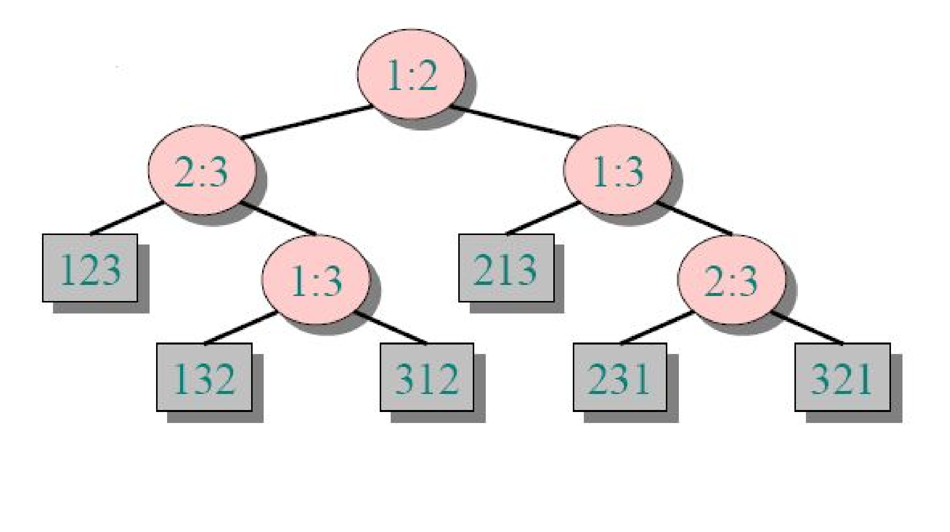
Tópico 5
Algoritmos de Ordenação

Parte II - métodos de ordenação: counting sort, radix sort e bucket sort.

Árvore de Decisão

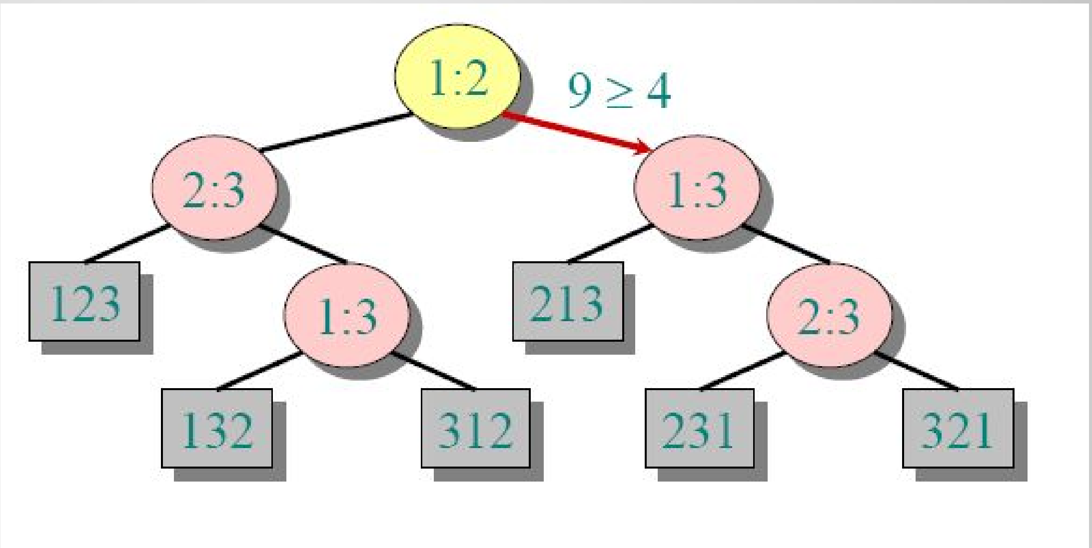
- Todos os algoritmos descritos anteriormente utilizam comparações para determinar a ordem relativa entre as entradas.
- Considere a sequência de entrada $\langle a_1, a_2, \ldots, a_n \rangle$



- Lado esquerdo da árvore: $a_i \leq a_j$.
- Lado direito da árvore: $a_i \geq a_j$.

Árvore de Decisão

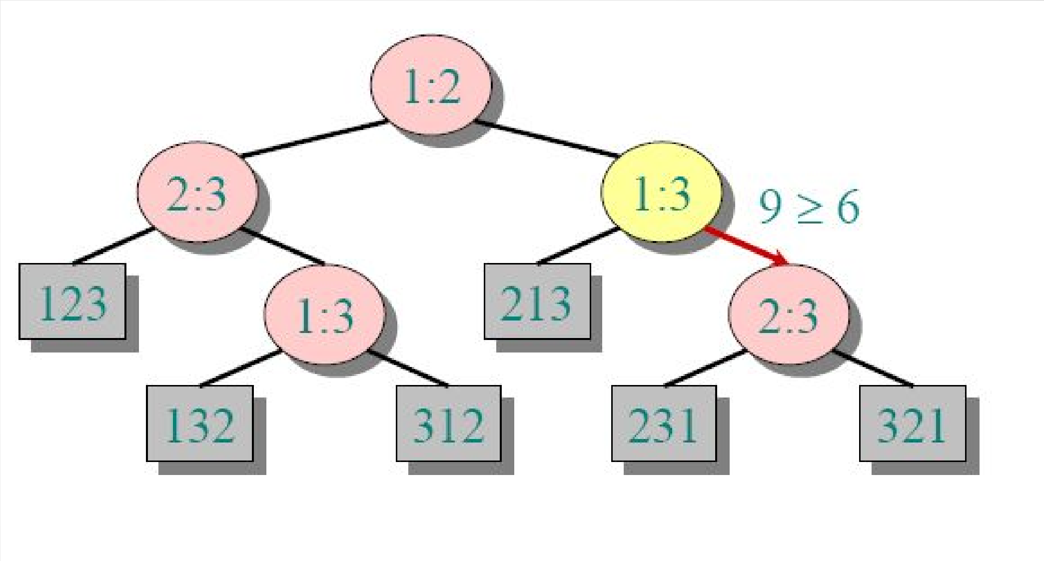
$\langle a_1, a_2, a_3 \rangle$
$= \langle 9, 4, 6 \rangle$



- Lado esquerdo da árvore: $a_i \leq a_j$.
- Lado direito da árvore: $a_i \geq a_j$.

Árvore de Decisão

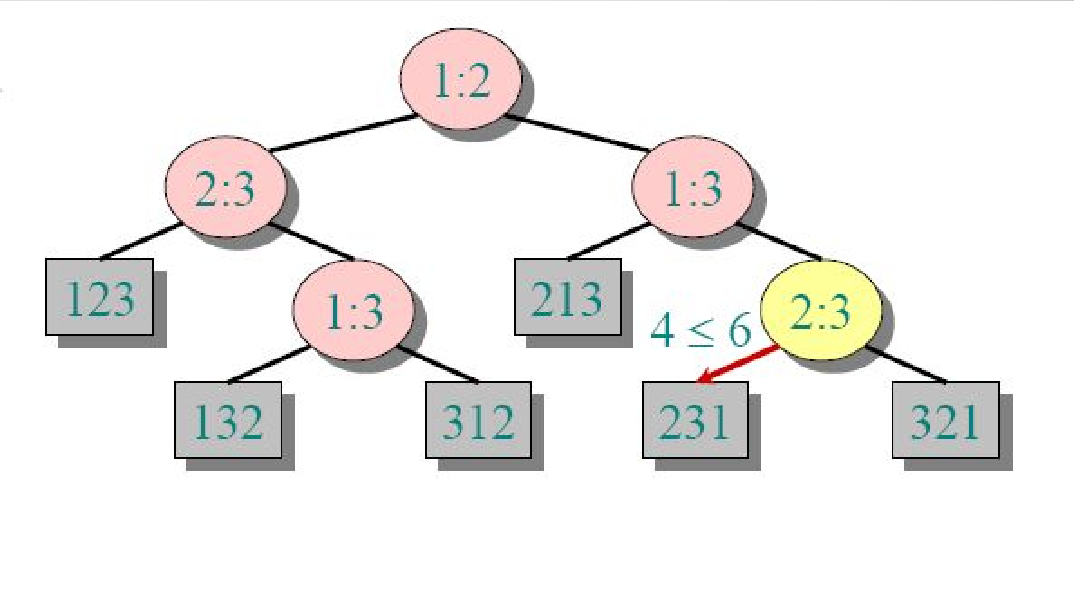
$\langle a_1, a_2, a_3 \rangle$
$= \langle 9, 4, 6 \rangle$



- Lado esquerdo da árvore: $a_i \leq a_j$.
- Lado direito da árvore: $a_i \geq a_j$.

Árvore de Decisão

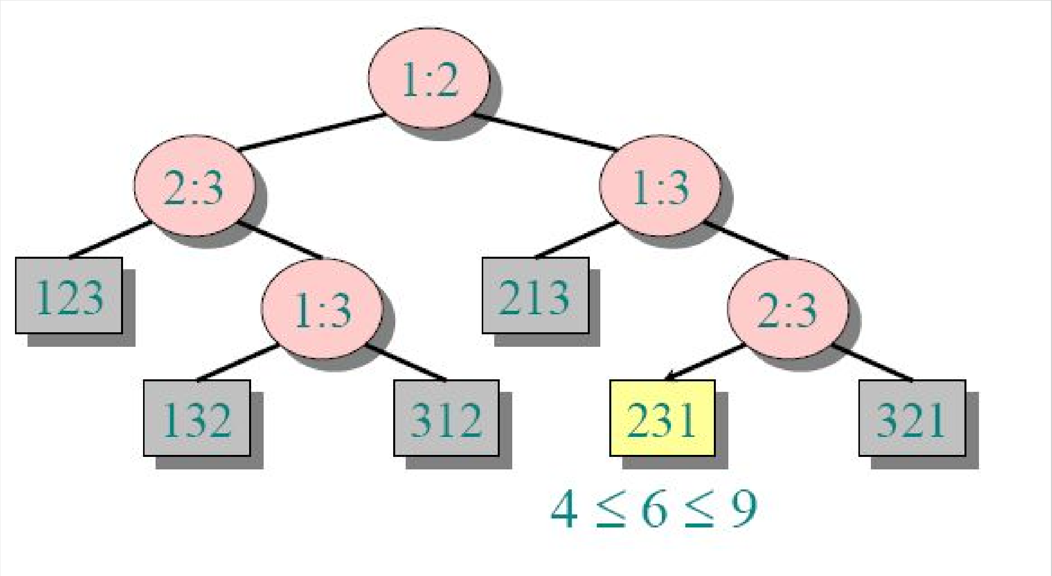
$\langle a_1, a_2, a_3 \rangle$
$= \langle 9, 4, 6 \rangle$



- Lado esquerdo da árvore: $a_i \leq a_j$.
- Lado direito da árvore: $a_i \geq a_j$.

Árvore de Decisão

$\langle a_1, a_2, a_3 \rangle$
$= \langle 9, 4, 6 \rangle$



- Os nós folhas representam permutações $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ que indicam a ordem $a_{\pi(1)} \leq a_{\pi(2)} \leq \ldots. \leq a_{\pi(n)}$.

Árvore de Decisão

- Uma árvore de decisão pode modelar a execução de qualquer algoritmo de comparação.
- Há uma árvore de decisão para cada tamanho de vetor *n*.
- Há um caminho na árvore de decisão para cada entrada do vetor de tamanho n.
- O tempo de execução do algoritmo é dado pelo tamanho do caminho.
- O pior caso é dado pela altura da árvore.

Limitante inferior para ordenação com Árvore de decisão

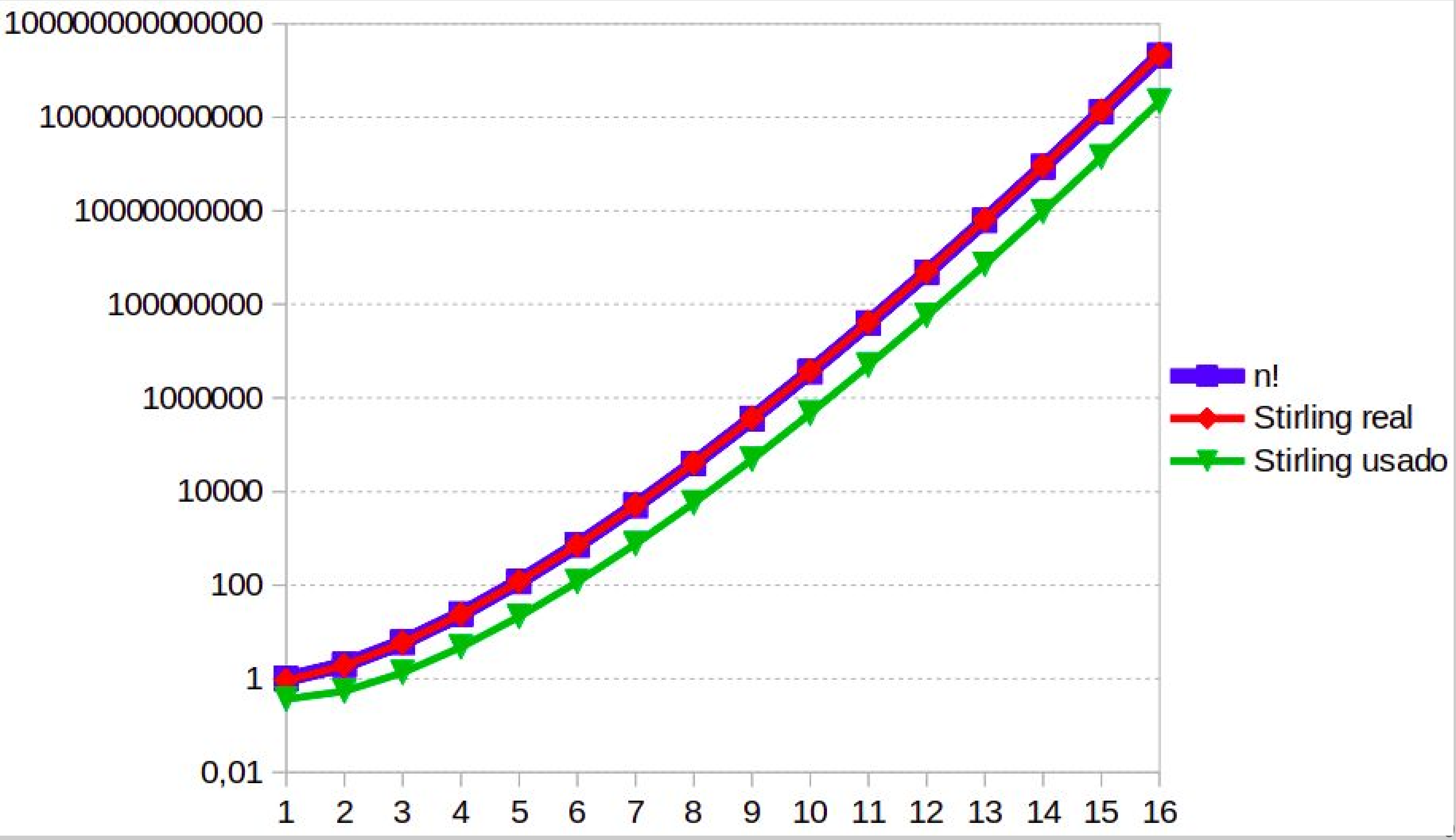
Teorema: Qualquer árvore de decisão que pode ordenar *n* elementos precisa ter altura $\mathbf{\Omega}(n \lg n)$.

Prova: A árvore precisa ter pelo menos $n!$ folhas, desde que há $n!$ possíveis permutações. Na altura h, a árvore binária possui no máximo 2^h folhas. Logo, $n! \leq 2^h$.

$$
\begin{aligned}
\therefore\ h &\geq \lg(n!) && \text{(lg é monotonicamente crescente)} \\
&\geq \lg((n/e)^n) && \text{(Stirling's formula)} \\
&= n \lg n - n \lg e \\
&= \Omega(n \lg n)\,. \quad \square
\end{aligned}
$$

L5.9

Aproximação de Stirling



Limitante inferior para ordenação com Árvore de decisão

Corolário: Merge and Heap sort são algorimtos de ordenação por comparação assintoticamente ótimos.

L5.9

Counting Sort

- **Counting Sort** ou **Ordenação por Contagem**
- Algoritmo inventado em **1954** por Harold H. Seward.
- **Não** há **comparações** entre elementos.
- Assume que cada uma das n entradas é um inteiro entre 0…k para algum inteiro k.
- Entrada: A[1..*n*], onde A[j] ∈ {0, 1, 2, …, *k*}
- Saída: B[1..*n*], ordenado
- C[1..*k*] é utilizado para armazenamento auxiliar.

Counting Sort

```
Counting-Sort(A, B, k)
 for i = 0 to k
     C[i]= 0;
 for j = 1 to A.length
     C[A[j]] = C[A[j]] + 1;
 for i = 1 to k
     C[i] = C[i] + C[i-1];
 for j = A.length downto 1
     B[C[A[j]]] = A[j];
     C[A[j]] = C[A[j]] - 1;
```

Counting Sort

```
Counting-Sort(A, B, k)
  for i = 0 to k
      C[i]= 0;
for j = 1 to A.length
      C[A[j]] = C[A[j]] + 1;
for i = 1 to k
      C[i] = C[i] + C[i-1];
  for j = A.length downto 1
      B[C[A[j]]] = A[j];
      C[A[j]] = C[A[j]] - 1;
```

$\Theta(k)$

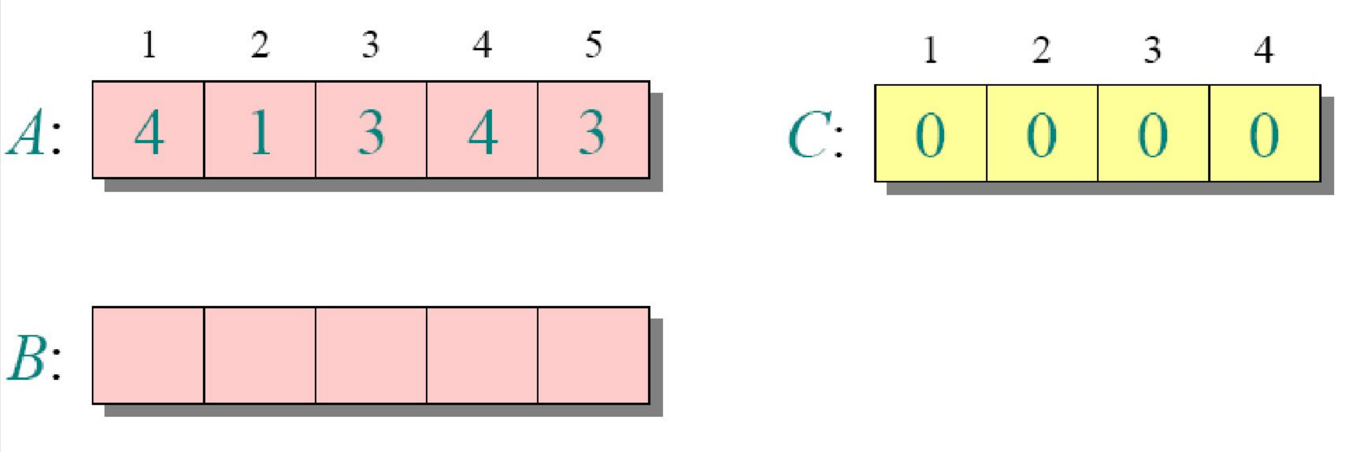
$\Theta(n)$

$\Theta(k)$

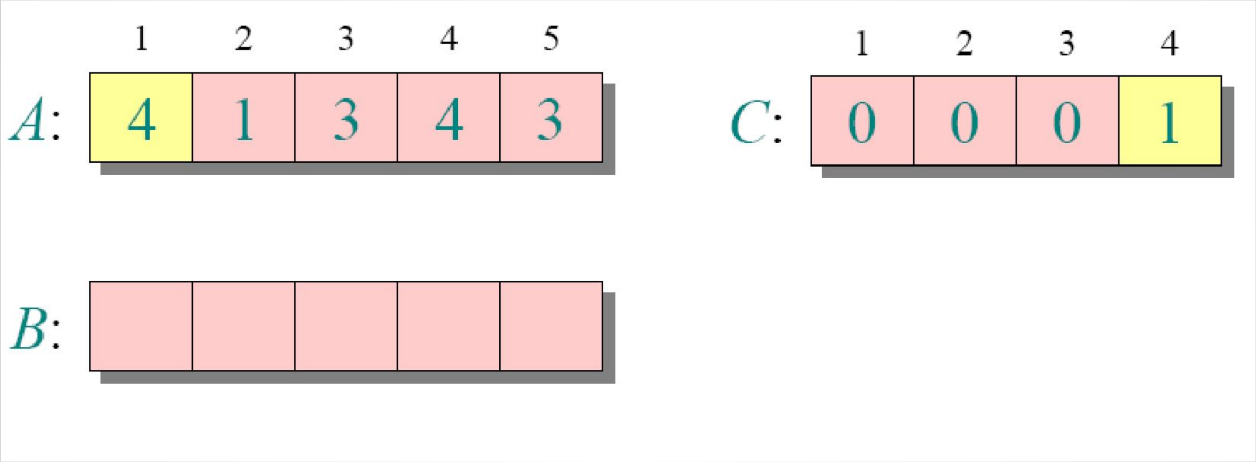
$\Theta(n)$

$= \Theta(n+k)$

Loop 1



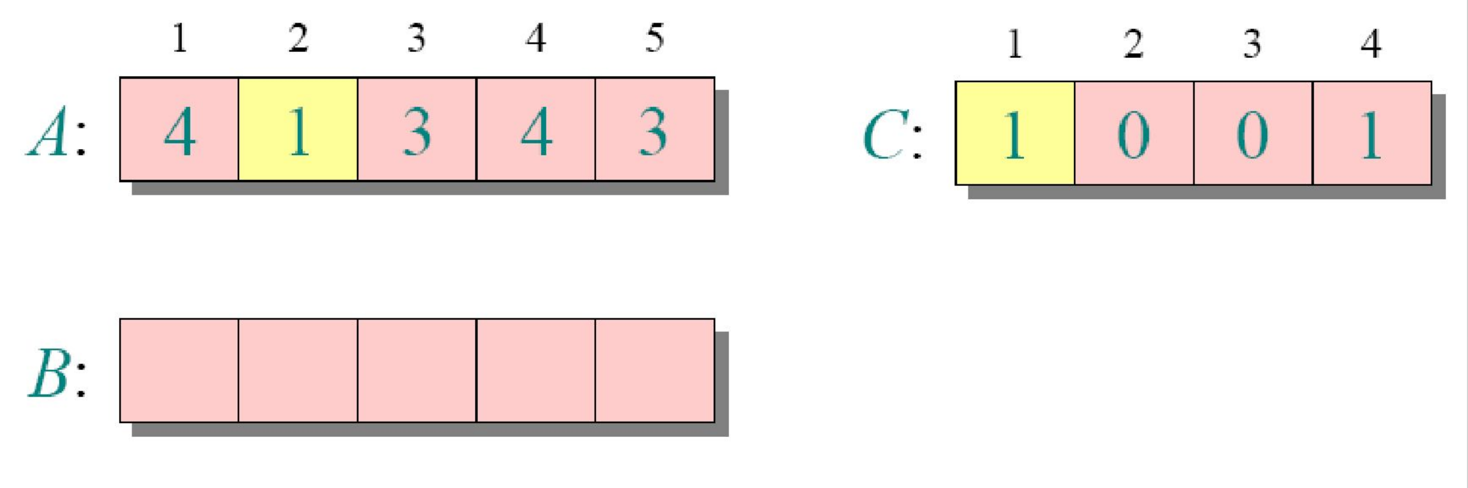
Loop 2



for j=1 **to** n

do $C[A[j]] = C[A[j]] + 1$

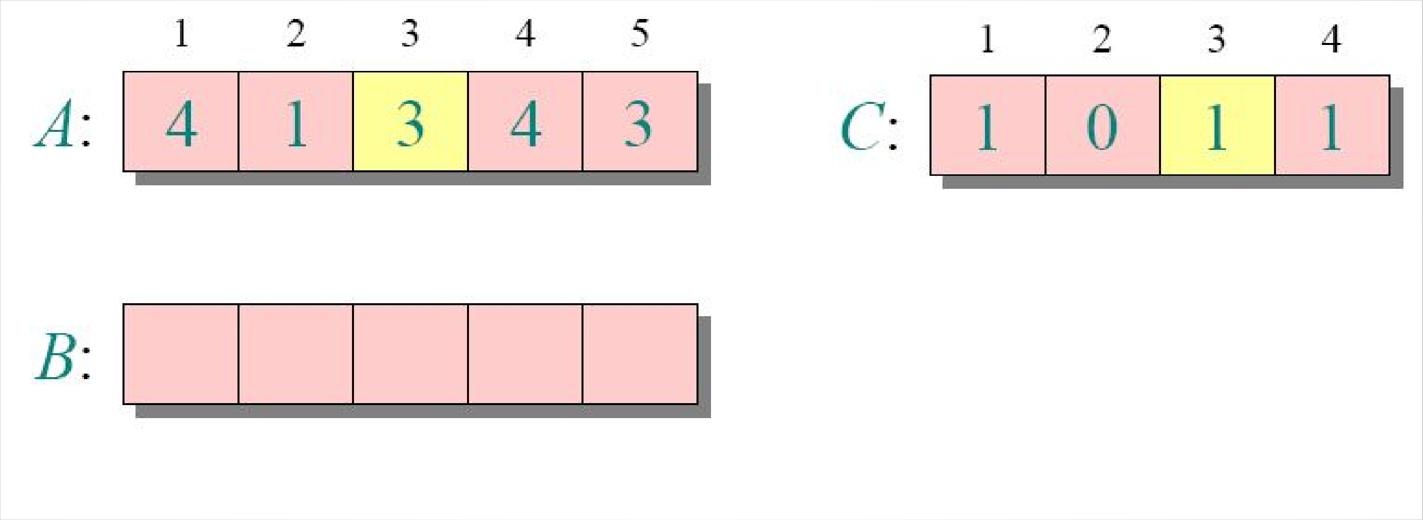
Loop 2



for j=1 **to** n

do $C[A[j]] = C[A[j]] + 1$

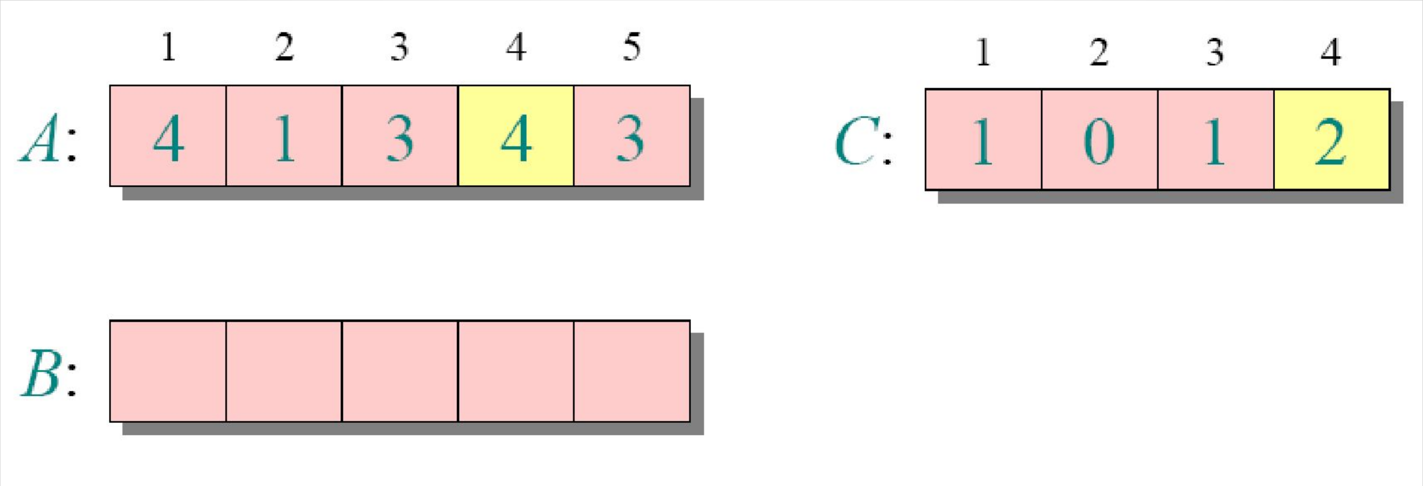
Loop 2



for j=1 **to** n

do $C[A[j]] = C[A[j]] + 1$

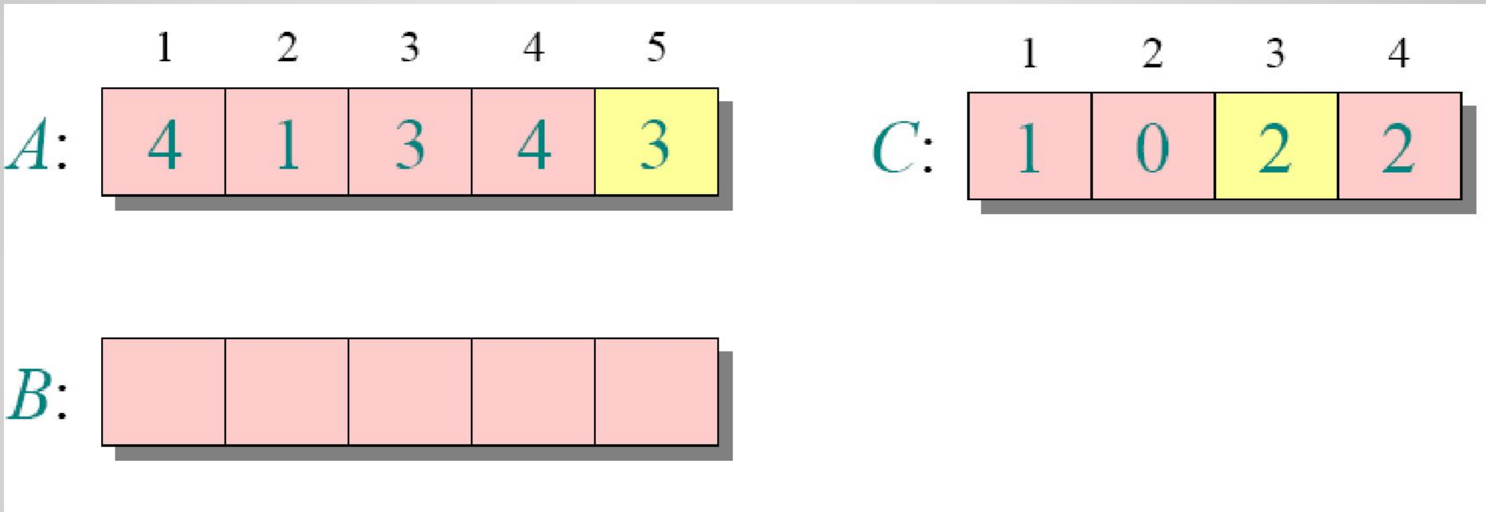
Loop 2



for j=1 **to** n

do $C[A[j]] = C[A[j]] + 1$

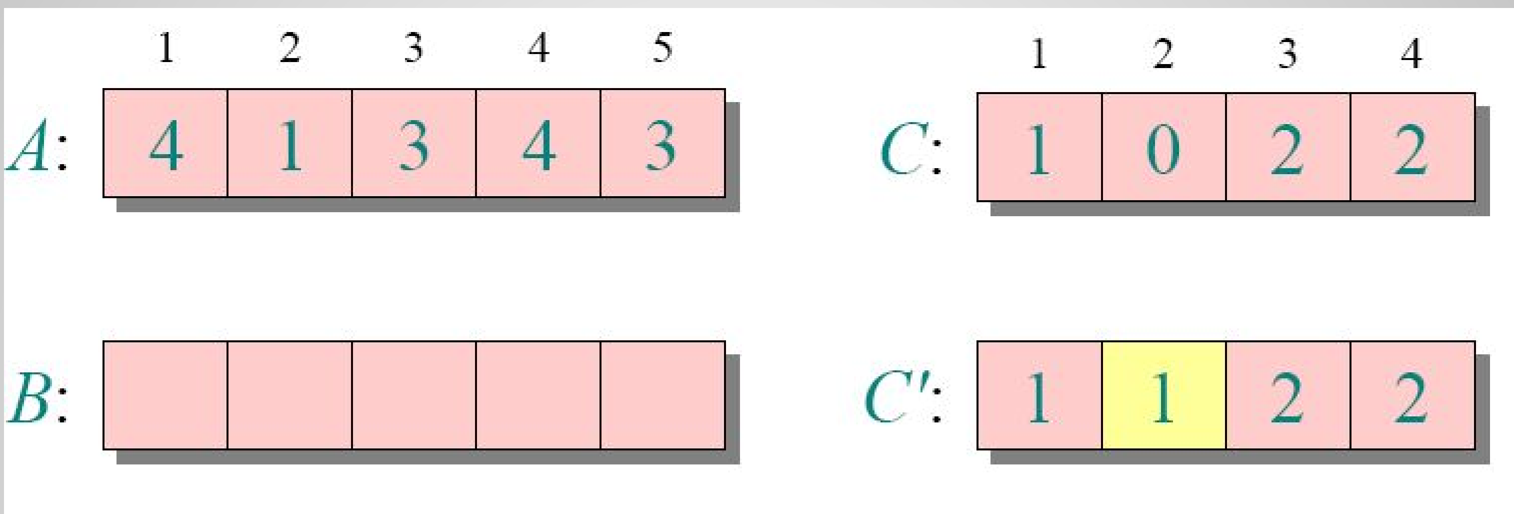
Loop 2



for j=1 **to** n

do $C[A[j]] = C[A[j]] + 1$

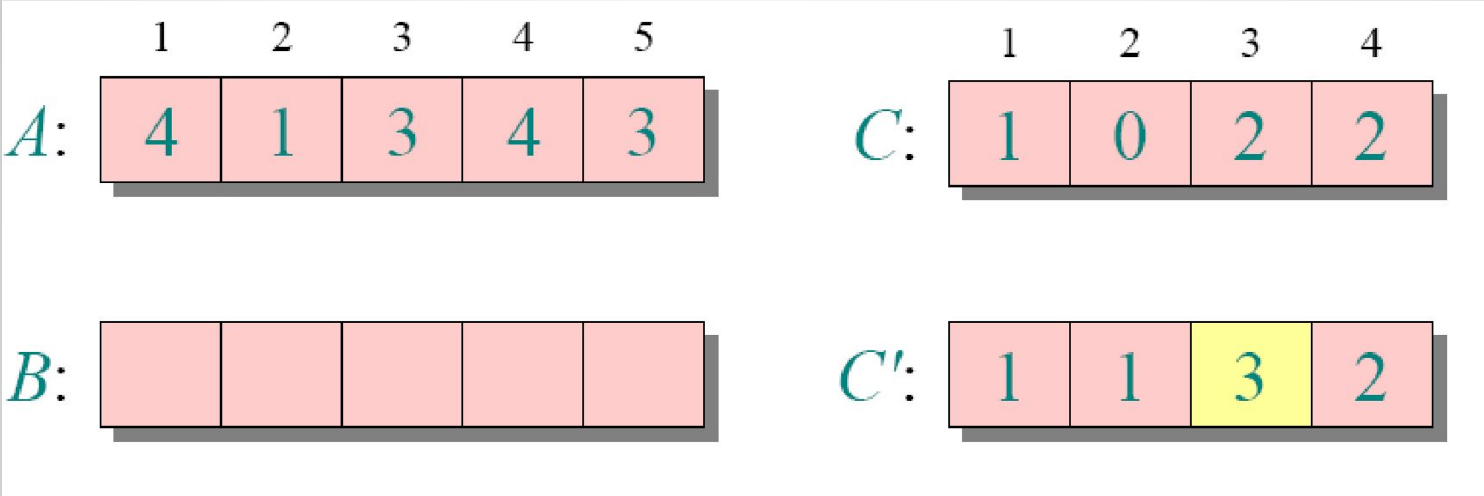
Loop 3



for i=2 **to** k

do $C[i] = C[i] + C[i-1]$

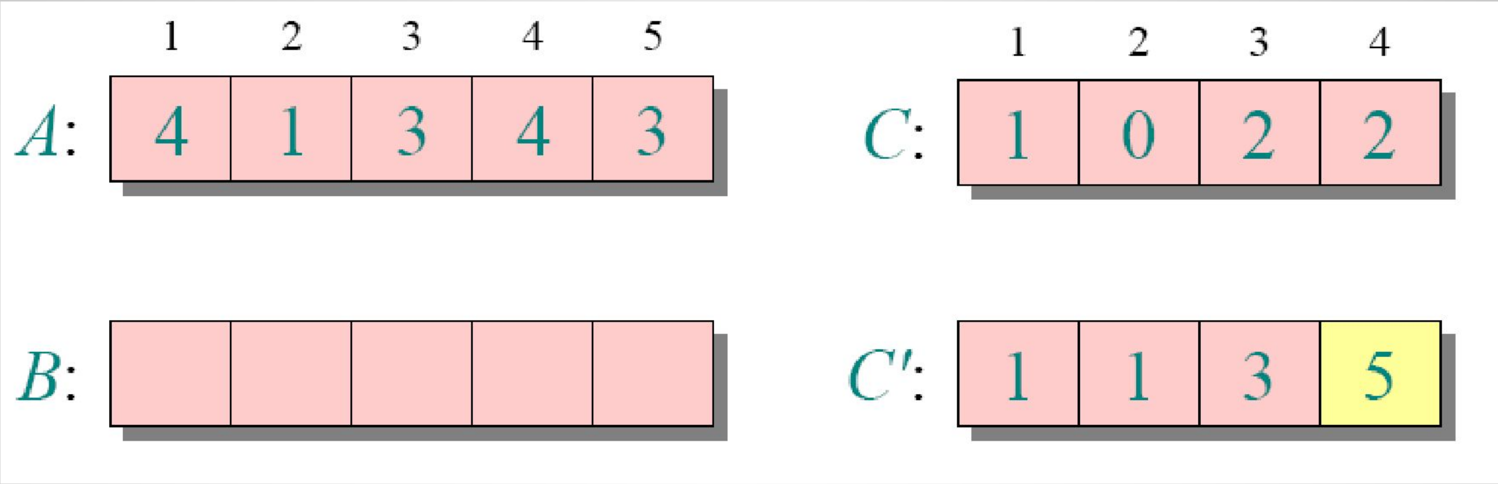
Loop 3



for i=2 **to** k

do $C[i] = C[i] + C[i-1]$

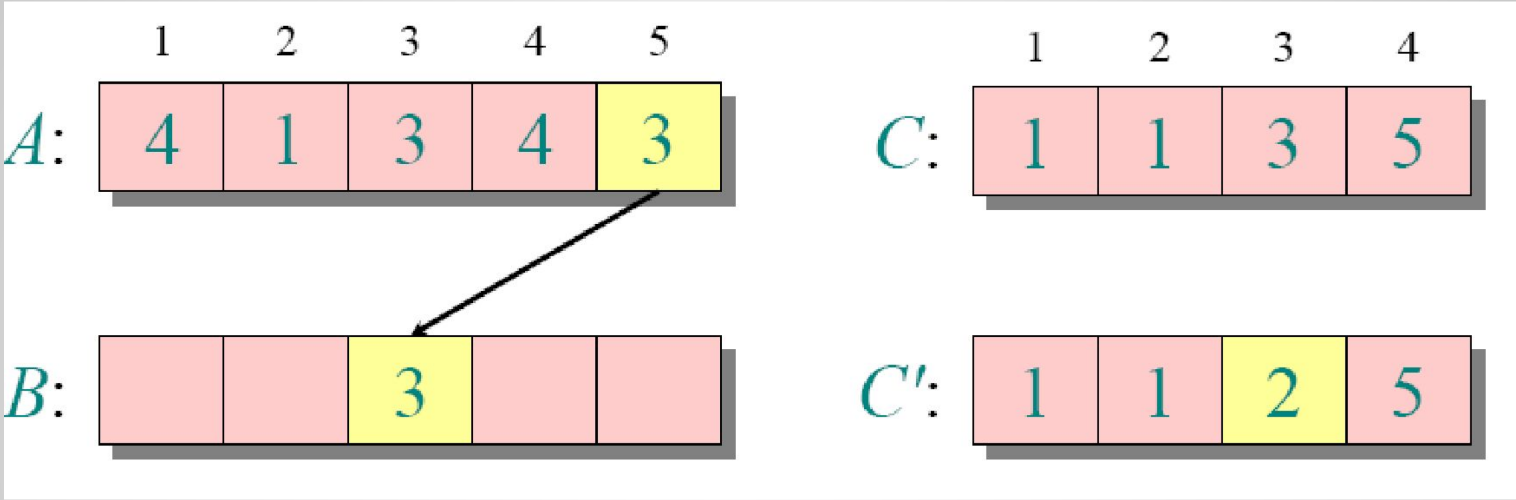
Loop 3



for i=2 **to** k

do $C[i] = C[i] + C[i-1]$

Loop 4

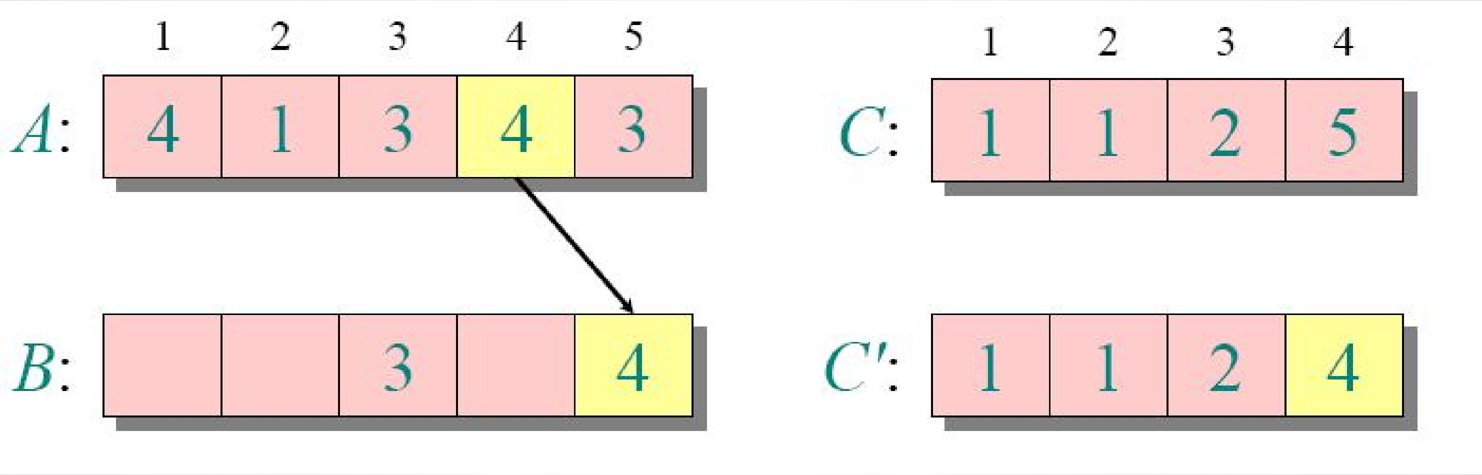


for $j = n$ **downto** 1

do $B[C[A[j]]] = \mathrm{A}[j]$

$C[A[j]] = C[A[j]] - 1$

Loop 4

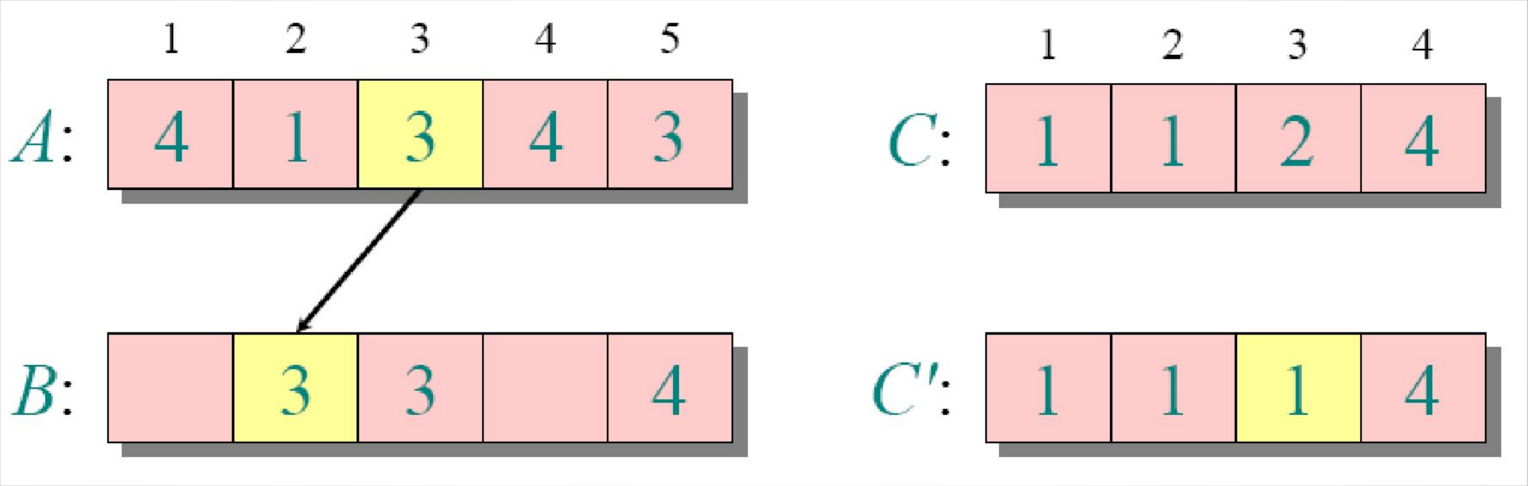


for $j = n$ **downto** 1

do $B[C[A[j]]] = \mathrm{A}[j]$

$C[A[j]] = C[A[j]] - 1$

Loop 4

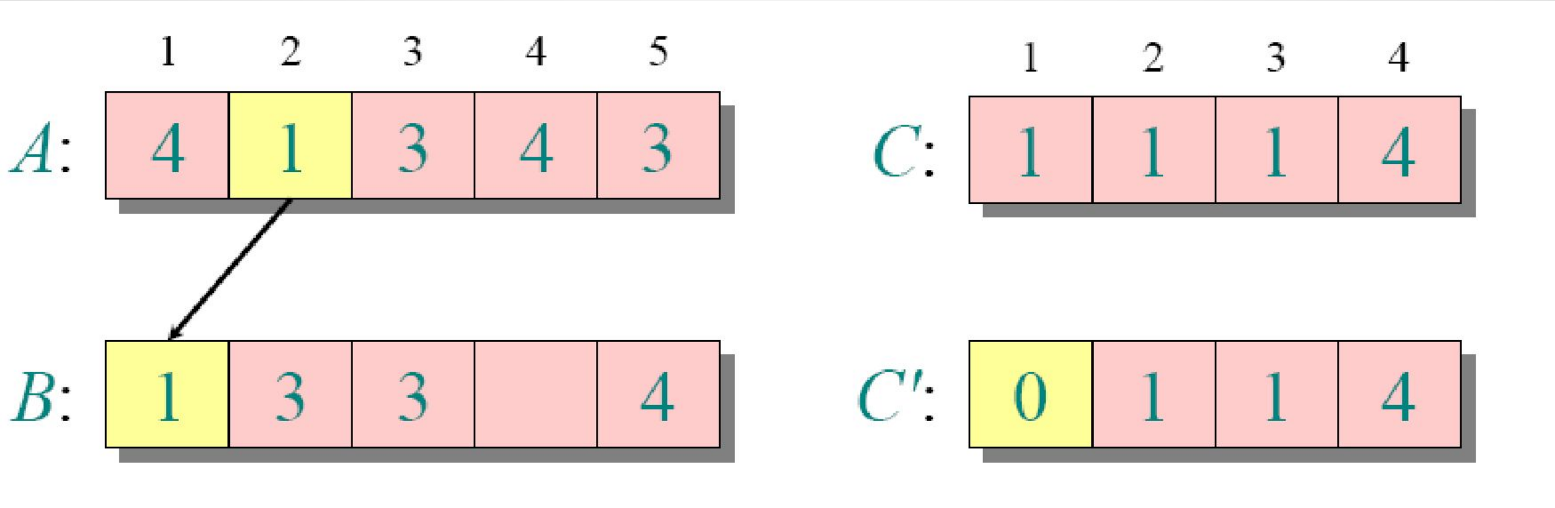


for $j=n$ **downto** 1

do $B[C[A[j]]] = \mathrm{A}[j]$

$C[A[j]] = C[A[j]] - 1$

Loop 4

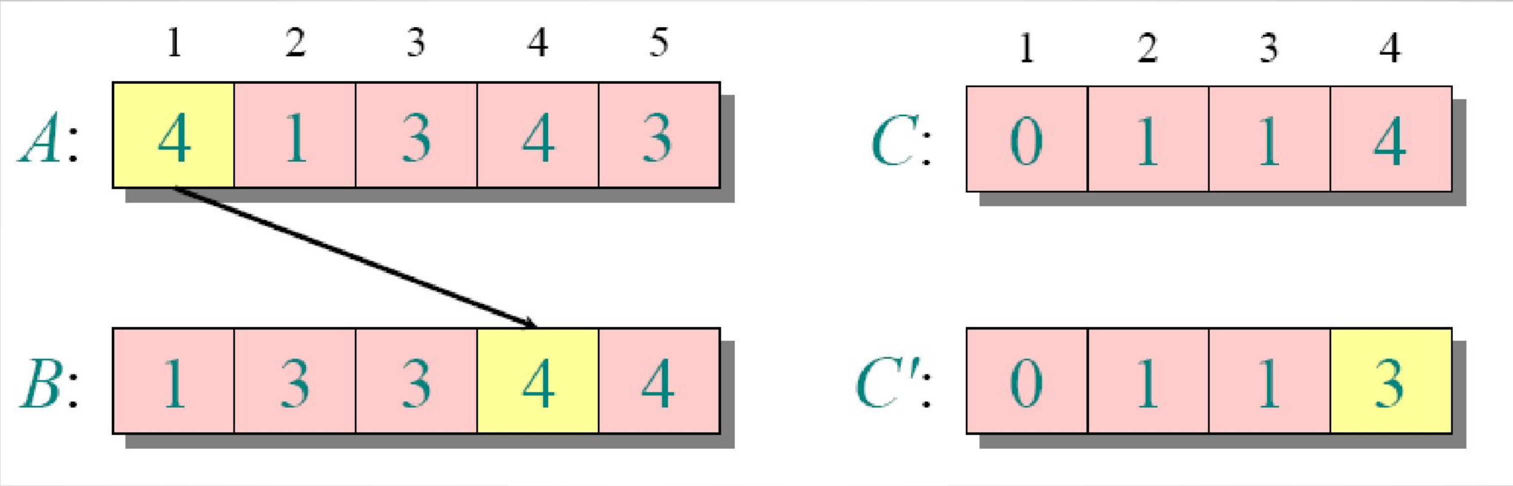


for j=n **downto** 1

do $B[C[A[j]]]$ =A[j]

$C[A[j]]$ =$C[A[j]]$ –1

Loop 4



for $j = n$ **downto** 1

do $B[C[A[j]]] = \mathrm{A}[j]$

$C[A[j]] = C[A[j]] - 1$

Counting Sort

- Counting sort deveria ser utilizado sempre, certo?
- Dependência em relação aos *k* elementos
- *Exemplo: Ordenar inteiros codificados em 32 bits* ⇒ $k = 2^{32}$
- Counting sort necessita de memória auxiliar, logo não faz ordenação *in-place*.
- Complexidade $\Theta(n+k)$ no melhor, pior e caso médio.

Counting Sort

- Counting sort ordena apenas valores inteiros positivos.

- Como ordenar os números:
  A = {-10, -12, -4, -3, 50, -20, 5}
- Como melhorar a ordenação dos números:
  A = {1.000.151, 1.000.093, 1.000.099, 1.000.041, 1.000.011, 1.000.057, 1.000.060}
- Como ordenar os números:
  A = {-10,5; -12,6; -4,1; -3,0; 50,9; -20,7; 5,8}

Counting Sort

- Como ordenar os números:

  A = {-10, -12, -4, -3, 50, -20, 5}

  Primeiro some 20 a todos os números:

  A = {10, 8, 16, 17, 70, 0, 25}

  Então ordene os números com counting sort:

  A = {0, 8, 10, 16, 17, 25, 70}

  Por fim, subtraia 20 de todos os números:

  A = {-20, -12, -10, -4, -4, 5, 50}

Counting Sort

- Como melhorar a ordenação dos números:

  A = {1.000.151, 1.000.093, 1.000.099, 1.000.041, 1.000.011, 1.000.057, 1.000.060}

  Primeiro subtraia 1.000.000 de todos os números:

  A = {151, 93, 99, 41, 11, 57, 60}

  Então ordene os números com counting sort:

  A = {11, 41, 57, 60, 93, 99, 151}

  Por fim, some 1.000.000 a todos os números:

  A = {1.000.011, 1.000.041, 1.000.057, 1.000.060, 1.000.093, 1.000.099, 1.000.151}

Counting Sort

- Como ordenar os números:

  A = {-10,5; -12,6; -4,1; -3,0; 50,9; -20,7; 5,8}

  Primeiro some 20,7 a todos os números:

  A = {10,2; 8,1; 16,6; 17,7; 71,6; 0,0; 26,5}

  Multiplique os números por 10.

  E carregue-os em um novo vetor A':

  A' = {102; 81; 166; 177; 716; 0; 265}

  Então ordene os números de A' com counting sort:

  A' = { 0; 81; 102; 166; 177; 265; 716}

  Divida os números por 10 e jogue-os no vetor antigo.

  A = { 0,0; 8,1; 10,2; 16,6; 17,7; 26,5; 71,6}

  Por fim, subtraia 20,7 de todos os números:

  A = { -20,7; -12,6; -10,5; -4,1; -3,0; 5,8; 50,9}

Ordenação Estável
Stable sorting

- Algoritmos de ordenação estável preservam a ordem entre elementos iguais.

- Counting sort é estável

	1	2	3	4	5	6	7	8
A	2_1	5_1	3_1	0_1	2_2	3_2	0_2	3_3

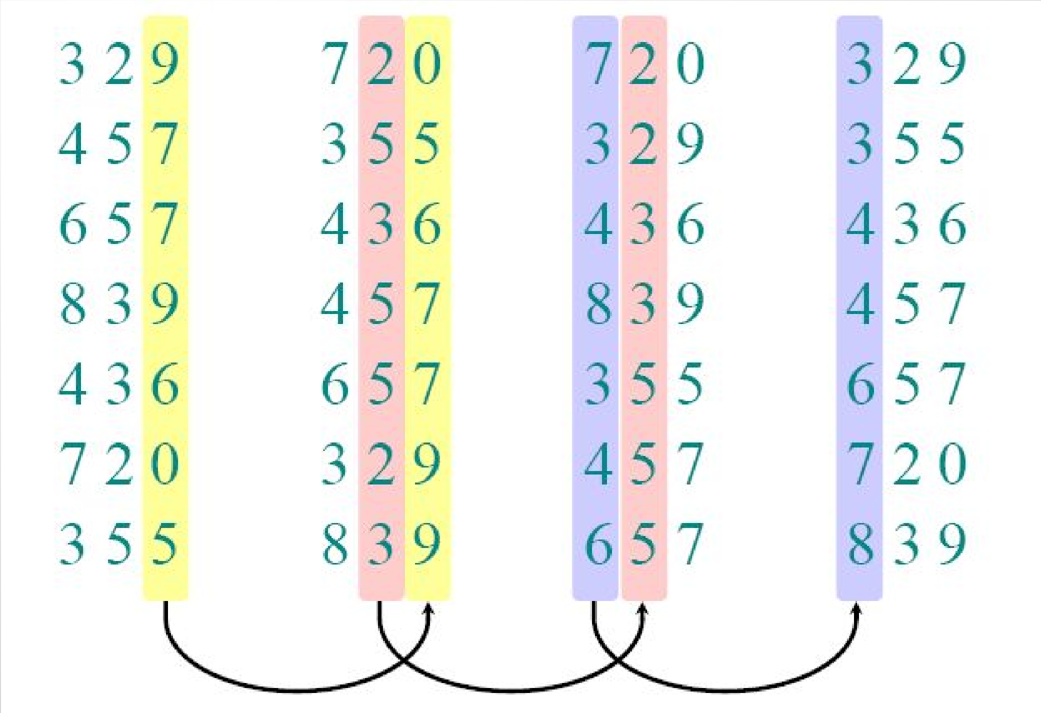
	1	2	3	4	5	6	7	8
B	0_1	0_2	2_1	2_2	3_1	3_2	3_3	5_1

Radix Sort

- **Radix Sort** ou **Ordenação da Raiz**
- Método usado em cartões perfurados por Herman Hollerith.
- Primeiro algoritmo computacional para o radix sort foi inventado em **1954** no MIT por **Harold H. Seward.**
- Ordenação dígito a dígito.
- Idéia original de Hollerith: classificar pelo dígito mais significativo.
- Boa ideia: classificar começando pelo dígito menos significativo utilizando uma ordenação estável auxiliar.

Radix Sort

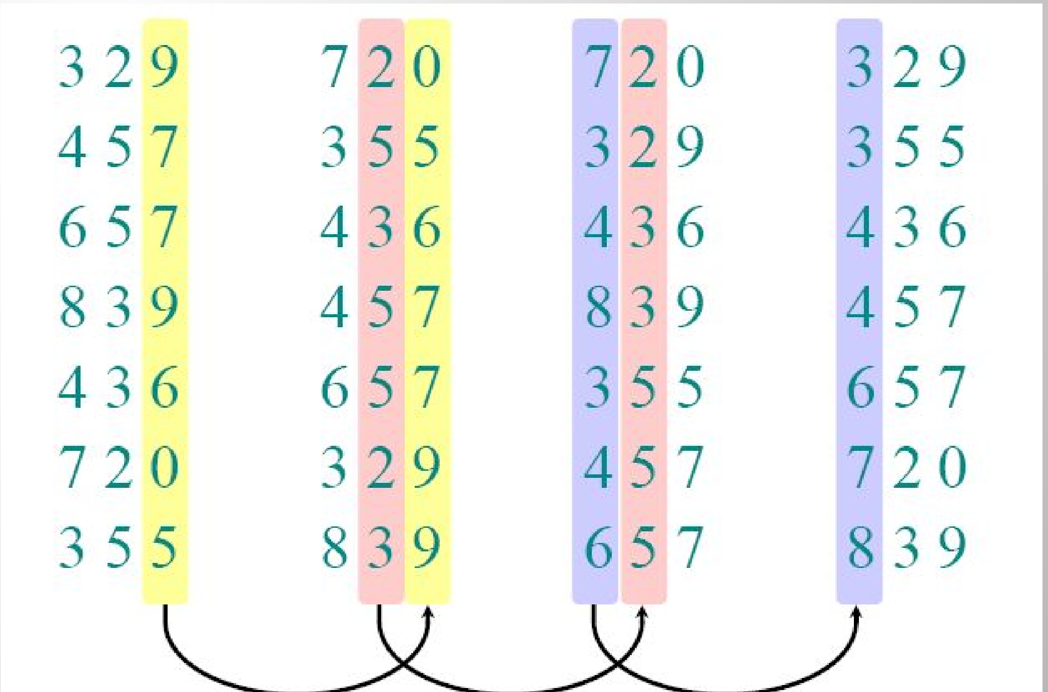
```
RADIX_SORT(A,d)
for i = 1 to d
    usar uma ordenação estável para ordenar o arranjo A sobre o dígito i
```



Radix Sort

```
RADIX_SORT(A,d)
for i = 1 to d
    usar uma ordenação estável para ordenar o arranjo A sobre o dígito i
```

- Em geral usa-se o counting sort como ordenação estável.
- Dessa forma, a ordenação não é *in-place*.



Radix Sort

```
RADIX_SORT(A,d)
for i = 1 to d
    usar uma ordenação estável para ordenar o arranjo A sobre o dígito i
```

- Executar o radix sort para a entrada:
  - A = [713, 131, 312, 124, 245, 457, 572, 724, 243, 437]

Radix Sort

Dado n números com d-dígitos, onde cada dígito pode assumir até k valores possíveis, RADIX-SORT ordena corretamente esses números em $\Theta(d(n + k))$.

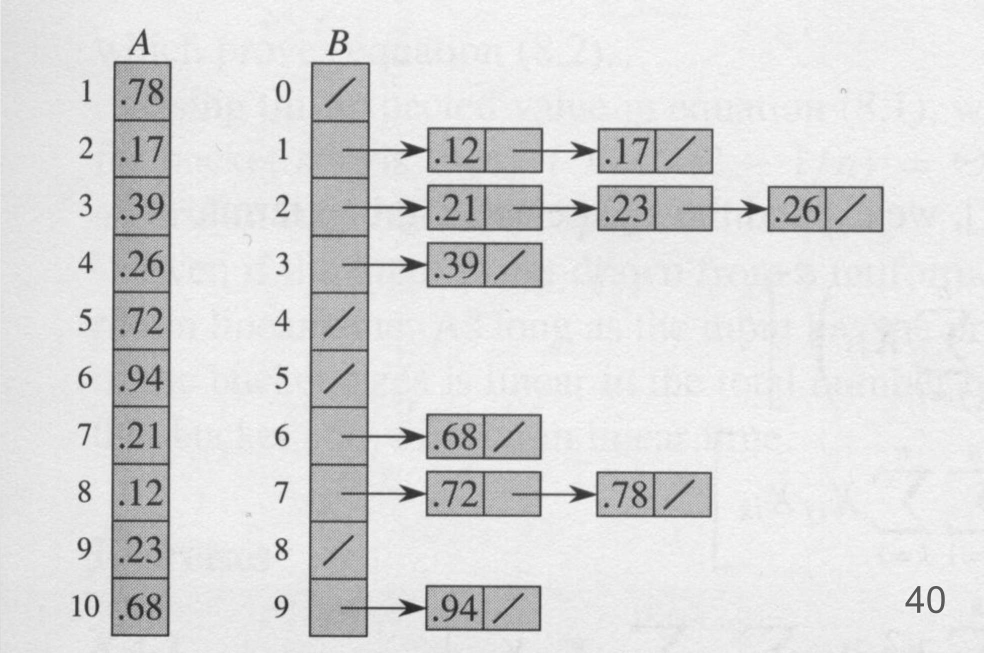
- Complexidade de pior caso: $\Theta(d(n + k))$
- Complexidade de melhor caso: $\Theta(d(n + k))$
- Complexidade de caso médio: $\Theta(d(n + k))$

Bucket Sort

- **Bucket Sort** ou **Bin Sort** ou **Ordenação por Balde**
- Algoritmo inventado em **1956** por **E. J. Isaac e R. C. Singleton**.
- Bucket sort, assim como counting sort, assume algo a respeito da entrada.
  - ✓ Counting sort: assume que as entradas são inteiras em um pequeno intervalo.
  - ✓ Bucket sort: assume que a entrada é gerada aleatoriamente com forma uniforme e independente no intervalo [0,1).

Bucket Sort

- Cria *n* listas ligadas para dividir o intervalo [0,1) em subintervalos de tamanho 1/*n*
- Adiciona cada elemento da entrada à lista apropriada e ordena cada lista com insertion sort.



Bucket Sort

```
Bucket-Sort(A)
 Seja B[0..n-1] um novo vetor
 n = A.length
for i=0 to n-1
      Faça B[i] uma lista vazia
for i = 1 to n
      Insira A[i] na lista B[ ⌊nA[i]⌋ ]
 for i = 0 to n-1
      Ordena  B[i] com insertion sort
Concatena listas B[0], B[1],…, B[n-1]
```

Análise Bin Sort: Θ(n)

Bucket Sort

- ➢ Executar algoritmo Bucket Sort para ordenar o vetor:
  - ✓ A = [0.65, 0.24, 0.92, 0.36]
  - ✓ A = [0.65, 0.24, 0.92, 0.36, 0.62, 0.14, 0.53, 0.19, 0.28, 0.71]
- ➢ Como ordenar o vetor com o método Bucket sort:
  - ✓ A = [5, 1, 7, 2, 8, 9, 3, 4]
  - ✓ A = [5.2, 1.5, 7.7, 2.4, 8.9, 9.2, 3.1, 4.0]