MC458 — Projeto e Análise de Algoritmos I

C.C. de Souza   C.N. da Silva   O. Lee

Antes de mais nada...

- Uma versão anterior deste conjunto de slides foi preparada por Cid Carvalho de Souza e Cândida Nunes da Silva para uma instância anterior desta disciplina.
- O que vocês tem em mãos é uma versão modificada preparada para atender a meus gostos.
- Nunca é demais enfatizar que o material é apenas um guia e não deve ser usado como única fonte de estudo. Para isso consultem a bibliografia (em especial o CLR ou CLRS).

Orlando Lee

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
  - José Coelho de Pina
  - Orlando Lee
  - Paulo Feofiloff
  - Pedro Rezende
  - Ricardo Dahab
  - Zanoni Dias

Cota inferior de complexidade

O Problema da Ordenação - cota inferior

- Estudamos diversos algoritmos para o Problema da 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.

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.

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 implicam em uma cota superior para o número de comparações efetuadas por um algoritmo que resolva o Problema da Ordenação.

- 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 implicam em 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 MergeSort e o HeapSort, que efetuam $\Theta(n \log n)$ comparações no pior caso.

O Problema da Ordenação - cota inferior

O Problema da Ordenação - cota inferior

- Será que é possível projetar um algoritmo de ordenação baseado em comparações ainda mais eficiente?

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!

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.

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.

- 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 π dos índices $1 \leq i \leq n$ tal que $x_{\pi(1)} \leq x_{\pi(2)} \leq \ldots \leq x_{\pi(n)}$.

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 π dos índices $1 \leq i \leq n$ tal que $x_{\pi(1)} \leq x_{\pi(2)} \leq \ldots \leq x_{\pi(n)}$.

- É possível representar um algoritmo para o Problema da Ordenação através de uma árvore binária de decisão da seguinte forma:

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 π dos índices $1 \leq i \leq n$ tal que $x_{\pi(1)} \leq x_{\pi(2)} \leq \ldots \leq x_{\pi(n)}$.

- É possível representar um algoritmo para o Problema da Ordenação através de uma árvore binária de decisão da seguinte forma:
  - Os nós internos representam comparações entre dois elementos do conjunto, digamos $x_i \leq x_j$.

- 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 π dos índices $1 \leq i \leq n$ tal que $x_{\pi(1)} \leq x_{\pi(2)} \leq \ldots \leq x_{\pi(n)}$.

- É possível representar um algoritmo para o Problema da Ordenação através de uma árvore binária 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 $x_i > x_j$.

Árvores de decisão para o Problema da Ordenação

- Considere a seguinte definição alternativa do problema da ordenação:

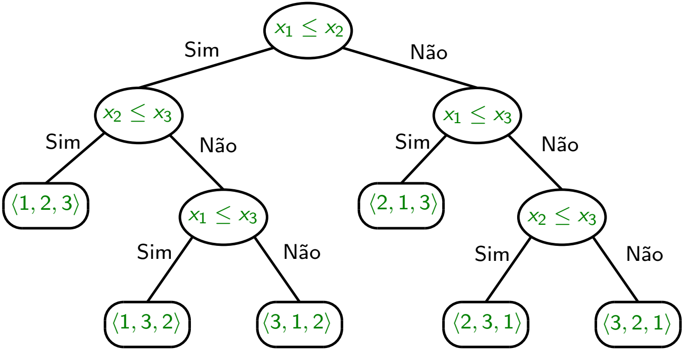
Problema da Ordenação:

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

- É possível representar um algoritmo para o Problema da Ordenação através de uma árvore binária 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 binária 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.

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 seqüências distintas de decisões terminem no mesmo resultado).

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 seqüências distintas de decisões terminem no mesmo resultado).
- O caminho mais longo da raiz a uma folha representa o pior caso de execução do algoritmo.

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 seqüências 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 é a altura mínima, em função de n, de uma árvore binária de decisão com pelo menos $n!$ folhas?

Cota inferior

- Qual é 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.

Cota inferior

- Qual é 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 $n! \leq 2^h$, ou seja, $h \geq \lg n!$.

Cota inferior

- Qual é 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 $n! \leq 2^h$, ou seja, $h \geq \lg n!$.
- Mas,

$$
\begin{aligned}
\lg n! &= \sum_{i=1}^{n} \lg i \\
&\geq \sum_{i=\lceil (n+1)/2 \rceil}^{n} \lg i \\
&\geq \sum_{i=\lceil (n+1)/2 \rceil}^{n} \lg n/2 \\
&\geq n/2 \lg n/2 \\
&= n/2(\lg n - 1) \\
&\geq n/2 \lg n - n/2
\end{aligned}
$$

Cota inferior

- Qual é 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 $n! \leq 2^h$, ou seja, $h \geq \lg n!$.
- Mas,

$$
\begin{aligned}
\lg n! &= \sum_{i=1}^{n} \lg i \\
&\geq \sum_{i=\lceil (n+1)/2 \rceil}^{n} \lg i \\
&\geq \sum_{i=\lceil (n+1)/2 \rceil}^{n} \lg n/2 \\
&\geq n/2 \lg n/2 \\
&= n/2(\lg n - 1) \\
&\geq n/2 \lg n - n/2
\end{aligned}
$$

- Então, $h \in \Omega(n \log n)$.

Conclusão

- Provamos então que $\Omega(n \log n)$ é uma cota inferior para o Problema da Ordenação.

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 (neste modelo).

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 (neste modelo).
- Veremos depois algoritmos lineares para o Problema da Ordenação, ou seja, que têm complexidade $O(n)$.
  (Como???)

Cotas inferiores de problemas

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 cota inferior trivial $\Omega(n)$.

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 cota inferior trivial $\Omega(n)$.

- São **pouquíssimos problemas** para os quais se conhece uma cota inferior que coincide com a cota superior, ou mesmo uma cota inferior não-trivial.

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 cota inferior trivial $\Omega(n)$.

- São **pouquíssimos problemas** para os quais se conhece uma cota inferior que coincide com a cota superior, ou mesmo uma cota inferior não-trivial.

- Um deles é o Problema da Ordenação.

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 cota inferior trivial $\Omega(n)$.

- São **pouquíssimos problemas** para os quais se conhece uma cota inferior que coincide com a cota superior, ou mesmo uma cota inferior não-trivial.
- Um deles é o Problema da Ordenação.
- Veremos mais dois exemplos: Problema da Busca em Vetor Ordenado e o Problema do Máximo (de um conjunto).

$\triangleright$ **Entrada:** Um **vetor ordenado** $A[p\,..\,r]$ e um elemento x.
$\triangleright$ **Saída:** Índice $p \leq i \leq r$ tal que $A[i] = x$ ou $i = -1$.

Busca-Binária(A, p, r, x)

se $p = r$ $\triangleright$ $n = 1$
    então
        se $A[p] = x$ **então devolva** p
                        senão devolva -1
    senão
        $i := \lceil (p + r)/2 \rceil$
        se $x < A[i]$
            então devolva Busca-Binária$(A, p, i - 1, x)$
            senão devolva Busca-Binária(A, i, r, x)

Número de comparações: $O(\lg n)$.

Busca em Vetor Ordenado

Busca em Vetor Ordenado

- É possível projetar um algoritmo mais rápido?

Busca em Vetor Ordenado

- É possível projetar um algoritmo mais rápido?
- Não, se o algoritmo baseia-se em comparações do tipo $A[i] < x$, $A[i] > x$ ou $A[i] = x$.

Busca em Vetor Ordenado

- É possível projetar um algoritmo mais rápido?
- Não, se o algoritmo baseia-se 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 $\Omega(\lg n)$.

Busca em Vetor Ordenado

- É possível projetar um algoritmo mais rápido?
- Não, se o algoritmo baseia-se 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

Cota inferior

- Todo algoritmo para o Problema da Busca em Vetor Ordenado baseado em comparações pode ser representado através de uma árvore binária de decisão.

Cota inferior

- Todo algoritmo para o Problema da Busca em Vetor Ordenado baseado em comparações pode ser representado através de uma árvore binária de decisão.
- Cada nó interno corresponde a uma comparação com o elemento procurado x.

Cota inferior

- Todo algoritmo para o Problema da Busca em Vetor Ordenado baseado em comparações pode ser representado através de uma árvore binária de decisão.
- Cada nó interno corresponde a uma comparação com o elemento procurado x.
- As ramificações correspondem ao resultado da comparação.

Cota inferior

- Todo algoritmo para o Problema da Busca em Vetor Ordenado baseado em comparações pode ser representado através de uma árvore binária 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.

Cota inferior

- Todo algoritmo para o Problema da Busca em Vetor Ordenado baseado em comparações pode ser representado através de uma árvore binária 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 $\Omega(\lg n)$.

Cota inferior

- Todo algoritmo para o Problema da Busca em Vetor Ordenado baseado em comparações pode ser representado através de uma árvore binária 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 $\Omega(\lg n)$.

Qualquer algoritmo baseado em comparações para o Problema da Busca em Vetor Ordenado deve fazer **pelo menos** $\Omega(\lg n)$ comparações.

Problema do Máximo

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

Problema do Máximo

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

- Existe um algoritmo que faz o serviço com $n-1$ comparações.

Problema:

Encontrar o maior elemento de um vetor $A[1\,..\,n]$.

- Existe um algoritmo que faz o serviço com $n - 1$ comparações.
- Existe um algoritmo que faz menos comparações?

Problema do Máximo

Problema:

Encontrar o maior elemento de um vetor $A[1 \,..\, n]$.

- 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.

Problema do Máximo

Problema:
Encontrar o maior elemento de um vetor $A[1 \mathinner{.\,.} n]$.

- 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?

Problema do Máximo

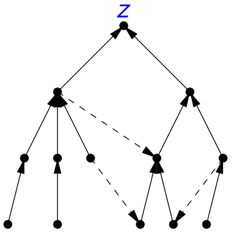
Seja V o conjunto dos elementos do vetor A. O algoritmo consiste, no fundo, em encontrar um conjunto $\mathcal{A}$ de pares ou arcos (u, v) de elementos distintos em V tais que $u < v$.
Considere o grafo orientado $D = (V, \mathcal{A})$.

Eis o paradigma de um algoritmo baseado em comparações:

MÁXIMO(A, n)
$\mathcal{A} \leftarrow \emptyset$
enquanto $D = (V, \mathcal{A})$ não possui super-sorvedouro **faça**
  escolha elementos u, v em V
  se $u < v$
    então $\mathcal{A} \leftarrow \mathcal{A} \cup (u, v)$
    senão $\mathcal{A} \leftarrow \mathcal{A} \cup (v, u)$
devolva $\mathcal{A}$

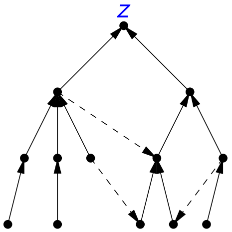
Problema do Máximo

Um super-sorvedouro de D é um elemento z tal que para todo vértice x existe um caminho orientado de x a z.



Problema do Máximo

Um super-sorvedouro de D é um elemento z tal que para todo vértice x existe um caminho orientado de x a z.



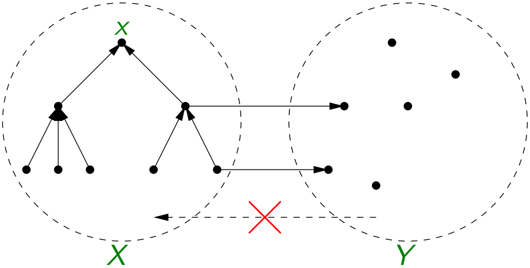
Para um algoritmo devolver corretamente o máximo, D deve conter um super-sorvedouro. (Por quê?)

Problema do Máximo

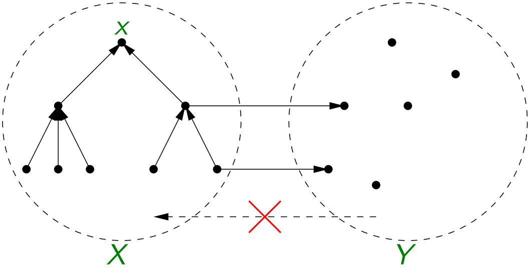
Suponha que existe alguma instância tal que $D = (V, \mathcal{A})$ não possui um super-sorvedouro.

Problema do Máximo

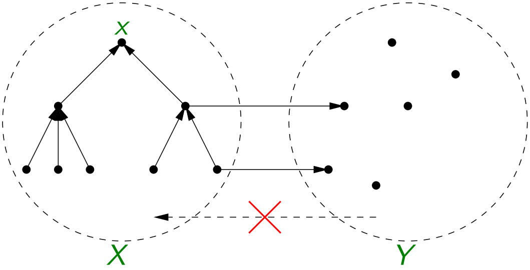
Suponha que existe alguma instância tal que $D = (V, \mathcal{A})$ não possui um super-sorvedouro. Seja x o elemento que o algoritmo devolveu como sendo o máximo. Seja X o conjunto dos vértices para os quais existe um caminho orientado dele até x e seja $Y = V - X$. Assim, não existe arco ligando Y a X.



Problema do Máximo

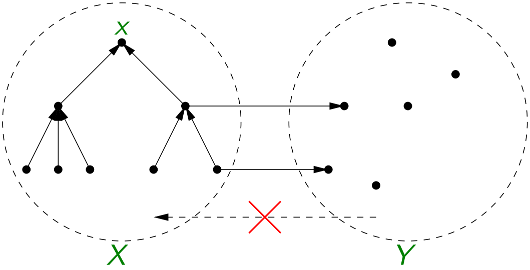


Problema do Máximo



Considere a instância modificada na qual aumentamos cada elemento de Y de um valor suficientemente grande. O algoritmo faz exatamente as mesmas comparações para esta nova instância e devolve x (Por quê?). No entanto, agora o máximo necessariamente está em Y.

Problema do Máximo

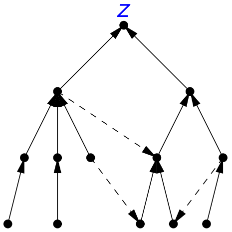


Considere a instância modificada na qual aumentamos cada elemento de Y de um valor suficientemente grande. O algoritmo faz exatamente as mesmas comparações para esta nova instância e devolve x (Por quê?). No entanto, agora o máximo necessariamente está em Y.

Assim, se o algoritmo devolve corretamente o máximo, então D possui um super-sorvedouro z.

Conclusão

Portanto, D contém uma árvore enraizada no super-sorvedouro z e portanto, possui pelo menos $n-1$ arcos.



Qualquer algoritmo baseado em comparações que resolve o Problema do Máximo em um vetor $A[1\,..\,n]$ faz **pelo menos** $n-1$ comparações.

Ordenação em Tempo Linear

Algoritmos lineares para ordenação

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

Algoritmos lineares para ordenação

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

- Counting Sort: Elementos são números inteiros "pequenos"; mais precisamente, inteiros $x \in O(n)$.

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.

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 $[0..1)$.

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 $[0..1)$.

Observação: note que em todos estes algoritmos há hipóteses adicionais sobre a entrada.

Counting Sort

- Considere o problema de ordenar um vetor $A[1 \, . \, . \, n]$ de inteiros quando se sabe que todos os inteiros estão no intervalo $[0, k]$.

Counting Sort

- Considere o problema de ordenar um vetor $A[1\,..\,n]$ de inteiros quando se sabe que todos os inteiros estão no intervalo $[0, k]$.
- Podemos ordenar o vetor simplesmente contando, para cada inteiro i no vetor, quantos elementos do vetor são iguais a i.

Counting Sort

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

Counting Sort

Recebe um vetor $A[1\,..\,n]$ com inteiros em $[0, k]$ e devolve um vetor $B[1\,..\,n]$ com os elementos originais ordenados.

$\textsc{Counting-Sort}(A, B, n, k)$

para $i \leftarrow 0$ **até** k **faça**
    $C[i] \leftarrow 0$
para $j \leftarrow 1$ **até** n **faça**
    $C[A[j]] \leftarrow C[A[j]] + 1$
  $\triangleright$ $C[i]$ é o número de elementos iguais a i
para $i \leftarrow 1$ **até** k **faça**
    $C[i] \leftarrow C[i] + C[i-1]$
  $\triangleright$ $C[i]$ é o número de elementos menores ou iguais i
para $j \leftarrow n$ **decrescendo até** 1 **faça**
    $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

Counting Sort – Exemplo

$n = 8$, $k = 5$

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

	0	1	2	3	4	5
C	0	0	0	0	0	0

	1	2	3	4	5	6	7	8
B								

para $i \leftarrow 0$ **até** k **faça**
    $C[i] \leftarrow 0$

Counting Sort – Exemplo

$n = 8$, $k = 5$

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

	0	1	2	3	4	5
C	2	0	2	3	0	1

	1	2	3	4	5	6	7	8
B								

3 **para** $j \leftarrow 1$ **até** n **faça**
4 $\quad C[A[j]] \leftarrow C[A[j]] + 1$
$\triangleright$ $C[i]$ é o número de elementos iguais a i

$n = 8$, $k = 5$

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

	0	1	2	3	4	5
C	2	2	4	7	7	8

	1	2	3	4	5	6	7	8
B								

5 **para** $i \leftarrow 1$ **até** k **faça**
6 $\quad C[i] \leftarrow C[i] + C[i-1]$
$\triangleright$ $C[i]$ é o número de elementos menores ou iguais i

Counting Sort – Exemplo

$n = 8$, $k = 5$

	1	2	3	4	5	6	7	j
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	2	2	4	7	7	8

	1	2	3	4	5	6	7	8
B								

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

$n = 8$, $k = 5$

	1	2	3	4	5	6	j	8
A	2	5	3	0	2	3	0	3

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

	1	2	3	4	5	6	7	8
B							3	

para $j \leftarrow n$ **decrescendo até** 1 **faça**
    $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

Counting Sort – Exemplo

$n = 8$, $k = 5$

	1	2	3	4	5	j	7	8
A	2	5	3	0	2	3	0	3

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

	1	2	3	4	5	6	7	8
B		0					3	

para $j \leftarrow n$ **decrescendo até** 1 **faça**
  $B[C[A[j]]] \leftarrow A[j]$
  $C[A[j]] \leftarrow C[A[j]] - 1$

Counting Sort – Exemplo

$n = 8$, $k = 5$

	1	2	3	4	j	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	1	2	4	5	7	8

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

para $j \leftarrow n$ **decrescendo até** 1 **faça**
    $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

$n = 8$, $k = 5$

	1	2	3	j	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	1	2	3	5	7	8

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

para $j \leftarrow n$ **decrescendo até** 1 **faça**
    $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

$n = 8$, $k = 5$

	1	2	j	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	0	2	3	5	7	8

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

para $j \leftarrow n$ **decrescendo até** 1 **faça**
    $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

Counting Sort – Exemplo

$n = 8$, $k = 5$

	1	j	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	0	2	3	4	7	8

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

7 **para** $j \leftarrow n$ **decrescendo até** 1 **faça**
8 $\quad B[C[A[j]]] \leftarrow A[j]$
9 $\quad C[A[j]] \leftarrow C[A[j]] - 1$

$n = 8$, $k = 5$

	j	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	0	2	3	4	7	7

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

para $j \leftarrow n$ **decrescendo até** 1 **faça**
  $B[C[A[j]]] \leftarrow A[j]$
  $C[A[j]] \leftarrow C[A[j]] - 1$

Counting Sort – Exemplo

$n = 8$, $k = 5$

j	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	0	2	2	4	7	7

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

para $j \leftarrow n$ **decrescendo até** 1 **faça**
    $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

Recebe um vetor $A[1\,..\,n]$ com inteiros em $[0, k]$ e devolve um vetor $B[1\,..\,n]$ com os elementos originais ordenados.

$\textsc{Counting-Sort}(A, B, n, k)$

para $i \leftarrow 0$ **até** k **faça**
    $C[i] \leftarrow 0$
para $j \leftarrow 1$ **até** n **faça**
    $C[A[j]] \leftarrow C[A[j]] + 1$
  $\triangleright$ $C[i]$ é o número de elementos iguais a i
para $i \leftarrow 1$ **até** k **faça**
    $C[i] \leftarrow C[i] + C[i-1]$
  $\triangleright$ $C[i]$ é o número de elementos menores ou iguais i
para $j \leftarrow n$ **decrescendo até** 1 **faça**
    $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

Counting Sort - Complexidade

Counting Sort - Complexidade

- COUNTING-SORT usa espaço adicional $\Theta(k)$.
- Qual a complexidade do algoritmo COUNTING-SORT?

Counting Sort - Complexidade

- Counting-Sort usa espaço adicional $\Theta(k)$.
- Qual a complexidade do algoritmo Counting-Sort?
- O algoritmo não faz comparações entre elementos de A!

- COUNTING-SORT usa espaço adicional $\Theta(k)$.
- 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.

Counting Sort - Complexidade

- COUNTING-SORT usa espaço adicional $\Theta(k)$.
- 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 $O(n)$.

Counting Sort - Complexidade

- COUNTING-SORT usa espaço adicional $\Theta(k)$.
- 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*

- Um algoritmo de ordenação é in-place se a memória adicional requerida é independente do tamanho do vetor que está sendo ordenado.

Algoritmos *in-place*

- Um algoritmo de ordenação é in-place se a memória adicional requerida é independente do tamanho do vetor que está sendo ordenado.
- Insertion-Sort, Selection-Sort e HeapSort são métodos de ordenação *in-place*.

Algoritmos *in-place*

- Um algoritmo de ordenação é in-place se a memória adicional requerida é independente do tamanho do vetor que está sendo ordenado.
- INSERTION-SORT, SELECTION-SORT e HEAPSORT são métodos de ordenação *in-place*.
- MERGESORT e COUNTING-SORT não são in-place.

Algoritmos *in-place*

- Um algoritmo de ordenação é in-place se a memória adicional requerida é independente do tamanho do vetor que está sendo ordenado.
- Insertion-Sort, Selection-Sort e HeapSort são métodos de ordenação *in-place*.
- MergeSort e Counting-Sort não são in-place.
- A implementação que vimos do QuickSort é in-place?

Algoritmos *estáveis*

- Um algoritmo de ordenação é *estável* se elementos iguais ocorrem no vetor ordenado na mesma ordem em que são passados na entrada.

Algoritmos *estáveis*

- Um algoritmo de ordenação é *estável* se elementos iguais ocorrem no vetor ordenado na mesma ordem em que são passados na entrada.
- Insertion-Sort, Selection-Sort e QuickSort são exemplos de algoritmos *estáveis* (desde que certos cuidados sejam tomados na implementação).

Algoritmos *estáveis*

- Um algoritmo de ordenação é *estável* se elementos iguais ocorrem no vetor ordenado na mesma ordem em que são passados na entrada.
- Insertion-Sort, Selection-Sort e QuickSort são exemplos de algoritmos *estáveis* (desde que certos cuidados sejam tomados na implementação).
- O Counting-Sort é um algoritmo estável. (Exercício!)

Algoritmos *estáveis*

- Um algoritmo de ordenação é *estável* se elementos iguais ocorrem no vetor ordenado na mesma ordem em que são passados na entrada.
- Insertion-Sort, Selection-Sort e QuickSort são exemplos de algoritmos *estáveis* (desde que certos cuidados sejam tomados na implementação).
- O Counting-Sort é um algoritmo estável. (Exercício!)
- HeapSort não é um algoritmo *estável*.

Algoritmos *estáveis*

- Um algoritmo de ordenação é *estável* se elementos iguais ocorrem no vetor ordenado na mesma ordem em que são passados na entrada.
- INSERTION-SORT, SELECTION-SORT e QUICKSORT são exemplos de algoritmos *estáveis* (desde que certos cuidados sejam tomados na implementação).
- O COUNTING-SORT é um algoritmo estável. (Exercício!)
- HEAPSORT não é um algoritmo *estável*.
- MERGESORT é *estável*?

Algoritmos *estáveis*

- Um algoritmo de ordenação é *estável* se elementos iguais ocorrem no vetor ordenado na mesma ordem em que são passados na entrada.
- INSERTION-SORT, SELECTION-SORT e QUICKSORT são exemplos de algoritmos *estáveis* (desde que certos cuidados sejam tomados na implementação).
- O COUNTING-SORT é um algoritmo estável. (Exercício!)
- HEAPSORT não é um algoritmo *estável*.
- MERGESORT é *estável*? Depende da versão de INTERCALA.

Radix Sort

- Considere agora o problema de ordenar um vetor $A[1 \,.\,.\, n]$ de inteiros no qual os inteiros podem ser representados com apenas d dígitos, onde d é uma constante.

Radix Sort

- Considere agora o problema de ordenar um vetor $A[1\,..\,n]$ de inteiros no qual os inteiros podem ser representados com apenas d dígitos, onde d é uma constante.
- Por exemplo, os elementos de A são CEPs, ou seja, inteiros de 8 dígitos.

Radix Sort

- Considere agora o problema de ordenar um vetor $A[1\,..\,n]$ de inteiros no qual os inteiros podem ser representados com apenas d dígitos, onde d é uma constante.
- Por exemplo, os elementos de A são CEPs, ou seja, inteiros de 8 dígitos.
- Outro exemplo: os elementos de A são números em base 16 com no máximo d "dígitos" em $\{0, 1, \ldots, 9, A, B, C, D, E, F\}$.

Radix Sort

- Considere agora o problema de ordenar um vetor $A[1\,..\,n]$ de inteiros no qual os inteiros podem ser representados com apenas d dígitos, onde d é uma constante.
- Por exemplo, os elementos de A são CEPs, ou seja, inteiros de 8 dígitos.
- Outro exemplo: os elementos de A são números em base 16 com no máximo d "dígitos" em $\{0, 1, \ldots, 9, A, B, C, D, E, F\}$.
- Outro exemplo: os elementos de A são strings de comprimento no máximo d, i.e., inteiros em base 256 com no máximo d "dígitos".

$$\text{``}ACB'' = 65 \times 256^2 + 67 \times 256^1 + 66 \times 256^0.$$

- Poderíamos ordenar os elementos do vetor dígito a dígito da seguinte forma:

- 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 (mais à esquerda).

- 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 (mais à esquerda).
  - Em seguida, ordenamos os elementos em cada grupo pelo mesmo método, levando em consideração apenas os $d - 1$ dígitos menos significativos.

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 (mais à esquerda).
  - Em seguida, ordenamos os elementos em cada grupo pelo mesmo método, levando em consideração apenas os $d - 1$ dígitos menos significativos.
- Esse algoritmo 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.

Radix Sort

- Podemos evitar o uso excessivo de memória adicional começando pelo dígito menos significativo.
- É isso o que faz o algoritmo Radix Sort.

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 algoritmo de ordenação *estável*.
  Por exemplo, COUNTING-SORT.

Radix Sort

Radix Sort supõe que os inteiros de $A[1\,..\,n]$ podem ser representados com apenas d dígitos.

Radix Sort

Radix Sort supõe que os inteiros de $A[1\,..\,n]$ podem ser representados com apenas d dígitos.

$\textsc{Radix-Sort}(A, n, d)$

para $i \leftarrow 1$ até d **faça**
  Ordene $A[1\,..\,n]$ pelo i-ésimo dígito
    usando um algoritmo de ordenação estável

Radix Sort - Exemplo

329
457
657
839
436
720
355

Radix Sort - Exemplo

$$
\begin{array}{ccccc}
329 & & 720 & & \\
457 & & 355 & & \\
657 & & 436 & & \\
839 & \rightarrow & 457 & \rightarrow & \\
436 & & 657 & & \\
720 & & 329 & & \\
355 & & 839 & & \\
 & & \uparrow & &
\end{array}
$$

Radix Sort - Exemplo

$$
\begin{array}{ccccccc}
329 & & 720 & & 720 & & \\
457 & & 355 & & 329 & & \\
657 & & 436 & & 436 & & \\
839 & & 457 & & 839 & & \\
436 & \rightarrow & 657 & \rightarrow & 355 & \rightarrow & \\
720 & & 329 & & 457 & & \\
355 & & 839 & & 657 & & \\
 & & \uparrow & & \uparrow & &
\end{array}
$$

Radix Sort - Exemplo

$$\begin{array}{ccccccc}
329 & & 720 & & 720 & & 329 \\
457 & & 355 & & 329 & & 355 \\
657 & & 436 & & 436 & & 436 \\
839 & \rightarrow & 457 & \rightarrow & 839 & \rightarrow & 457 \\
436 & & 657 & & 355 & & 657 \\
720 & & 329 & & 457 & & 720 \\
355 & & 839 & & 657 & & 839 \\
 & & \quad\uparrow & & \uparrow & & \uparrow\quad
\end{array}$$

O seguinte argumento indutivo garante a corretude do algoritmo:

Radix Sort - Corretude

O seguinte argumento indutivo garante a corretude do algoritmo:

- Hipótese de indução: os números estão ordenados com relação aos $i - 1$ dígitos menos significativos.

O seguinte argumento indutivo garante a corretude 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?

Radix Sort - Corretude

O seguinte argumento indutivo garante a corretude 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.

Radix Sort - Corretude

O seguinte argumento indutivo garante a corretude 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.

- Qual é a complexidade de RADIX-SORT?

- Qual é a complexidade de Radix-Sort?
- Depende da complexidade do algoritmo estável usado para ordenar cada dígito.

- Qual é a complexidade de 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 $\Theta(d\,f(n))$.

Radix Sort - Complexidade

- Qual é a complexidade de 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 $\Theta(d\,f(n))$.
- Se d é **constante**, então a complexidade é $\Theta(f(n))$.

- Qual é a complexidade de 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 $\Theta(d\,f(n))$.
- Se d é **constante**, então a complexidade é $\Theta(f(n))$.
- Se o algoritmo estável for, por exemplo, o COUNTING-SORT, obtemos a complexidade $\Theta(n + k)$.

Radix Sort - Complexidade

- Qual é a complexidade de 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 $\Theta(d\,f(n))$.
- Se d é **constante**, então a complexidade é $\Theta(f(n))$.
- 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 MERGESORT teria complexidade $\Theta(n \lg n)$.

Radix Sort - Complexidade

- Em contraste, um algoritmo por comparação como o MERGESORT teria complexidade $\Theta(n \lg n)$.
- Assim, RADIX-SORT é mais vantajoso que MERGESORT quando $d < \lg n$, ou seja, o número de dígitos for menor que $\lg n$.

Radix Sort - Complexidade

- Em contraste, um algoritmo por comparação como o MERGESORT teria complexidade $\Theta(n \lg n)$.
- Assim, RADIX-SORT é mais vantajoso que MERGESORT 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 de um número da entrada, então $O(\log n)$ é uma estimativa para a quantidade de dígitos dos números.

Radix Sort - Complexidade

- Em contraste, um algoritmo por comparação como o MERGESORT teria complexidade $\Theta(n \lg n)$.
- Assim, RADIX-SORT é mais vantajoso que MERGESORT 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 de um número da entrada, 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 MERGESORT e do RADIX-SORT?

Radix Sort - Complexidade

Radix Sort - Complexidade

- O nome *Radix Sort* vem da **base** (em inglês *radix*) em que interpretamos os dígitos.

- O nome *Radix Sort* vem da **base** (em inglês *radix*) em que interpretamos os dígitos.
- A vantagem de se usar Radix-Sort fica evidente quando interpretamos os dígitos de forma mais geral do que simplesmente $0, 1, \ldots, 9$.

Radix Sort - Complexidade

- O nome *Radix Sort* vem da **base** (em inglês *radix*) em que interpretamos os dígitos.
- A vantagem de se usar Radix-Sort fica evidente quando interpretamos os dígitos de forma mais geral do que simplesmente $0, 1, \ldots, 9$.
- Tomemos o seguinte exemplo: suponha que queremos ordenar um conjunto de $n = 2^{20}$ números de 64 bits. Então, MergeSort faria cerca de $n \lg n = 20 \times 2^{20}$ comparações e usaria um vetor auxiliar de tamanho 2^{20}.

Radix Sort - Complexidade

Radix Sort - Complexidade

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

- Agora suponha que interpretamos cada número do conjunto 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 MERGESORT. Mas, note que utilizamos dois vetores auxiliares, de tamanhos 2^{16} e 2^{20}.

Radix Sort - Complexidade

- Agora suponha que interpretamos cada número do conjunto 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 MERGESORT. 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*.

- Agora suponha que interpretamos cada número do conjunto 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 MERGESORT. 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

- Hipótese: Supõe que os n elementos da entrada estão distribuídos uniformemente no intervalo $[0, 1)$.

Bucket Sort

- Hipótese: Supõe que os n elementos da entrada estão distribuídos uniformemente no intervalo $[0, 1)$.
- A idéia é dividir o intervalo $[0, 1)$ em n segmentos de mesmo tamanho (*buckets*) e distribuir os n elementos nos seus respectivos *buckets*. Como os elementos estão distribuídos uniformemente, espera-se que o número de elementos seja aproximadamente o mesmo em cada *bucket*.

Bucket Sort

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

Bucket Sort - Pseudocódigo

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

$$
\begin{aligned}
A[i] \in [0, 1/n) &\implies A[i] \in B[0] \\
A[i] \in [1/n, 2/n) &\implies A[i] \in B[1] \\
&\vdots \\
A[i] \in [j/n, (j+1)/n) &\implies A[i] \in B[j] \\
&\vdots \\
A[i] \in [(n-1)/n, 1) &\implies A[i] \in B[n-1]
\end{aligned}
$$

Bucket Sort - Pseudocódigo

BUCKET-SORT(A, n)

para $i \leftarrow 1$ **até** n **faça**

  insira $A[i]$ na lista ligada $B[\lfloor n\,A[i] \rfloor]$

para $i \leftarrow 0$ **até** $n - 1$ **faça**

  ordene a lista $B[i]$ com INSERTION-SORT

Concatene as listas $B[0], B[1], \ldots, B[n-1]$

Bucket Sort - Pseudocódigo

BUCKET-SORT(A, n)

para $i \leftarrow 1$ **até** n **faça**

    insira $A[i]$ na lista ligada $B[\lfloor n\,A[i] \rfloor]$

para $i \leftarrow 0$ **até** $n - 1$ **faça**

    ordene a lista $B[i]$ com INSERTION-SORT

Concatene as listas $B[0], B[1], \ldots, B[n-1]$

Por que usar INSERTION-SORT?

Bucket Sort - Pseudocódigo

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

Por que usar INSERTION-SORT?

Porque é um algoritmo simples e funciona bem para sequências pequenas.

Bucket Sort - Exemplo

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

Bucket Sort - Exemplo

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

Bucket Sort - Corretude

Bucket Sort - Corretude

- Dois elementos x e y de A tais que $x < y$, ou terminam na mesma lista ou são colocados em *buckets* diferentes $B[i]$ e $B[j]$.

Bucket Sort - Corretude

- Dois elementos x e y de A tais que $x < y$, ou terminam na mesma lista ou são colocados em *buckets* diferentes $B[i]$ e $B[j]$.
- A primeira possibilidade implica que x aparecerá antes de y na concatenação final, já que cada *bucket* foi ordenado.

Bucket Sort - Corretude

- Dois elementos x e y de A tais que $x < y$, ou terminam na mesma lista ou são colocados em *buckets* diferentes $B[i]$ e $B[j]$.
- A primeira possibilidade implica que x aparecerá antes de y na concatenação final, já que cada *bucket* foi ordenado.
- 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 - Corretude

- Dois elementos x e y de A tais que $x < y$, ou terminam na mesma lista ou são colocados em *buckets* diferentes $B[i]$ e $B[j]$.
- A primeira possibilidade implica que x aparecerá antes de y na concatenação final, já que cada *bucket* foi ordenado.
- 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.
- Assim, Bucket-Sort é correto.

Bucket Sort - Complexidade

- É claro que o tempo de pior caso do Bucket-Sort é quadrático, supondo-se que as ordenações dos *buckets* seja feita pelo Insertion-Sort.

Bucket Sort - Complexidade

- É claro que o tempo de pior caso do BUCKET-SORT é quadrático, supondo-se que as ordenações dos *buckets* seja feita pelo INSERTION-SORT.
- Entretanto, o tempo esperado é linear. Ou equivalentemente, o tempo de caso médio é linear.

- É claro que o tempo de pior caso do BUCKET-SORT é quadrático, supondo-se que as ordenações dos *buckets* seja feita pelo INSERTION-SORT.
- Entretanto, o tempo esperado é linear. Ou equivalentemente, o tempo de caso médio é linear.
- A idéia da demonstração é que, como os n elementos estão distribuídos uniformemente em $[0, 1)$, então o tamanho esperado de cada um dos n *buckets* é $O(1)$.

- É claro que o tempo de pior caso do BUCKET-SORT é quadrático, supondo-se que as ordenações dos *buckets* seja feita pelo INSERTION-SORT.
- Entretanto, o tempo esperado é linear. Ou equivalentemente, o tempo de caso médio é linear.
- A idéia da demonstração é que, como os n elementos estão distribuídos uniformemente em $[0, 1)$, então o tamanho esperado de cada um dos n *buckets* é $O(1)$.
- Portanto, as ordenações dos n *buckets* leva tempo total esperado $\Theta(n)$.

Bucket Sort - Complexidade

- É claro que o tempo de pior caso do BUCKET-SORT é quadrático, supondo-se que as ordenações dos *buckets* seja feita pelo INSERTION-SORT.
- Entretanto, o tempo esperado é linear. Ou equivalentemente, o tempo de caso médio é linear.
- A idéia da demonstração é que, como os n elementos estão distribuídos uniformemente em $[0, 1)$, então o tamanho esperado de cada um dos n *buckets* é $O(1)$.
- Portanto, as ordenações dos n *buckets* leva tempo total esperado $\Theta(n)$.
- Os detalhes podem ser vistos no CLRS.

Resumo da ópera

Resumo da ópera

- O problema de ordenação tem cota inferior $\Theta(n \log n)$.

Resumo da ópera

- O problema de ordenação tem cota inferior $\Theta(n \log n)$.
- Entretanto, quando a sequência a ser ordenada satisfaz certas propriedades é possível resolver o problema em tempo linear.