Projeto e Análise de Algoritmos*

Cota inferior para ordenação e ordenação em tempo linear

Segundo Semestre de 2019

*Criado por C. de Souza, C. da Silva, O. Lee, F. Miyazawa et al.

A maior parte deste conjunto de slides foi inicialmente preparada por Cid Carvalho de Souza e Cândida Nunes da Silva para cursos de Análise de Algoritmos. Além desse material, diversos conteúdos foram adicionados ou incorporados por outros professores, em especial por Orlando Lee e por Flávio Keidi Miyazawa. Os slides usados nessa disciplina são uma junção dos materiais didáticos gentilmente cedidos por esses professores e contêm algumas modificações, que podem ter introduzido erros.

O conjunto de slides de cada unidade do curso será disponibilizado como guia de estudos e deve ser usado unicamente para revisar as aulas. Para estudar e praticar, leia o livro-texto indicado e resolva os exercícios sugeridos.

Lehilton

Agradecimentos (Cid e Cândida)

- Várias pessoas contribuíram direta ou indiretamente com a preparação deste material.
- Algumas destas pessoas cederam gentilmente seus arquivos digitais enquanto outras cederam gentilmente o seu tempo fazendo correções e dando sugestões.
- Uma lista destes "colaboradores" (em ordem alfabética) é dada abaixo:
  - Célia Picinin de Mello
  - Flávio Keidi Miyazawa
  - José Coelho de Pina
  - Orlando Lee
  - Paulo Feofiloff
  - Pedro Rezende
  - Ricardo Dahab
  - Zanoni Dias

Cota inferior para ordenação

O problema da ordenação - cota inferior

- Estudamos diversos algoritmos para o problema da ordenação.
- Todos eles têm algo em comum: usam somente comparações entre dois elementos do conjunto a ser ordenado para definir a posição relativa desses elementos.
- Isto é, o resultado da comparação de x_i com x_j, $i \neq j$, define se x_i será posicionado antes ou depois de x_j no conjunto ordenado.
- Todos os algoritmos dão uma cota superior para o número de comparações efetuadas por um algoritmo que resolva o problema da ordenação.
- A menor cota superior é dada pelos algoritmos MERGE-SORT e o HEAP-SORT, que efetuam $\Theta(n \log n)$ comparações no pior caso.

O problema da ordenação - cota inferior

- Será que é possível projetar um algoritmo de ordenação baseado em comparações ainda mais eficiente?
- Veremos a seguir que não!
- É possível provar que qualquer algoritmo que ordena n elementos baseado apenas em comparações de elementos efetua **no mínimo** $\Omega(n \log n)$ comparações no pior caso.
- Para demonstrar esse fato, vamos representar os algoritmos de ordenação em um modelo computacional abstrato, denominado árvore (binária) de decisão.

Árvores de Decisão - Modelo Abstrato

- Os nós internos de uma árvore de decisão representam comparações feitas pelo algoritmo.
- As subárvores de cada nó interno representam possibilidades de continuidade das ações do algoritmo após a comparação.
- No caso das árvores binárias de decisão, cada nó possui apenas duas subárvores. Tipicamente, as duas subárvores representam os caminhos a serem seguidos conforme o resultado (verdadeiro ou falso) da comparação efetuada.
- As folhas são as respostas possíveis do algoritmo após as decisões tomadas ao longo dos caminhos da raiz até as folhas.

Árvores de decisão para o problema da ordenação

- Considere a seguinte definição alternativa do problema da ordenação:

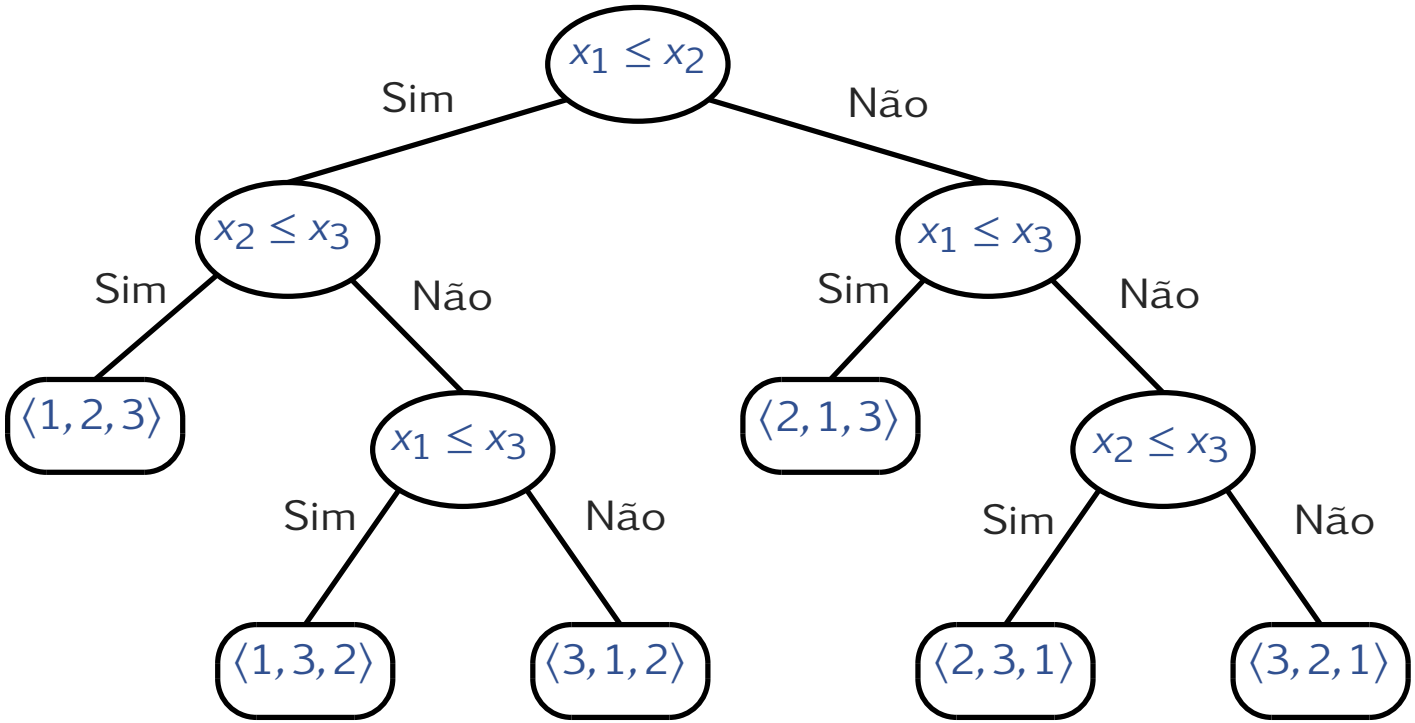
Problema da Ordenação:

Dado um conjunto de n inteiros $x_1, x_2, \ldots, x_n$, encontre uma permutação p dos índices $1 \leq i \leq n$ tal que $x_{p(1)} \leq x_{p(2)} \leq \ldots \leq x_{p(n)}$.

- É possível representar um algoritmo para o problema da ordenação através de uma árvore de decisão da seguinte forma:
  - Os nós internos representam comparações entre dois elementos do conjunto, digamos $x_i \leq x_j$.
  - As ramificações representam os possíveis resultados da comparação: verdadeiro se $x_i \leq x_j$, ou falso se $x_i > x_j$.
  - As folhas representam possíveis soluções: as diferentes permutações dos n índices.

Árvores de Decisão para o Problema da Ordenação

Veja a árvore de decisão que representa o comportamento do *Insertion Sort* para um conjunto de 3 elementos:



Árvores de decisão para o problema da ordenação

- Ao representarmos um algoritmo de ordenação qualquer baseado em comparações por uma árvore binária de decisão, todas as permutações de n elementos devem ser possíveis soluções.
- Assim, a árvore binária de decisão deve ter pelo menos $n!$ folhas, podendo ter mais (nada impede que duas sequências distintas de decisões terminem no mesmo resultado).
- O caminho mais longo da raiz a uma folha representa o pior caso de execução do algoritmo.
- A altura mínima de uma árvore binária de decisão com pelo menos $n!$ folhas fornece o número mínimo de comparações que o melhor algoritmo de ordenação baseado em comparações deve efetuar.

Cota inferior

- Qual a altura mínima, em função de n, de uma árvore binária de decisão com pelo menos $n!$ folhas?
- Uma árvore binária de decisão T com altura h tem, no máximo, 2^h folhas.
- Portanto, se T tem pelo menos $n!$ folhas, então $2^h \geq n!$, ou seja, $h \geq \log_2 n!$.
- Mas,

$$\begin{aligned}
\log_2 n! &= \sum_{i=1}^{n} \log i \\
&\geq \sum_{i=\lceil n/2 \rceil}^{n} \log i \\
&\geq \sum_{i=\lceil n/2 \rceil}^{n} \log n/2 \\
&\geq (n/2 - 1) \log n/2 \\
&= n/2 \log n - n/2 - \log n + 1 \\
&\geq n/4 \log n, \text{ para } n \geq 16.
\end{aligned}$$

- Então, $h \in \Omega(n \log n)$.

Outro jeito

Devemos ter $n! \leq 2^h$, ou seja $\lg n! \leq h$.

Temos que

$$(n!)^2 = \prod_{i=0}^{n-1} (n-i)(i+1) \geq \prod_{i=1}^{n} n = n^n$$

Portanto,

$$h \geq \lg(n!) \geq \frac{1}{2} n \lg n.$$

Conclusão

- Provamos então que $\Omega(n \log n)$ é uma cota inferior para o problema da ordenação.
- Portanto, os algoritmos *Mergesort* e *Heapsort* são algoritmos ótimos.
- Veremos depois algoritmos lineares para ordenação, ou seja, que têm complexidade $O(n)$. (Como???)

Cotas inferiores de problemas

- Em geral é muito difícil provar cotas inferiores não triviais de um problema.

  Um problema com entrada de tamanho n tem como cota inferior trivial $\Omega(n)$.
- São pouquíssimos problemas para os quais se conhece uma cota inferior que coincide com a cota superior.
- Um deles é o problema da ordenação.
- Veremos mais dois exemplos: busca em um vetor ordenado e o problema de encontrar o máximo.

Busca em vetor ordenado

Dado um vetor crescente $A[p \ldots r]$ e um elemento x, devolver um índice i tal que $A[i] = x$ ou -1 se tal índice não existe.

```
BUSCA-BINÁRIA(A, p, r, x)
1   se p ≤ r
2      então q ← ⌊(p + r)/2⌋
3            se A[q] > x
4               então devolva BUSCA-BINÁRIA(A, p, q − 1, x)
5            se A[q] < x
6               então devolva BUSCA-BINÁRIA(A, q + 1, r, x)
7            devolva q ▷ A[q] = x
8      senão
9         devolva −1
```

Número de comparações: $O(\lg n)$.

Busca em vetor ordenado

- É possível projetar um algoritmo mais rápido?
- Não, se o algoritmo se baseia em comparações do tipo $A[i] < x$, $A[i] > x$ ou $A[i] = x$.
- A cota inferior do número de comparações para o problema da busca em vetor ordenado é $\Omega(\lg n)$.
- Pode-se provar isso usando o modelo de árvore de decisão.

Cota inferior

- Todo algoritmo para o problema da busca em vetor ordenado baseado em comparações pode ser representado através de uma árvore de decisão.
- Cada nó interno corresponde a uma comparação com o elemento procurado x.
- As ramificações correspondem ao resultado da comparação.
- As folhas correspondem às possíveis respostas do algoritmo. Então tal árvore deve ter pelo menos $n + 1$ folhas.
- Logo, a altura da árvore é pelo menos $\Omega(\lg n)$.

Problema do Máximo

Problema:
Encontrar o maior elemento de um vetor $A[1 \ldots n]$.

- Existe um algoritmo que faz o serviço com $n - 1$ comparações.
- Existe um algoritmo que faz menos comparações?
- Não, se o algoritmo é baseado em comparações.
- Considere um algoritmo genérico baseado em comparações que resolve o problema.
  Que "cara" ele tem?

Máximo

O algoritmo consiste, no fundo, na determinação de uma coleção $\mathcal{A}$ de pares (i,j) de elementos distintos em $\{1,\ldots,n\}$

- Um par (i,j) é um *arco* tal que $A[i] < A[j]$
- O algoritmo deve continuar enquanto existir **mais** de um "sorvedouro".

Eis o paradigma de um algoritmo baseado em comparações:

```
MÁXIMO(A, n)
1   A ← ∅
2   enquanto A "não possui sorvedouro único" faça
3       Escolha índice i e j em {1,...,n}
4       se A[i] < A[j]
5           então A ← A ∪ (i,j)
6           senão A ← A ∪ (j,i)
7   devolva A
```

Conclusão

Qualquer conjunto $\mathcal{A}$ devolvido pelo método contém uma "árvore enraizada" e portanto contém pelo menos $n - 1$ arcos.

Assim, qualquer algoritmo baseado em comparações que encontra o maior elemento de um vetor $A[1 \ldots n]$ faz **pelo menos** $n - 1$ comparações.

Ordenação em Tempo Linear

Algoritmos lineares para ordenação

Os seguintes algoritmos de ordenação têm complexidade $O(n)$:

- Counting Sort: Elementos são números inteiros "pequenos"; mais precisamente, inteiros $x \in O(n)$.
- Radix Sort: Elementos são números inteiros de comprimento máximo constante, isto é, independente de n.
- Bucket Sort: Elementos são números reais uniformemente distribuídos no intervalo $[0..1)$.

Counting Sort

- Considere o problema de ordenar um vetor $A[1 \ldots n]$ de inteiros quando se sabe que todos os inteiros estão no intervalo entre 0 e k.
- Podemos ordenar o vetor simplesmente contando, para cada inteiro i no vetor, quantos elementos do vetor são menores que i.
- É exatamente o que faz o algoritmo *Counting Sort*.

Counting Sort

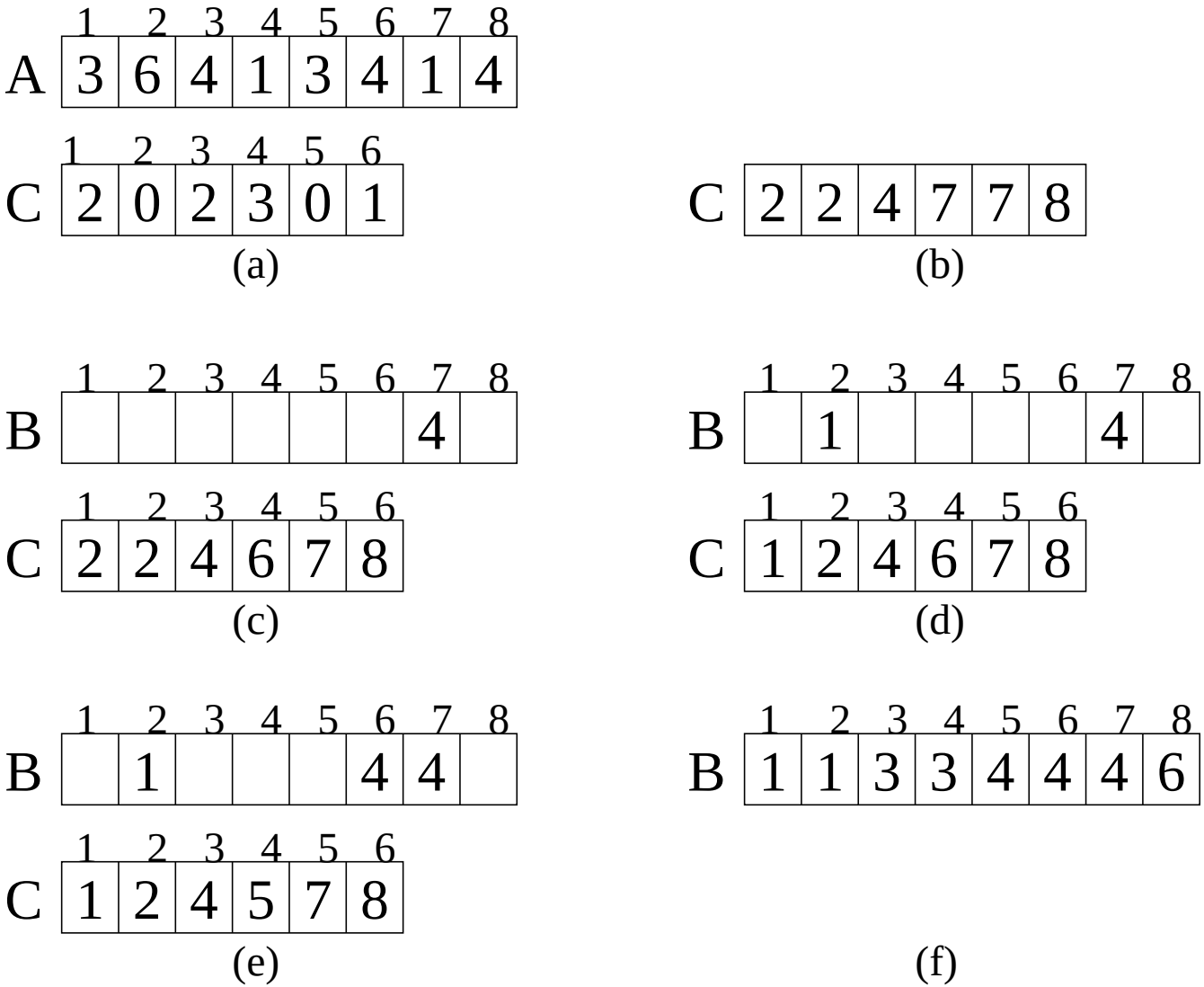
```
COUNTING-SORT(A, B, n, k)
1   para i ← 0 até k faça
2       C[i] ← 0

3   para j ← 1 até n faça
4       C[A[j]] ← C[A[j]] + 1
    ▷ C[i] é o número de js tais que A[j] = i

5   para i ← 1 até k faça
6       C[i] ← C[i] + C[i − 1]
    ▷ C[i] é o número de js tais que A[j] ≤ i

7   para j ← n decrescendo até 1 faça
8       B[C[A[j]]] ← A[j]
9       C[A[j]] ← C[A[j]] − 1
```

Counting Sort



Counting Sort - Complexidade

- Qual a complexidade do algoritmo COUNTING-SORT?
- O algoritmo não faz comparações entre elementos de A!
- Sua complexidade deve ser medida em função do número das outras operações, aritméticas, atribuições, etc.
- Claramente, a complexidade de COUNTING-SORT é $O(n+k)$. Quando $k \in O(n)$, ele tem complexidade $O(n)$.

Há algo de errado com o limite inferior de $\Omega(n \log n)$ para ordenação?

Algoritmos *in-place* e *estáveis*

- Algoritmos de ordenação podem ser ou não *in-place* ou *estáveis*.
- Um algoritmo de ordenação é in-place se a memória adicional requerida é independente do tamanho do vetor que está sendo ordenado.
- Exemplos: QuickSort e Heap-Sort são métodos de ordenação *in-place*, já Merge-Sort e Counting-Sort não são.
- Um método de ordenação é estável se elementos iguais ocorrem no vetor ordenado na mesma ordem em que são passados na entrada.
- Exemplos: Counting-Sort e QuickSort são exemplos de métodos estáveis (desde que certos cuidados sejam tomados na implementação). Heap-Sort não é.

Radix Sort

- Considere agora o problema de ordenar um vetor $A[1 \ldots n]$ inteiros quando se sabe que todos os inteiros podem ser representados com apenas d dígitos, onde d é uma constante.
- Por exemplo, os elementos de A podem ser CEPs, ou seja, inteiros de 8 dígitos.

Radix Sort

- Poderíamos ordenar os elementos do vetor dígito a dígito da seguinte forma:
  - Separamos os elementos do vetor em grupos que compartilham o mesmo dígito mais significativo.
  - Em seguida, ordenamos os elementos em cada grupo pelo mesmo método, levando em consideração apenas os $d - 1$ dígitos menos significativos.
- Esse método funciona, mas requer o uso de bastante memória adicional para a organização dos grupos e subgrupos.

Radix Sort

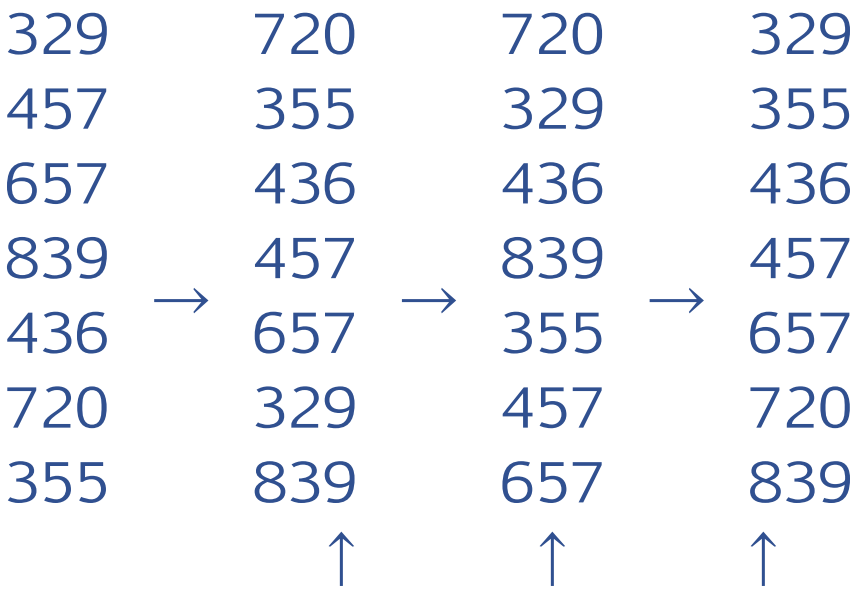
- Podemos evitar o uso excessivo de memória adicional começando pelo dígito menos significativo.
- É isso o que faz o algoritmo Radix Sort.
- Para que Radix Sort funcione corretamente, ele deve usar um método de ordenação *estável*.
- Por exemplo, o Counting-Sort.

Radix Sort

Suponha que os elementos do vetor A a ser ordenado sejam números inteiros de até d dígitos. O *Radix Sort* é simplesmente:

```
RADIX-SORT(A, n, d)
1   para i ← 1 até d faça
2       Ordene A[1...n] pelo i-ésimo dígito
        usando um método estável
```

Radix Sort - Exemplo



Radix Sort - Correção

O seguinte argumento indutivo garante a correção do algoritmo:

- Hipótese de indução: os números estão ordenados com relação aos $i - 1$ dígitos menos significativos.
- O que acontece ao ordenarmos pelo i-ésimo dígito?
- Se dois números têm i-ésimo dígitos distintos, o de menor i-ésimo dígito aparece antes do de maior i-ésimo dígito.
- Se ambos possuem o mesmo i-ésimo dígito, então a ordem dos dois também estará correta pois o método de ordenação é estável e, pela HI, os dois elementos já estavam ordenados segundo os $i - 1$ dígitos menos significativos.

Radix Sort - Complexidade

- Qual é a complexidade do RADIX-SORT?
- Depende da complexidade do algoritmo estável usado para ordenar cada dígito.
- Se essa complexidade for $\Theta(f(n))$, obtemos uma complexidade total de $\Theta(d\, f(n))$.
- Como d é constante, a complexidade é então $\Theta(f(n))$.
- Se o algoritmo estável for, por exemplo, o COUNTING-SORT, obtemos a complexidade $\Theta(n+k)$.
- Se $k \in O(n)$, isto resulta em uma complexidade linear em n.

E o limite inferior de $\Omega(n \log n)$ para ordenação?

Radix Sort - Complexidade

- Em contraste, um algoritmo por comparação como o MERGE-SORT teria complexidade $\Theta(n \lg n)$.
- Assim, RADIX-SORT é mais vantajoso que MERGE-SORT quando $d < \lg n$, ou seja, o número de dígitos for menor que $\lg n$.
- Se n for um limite superior para o maior valor a ser ordenado, então $O(\log n)$ é uma estimativa para a quantidade de dígitos dos números.
- Isso significa que não há diferença significativa entre o desempenho do MERGE-SORT e do RADIX-SORT?

Radix Sort - Complexidade

- O nome *Radix Sort* vem da **base** (em inglês *radix*) em que intepretamos os dígitos.
- A vantagem de se usar RADIX-SORT fica evidente quando interpretamos os dígitos de forma mais geral que simplesmente 0..9.
- Tomemos o seguinte exemplo: suponha que desejemos ordenar um conjunto de $n = 2^{20}$ números de 64 bits. Então, MERGE-SORT faria cerca de $n \lg n = 20 \times 2^{20}$ comparações e usaria um vetor auxiliar de tamanho 2^{20}.

Radix Sort - Complexidade

- Agora suponha que interpretamos cada número como tendo $d = 4$ dígitos em base $k = 2^{16}$, e usarmos RADIX-SORT com o *Counting Sort* como método estável.

  Então a complexidade de tempo seria da ordem de $d(n + k) = 4(2^{20} + 2^{16})$ operações, bem menor que 20×2^{20} do MERGE-SORT. Mas, note que utilizamos dois vetores auxiliares, de tamanhos 2^{16} e 2^{20}.
- 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 é 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 d-upla ordenada de itens comparáveis.

Bucket Sort

- Supõe que os n elementos da entrada estão distribuídos uniformemente no intervalo $[0, 1)$.
- A ideia é dividir o intervalo $[0, 1)$ em n segmentos de mesmo tamanho (*buckets*) e distribuir os n elementos nos seus respectivos segmentos. Como os elementos estão distribuídos uniformemente, espera-se que o número de elementos seja aproximadamente o mesmo em todos os segmentos.
- Em seguida, os elementos de cada segmento são ordenados por um método qualquer. Finalmente, os segmentos ordenados são concatenados em ordem crescente.

Bucket Sort - Pseudocódigo

```
BUCKET-SORT(A, n)
1   para i ← 1 até n faça
2       faça B[i] uma lista ligada vazia
3   para i ← 1 até n faça
4       insira A[i] na lista ligada B[⌊n A[i]⌋]
5   para i ← 0 até n − 1 faça
6       ordene a lista B[i] com INSERTION-SORT
7   Concatene as listas B[0], B[1], ..., B[n − 1]
```

Bucket Sort - Exemplo

$$A = \begin{array}{c|l} 1 & .78 \\ 2 & .17 \\ 3 & .39 \\ 4 & .26 \\ 5 & .72 \\ 6 & .94 \\ 7 & .21 \\ 8 & .12 \\ 9 & .23 \\ 10 & .68 \end{array} \qquad B = \begin{array}{c|l} 0 & \\ 1 & .12,.17 \\ 2 & .21,.23,.26 \\ 3 & .39 \\ 4 & \\ 5 & \\ 6 & .68 \\ 7 & .72,.78 \\ 8 & \\ 9 & .94 \end{array}$$

Bucket Sort - Correção

- Dois elementos x e y de A, $x < y$, ou terminam na mesma lista ou são colocados em listas diferentes $B[i]$ e $B[j]$.
- A primeira possibilidade implica que x aparecerá antes de y na concatenação final, já que cada lista é ordenada.
- No segundo caso, como $x < y$, segue que $i = \lfloor nx \rfloor \leq \lfloor ny \rfloor = j$. Como $i \neq j$, temos $i < j$. Assim, x aparecerá antes de y na lista final.

Bucket Sort - Complexidade

- É claro que o pior caso do *Bucket Sort* é quadrático, supondo-se que as ordenações das listas seja feita com ordenação por inserção.
- Entretanto, o tempo esperado é linear.
  Intuitivamente, a ideia da demonstração é que, como os n elementos estão distribuídos uniformemente no intervalo $[0, 1)$, então o tamanho esperado das listas é pequeno.
- Portanto, as ordenações das n listas $B[i]$ leva tempo total esperado $\Theta(n)$.

Bucket Sort - Complexidade

- Seja n_i variável aleatória que denota o número de elementos em $B[i]$. Claramente $n = \sum_{i=1}^{n} n_i$.
- Seja $T(n)$ variável aleatória que denota o tempo de execução do algoritmo
- Inserção em listas ligadas é feita em $\Theta(1)$. Assim, a inserção dos n elementos em $B[\,]$ gasta $\Theta(n)$.
- Execução do InsertionSort em uma lista com n_i elementos é feita em tempo $O(n_i^2)$

$$E[T(n)] = E\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right]$$

$$= \Theta(n) + \sum_{i=0}^{n-1} E[O(n_i^2)]$$

$$= \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2])$$

Bucket Sort - Complexidade

Seja X_{ij} a variável aleatória binária que indica se o elemento j é inserido em $B[i]$. Claramente $n_i = \sum_{j=1}^{n} X_{ij}$.

$$
\begin{aligned}
E[n_i^2] &= E\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right] \\
&= E\left[\left(\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij}X_{ik}\right)\right] \\
&= E\left[\sum_{j=1}^{n} X_{ij}^2 + \sum_{j=1}^{n}\sum_{k=1,\ k\neq j}^{n} X_{ij}X_{ik}\right] \\
&= \sum_{j=1}^{n} E[X_{ij}^2] + \sum_{j=1}^{n}\sum_{k=1,\ k\neq j}^{n} E[X_{ij}X_{ik}]
\end{aligned}
$$

Bucket Sort - Complexidade

$$
\begin{aligned}
E[n_i^2] &= \sum_{j=1}^{n} E[X_{ij}^2] + \sum_{j=1}^{n} \sum_{k=1,\ k\neq j}^{n} E[X_{ij}X_{ik}] \\
&= \sum_{j=1}^{n} E[X_{ij}] + \sum_{j=1}^{n} \sum_{k=1,\ k\neq j}^{n} E[X_{ij}]E[X_{ik}] \\
&= \sum_{j=1}^{n} \frac{1}{n} + \sum_{j=1}^{n} \sum_{k=1,\ k\neq j}^{n} \frac{1}{n}\frac{1}{n} \\
&= 1 + n(n-1)\frac{1}{n^2} \\
&= 2 - \frac{1}{n}
\end{aligned}
$$

Bucket Sort - Complexidade

Voltando a análise de $E[T(n)]$, temos

$$
\begin{aligned}
E[T(n)] &= \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2]) \\
&= \Theta(n) + \sum_{i=0}^{n-1} O(1) \\
&= \Theta(n)
\end{aligned}
$$