Algoritmos de Ordenação Linear

Disciplina: Estruturas de Dados II

Prof. Fermín Alfredo Tang Montané

Curso: Ciência da Computação
Universidade Estadual do Norte Fluminense

Algoritmos de Ordenação

Tempo Linear

- Ordenação por Contagem (*Counting Sort*);
- *Radix Sort;*
- *Bucket Sort;*

Counting Sort

Descrição

- Dado um conjunto de n elementos a serem ordenados, o algoritmo de ordenação por contagem (*Counting Sort*) exige que todos os elementos sejam inteiros. Além disso, exige que todos os valores se encontrem entre 0 e k, um valor máximo conhecido.
- A ordenação tem complexidade $O(n)$.
- O algoritmo por contagem realiza a ordenação de um conjunto de elementos com base na seguinte ideia:
  - Para cada elemento x, determina por contagem, o número de elementos menores que x;
  - Utiliza a informação sobre o número de elementos menores que x, para posicionar o elemento x na posição correta na lista ordenada. p.e. se existem 17 elementos menores que x, deverá colocar x na posição 18.
  - Modificar o princípio de contagem para contemplar o caso em que vários elementos possuem o mesmo valor, evitando que sejam colocados na mesma posição.

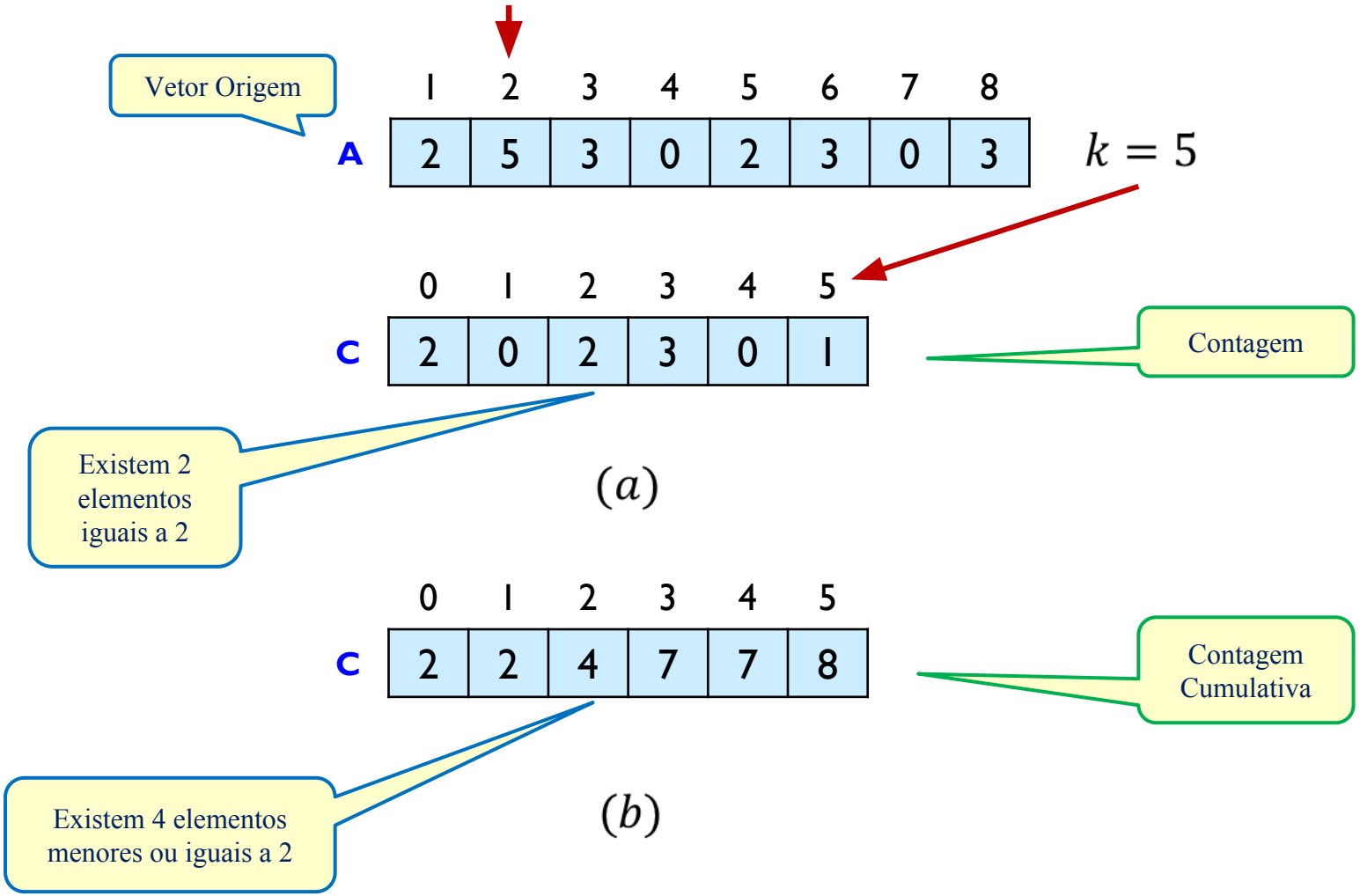
Counting Sort

Algoritmo

- No algoritmo mostrado considera-se que:
  - Os dados de entrada se encontram em um vetor A[1 ... n] com comprimento A.*length*= n;
  - Os dados de saída, ordenados serão colocados em um vetor B[1 ... n] com comprimento B.*length*= n;
  - Os dados sobre contagem de elementos são armazenados em um vetor C[0 ... k] com comprimento C.*length*= $k + 1$

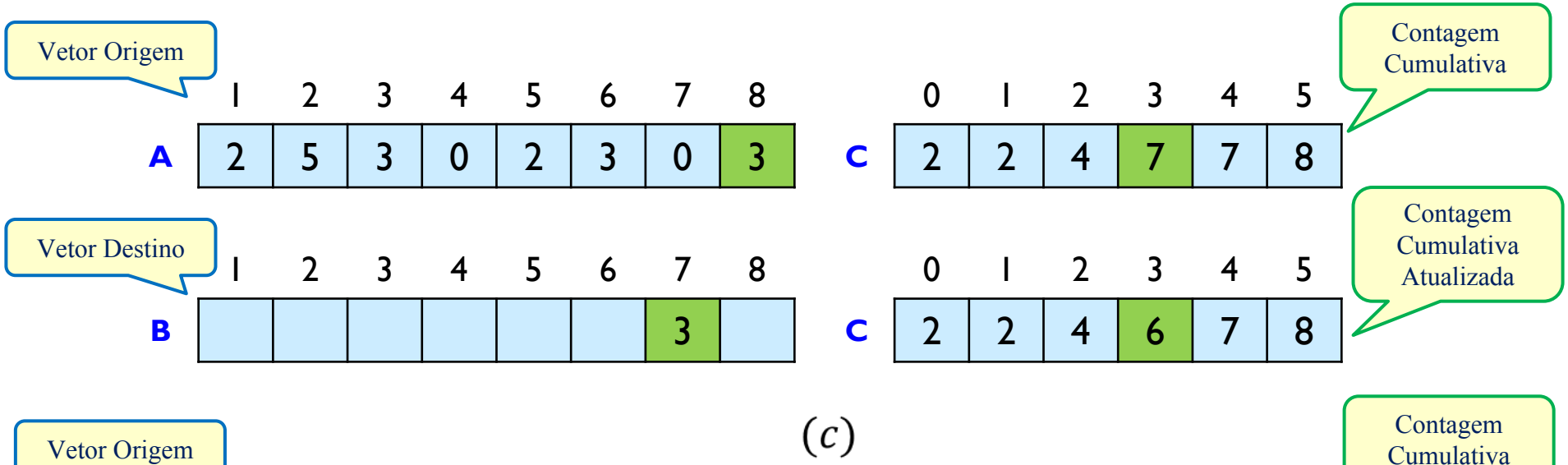
Counting Sort

Exemplo

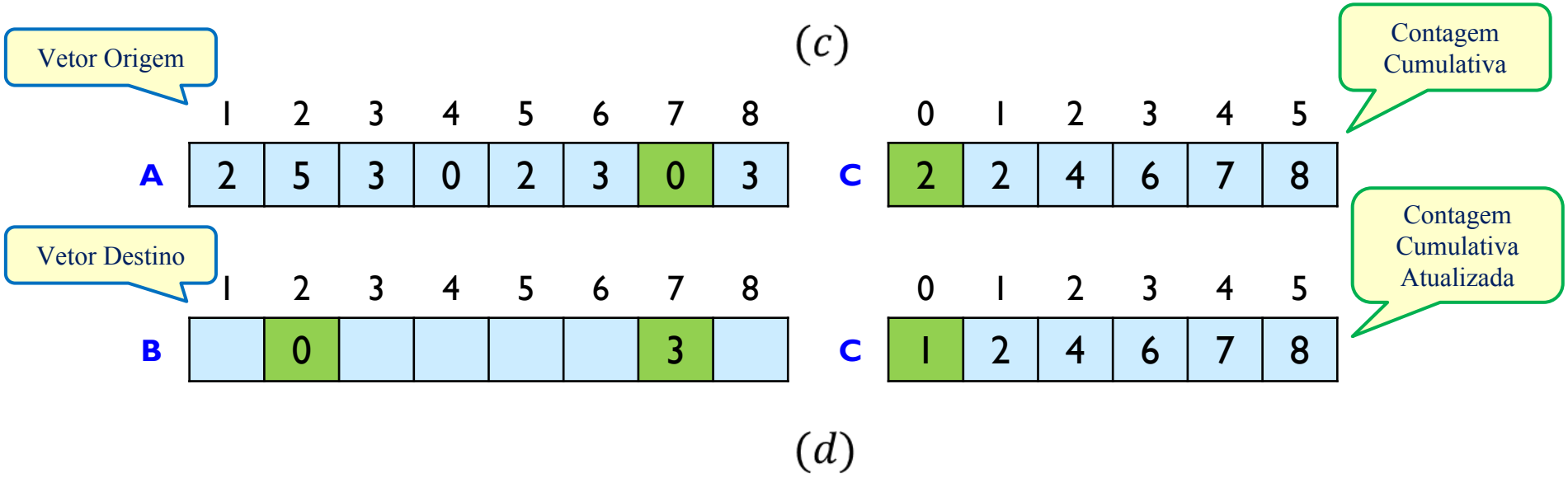


Counting Sort

Exemplo



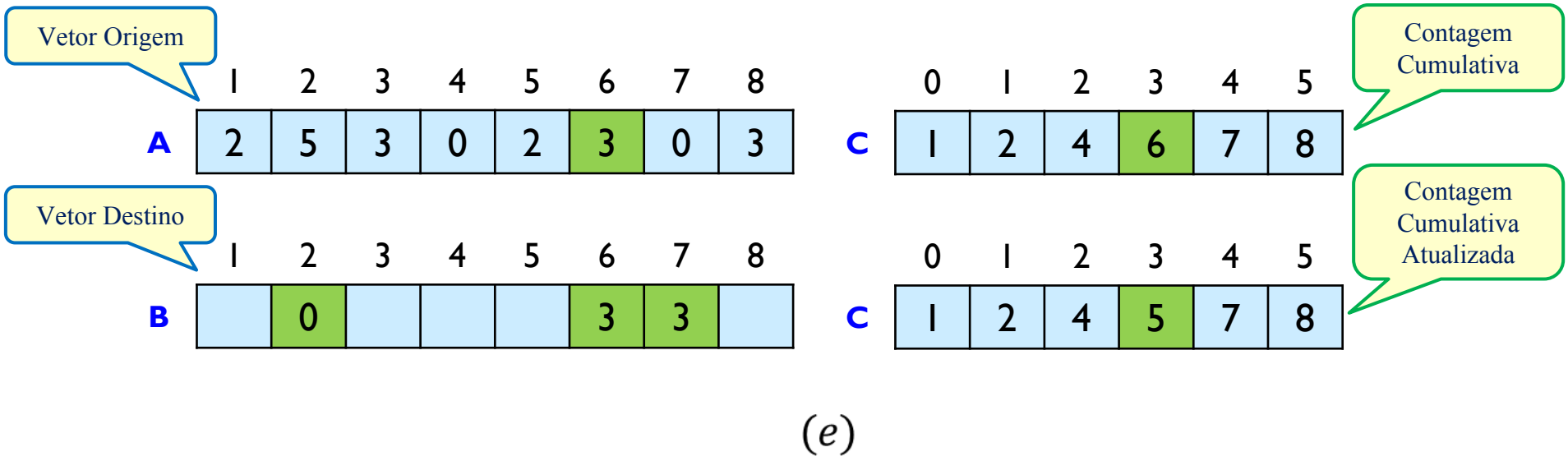
(c)



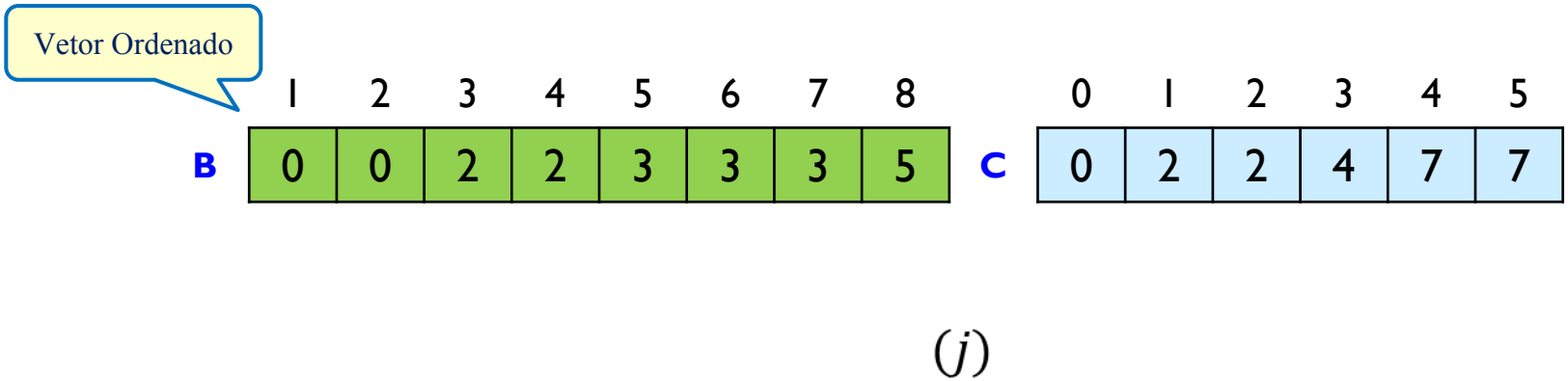
(d)

Counting Sort

Exemplo



(e)



(j)

Counting Sort
Algoritmo

- O algoritmo correspondente ao Counting Sort.

```
COUNTING-SORT (A, B, C)
1 Seja C[0..k] um novo vetor
2 for i=0 to k
3     C[i] = 0
4 for j = 1 to A.length
5     C[A[j]] = C[A[j]] + 1
6  // C[i]  contém o número de elementos iguais a i
7 for i =1 to k
8     C[i] = C[i] + C[i -1]
9 // C[i] contém o número de elementos menores ou iguais a i.
10 for j = A.length downto 1
11   B[C[A[j]]] =  A[j]
12   C[A[j]] = C[A[j]] - 1
```

Counting Sort
Algoritmo Estável

- O algoritmo *Counting Sort* possui a propriedade de ser **Estável**.
- Números com o mesmo valor aparecem no vetor de saída na mesma ordem em que apareciam no vetor de entrada.

Radix Sort

Descrição

- Radix sort é um algoritmo usado por máquinas de ordenação de cartões perfurados, que agora somente existem em museus.
- Os cartões perfurados possuíam 80 colunas e em cada coluna a máquina podia fazer um furo em apenas UM de 12 lugares.
- A máquina podia ser programada para examinar uma dada coluna em cada cartão, em uma pilha de cartões, e distribuir cada cartão em UM de 12 caixas possíveis, de acordo com o lugar do furo na coluna.
- Um operador poderia então coletar os cartões, caixa por caixa, de maneira que os cartões com o primeiro lugar perfurado estivessem no topo, seguidos dos cartões com o segundo lugar perfurado e assim sucessivamente.

Radix Sort

Descrição

- Para dígitos decimais, cada coluna usava somente 10 lugares (0 a 9). Assim, um número com d-dígitos ocuparia d colunas.
- Como a máquina para ordenar cartões somente pode processar uma coluna de cada vez, o problema de ordenar n-cartões com números de d-dígitos requereria um algoritmo de ordenação.

Radix Sort

Descrição

- De maneira intuitiva poderíamos ordenar os números, com base no seu digito mais significativo.
- Depois ordenar cada uma das caixas resultantes de maneira recursiva e combinar as pilhas resultantes em ordem.
- No entanto, este procedimento um número muito grande de pilhas de cartas intermediárias.

Radix Sort

Descrição

- O algoritmo *Radix Sort* resolve o problema de ordenação de maneira contrária a intuição.
- Primeiro ordena com base no dígito menos significativo, produzindo 10 pilhas de cartas, que são colocadas juntas novamente, com aquelas na caixa 0, precedendo aquelas na caixa 1, e estas precedendo aquelas na caixa 2 e assim sucessivamente.
- Depois ordena a nova pilha com base no segundo dígito menos significativo, repetindo o processo de separação e recombinação.
- O processo termina quando a pilha é ordenada com base no dígito d. Ou seja, quando passou pela ordenação de todos os dígitos.
- Dessa forma o algoritmo somente realiza d avaliações (passagens) do conjunto de cartas para conseguir ordenar.

- **Obs.** Para que o algoritmo funcione corretamente, o algoritmo usado para ordenar por dígitos deve ser um **algoritmo estável**. O *counting sort* costuma ser usado como subrotina no *Radix Sort*.

Radix Sort

Algoritmo

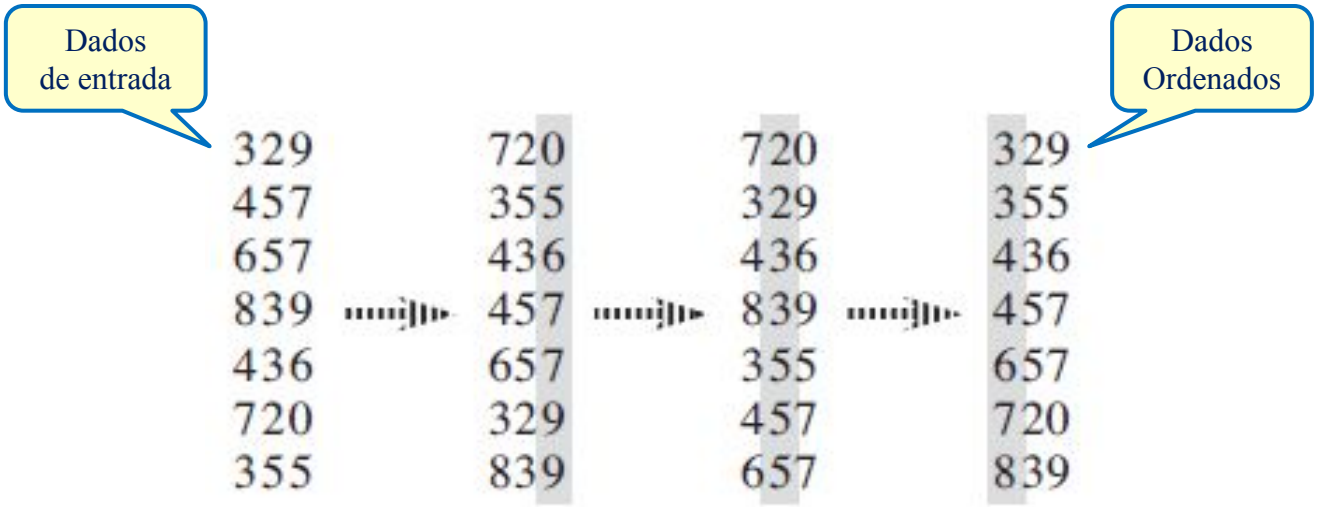
- O algoritmo *Radix Sort* assume que todos os elementos do vetor A, são inteiros e possuem d digitos. Onde 1 é o dígito de menor ordem e d o dígito de maior ordem. O algoritmo é basicamente um loop com d repetições.

```
RADIX-SORT (A, d)
1 for i = 1 to d
2 // Use um algoritmo estável
3     Ordene os elementos do vetor A com base no dígito i
```

Radix Sort

Exemplo

- O exemplo mostra a ordenação de sete números de 3 dígitos usando o algoritmo Radix Sort.
- Os dados de entrada se encontram na coluna mais a esquerda.
- As colunas seguintes mostram a lista de elementos após ordenações sucessivas com base nos dígitos significativos crescentes.
- O sombreamento indica a posição do digito usado para ordenar cada lista.



Radix Sort
Aplicação

- Utiliza-se o algoritmo *Radix sort* para ordenar registros de informação que possuam múltiplos campos chave.
- Por exemplo, considere que desejamos ordenar datas, considerando as três chaves: ano, mês e dia.
- Poderíamos executar um algoritmo de ordenação que utilize uma função de comparação, que dadas duas datas, compara-se os anos, se houver empate, compara-se os meses e se houvesse outro empate compara-se os dias.
- Alternativamente, podemos ordenar a informação três vezes usando uma ordenação estável: primeiro por dia, depois por mês e finalmente por ano.

Bucket Sort

Descrição

- De maneira similar aos algoritmos anteriores, algoritmo *Bucket Sort*, ou ordenação por caixas (ou literalmente baldes), consegue ser rápido porque assume alguma coisa sobre os dados de entrada.
- O algoritmo *Bucket Sort* assume que a entrada de dados é extraída a partir de uma **distribuição uniforme**, e possui complexidade de caso médio $O(n)$.
- O algoritmo *Bucket Sort* assume que os dados de entrada foram gerados mediante um processo aleatório que distribui os elementos de maneira uniforme e independente no intervalo [0, 1).
- Basicamente, isso significa que um dado qualquer tem igual chance de se encontrar em uma porção do intervalo que outro. E que os dados estão distribuídos de forma igualitária ao longo do intervalo.

Bucket Sort

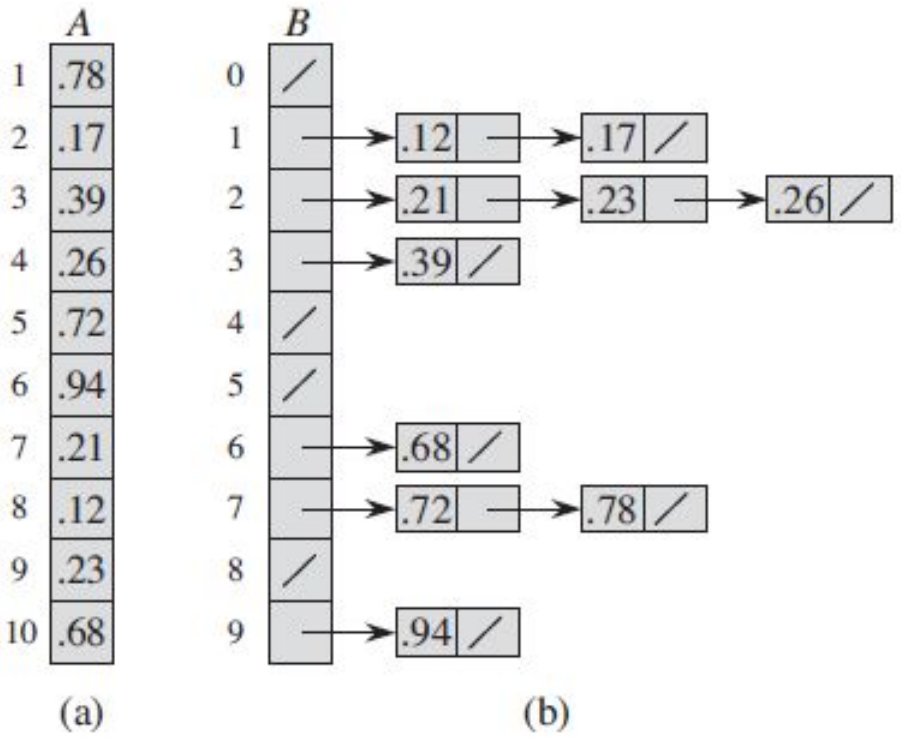
Descrição

- O algoritmo *Bucket Sort* divide o intervalo [0,1) em n subintervalos de igual tamanho, ou *buckets* (caixas ou baldes), e distribui os n dados de entrada entre esses *buckets*.
- Como a entrada de dados está distribuída de maneira uniforme no intervalo [0,1), não se espera que existam muitos números em cada *bucket*. Para realizar a ordenação, o algoritmo simplesmente ordena os números em cada *bucket* e depois percorre os *buckets* em ordem listando cada um dos elementos contidos neles.
- O algoritmo assume que os dados se encontram em um vetor A de n elementos, e que cada elemento $A[i]$ satisfaz $0 \leq A[i] < 1$.
- O algoritmo utiliza um vetor auxiliar $B[0..n-1]$ de listas encadeadas. Cada lista encadeada representa um *bucket*. Entende-se que as operações para manutenção das listas encadeadas estão disponíveis.

Bucket Sort

Exemplo

- A figura ilustra um exemplo de execução *Bucket Sort* para $n = 10$.
- Os dados de entrada se encontram no vetor $A[1..10]$.
- O vetor $B[1..10]$ de listas encadeadas (*buckets*) permite acessar as listas em ordem. Cada lista (*bucket*) contém valores no intervalo $[{}^{i}/_{10}, {}^{(i+1)}/_{10})$



- A saída ordenada é obtida pela concatenação em ordem das listas $B[0] \ldots B[9]$.

Bucket Sort

Algoritmo

- Dados dois elementos $A[i]$ e $A[j]$ tal que $A[i] \leq A[j]$. Com isso temos que $\lfloor nA[i] \rfloor \leq \lfloor nA[i] \rfloor$, sendo que $A[i]$ pode ir no mesmo *bucket* que $A[j]$, ou então em um *bucket* com índice menor.

```
BUCKET-SORT(A)
1  let B[0..n − 1] be a new array
2  n = A.length
3  for i = 0 to n − 1
4       make B[i] an empty list
5  for i = 1 to n
6       insert A[i] into list B[⌊nA[i]⌋]
7  for i = 0 to n − 1
8       sort list B[i] with insertion sort
9  concatenate the lists B[0], B[1], . . . , B[n − 1] together in order
```

Dados de entrada

Cria as listas vazias

Insere elementos nas listas

Ordena cada lista usando Insertion sort

Referências

- Thomas **Cormen**, Charles **Leiserson,** et al.**.** Algoritmos. Teoria e Prática. 2ª Edição. 2002.