

Estrutura de Dados

Métodos de Ordenação sem comparação de chaves

Professores: Anisio Lacerda
Wagner Meira Jr.

Estrutura de Dados

Professores: Anisio Lacerda
Wagner Meira Jr.

Até Agora...

Todos os algoritmos de ordenação com base em:

- ❑ Comparação
 - ❑ Troca
-
- Ordenação Comparativa

**Algoritmos
simples
 $O(n^2)$**

Insertion sort
Selection sort
Shell sort
...

**Algoritmos
sofisticados
 $O(n \log n)$**

Merge sort
Quick sort
Heap sort
...

**Limite inferior
 $\Omega(n \log n)$**

**Algoritmos
especializados
 $O(n)$**

Counting sort
Bucket sort
Radix sort
...

**Grandes
volumes de
dados**

Ordenação
Externa

Hoje

- Existem métodos que **não requerem comparações de chaves**. Por exemplo:
 - Counting Sort
 - Bucket Sort
 - Radix Sort

COUNTING SORT

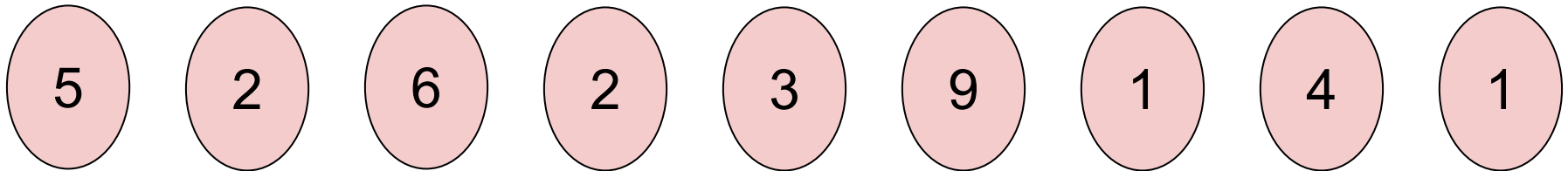
Ordenação por Contagem

Considere o seguinte problema:

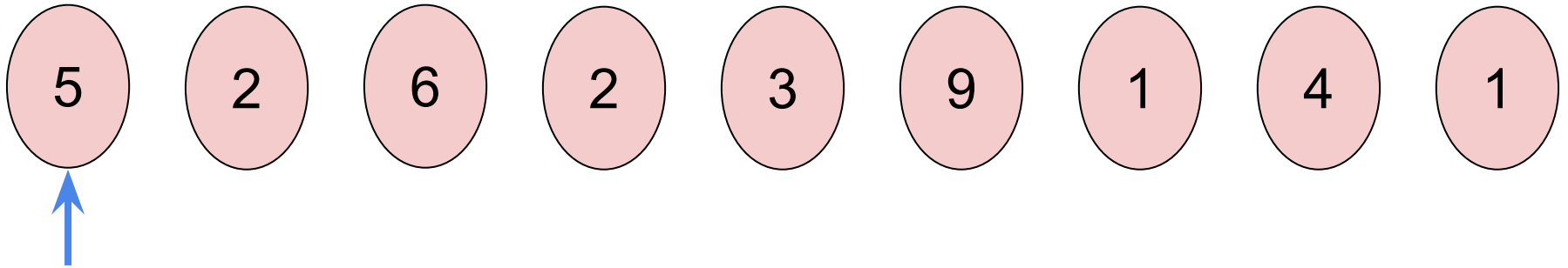
- Dada uma lista/vetor de elementos
 - cada elemento: $[0, \text{max})$
- Max é conhecido antecipadamente

Ordenação por Contagem

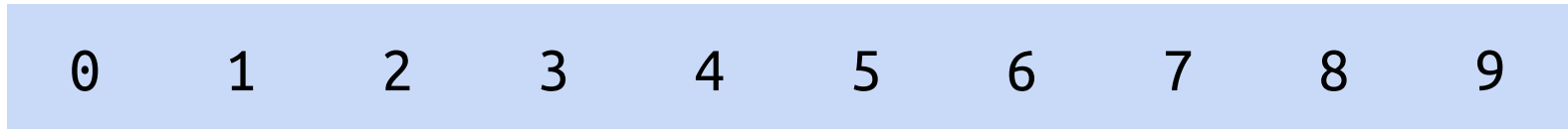
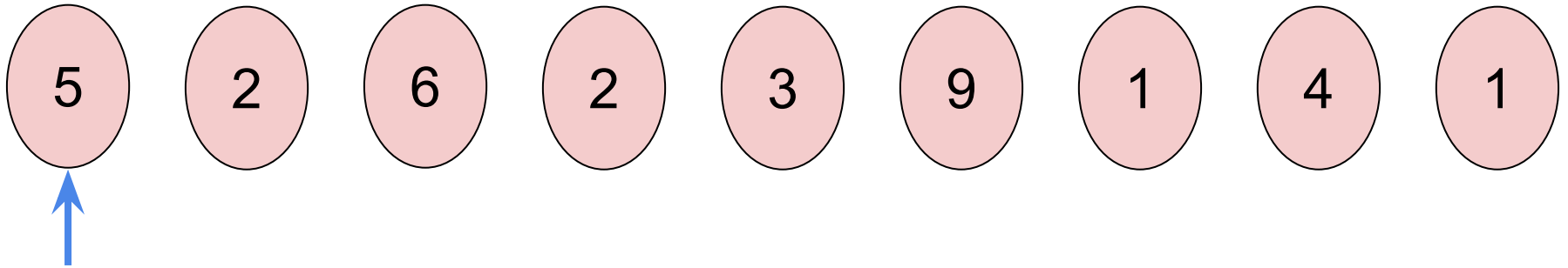
- Dada uma lista/vetor de elementos entre $[0, \text{max})$
 - Max é conhecido antecipadamente
 - $\text{max} = 9$



Contagem

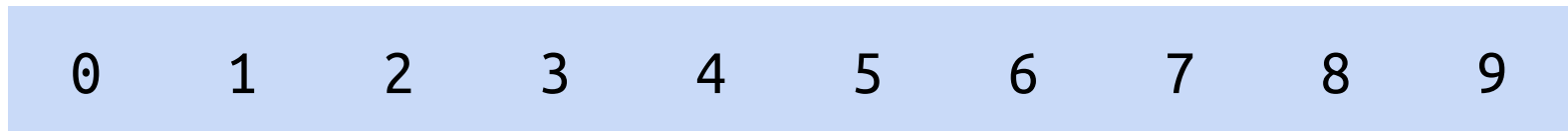
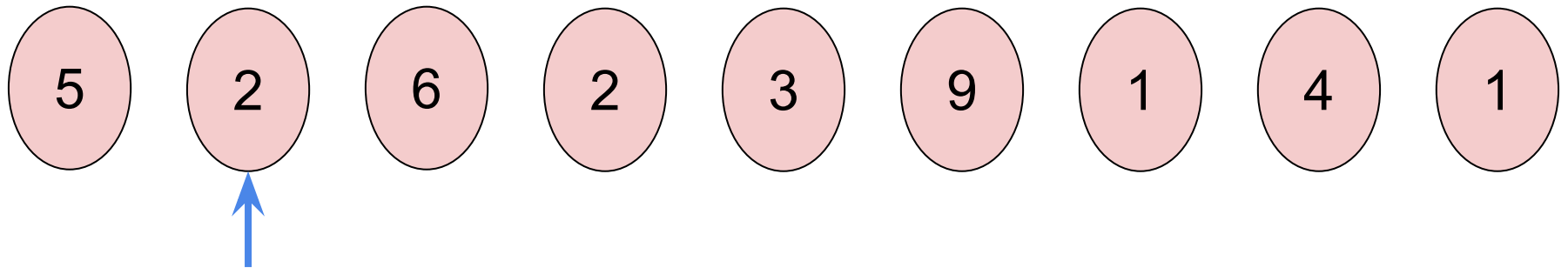


Contagem



1

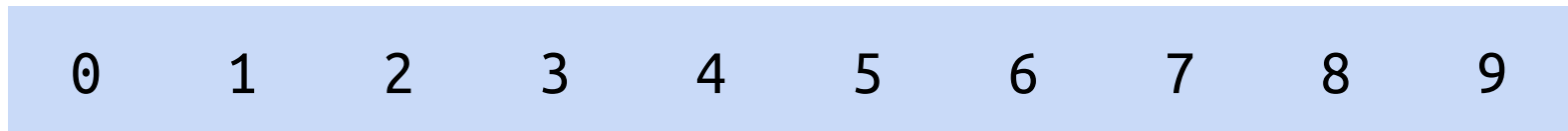
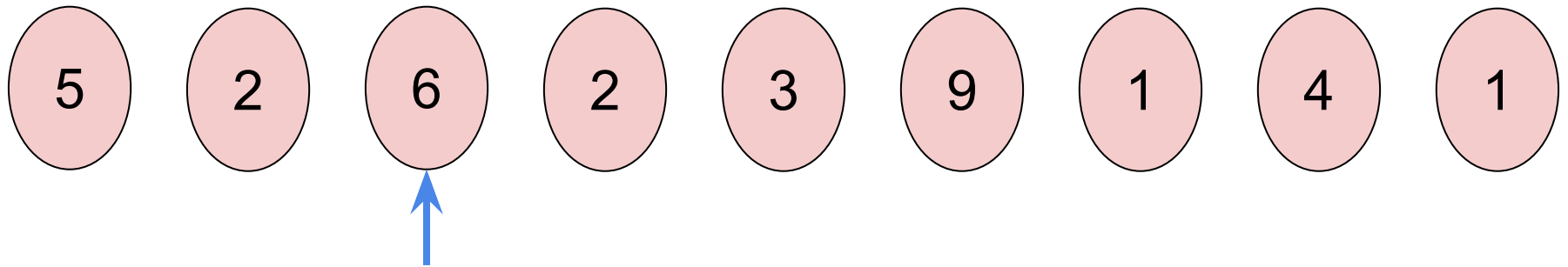
Contagem



1

1

Contagem

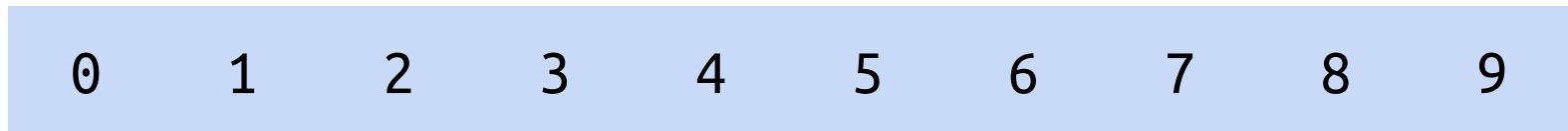
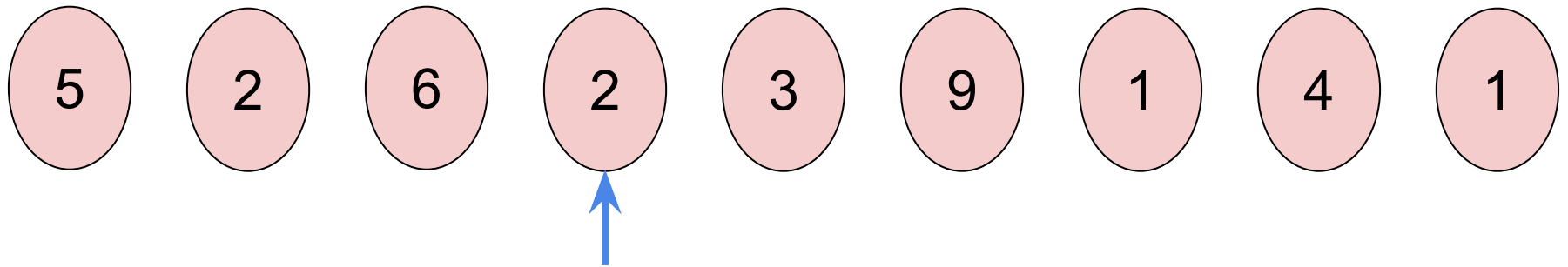


1

1

1

Contagem

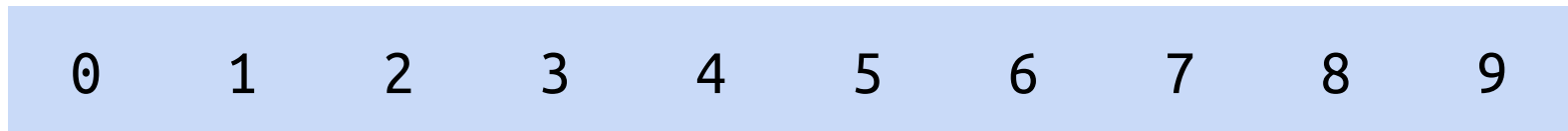
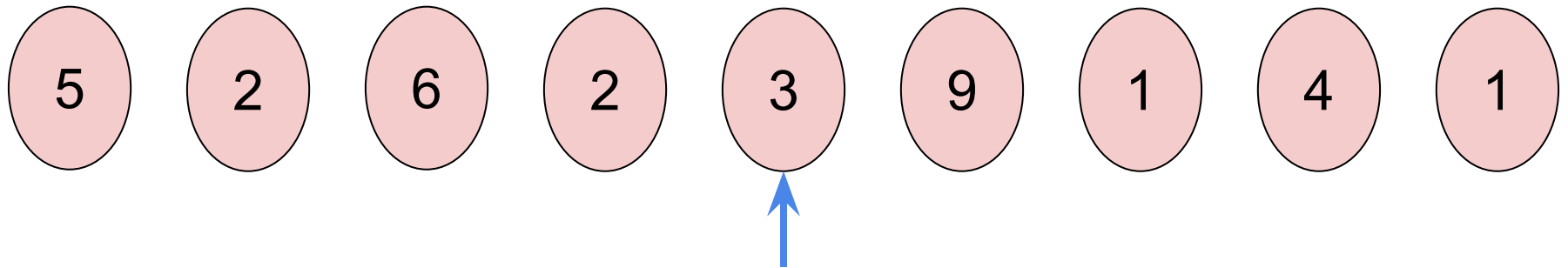


2

1

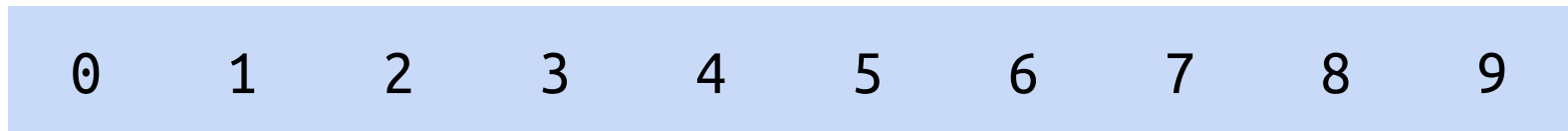
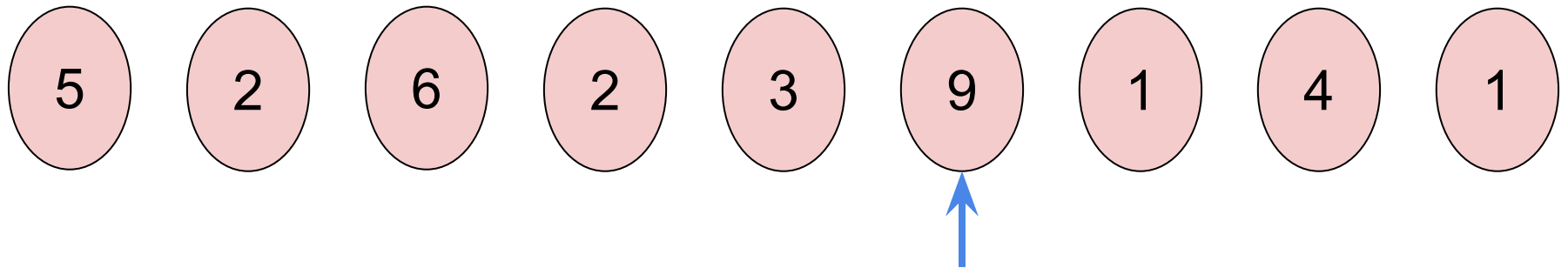
1

Contagem



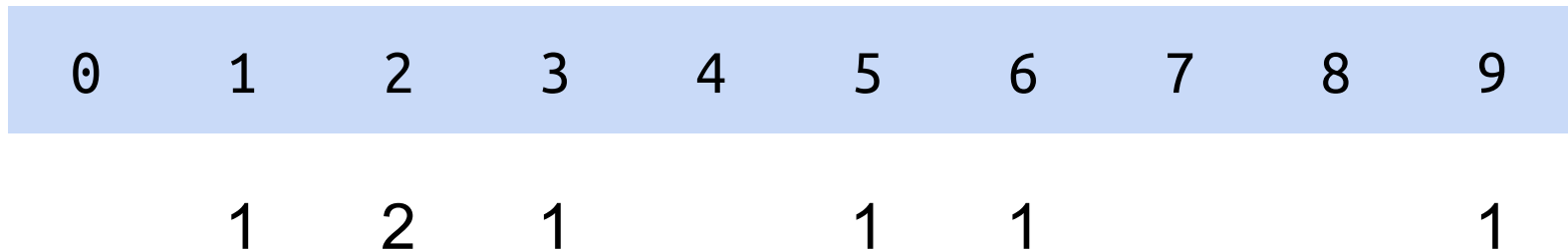
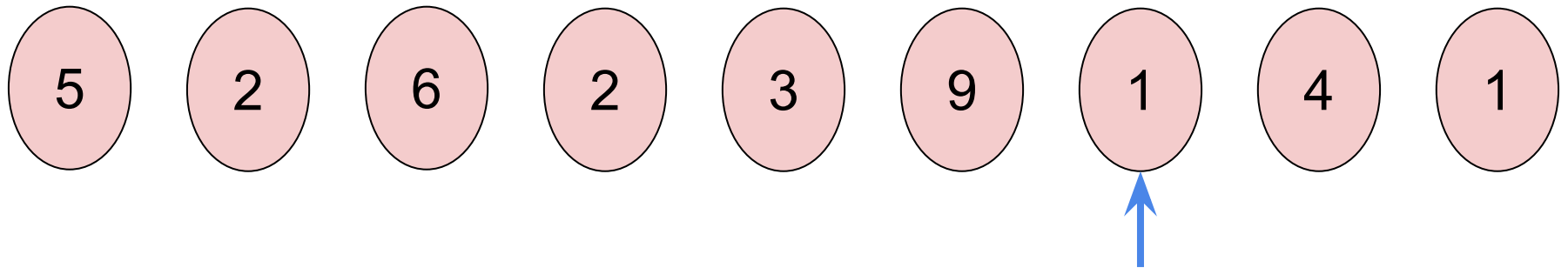
2 1 1 1

Contagem

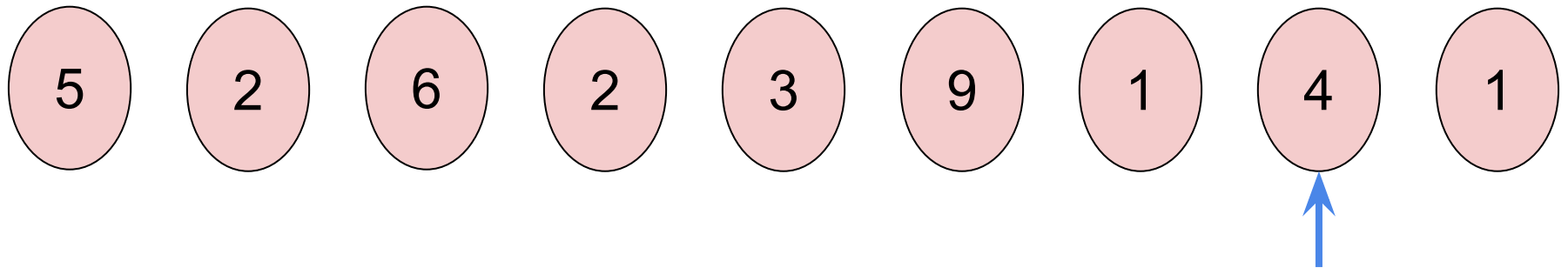


2 1 1 1 1

Contagem

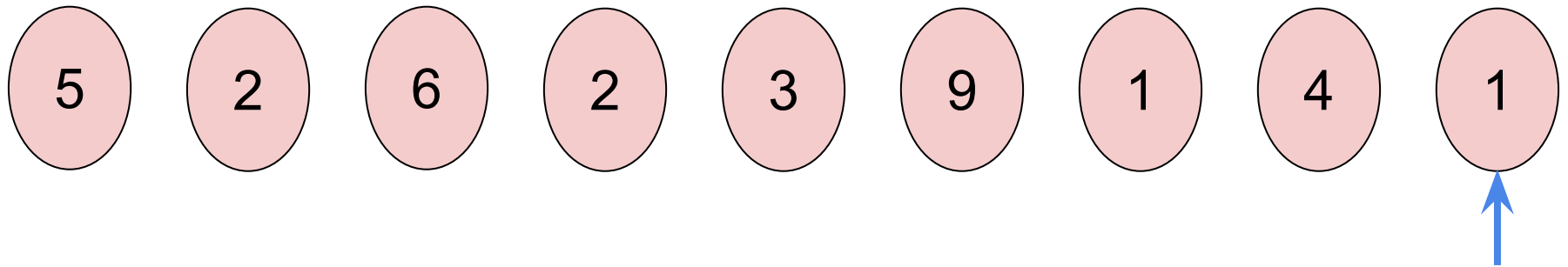


Contagem



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

Contagem



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


Sabendo das contagens

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

- Como ordenar?


Sabendo das contagens

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



Sabendo das contagens

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



Sabendo das contagens

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

0	1	2	1	1	1	1	0	0	1
---	---	---	---	---	---	---	---	---	---

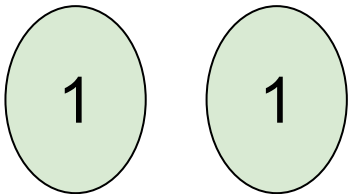


1

Sabendo das contagens

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

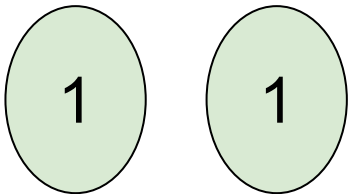
0	0	2	1	1	1	1	0	0	1
---	---	---	---	---	---	---	---	---	---



Sabendo das contagens

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

0	0	2	1	1	1	1	0	0	1
---	---	---	---	---	---	---	---	---	---



Sabendo das contagens

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

0	0	1	1	1	1	1	0	0	1
---	---	---	---	---	---	---	---	---	---



1

1

2

Sabendo das contagens

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

0	0	0	1	1	1	1	0	0	1
---	---	---	---	---	---	---	---	---	---



1

1

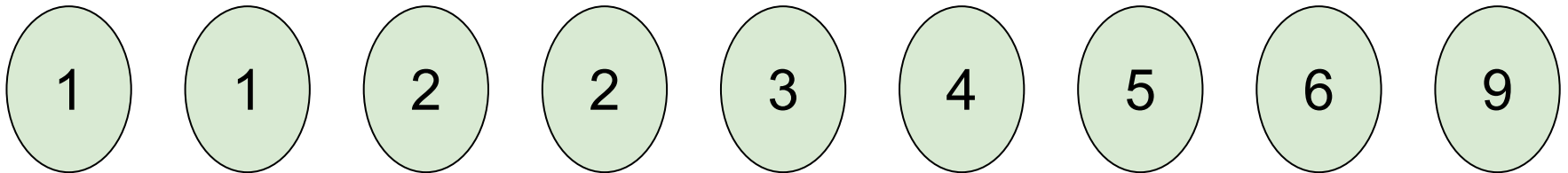
2

2

Sabendo das contagens

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

0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---



CountingSort (Ordenação por Contagem)

- Cria um vetor de contadores
- Conta todos os elementos
- Só funciona com limites de entrada bem definidos
 - Sem saber a entrada não temos como alocar a contagem de forma eficaz
 - Custo de memória

CountingSort

```
void countingsort(int *values, int n, int max) {  
    int *counts = (int *) calloc(max, sizeof(int));  
    int i, j;  
    for (i = 0; i < n; i++)  
        counts[values[i]]++;  
    i = 0;  
    for(j = 0; j < max; j++)  
        while(counts[j] > 0) {  
            values[i++] = j;  
            counts[j]--;  
        }  
    free(counts);  
}
```

CountingSort

```
void countingsort(int *values, int n, int max) {  
    int *counts = (int *) calloc(max, sizeof(int));  
    int i, j;  
    for (i = 0; i < n; i++)  
        counts[values[i]]++;  
    i = 0;  
    for(j = 0; j < max; j++)  
        while(counts[j] > 0) {  
            values[i++] = j;  
            counts[j]--;  
        }  
    free(counts);  
}
```

— Conta os elementos

— Percorre o vetor
usado para contar e
ordena

Complexidade

- Considere n o número de elementos e k o valor do maior elemento (max)
- Tempo
 - Uma passagem pelo vetor para contagem: $O(n)$
 - Uma passagem pelo “contador” para imprimir os elementos: $O(k)$ [com $O(n)$ impressões]
 - Total: $O(n+k)$
- Espaço
 - Memória extra: $O(k)$

Counting Sort - Considerações

■ Vantagem

- Ordenação em $O(n)$

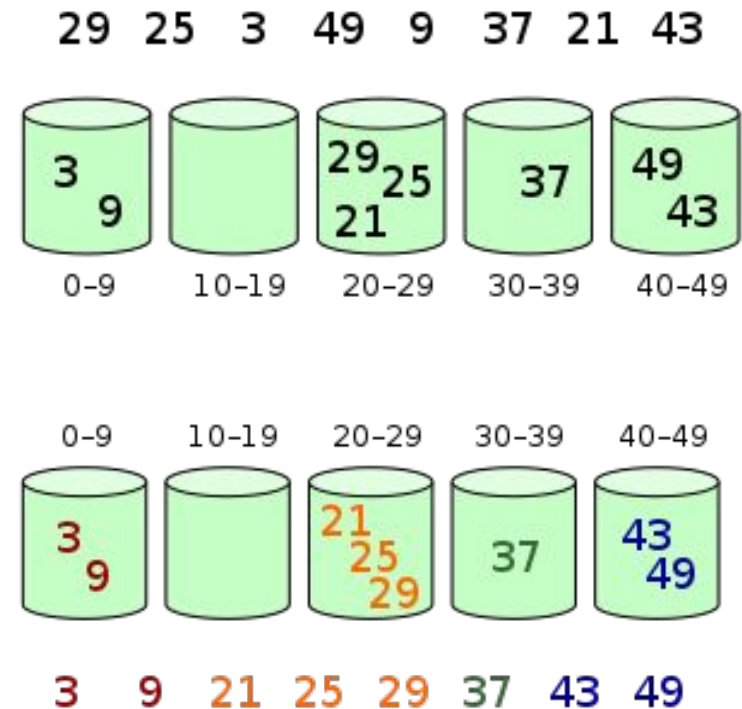
■ Desvantagens

- Muita memória extra
- Só é viável se soubermos a faixa de elementos a priori
 - Tamanho do vetor contador

BUCKET SORT

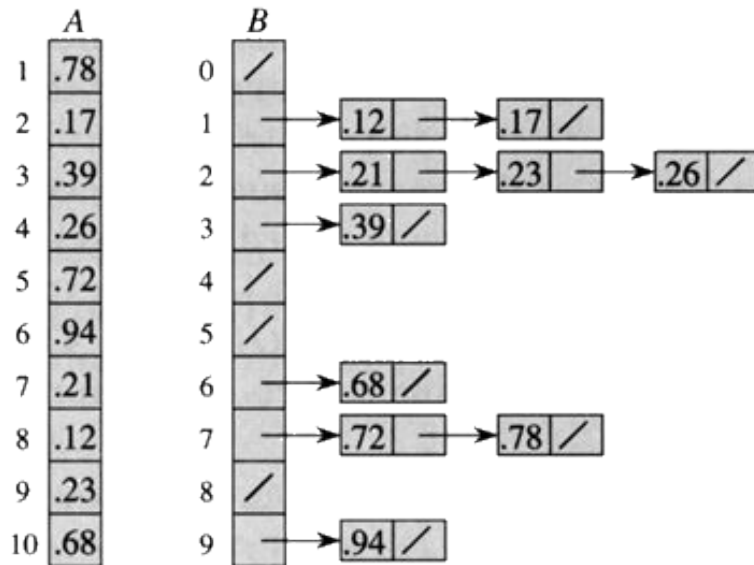
Bucket Sort

- Separa os elementos em *buckets* (baldes) de tamanho menor
- Ordena separadamente cada um dos baldes usando um dos algoritmos tradicionais



Bucket Sort

- Alternativamente, pode-se implementar os *buckets* como listas encadeadas e já inserir de forma ordenada em cada lista



Exemplo: <https://www.cs.usfca.edu/~galles/visualization/BucketSort.html>

Complexidade

- Considere n o número de elementos e k buckets (listas ordenadas)
- Tempo
 - Pior caso: $O(n^2)$
 - Melhor caso: $O(n)$
 - Caso médio: $O(n)$ (certas condições, próximo slide)
- Espaço
 - Memória extra: $O(n+k)$
 - k listas, mas com n elementos no total

Complexidade

O seguinte resultado é provado em [CLRS]

Assuma:

1. O número de buckets é igual ao número de chaves (i.e., $k = n$).
2. As chaves são uniformemente e independentemente distribuídas.

Então o custo esperado é $O(n)$

Bucket Sort - Considerações

- Vantagens:

- Ordenação em tempo quasi-linear quando k cresce

- Desvantagens

- Muita memória extra para armazenar os buckets

RADIX SORT

Radixsort

- **Radix Sort** é uma classe de algoritmos que usa a representação binária das chaves para a ordenação

Radixsort

- **Radix Sort** é uma classe de algoritmos que usa a representação binária das chaves para a ordenação
- **Idéia Geral:**
 - chaves cujo bit mais à esquerda é 0, vem antes de chaves cujo bit é 1
 - Repetindo-se isso para todos os bits de forma adequada é possível ordenar

Radixsort

- Requer a representação binária da chave
- Como extrair os bits
 - Formas simples: divisão e resto
 - Considerando bits indexados de 0 a n da dir para esq, para extrair o bit i de um número X, temos $(X / 2^i) \% 2$
 - $X = 150 \rightarrow 10010110$
 - $(10010\mathbf{1}10) \rightarrow i = 2 \rightarrow (150/2^2)\%2 = 37\%2 = 1$
 - Formas eficientes em c: and (&) e shift(>>)
 - `bit = x & 00000001; x >> 1;`

Radix Exchange Sort

- Algoritmo analisa os bits da esquerda para a direita
- Funcionamento similar ao do Quicksort, mas a partição é feita comparando-se bits ao invés de chaves
- Chamadas recursivas ordenando os subvetores pelo bit $i-1$

Exemplo

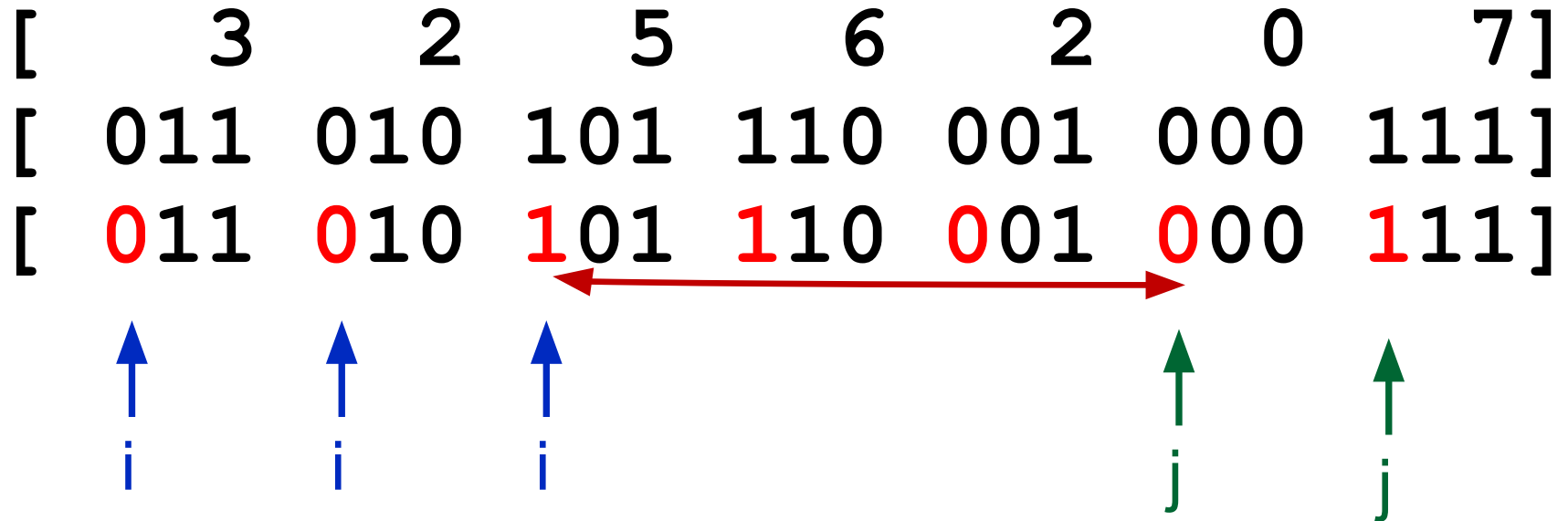
[3 2 5 6 2 0 7]

[011 010 101 110 001 000 111]

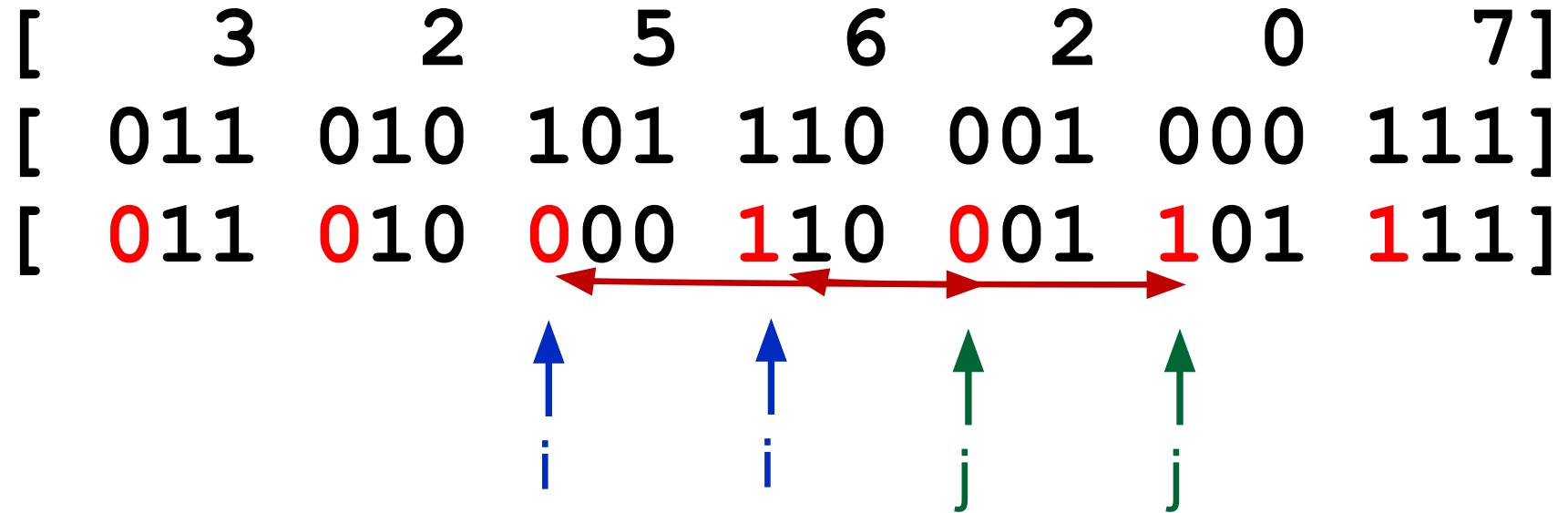
Exemplo

[3	2	5	6	2	0	7]
[011	010	101	110	001	000	111]
[011	010	101	110	001	000	111]

Exemplo



Exemplo



Exemplo

[3	2	5	6	2	0	7]
[011	010	101	110	001	000	111]
[011	010	000	001	110	101	111]

Diagram illustrating a step in a sorting algorithm (likely Quicksort) on an array of binary strings. The array is shown with indices 3 through 7. The pivot is the element at index 5, which is '110'. A vertical red line separates the pivot from the rest of the array. A red double-headed arrow indicates the partitioning process. Below the array, two pairs of arrows (one blue pointing up, one green pointing down) are shown, labeled 'i' and 'j', indicating the current pointers used in the partitioning process.

Exemplo

[3	2	5	6	2	0	7]
[011	010	101	110	001	000	111]
[011	010	101	110	001	000	111]
[011	010	000	001	110	101	111]

Exemplo

[3	2	5	6	2	0	7]
[011	010	101	110	001	000	111]
[011	010	101	110	001	000	111]
[011	010	000	001	110	101	111]
[001	000	010	011			

Exemplo

[3 2 5 6 2 0 7]
[011 010 101 110 001 000 111]
[011 010 101 110 001 000 111]
[011 010 000 001 | 110 101 111]
[001 000 | 010 011]
[000 | 001]
[010 | 011]

Exemplo

[3 2 5 6 2 0 7]
[011 010 101 110 001 000 111]
[011 010 101 110 001 000 111]
[011 010 000 001 | 110 101 111]
[001 000 | 010 011]
[000 | 001]
[010 | 011]
[101 | 110 111]
[110 | 111]

Exemplo

[3	2	5	6	2	0	7]
[011	010	101	110	001	000	111]
[011	010	101	110	001	000	111]
[011	010	000	001	110	101	111]
[001	000	010	011]			
[000	001]					
		[010	011]				
			[101	110	111]		
				[110	111]		
[0	1	2	3	5	6	7]

Radix Exchange Sort

```
quicksortB(int a[], int l, int r, int w) {
    int i = l, j = r;

    if (r <= l || w > 0) return;
    while (j != i) {
        while (digit(a[i], w) == 0 && (i < j)) i++;
        while (digit(a[j], w) == 1 && (j > i)) j--;

        exch(a[i], a[j]);
    }
    if (digit(a[r], w) == 0) j++;
    quicksortB(a, l, j-1, w-1);
    quicksortB(a, j, r, w-1);
}

void sort(Item a[], int l, int r) {
    quicksortB(a, l, r, numbits - 1);
}
```

*Fonte: Algorithms in C
Robert Sedgewick*

Radix Exchange Sort

```
quicksortB(int a[], int l, int r, int w) {  
    int i = l, j = r;
```

```
    if (r <= l || w > 0) return;  
    while (j != i) {  
        while (digit(a[i], w) == 0 && (i < j)) i++;  
        while (digit(a[j], w) == 1 && (j > i)) j--;  
  
        exch(a[i], a[j]);  
    }  
    if (digit(a[r], w) == 0) j++;  
    quicksortB(a, l, j-1, w-1);  
    quicksortB(a, j, r, w-1);  
}
```

Partição
baseada
nos bits

Chamada
recursiva

```
void sort(Item a[], int l, int r) {  
    quicksortB(a, l, r, numbits - 1);  
}
```

*Fonte: Algorithms in C
Robert Sedgewick*

Complexidade

- Considere n o número de elementos e k o número de bits de cada chave
- O *radix exchange sort* faz k passagens pelo vetor de n elementos: $O(n.k)$ comparações de bits.
 - Considerando $k = \log(n)$, temos $O(n.\log(n))$ comparações de bits
- A eficiência do algoritmo depende do custo para se extrair os bits...

Radix Sort

Podemos generalizar o Radix Sort para ordenação de múltiplos campos em ordem lexicográfica

Ordem lexicográfica: ordene pelo campo mais importante, empates serão resolvidos pelos próximos campos sucessivamente

Ordem das palavras no dicionário

Radix Sort

Exemplos

Palavras no dicionário

Datas: (dia, mês, ano)

Múltiplos dígitos (e.g. 3 dígitos)

293: (2,9,3)

71: (0,7,1)

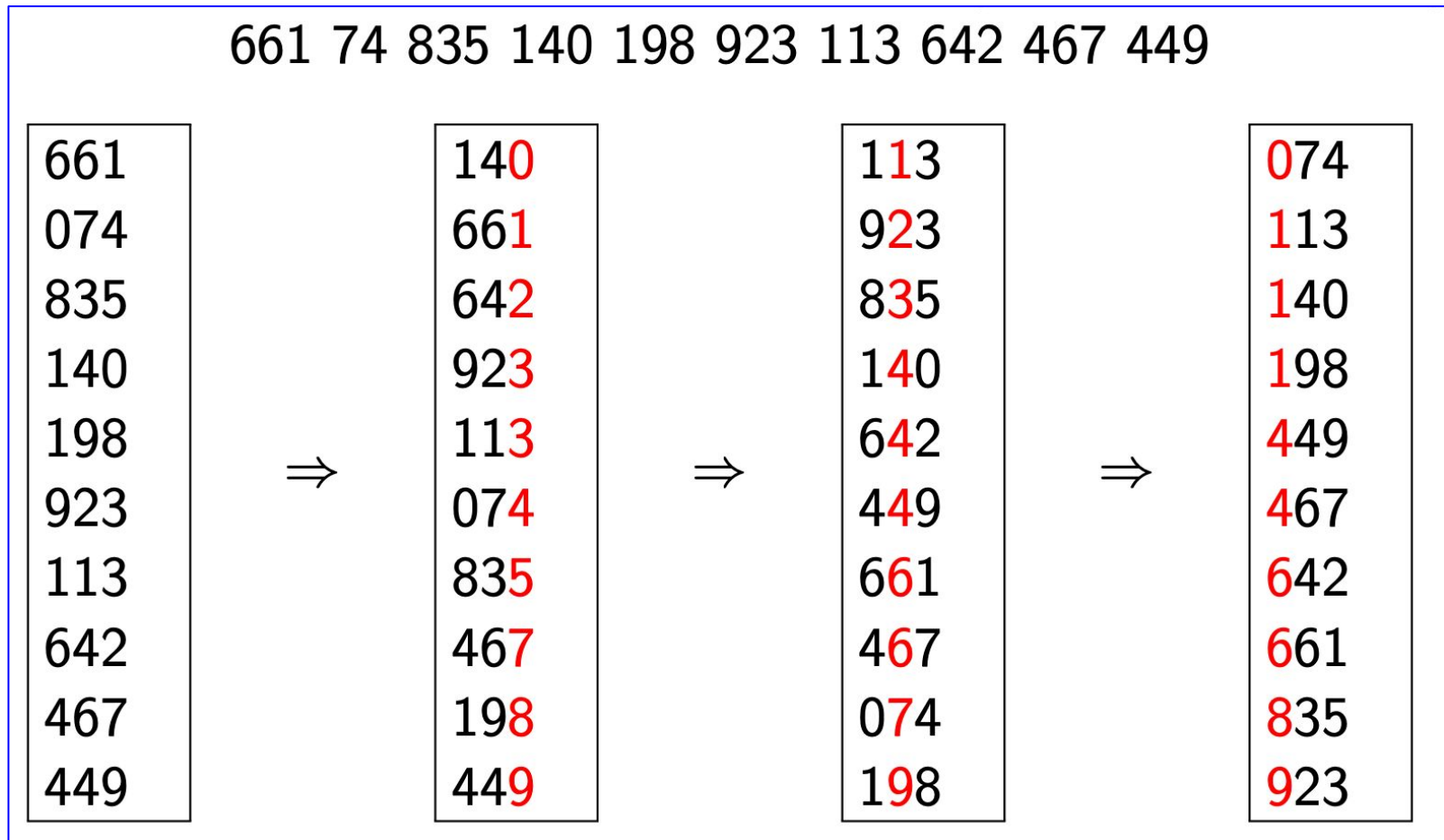
Radix Sort

Ordena cada campo da chave, um por vez

Comece pelo campo menos-significativo

Use um método estável

Radix Sort - exemplo dígitos



Note a importância da estabilidade

Empates são resolvidos e mantidos corretamente

Radix Sort - Complexidade

- n é o número de elementos
- b é o tamanho de cada faixa
 - Cada campo é um número na faixa $0 \dots b-1$
 - Isto é verdade se os números são inteiros na base b
- d é o número de campos
 - E.g., se cada elemento é um número na *base b* com *d dígitos (entre 0 e $b^d - 1$)*
- Cada campo é ordenado usando o Bucket Sort ou o Counting Sort
- Então o tempo de execução: $O(d(n+b))$