armazenar o número de ocorrências da chave `val` em `ocorr_pred[val + 1]`.
- Com isso, a princípio a posição `ocorr_pred[val]`
  - terá o número de ocorrências de `val - 1`.
- Lembrando que, para `val > 0`,
  - $ocorr\_pred[val] = occorrs[val - 1] + occorrs[val - 1]$ .
- Assim, para que `ocorr_pred[val]` passe a armazenar
  - o número de ocorrências dos predecessores
    - basta somar a ele `ocorr_pred[val - 1]`,
  - já que o número de ocorrências de `val - 1` já está lá.

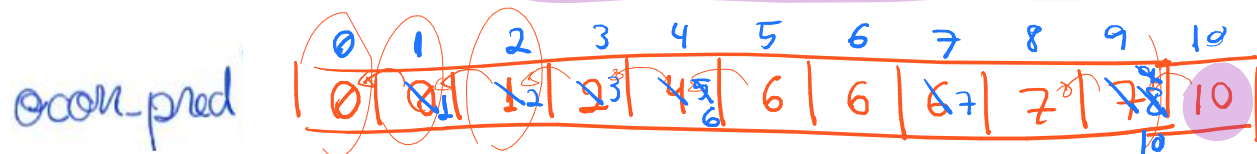
Exemplo:



`ocorr_pred[valor]` é o # de ocorrências de `valor - 1`

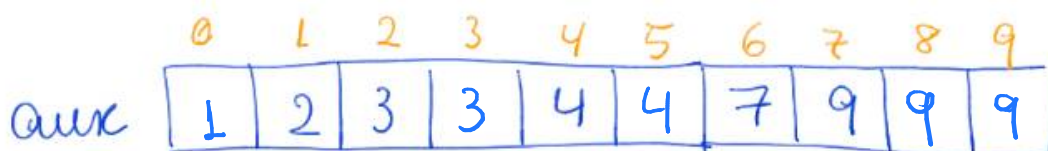


Para (valor = 1 até  $R$ ) faça { `ocorr_pred[valor] += occorrs[valor - 1]` }

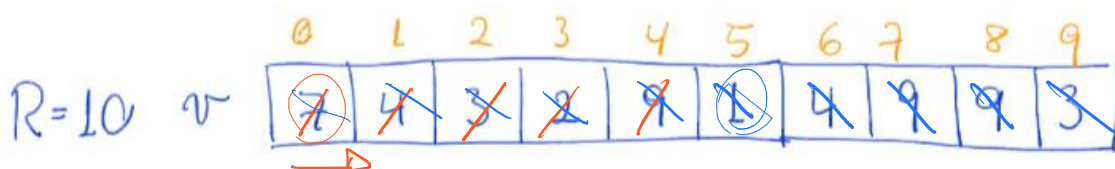


`ocorr_pred[valor]` passa a ser o # de ocorrências dos predecessores de `valor`

aux após percorrer  $v$  de 0 até 4



Vetor  $v$  original, para nos ajudar a mover os elementos para aux





Código:

```
// ordena um vetor v[0 .. n-1] de inteiros em [0, R) → {0, ..., R-1}
void countingSort2(int v[], int n, int R) {
    int valor, i;
    int *ocorr_pred, *aux;
    ocorr_pred = malloc((R + 1) * sizeof(int));
    aux = malloc(n * sizeof(int));
    for (valor = 0; valor <= R; valor++) — inic.
        ocorr_pred[valor] = 0;
    for (i = 0; i < n; i++) { — percorra v
        valor = v[i];
        ocorr_pred[valor + 1] += 1;
    }
    // ocorr_pred[valor] é o núm. de ocorrências de valor - 1
    for (valor = 1; valor <= R; valor++)
        [ocorr_pred[valor] += ocorr_pred[valor - 1]];
    // ocorr_pred[valor] é o núm. de ocorrencias dos predecessores de
    // valor. Logo, a cadeia de elementos iguais a valor deve
    // começar no índice ocorr_pred[valor] no vetor ordenado.
    for (i = 0; i < n; i++) { — percorra v colocando em ordem em aux
        valor = v[i];
        aux[ocorr_pred[valor]] = v[i];
        ocorr_pred[valor]++; // atualiza o número de predecessores
    }
    // aux[0 .. n-1] está em ordem crescente
    for (i = 0; i < n; ++i) v[i] = aux[i];
    free(ocorr_pred);
    free(aux);
}
```

Curiosidade: note que, `ocorr_pred` foi alocado com uma posição a mais,

- mas o único motivo para tanto é evitar que, no segundo laço
  - seja acessada uma posição de memória inválida.
- Isso porque, se  $valor = v[i] = R - 1$  temos que
  - $ocorr\_pred[valor + 1] = occorr\_pred[R]$  recebe uma atribuição.

Eficiência de tempo: countingSort leva tempo da ordem de  $n + R$ , i.e.,  $O(n + R)$ .

- Isso porque cada laço itera por  $R$  ou  $n$  vezes,
  - e não temos laços aninhados.
- Se  $R$  é pequeno (da ordem de  $n$  no máximo),
  - isso é melhor que a eficiência  $O(n \log n)$  de algoritmos como
    - mergeSort, quickSort e heapSort.
- Por isso, countingSort é o método preferido para ordenar
  - vetores cujas chaves são inteiros pequenos.

Eficiência de espaço:  $O(n + R)$ , já que precisamos de

- um vetor *ocorr\_pred* de tamanho proporcional a  $R$ 
  - e um vetor aux de tamanho proporcional a  $n$ .

Estabilidade:

- countingSort é estável.

Curiosidade: A estabilidade do countingSort é a propriedade chave

- que permite aplicá-lo ao **LSD radix sort**, que veremos na próxima aula.

Quiz1: A ordem de alguns laços do algoritmo é decisiva

- para manter a estabilidade, enquanto outros laços
  - poderiam ser invertidos ou seguir uma ordem arbitrária.
- Identifique a relevância da ordem de cada laço do algoritmo.

Quiz2: Como generalizar o countingSort para lidar com inteiros

- que estão em um intervalo pequeno que não começa em 0?
- Na resposta, suponha que os inteiros estão no intervalo **[lim\_inf, lim\_sup)**.