Estruturas de Dados 2

Algoritmos de Ordenação em Tempo Linear

Ordenação em Tempo Linear

- ***Algoritmos de Ordenação em Tempo Linear***
  - *Limite Assintótico para Algoritmos de Ordenação baseados em Comparação*
  - *Ordenação em Tempo Linear: Counting Sort*
  - *Algoritmos “in-place” e algoritmos “estáveis”*
  - *Ordenação em Tempo Linear: Radix Sort*
  - *Ordenação em Tempo Linear: Bucket Sort*
  - *Resumo*

Ordenação em Tempo Linear

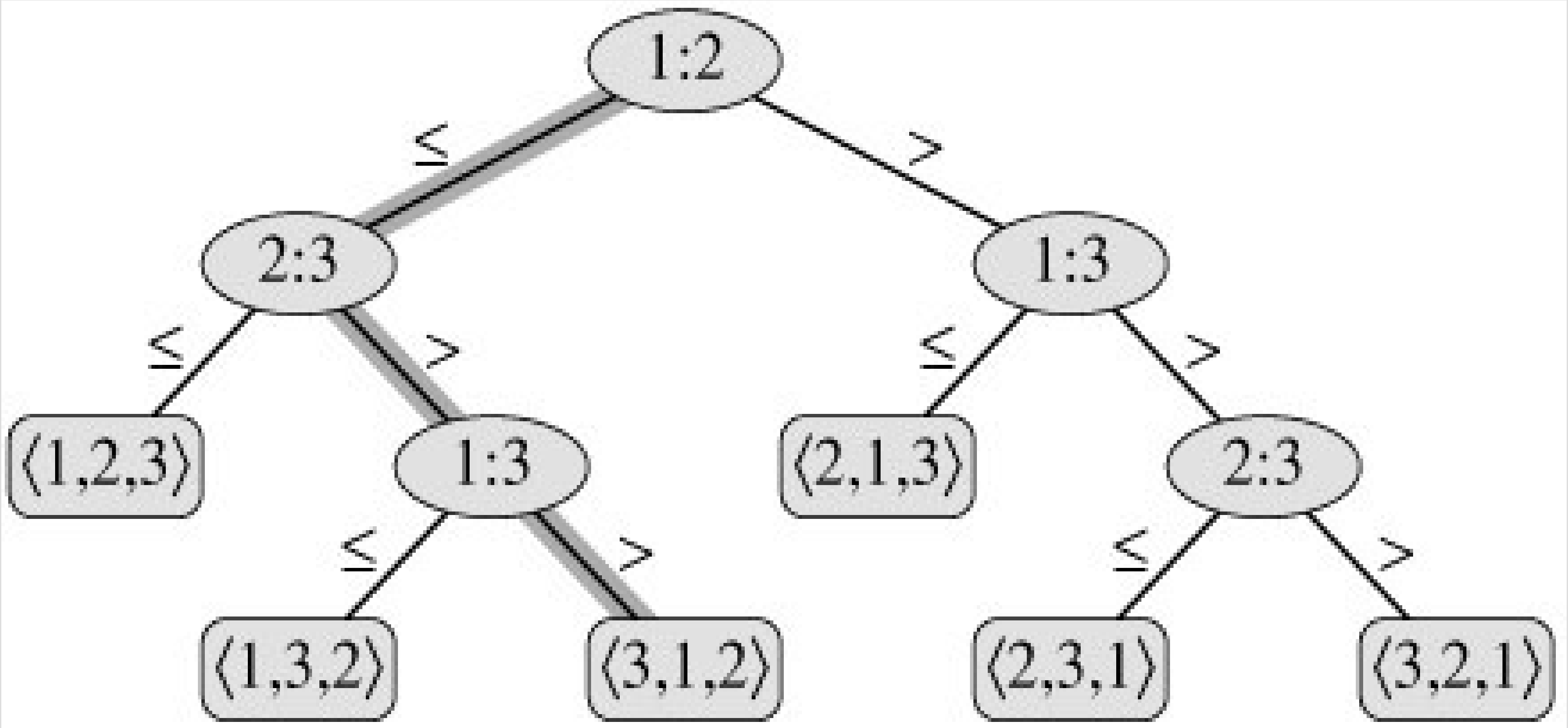
- ***Limite assintótico para Algoritmos de Ordenação baseados em Comparação***
- Qual será o “limite matemático” para a eficiência de algoritmos de ordenação (menor **x** tal que possa existir um algoritmo **O(x)**?)
- Para tentar responder a esta questão, vamos utilizar árvores de decisão para analisar os algoritmos de ordenação estudados até agora (observe que são todos baseados em comparação de valores do vetor...será que existe outra forma?)

Ordenação em Tempo Linear

- ***Limite assintótico para Algoritmos de Ordenação baseados em Comparação***
  - Supondo (sem perda de generalidade) que os valores a serem ordenados são sempre distintos entre si.
  - Construindo uma árvore de decisão em que cada nodo é associado à uma comparação entre dois nodos que possuem os elementos x_i and x_j;
  - Cada nodo folha acaba sendo associado à uma permutação dos elementos do vetor (chegando ao total de *n!* permutações, para um vetor de tamanho *n*)
  - Qualquer algoritmo que deseje ordenar um determinado vetor corresponde à percorrer um caminho desta árvore, da raiz até a folha que corresponde ao vetor ordenado.

Limite inferior para Algoritmos de Ordenação baseados em Comparação

- **N=3 (3!=6 nodos folhas)**



- **Número de Folhas da Árvore de Decisão = n!**

Limite inferior para Algoritmos de Ordenação baseados em Comparação

- **Lemma:**
  - Seja L o número de folhas de uma árvore binária, e h sua altura.
  - Então $L <= 2^h$, and $h >= \lceil lg\ L \rceil$
  - Para um dado número n, $L = n!$, a árvore de decisão para qualquer algoritmo que ordena por comparação de valores tem altura de pelo penos $\lceil lg\ n! \rceil$.

Limite inferior para Algoritmos de Ordenação baseados em Comparação

Teorema:

- Qualquer algoritmo que ordene n elementos por comparação precisa fazer ao menos $\lceil lg\ n! \rceil$, ou seja, aproximadamente $\lceil n\ lg\ n - 1.443\ n \rceil$, comparações no pior caso.

Teorema:

- Qualquer algoritmo que ordene n elementos por comparação precisa fazer, na média $lg\ n!$, ou seja, aproximadamente $n\ lg\ n - 1.443\ n$, comparações.
- **A única diferença é que no pior caso há arredondamento para cima.....**
- **A média não precisa ser um inteiro, mas o pior caso precisa.**

Limite inferior para Algoritmos de Ordenação baseados em Comparação

- Desta forma, fica provado que os algoritmos estudados até agora (O(nlgn)) são os melhores possíveis, portanto não há mais porque se preocupar com isto....(certo???)
- Não necessariamente......
- Mudando nossas premissas, podemos chegar a conclusões inusitadas!!!!
- Esta prova apenas garante que não existe algoritmo “baseado em comparações” melhor que este limite....
- Então, passemos aos algoritmos que NÃO SÃO baseados em comparações....(mas exigem algum outro tipo de conhecimento sobre os dados a serem ordenados, portanto não são tão genéricos como os vistos até agora!!!)

Ordenação em Tempo Linear

• Os seguintes algoritmos de ordenação têm complexidade **O(n)**, onde n é o tamanho do vetor A a ser ordenado:

- ***Counting Sort:*** Os elementos a serem ordenados são números inteiros “pequenos", isto é, **inteiros x onde $x \in O(n)$.**
- ***Radix Sort:*** Os elementos a serem ordenados são números inteiros de **comprimento máximo constante**, isto é, independente de n.
- ***Bucket Sort:*** Os elementos são números reais uniforme-mente distribuídos no intervalo [0::1).

Ordenação em Tempo Linear

- ***Ordenação em Tempo Linear: Counting Sort***
  - Suponha que um vetor A a ser ordenado contenha n números inteiros, todos menores ou iguais a k, (k $\in$ O(n)).
  - O algoritmo "***Counting Sort***" ordena estes n números em tempo O(n + k) (equivalente a O(n)).
  - O algoritmo usa dois vetores auxiliares:

    1 - **C de tamanho k** que guarda em C[i] o número de ocorrências de elementos i em A.

    2 - **B de tamanho n** onde se constrói o vetor ordenado

Ordenação em Tempo Linear

•***Ordenação em Tempo Linear: Counting Sort***

- **CountingSort(A, k)**

▷ **Entrada:** - Vetor A de tamanho n

- o valor k do maior inteiro em A

▷ **Saida:** O vetor A em ordem crescente

para $i := 1$ **até** k **faça** $C[i] := 0$

para $j := 1$ **até** n **faça** $C[A[j]] := C[A[j]] + 1$

para $i := 2$ **até** k **faça** $C[i] := C[i] + C[i-1]$

para $j := n$ **até** 1 **faça**

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

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

retorne(B)

Ordenação em Tempo Linear

- ***Ordenação em Tempo Linear: Counting Sort***
  - O algoritmo **não faz comparações entre elementos de A**.
  - Sua complexidade deve ser medida com base nas outras operações (aritméticas, atribuições, etc.)
  - Claramente, o número de tais operações é uma função em O(n + k), já que temos dois loops simples com n iterações e dois com k iterações.
  - Assim, quando $k \in O(n)$, este algoritmo tem complexidade O(n).
  - Algo de errado com o limite assintótico de (n log n) para ordenação?
  - Simples: este limite (mínimo) só é válido se o algoritmo é baseado em comparações, coisa que o ***Couting*** não é!!!

Ordenação em Tempo Linear

- ***Algoritmos "in-place" e algoritmos "estáveis"***
  - Algoritmos de ordenação podem ou não ser *in-place* ou estáveis.
  - *In-place*: se usa apenas uma quantidade fixa de memória adicional (além da memória necessária para armazenar o vetor a ser ordenado).
  - Exemplos: o Quicksort e o Heapsort são métodos de ordenação *in-place*, já o Mergesort e o Counting Sort não.
  - *Estável*: se a posição relativa de elementos iguais que ocorrem no vetor ordenado permanecem na mesma ordem em que aparecem na entrada (pode ser relevante.....).
  - Exemplos: o Counting Sort e o Mergesort são exemplos de métodos estáveis (desde que certos cuidados sejam tomados na implementação). O Quicksort e o Heapsort não são estáveis.

Ordenação em Tempo Linear

- ***Ordenação em Tempo Linear: Radix Sort***
  - O algoritmo Radix Sort ordena um vetor A de n números inteiros com um número constante d de dígitos, através de ordenações parciais dígito a dígito.
  - Esse algoritmo surgiu no contexto de ordenação de cartões perfurados; a máquina ordenadora so era capaz de ordenar os cartões segundo um de seus dígitos.
  - Poderamos então ordenar os números iniciando a ordenação a partir do dígito mais significativo i, mas, para prosseguir assim, teríamos que separar os elementos da entrada em subconjuntos com o mesmo valor no dígito i.
  - Podemos também ordenar os números ordenando-os segundo cada um de seus dígitos, começando pelo menos significativo.

Ordenação em Tempo Linear

- ***Ordenação em Tempo Linear: Radix Sort***
  - Ordenar um vetor A contendo inteiros de no máximo d dígitos!

RadixSort (A, d)

1. **para** $i := 1$ **até** d **faça**
2. ordene os elementos de A pelo i-ésimo dígito usando um método **estável**

Ordenação em Tempo Linear

- ***Ordenação em Tempo Linear: Radix Sort (Exemplo)***

```
329
457
657
839
436
720
355
  ^
```

Ordenação em Tempo Linear

- ***Ordenação em Tempo Linear: Radix Sort (Exemplo)***

```
329         720
457         355
657         436
839  ->     457
436         657
720         329
355         839
               ^
```

Ordenação em Tempo Linear

- ***Ordenação em Tempo Linear: Radix Sort (Exemplo)***

```
329      720      720
457      355      329
657      436      436
839  ->  457  ->  839
436      657      355
720      329      457
355      839      657
                  ^
```

Ordenação em Tempo Linear

- ***Ordenação em Tempo Linear: Radix Sort (Exemplo)***

```
329        720        720        329
457        355        329        355
657        436        436        436
839  ->    457  ->    839  ->    457
436        657        355        657
720        329        457        720
355        839        657        839
                                 Ordenado!
```

Ordenação em Tempo Linear

- ***Radix Sort*** (Corretude)
  - O seguinte argumento indutivo garante a corretude do algoritmo:
  - Por **hipótese de indução**, assumimos que **os números estão ordenados com relação aos i - 1 dígitos menos significativos**.
  - Ordenando os números com relação ao **i-ésimo dígito**, com um algoritmo estável, teremos então os números ordenados com relação aos i dígitos menos significativos, pois:
    - para dois números com o i-ésimo dígito distintos, o de menor valor no dígito estará antes do de maior valor;
    - e se ambos possuem o i -ésimo dígito igual, então a ordem dos dois também estará correta pois utilizamos um método de ordenação estável e, por hipótese de indução, os dois elementos já estavam ordenados segundo os i-1dígitos menos significativos.

Ordenação em Tempo Linear

- ***Radix Sort*** (Complexidade)
  - A complexidade do Radix Sort depende da complexidade do algoritmo estável usado para ordenar cada dígito dos elementos.
  - Se essa complexidade estiver em $\Theta(f(n))$, obtemos uma complexidade total de $\Theta(d\ f(n))$ para o *Radix Sort*.
  - Como supomos d constante, a complexidade reduz-se para $\Theta(f(n))$.
  - Se o algoritmo estável for, por exemplo, o *Counting Sort*, obtemos a complexidade $\Theta(n + k)$.
  - Supondo $k \in O(n)$, resulta numa complexidade *linear* em n.

Ordenação em Tempo Linear

- ***Radix Sort*** (Complexidade)
  - Em contraste, um algoritmo por comparação como o MergeSort teria complexidade n log n.
  - Assim, o Radix Sort é mais vantajoso que o MergeSort quando d < log n, ou seja, o número de dígitos for menor que log n.
  - Se considerarmos que n também é um limite superior para o maior valor a ser ordenado, então O(log n) é uma estimativa para o tamanho, em dígitos, desse número.
  - Isso significa que não há diferença significativa entre o desempenho do MergeSort e do Radix Sort?

Ordenação em Tempo Linear

- ***Radix Sort*** (Complexidade)
  - Mas qual a vantagem de se usar o Radix Sort ????
  - Tomemos o seguinte exemplo: suponha que desejemos ordenar um conjunto de 2^{20} números de 64 bits. Então, o MergeSort faria cerca de 20×2^{20} comparações e usaria um vetor auxiliar de tamanho 2^{20} .
  - Se interpretarmos cada número do conjunto como tendo 4 dígitos em base 2^{16}, e usarmos o Radix Sort com o Counting Sort como método estável, a complexidade de tempo seria da ordem de $4 \times (2^{20} + 2^{16})$ operações, uma redução substancial.
  - Mas, note que utilizamos dois vetores auxiliares, um de tamanho 2^{16} e outro de tamanho 2^{20}

Ordenação em Tempo Linear

- ***Radix Sort*** (Complexidade)
  - Tomemos o seguinte exemplo: suponha que desejemos ordenar um conjunto de 10.000 números de 5 dígitos. Então, o MergeSort faria cerca de 10.000 x log 10.000 = 10.000 x 4 = 40.000 comparações e usaria um vetor auxiliar de tamanho 10.000 .
  - Usando o Radix Sort com o Counting Sort como método estável, a complexidade de tempo seria O(n+k), com n+k= 10.000 + 99.999 =~100.000, totalizando então 5 x 100.000 operações.....(pior!!!!)

Ordenação em Tempo Linear

- ***Radix Sort*** (Complexidade)
  - O nome Radix Sort vêm da base (em inglês radix) em que interpretamos os dígitos.
  - Veja que se o uso de memória auxiliar for muito limitado, então o melhor mesmo é usar um algoritmo de ordenação por comparação *in-place*.
  - Note que e possível usar o Radix Sort para ordenar outros tipos de elementos, como datas, palavras em ordem lexicográfica e qualquer outro tipo que possa ser visto como uma tupla ordenada de itens comparáveis.

Ordenação em Tempo Linear

- ***Ordenação em Tempo Linear: Bucket Sort***
  - O algoritmo **Bucket Sort** possui tempo linear, desde que os valores a serem ordenados sejam distribuídos uniformemente sobre o intervalo [0, 1).
  - O **Bucket Sort** (**Ordenação por Balde**?) divide o intervalo [0, 1) em n sub-intervalos iguais, denominados buckets (baldes), e então distribui os n números reais nos n buckets. Como a entrada é composta por dados distribuídos uniformemente, espera-se que cada balde possua, ao final deste processo, um número equivalente de elementos (usualmente 1).

Ordenação em Tempo Linear

- ***Ordenação em Tempo Linear: Bucket Sort***
  - Para obter o resultado, basta ordenar os elementos em cada bucket e então apresentá-los em ordem.
  - O pseudo-código assume que a entrada é um vetor A de n elementos, onde cada elemento A[i] satisfaz $0 \le A[i] < 1$.
  - É utilizado um vetor auxiliar B[0 .. n] de listas encadeadas alocadas dinamicamente, que são utilizadas pelo algoritmo InsertionSort para criar os buckets de forma mais eficiente.
  - O resultado é a concatenação das listas, na ordem dos buckets.

Ordenação em Tempo Linear

- ***Ordenação em Tempo Linear: Bucket Sort***

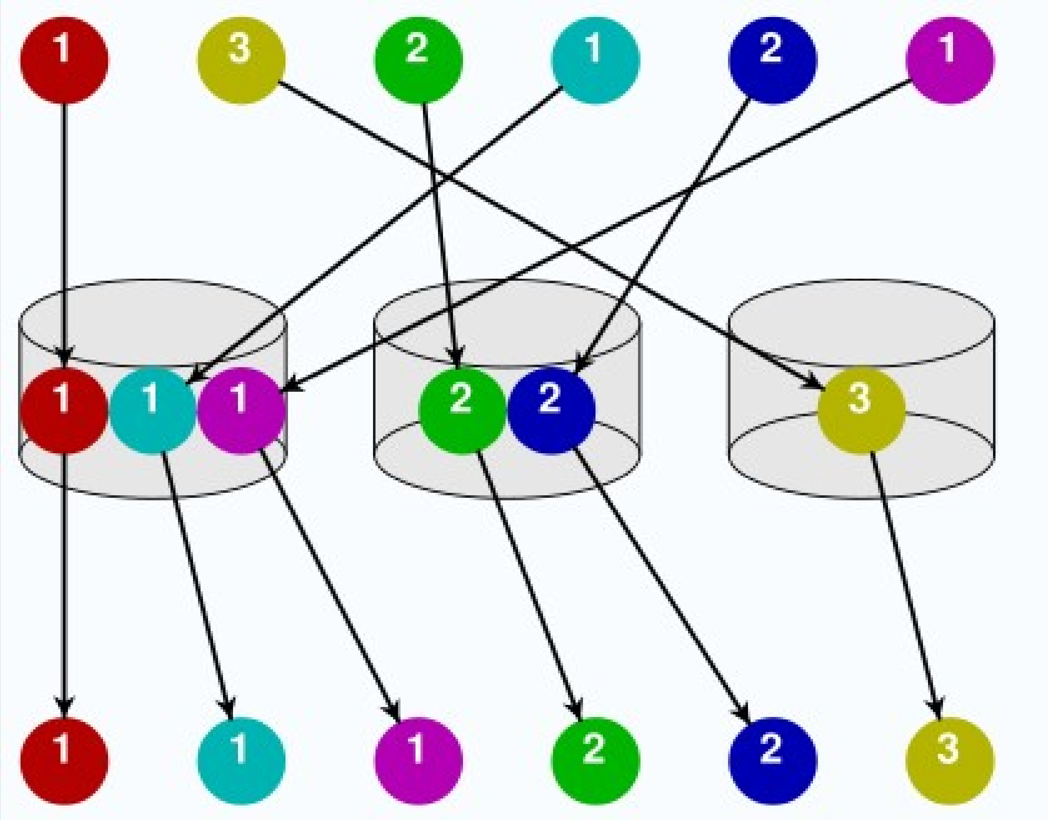
BucketSort(A)

1. $n :=$ comprimento de A
2. **para** $i := 1$ **até** n **faça**
3. insira $A[i]$ na lista ligada $B[\lfloor nA[i] \rfloor]$
4. **para** $i := 0$ **até** $n - 1$ **faça**
5. ordene a lista $B[i]$ com *Insertion Sort*
6. Concatene as listas $B[0], B[1], \ldots, B[n-1]$ nessa ordem.

Ordenação em Tempo Linear

- ***Ordenação em Tempo Linear: Bucket Sort***

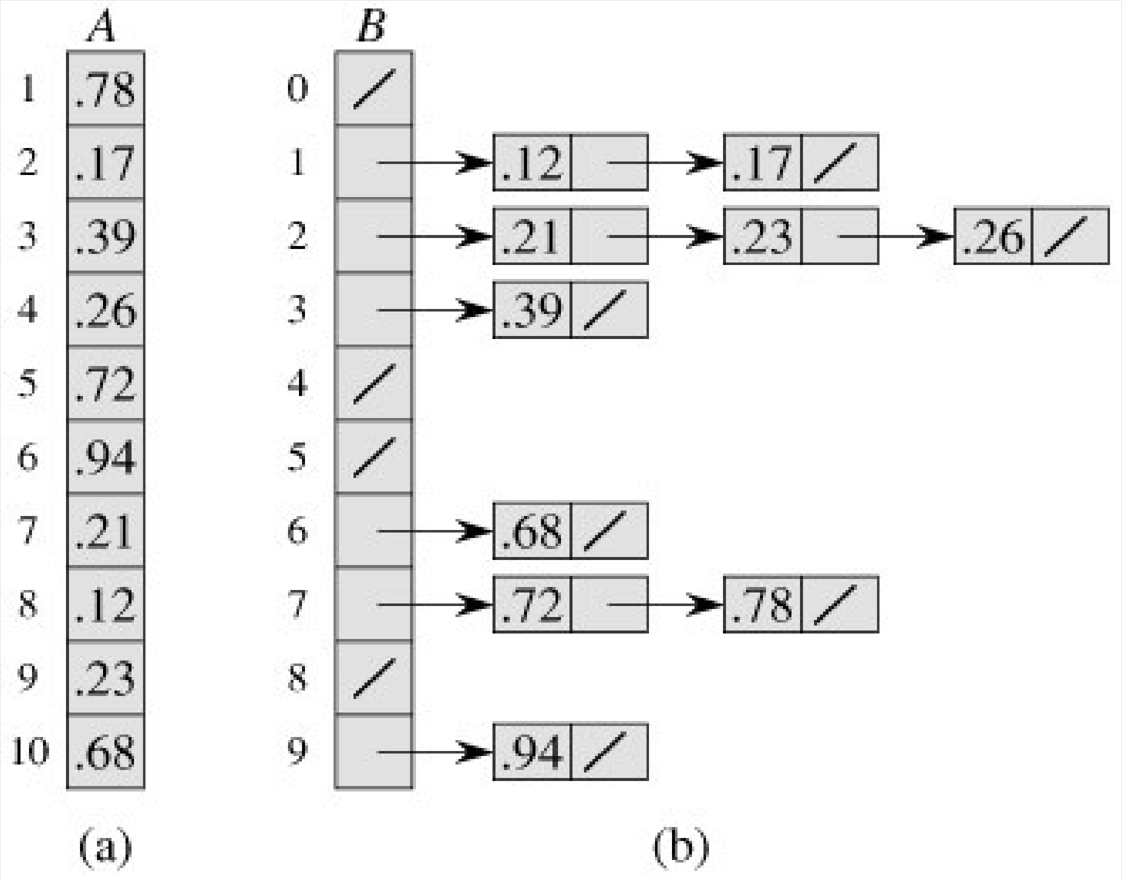
BucketSort – Exemplos (Wikipedia)



Ordenação em Tempo Linear

- ***Ordenação em Tempo Linear: Bucket Sort***

BucketSort – Exemplos (Cormen)



Ordenação em Tempo Linear

- ***Ordenação em Tempo Linear: Bucket Sort***

BucketSort – Corretude

- Considere 2 elementos A[i] e A[j].
- Assuma (sem perda de generalidade) que A[i] ≤ A[j].
- Como $\lfloor n\, A[i] \rfloor$ $\lfloor n\, A[j] \rfloor$, o elemento A[i] está ou no mesmo bucket que A[j], ou em um *Bucket* de menor índice
- Se A[i] e A[j] estão no mesmo *Bucket*, o loop nas linha 4-5 os coloca na ordem correta.
- Se A[i] e A[j] estão em *Buckets* diferentes, a linha 6 os coloca na ordem correta.
- Portanto, **Bucket Sort** funciona!

Ordenação em Tempo Linear
Resumo

- O ***Limite Assintótico Superior*** para ***Algoritmos de Ordenação*** baseados em ***Comparações*** é **O(n lg n)**

Ordenação em Tempo Linear
Resumo

- O ***Limite Assintótico Superior*** para ***Algoritmos de Ordenação*** baseados em ***Comparações*** é **O(n lg n)**
- Mas existem algoritmos de ordenação em tempo linear:

Ordenação em Tempo Linear
Resumo

- O ***Limite Assintótico Superior*** para ***Algoritmos de Ordenação*** baseados em ***Comparações*** é **O(n lg n)**
- Mas existem algoritmos de ordenação em tempo linear:
  - **Counting Sort**: ***O(n+k)***, onde se os elementos a serem ordenados são números inteiros “pequenos", sendo k o maior inteiro presente na entrada.

Ordenação em Tempo Linear
Resumo

- O ***Limite Assintótico Superior*** para ***Algoritmos de Ordenação*** baseados em ***Comparações*** é **O(n lg n)**
- Mas existem algoritmos de ordenação em tempo linear:
  - **Counting Sort**: ***O(n+k)***, onde se os elementos a serem ordenados são números inteiros “pequenos", sendo k o maior inteiro presente na entrada.
  - **Radix Sort**: Ordena um vetor de n números de d dígitos em tempo linear, usando ordenação estável. Se usar CountingSort, também é ***O(n+k)***.

Ordenação em Tempo Linear
Resumo

- O ***Limite Assintótico Superior*** para ***Algoritmos de Ordenação*** baseados em ***Comparações*** é **O(n lg n)**
- Mas existem algoritmos de ordenação em tempo linear:
  - **Counting Sort**: ***O(n+k)***, onde se os elementos a serem ordenados são números inteiros “pequenos", sendo k o maior inteiro presente na entrada.
  - **Radix Sort**: Ordena um vetor de n números de d dígitos em tempo linear, usando ordenação estável. Se usar CountingSort, também é ***O(n+k)***.
  - **Bucket Sort**: Ordena n números uniformemente distribuídos na faixa [0-1) em tempo **médio *O(n)* .**

Ordenação em Tempo Linear - Agradecimentos

- Estes slides são baseados no material disponibilizado pelos profs. **Cid Carvalho de Souza** e **Cândida Nunes da Silva**, da **UNICAMP**.

- Qualquer incorretude é, entretanto, de inteira responsabilidade do prof. João Alberto Fabro, da UTFPR.

Ordenação em Tempo Linear - Exercícios

- Implementar os algoritmos ***Counting Sort***, ***Radix Sort*** e ***Bucket Sort***, realizando experimentos que avaliem a quantidade de operações (comparações) e o tempo de execução para:
- 10 vetores com 1.000, 10.000, 100.000 e um milhão de números inteiros entre 0 e 99.999 (totalizando 40 execuções)
- Comparar estes algoritmos com o QuickSort para os mesmos vetores