Projeto e Análise de Algoritmos

Cotas inferiores e ordenação em tempo linear

Lehilton Pedrosa

Primeiro Semestre de 2020

Cotas inferiores

Cotas inferiores

- Cota inferior para ordenação

Número de comparações para ordenação

Vimos diversos algoritmos para o problema da ordenação

Todos eles baseados em comparações

- elas ditam a posição relativa de dois elementos
- a relação $x_i \leq x_j$ determina se x_i vem antes de x_j
- não usamos outro tipo de informação sobre elementos

Podemos contar o número de comparações

- o tempo de execução fornece uma cota superior
- as melhores cotas são de MERGE-SORT e HEAP-SORT
- eles executam $\Theta(n \log n)$ comparações no pior caso

Número de comparações para ordenação

Vimos diversos algoritmos para o problema da ordenação

Todos eles baseados em comparações

- elas ditam a posição relativa de dois elementos
- a relação $x_i \leq x_j$ determina se x_i vem antes de x_j
- não usamos outro tipo de informação sobre elementos

Podemos contar o número de comparações

- o tempo de execução fornece uma cota superior
- as melhores cotas são de MERGE-SORT e HEAP-SORT
- eles executam $\Theta(n \log n)$ comparações no pior caso

Número de comparações para ordenação

Vimos diversos algoritmos para o problema da ordenação

Todos eles baseados em comparações

- elas ditam a posição relativa de dois elementos
- a relação $x_i \leq x_j$ determina se x_i vem antes de x_j
- não usamos outro tipo de informação sobre elementos

Podemos contar o número de comparações

- o tempo de execução fornece uma cota superior
- as melhores cotas são de Merge-Sort e Heap-Sort
- eles executam $\Theta(n \log n)$ comparações no pior caso

Número de comparações para ordenação

Vimos diversos algoritmos para o problema da ordenação

Todos eles baseados em comparações

- elas ditam a posição relativa de dois elementos
- a relação $x_i \leq x_j$ determina se x_i vem antes de x_j
- não usamos outro tipo de informação sobre elementos

Podemos contar o número de comparações

- o tempo de execução fornece uma cota superior
- as melhores cotas são de MERGE-SORT e HEAP-SORT
- eles executam $\Theta(n \log n)$ comparações no pior caso

Número de comparações para ordenação

Vimos diversos algoritmos para o problema da ordenação

Todos eles baseados em comparações

- elas ditam a posição relativa de dois elementos
- a relação $x_i \leq x_j$ determina se x_i vem antes de x_j
- não usamos outro tipo de informação sobre elementos

Podemos contar o número de comparações

- o tempo de execução fornece uma cota superior
- as melhores cotas são de MERGE-SORT e HEAP-SORT
- eles executam $\Theta(n \log n)$ comparações no pior caso

Número de comparações para ordenação

Vimos diversos algoritmos para o problema da ordenação

Todos eles baseados em comparações

- elas ditam a posição relativa de dois elementos
- a relação $x_i \leq x_j$ determina se x_i vem antes de x_j
- não usamos outro tipo de informação sobre elementos

Podemos contar o número de comparações

- o tempo de execução fornece uma cota superior
- as melhores cotas são de MERGE-SORT e HEAP-SORT
- eles executam $\Theta(n \log n)$ comparações no pior caso

Número de comparações para ordenação

Vimos diversos algoritmos para o problema da ordenação

Todos eles baseados em comparações

- elas ditam a posição relativa de dois elementos
- a relação $x_i \leq x_j$ determina se x_i vem antes de x_j
- não usamos outro tipo de informação sobre elementos

Podemos contar o número de comparações

- o tempo de execução fornece uma cota superior
- as melhores cotas são de MERGE-SORT e HEAP-SORT
- eles executam $\Theta(n \log n)$ comparações no pior caso

Número de comparações para ordenação

Vimos diversos algoritmos para o problema da ordenação

Todos eles baseados em comparações

- elas ditam a posição relativa de dois elementos
- a relação $x_i \leq x_j$ determina se x_i vem antes de x_j
- não usamos outro tipo de informação sobre elementos

Podemos contar o número de comparações

- o tempo de execução fornece uma cota superior
- as melhores cotas são de MERGE-SORT e HEAP-SORT
- eles executam $\Theta(n \log n)$ comparações no pior caso

Número de comparações para ordenação

Vimos diversos algoritmos para o problema da ordenação

Todos eles baseados em comparações

- elas ditam a posição relativa de dois elementos
- a relação $x_i \leq x_j$ determina se x_i vem antes de x_j
- não usamos outro tipo de informação sobre elementos

Podemos contar o número de comparações

- o tempo de execução fornece uma cota superior
- as melhores cotas são de MERGE-SORT e HEAP-SORT
- eles executam $\Theta(n \log n)$ comparações no pior caso

Cota inferior para o número de comparações

Existe algoritmo para ordenar um vetor mais eficiente?

- se ele for baseado em comparações, vamos ver que não!
- qualquer algoritmo de ordenação precisa executar pelo menos $\Omega(n \log n)$ comparações

Para formalizar isso, restringiremos o modelo computacional

- é proibido acessar o valor de um elemento x_i diretamente
- só acessamos elementos por meio de comparação $x_i \leq x_j$
- a execução de um algoritmo nesse modelo computacional induz o que chamamos de árvore de decisão

Cota inferior para o número de comparações

Existe algoritmo para ordenar um vetor mais eficiente?

- se ele for baseado em comparações, vamos ver que não!
- qualquer algoritmo de ordenação precisa executar pelo menos $\Omega(n \log n)$ comparações

Para formalizar isso, restringiremos o modelo computacional

- é proibido acessar o valor de um elemento x_i diretamente
- só acessamos elementos por meio de comparação $x_i \leq x_j$
- a execução de um algoritmo nesse modelo computacional induz o que chamamos de árvore de decisão

Cota inferior para o número de comparações

Existe algoritmo para ordenar um vetor mais eficiente?

- se ele for baseado em comparações, vamos ver que não!
- qualquer algoritmo de ordenação precisa executar pelo menos $\Omega(n \log n)$ comparações

Para formalizar isso, restringiremos o modelo computacional

- é proibido acessar o valor de um elemento x_i diretamente
- só acessamos elementos por meio de comparação $x_i \leq x_j$
- a execução de um algoritmo nesse modelo computacional induz o que chamamos de árvore de decisão

Cota inferior para o número de comparações

Existe algoritmo para ordenar um vetor mais eficiente?

- se ele for baseado em comparações, vamos ver que não!
- qualquer algoritmo de ordenação precisa executar pelo menos $\Omega(n \log n)$ comparações

Para formalizar isso, restringiremos o modelo computacional

- é proibido acessar o valor de um elemento x_i diretamente
- só acessamos elementos por meio de comparação $x_i \leq x_j$
- a execução de um algoritmo nesse modelo computacional induz o que chamamos de árvore de decisão

Cota inferior para o número de comparações

Existe algoritmo para ordenar um vetor mais eficiente?

- se ele for baseado em comparações, vamos ver que não!
- qualquer algoritmo de ordenação precisa executar pelo menos $\Omega(n \log n)$ comparações

Para formalizar isso, restringiremos o modelo computacional

- é proibido acessar o valor de um elemento x_i diretamente
- só acessamos elementos por meio de comparação $x_i \leq x_j$
- a execução de um algoritmo nesse modelo computacional induz o que chamamos de árvore de decisão

Cota inferior para o número de comparações

Existe algoritmo para ordenar um vetor mais eficiente?

- se ele for baseado em comparações, vamos ver que não!
- qualquer algoritmo de ordenação precisa executar pelo menos $\Omega(n \log n)$ comparações

Para formalizar isso, restringiremos o modelo computacional

- é proibido acessar o valor de um elemento x_i diretamente
- só acessamos elementos por meio de comparação $x_i \leq x_j$
- a execução de um algoritmo nesse modelo computacional induz o que chamamos de árvore de decisão

Cota inferior para o número de comparações

Existe algoritmo para ordenar um vetor mais eficiente?

- se ele for baseado em comparações, vamos ver que não!
- qualquer algoritmo de ordenação precisa executar pelo menos $\Omega(n \log n)$ comparações

Para formalizar isso, restringiremos o modelo computacional

- é proibido acessar o valor de um elemento x_i diretamente
- só acessamos elementos por meio de comparação $x_i \leq x_j$
- a execução de um algoritmo nesse modelo computacional induz o que chamamos de árvore de decisão

Árvores de decisão

Uma árvore de decisão é uma árvore binária em que

- **nós internos** representam comparações executadas
- **subárvores** representam a continuação do algoritmo após a comparação
- **folhas** representam as saídas possíveis do algoritmo

Propriedades

- toda execução do algoritmo deve corresponder a um caminho entre a raiz e uma folha
- o número de comparações realizada corresponde ao número de nós internos nesse caminho
- o pior caso corresponde à altura da árvore

Árvores de decisão

Uma árvore de decisão é uma árvore binária em que

- **nós internos** representam comparações executadas
- **subárvores** representam a continuação do algoritmo após a comparação
- **folhas** representam as saídas possíveis do algoritmo

Propriedades

- toda execução do algoritmo deve corresponder a um caminho entre a raiz e uma folha
- o número de comparações realizada corresponde ao número de nós internos nesse caminho
- o pior caso corresponde à altura da árvore

Árvores de decisão

Uma árvore de decisão é uma árvore binária em que

- **nós internos** representam comparações executadas
- **subárvores** representam a continuação do algoritmo após a comparação
- **folhas** representam as saídas possíveis do algoritmo

Propriedades

- toda execução do algoritmo deve corresponder a um caminho entre a raiz e uma folha
- o número de comparações realizada corresponde ao número de nós internos nesse caminho
- o pior caso corresponde à altura da árvore

Árvores de decisão

Uma árvore de decisão é uma árvore binária em que

- **nós internos** representam comparações executadas
- **subárvores** representam a continuação do algoritmo após a comparação
- **folhas** representam as saídas possíveis do algoritmo

Propriedades

- toda execução do algoritmo deve corresponder a um caminho entre a raiz e uma folha
- o número de comparações realizada corresponde ao número de nós internos nesse caminho
- o pior caso corresponde à altura da árvore

Árvores de decisão

Uma árvore de decisão é uma árvore binária em que

- **nós internos** representam comparações executadas
- **subárvores** representam a continuação do algoritmo após a comparação
- **folhas** representam as saídas possíveis do algoritmo

Propriedades

- toda execução do algoritmo deve corresponder a um caminho entre a raiz e uma folha
- o número de comparações realizada corresponde ao número de nós internos nesse caminho
- o pior caso corresponde à altura da árvore

Árvores de decisão

Uma árvore de decisão é uma árvore binária em que

- **nós internos** representam comparações executadas
- **subárvores** representam a continuação do algoritmo após a comparação
- **folhas** representam as saídas possíveis do algoritmo

Propriedades

- toda execução do algoritmo deve corresponder a um caminho entre a raiz e uma folha
- o número de comparações realizada corresponde ao número de nós internos nesse caminho
- o pior caso corresponde à altura da árvore

Árvores de decisão

Uma árvore de decisão é uma árvore binária em que

- **nós internos** representam comparações executadas
- **subárvores** representam a continuação do algoritmo após a comparação
- **folhas** representam as saídas possíveis do algoritmo

Propriedades

- toda execução do algoritmo deve corresponder a um caminho entre a raiz e uma folha
- o número de comparações realizada corresponde ao número de nós internos nesse caminho
- o pior caso corresponde à altura da árvore

Árvores de decisão

Uma árvore de decisão é uma árvore binária em que

- **nós internos** representam comparações executadas
- **subárvores** representam a continuação do algoritmo após a comparação
- **folhas** representam as saídas possíveis do algoritmo

Propriedades

- toda execução do algoritmo deve corresponder a um caminho entre a raiz e uma folha
- o número de comparações realizada corresponde ao número de nós internos nesse caminho
- o pior caso corresponde à altura da árvore

Ordenação como problema de permutação

Podemos redefinir o problema da ordenação

- **Entrada:** vetor de n inteiros $x_1, x_2, \ldots, x_n$
- **Saída:** permutação p tal que $x_{p(1)} \leq x_{p(2)} \leq \ldots \leq x_{p(n)}$

Com isso, reinterpretamos a árvore de decisão

- um **nó interno** representa a comparação $x_i \leq x_j$
- o **ramo esquerdo** corresponde a vetores com $x_i \leq x_j$
- o **ramo direito** corresponde a vetores com $x_i > x_j$.
- uma **folha** representa uma permutações da entrada

Ordenação como problema de permutação

Podemos redefinir o problema da ordenação

- **Entrada:** vetor de n inteiros $x_1, x_2, \ldots, x_n$
- **Saída:** permutação p tal que $x_{p(1)} \leq x_{p(2)} \leq \ldots \leq x_{p(n)}$

Com isso, reinterpretamos a árvore de decisão

- um **nó interno** representa a comparação $x_i \leq x_j$
- o **ramo esquerdo** corresponde a vetores com $x_i \leq x_j$
- o **ramo direito** corresponde a vetores com $x_i > x_j$.
- uma **folha** representa uma permutações da entrada

Ordenação como problema de permutação

Podemos redefinir o problema da ordenação

- **Entrada:** vetor de n inteiros $x_1, x_2, \ldots, x_n$
- **Saída:** permutação p tal que $x_{p(1)} \leq x_{p(2)} \leq \ldots \leq x_{p(n)}$

Com isso, reinterpretamos a árvore de decisão

- um **nó interno** representa a comparação $x_i \leq x_j$
- o **ramo esquerdo** corresponde a vetores com $x_i \leq x_j$
- o **ramo direito** corresponde a vetores com $x_i > x_j$.
- uma **folha** representa uma permutações da entrada

Ordenação como problema de permutação

Podemos redefinir o problema da ordenação

- **Entrada:** vetor de n inteiros $x_1, x_2, \ldots, x_n$
- **Saída:** permutação p tal que $x_{p(1)} \leq x_{p(2)} \leq \ldots \leq x_{p(n)}$

Com isso, reinterpretamos a árvore de decisão

- um **nó interno** representa a comparação $x_i \leq x_j$
- o **ramo esquerdo** corresponde a vetores com $x_i \leq x_j$
- o **ramo direito** corresponde a vetores com $x_i > x_j$.
- uma **folha** representa uma permutações da entrada

Ordenação como problema de permutação

Podemos redefinir o problema da ordenação

- **Entrada:** vetor de n inteiros $x_1, x_2, \ldots, x_n$
- **Saída:** permutação p tal que $x_{p(1)} \leq x_{p(2)} \leq \ldots \leq x_{p(n)}$

Com isso, reinterpretamos a árvore de decisão

- um **nó interno** representa a comparação $x_i \leq x_j$
- o **ramo esquerdo** corresponde a vetores com $x_i \leq x_j$
- o **ramo direito** corresponde a vetores com $x_i > x_j$.
- uma **folha** representa uma permutações da entrada

Ordenação como problema de permutação

Podemos redefinir o problema da ordenação

- **Entrada:** vetor de n inteiros $x_1, x_2, \ldots, x_n$
- **Saída:** permutação p tal que $x_{p(1)} \leq x_{p(2)} \leq \ldots \leq x_{p(n)}$

Com isso, reinterpretamos a árvore de decisão

- um **nó interno** representa a comparação $x_i \leq x_j$
- o **ramo esquerdo** corresponde a vetores com $x_i \leq x_j$
- o **ramo direito** corresponde a vetores com $x_i > x_j$.
- uma **folha** representa uma permutações da entrada

Ordenação como problema de permutação

Podemos redefinir o problema da ordenação

- **Entrada:** vetor de n inteiros $x_1, x_2, \ldots, x_n$
- **Saída:** permutação p tal que $x_{p(1)} \leq x_{p(2)} \leq \ldots \leq x_{p(n)}$

Com isso, reinterpretamos a árvore de decisão

- um **nó interno** representa a comparação $x_i \leq x_j$
- o **ramo esquerdo** corresponde a vetores com $x_i \leq x_j$
- o **ramo direito** corresponde a vetores com $x_i > x_j$.
- uma **folha** representa uma permutações da entrada

Ordenação como problema de permutação

Podemos redefinir o problema da ordenação

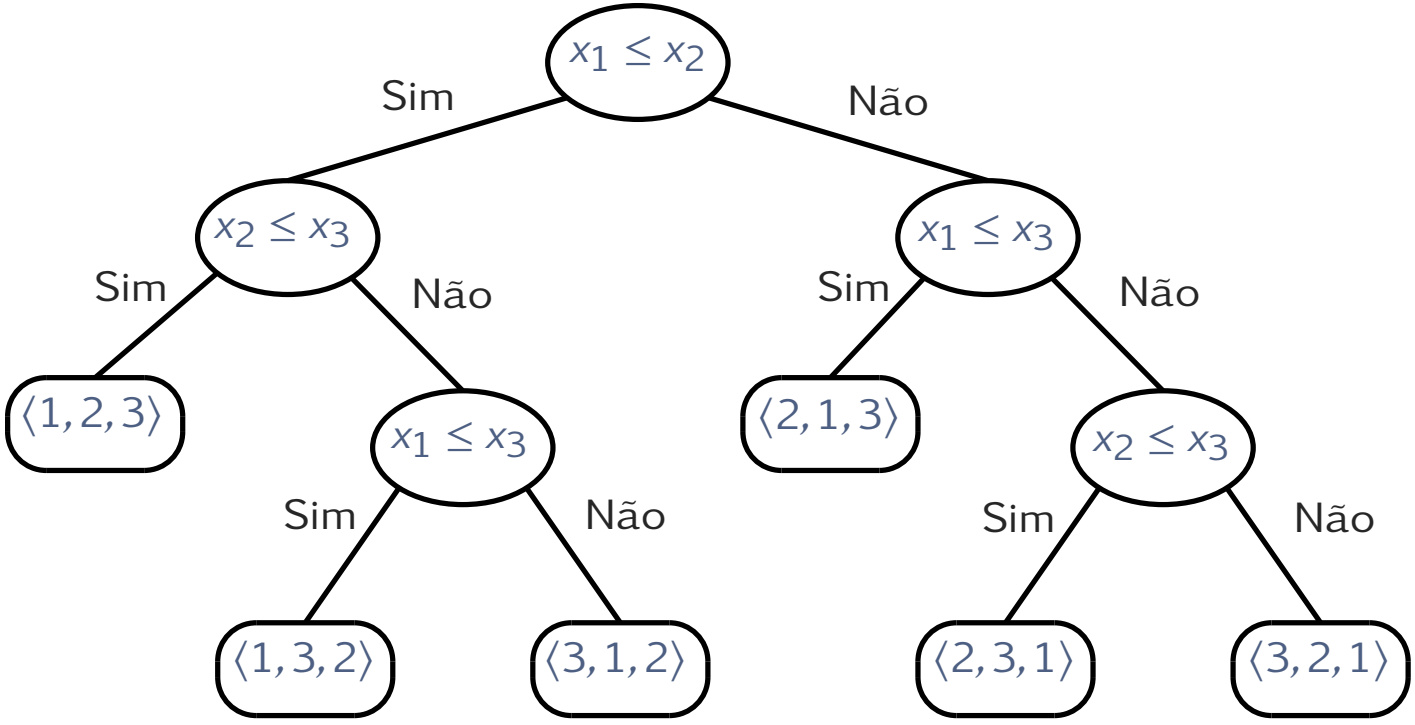
- **Entrada:** vetor de n inteiros $x_1, x_2, \ldots, x_n$
- **Saída:** permutação p tal que $x_{p(1)} \leq x_{p(2)} \leq \ldots \leq x_{p(n)}$

Com isso, reinterpretamos a árvore de decisão

- um **nó interno** representa a comparação $x_i \leq x_j$
- o **ramo esquerdo** corresponde a vetores com $x_i \leq x_j$
- o **ramo direito** corresponde a vetores com $x_i > x_j$.
- uma **folha** representa uma permutações da entrada

Exemplo de árvore de decisão

Árvore de decisão de INSERTION-SORT para 3 elementos:



Analisando o caso geral

Quantas folhas deve haver para um vetor com n elementos?

- cada permutação de n índices corresponde a pelo menos uma entrada do problema de ordenação
- todas as permutações devem estar representas
- a árvore de decisão deve ter pelo menos $n!$ folhas

Consequências para o tempo de execução

- o pior caso corresponde à altura da árvore
- então o melhor algoritmo tem a árvore mais baixa
- quão baixa pode ser uma árvore binária com $n!$ folhas?

Analisando o caso geral

Quantas folhas deve haver para um vetor com n elementos?

- cada permutação de n índices corresponde a pelo menos uma entrada do problema de ordenação
- todas as permutações devem estar representas
- a árvore de decisão deve ter pelo menos $n!$ folhas

Consequências para o tempo de execução

- o pior caso corresponde à altura da árvore
- então o melhor algoritmo tem a árvore mais baixa
- quão baixa pode ser uma árvore binária com $n!$ folhas?

Analisando o caso geral

Quantas folhas deve haver para um vetor com n elementos?

- cada permutação de n índices corresponde a pelo menos uma entrada do problema de ordenação
- todas as permutações devem estar representas
- a árvore de decisão deve ter pelo menos $n!$ folhas

Consequências para o tempo de execução

- o pior caso corresponde à altura da árvore
- então o melhor algoritmo tem a árvore mais baixa
- quão baixa pode ser uma árvore binária com $n!$ folhas?

Analisando o caso geral

Quantas folhas deve haver para um vetor com n elementos?

- cada permutação de n índices corresponde a pelo menos uma entrada do problema de ordenação
- todas as permutações devem estar representas
- a árvore de decisão deve ter pelo menos $n!$ folhas

Consequências para o tempo de execução

- o pior caso corresponde à altura da árvore
- então o melhor algoritmo tem a árvore mais baixa
- quão baixa pode ser uma árvore binária com $n!$ folhas?

Analisando o caso geral

Quantas folhas deve haver para um vetor com n elementos?

- cada permutação de n índices corresponde a pelo menos uma entrada do problema de ordenação
- todas as permutações devem estar representas
- a árvore de decisão deve ter pelo menos $n!$ folhas

Consequências para o tempo de execução

- o pior caso corresponde à altura da árvore
- então o melhor algoritmo tem a árvore mais baixa
- quão baixa pode ser uma árvore binária com $n!$ folhas?

Analisando o caso geral

Quantas folhas deve haver para um vetor com n elementos?

- cada permutação de n índices corresponde a pelo menos uma entrada do problema de ordenação
- todas as permutações devem estar representas
- a árvore de decisão deve ter pelo menos $n!$ folhas

Consequências para o tempo de execução

- o pior caso corresponde à altura da árvore
- então o melhor algoritmo tem a árvore mais baixa
- quão baixa pode ser uma árvore binária com $n!$ folhas?

Analisando o caso geral

Quantas folhas deve haver para um vetor com n elementos?

- cada permutação de n índices corresponde a pelo menos uma entrada do problema de ordenação
- todas as permutações devem estar representas
- a árvore de decisão deve ter pelo menos $n!$ folhas

Consequências para o tempo de execução

- o pior caso corresponde à altura da árvore
- então o melhor algoritmo tem a árvore mais baixa
- quão baixa pode ser uma árvore binária com $n!$ folhas?

Analisando o caso geral

Quantas folhas deve haver para um vetor com n elementos?

- cada permutação de n índices corresponde a pelo menos uma entrada do problema de ordenação
- todas as permutações devem estar representas
- a árvore de decisão deve ter pelo menos $n!$ folhas

Consequências para o tempo de execução

- o pior caso corresponde à altura da árvore
- então o melhor algoritmo tem a árvore mais baixa
- quão baixa pode ser uma árvore binária com $n!$ folhas?

Cota inferior

Considere uma árvore de decisão com altura h

- como ela é binária, há no máximo 2^h folhas
- mas há pelo menos $n!$ folhas, então $n! \leq 2^h$
- portanto, $h \geq \lg n!$

Desenvolvendo o limitante inferior

$$\lg n! = \sum_{i=1}^{n} \lg i \geq \sum_{i=\lceil n/2 \rceil}^{n} \lg n/2 \geq n/2 \cdot \lg n/2$$

Então $h \in \Omega(n \lg n)$.

Cota inferior

Considere uma árvore de decisão com altura h

- como ela é binária, há no máximo 2^h folhas
- mas há pelo menos $n!$ folhas, então $n! \leq 2^h$
- portanto, $h \geq \lg n!$

Desenvolvendo o limitante inferior

$$\lg n! = \sum_{i=1}^{n} \lg i \geq \sum_{i=\lceil n/2 \rceil}^{n} \lg n/2 \geq n/2 \cdot \lg n/2$$

Então $h \in \Omega(n \lg n)$.

Cota inferior

Considere uma árvore de decisão com altura h

- como ela é binária, há no máximo 2^h folhas
- mas há pelo menos $n!$ folhas, então $n! \leq 2^h$
- portanto, $h \geq \lg n!$

Desenvolvendo o limitante inferior

$$\lg n! = \sum_{i=1}^{n} \lg i \geq \sum_{i=\lceil n/2 \rceil}^{n} \lg n/2 \geq n/2 \cdot \lg n/2$$

Então $h \in \Omega(n \lg n)$.

Cota inferior

Considere uma árvore de decisão com altura h

- como ela é binária, há no máximo 2^h folhas
- mas há pelo menos $n!$ folhas, então $n! \leq 2^h$
- portanto, $h \geq \lg n!$

Desenvolvendo o limitante inferior

$$\lg n! = \sum_{i=1}^{n} \lg i \geq \sum_{i=\lceil n/2 \rceil}^{n} \lg n/2 \geq n/2 \cdot \lg n/2$$

Então $h \in \Omega(n \lg n)$.

Cota inferior

Considere uma árvore de decisão com altura h

- como ela é binária, há no máximo 2^h folhas
- mas há pelo menos $n!$ folhas, então $n! \leq 2^h$
- portanto, $h \geq \lg n!$

Desenvolvendo o limitante inferior

$$\lg n! = \sum_{i=1}^{n} \lg i \geq \sum_{i=\lceil n/2 \rceil}^{n} \lg n/2 \geq n/2 \cdot \lg n/2$$

Então $h \in \Omega(n \lg n)$.

Cota inferior

Considere uma árvore de decisão com altura h

- como ela é binária, há no máximo 2^h folhas
- mas há pelo menos $n!$ folhas, então $n! \leq 2^h$
- portanto, $h \geq \lg n!$

Desenvolvendo o limitante inferior

$$\lg n! = \sum_{i=1}^{n} \lg i \geq \sum_{i=\lceil n/2 \rceil}^{n} \lg n/2 \geq n/2 \cdot \lg n/2$$

Então $h \in \Omega(n \lg n)$.

Cota inferior

Considere uma árvore de decisão com altura h

- como ela é binária, há no máximo 2^h folhas
- mas há pelo menos $n!$ folhas, então $n! \leq 2^h$
- portanto, $h \geq \lg n!$

Desenvolvendo o limitante inferior

$$\lg n! = \sum_{i=1}^{n} \lg i \geq \sum_{i=\lceil n/2 \rceil}^{n} \lg n/2 \geq n/2 \cdot \lg n/2$$

Então $h \in \Omega(n \lg n)$.

Conclusão

O que aprendemos sobre o problema da ordenação?

- qualquer algoritmo executa $\Omega(n \log n)$ comparações
- assim, $\Omega(n \log n)$ é uma cota inferior para o problema
- portanto, Merge-Sort e Heap-Sort são algoritmos assintoticamente ótimos

É preciso atentar-se ao problema considerado!

- veremos algoritmos lineares para ordenação depois
- mas eles não são baseados em comparação
- e têm restrições adicionais sobre a entrada

Conclusão

O que aprendemos sobre o problema da ordenação?

- qualquer algoritmo executa $\Omega(n \log n)$ comparações
- assim, $\Omega(n \log n)$ é uma cota inferior para o problema
- portanto, Merge-Sort e Heap-Sort são algoritmos assintoticamente ótimos

É preciso atentar-se ao problema considerado!

- veremos algoritmos lineares para ordenação depois
- mas eles não são baseados em comparação
- e têm restrições adicionais sobre a entrada

Conclusão

O que aprendemos sobre o problema da ordenação?

- qualquer algoritmo executa $\Omega(n \log n)$ comparações
- assim, $\Omega(n \log n)$ é uma cota inferior para o problema
- portanto, Merge-Sort e Heap-Sort são algoritmos assintoticamente ótimos

É preciso atentar-se ao problema considerado!

- veremos algoritmos lineares para ordenação depois
- mas eles não são baseados em comparação
- e têm restrições adicionais sobre a entrada

Conclusão

O que aprendemos sobre o problema da ordenação?

- qualquer algoritmo executa $\Omega(n \log n)$ comparações
- assim, $\Omega(n \log n)$ é uma cota inferior para o problema
- portanto, Merge-Sort e Heap-Sort são algoritmos assintoticamente ótimos

É preciso atentar-se ao problema considerado!

- veremos algoritmos lineares para ordenação depois
- mas eles não são baseados em comparação
- e têm restrições adicionais sobre a entrada

Conclusão

O que aprendemos sobre o problema da ordenação?

- qualquer algoritmo executa $\Omega(n \log n)$ comparações
- assim, $\Omega(n \log n)$ é uma cota inferior para o problema
- portanto, Merge-Sort e Heap-Sort são algoritmos assintoticamente ótimos

É preciso atentar-se ao problema considerado!

- veremos algoritmos lineares para ordenação depois
- mas eles não são baseados em comparação
- e têm restrições adicionais sobre a entrada

Conclusão

O que aprendemos sobre o problema da ordenação?

- qualquer algoritmo executa $\Omega(n \log n)$ comparações
- assim, $\Omega(n \log n)$ é uma cota inferior para o problema
- portanto, Merge-Sort e Heap-Sort são algoritmos assintoticamente ótimos

É preciso atentar-se ao problema considerado!

- veremos algoritmos lineares para ordenação depois
- mas eles não são baseados em comparação
- e têm restrições adicionais sobre a entrada

Conclusão

O que aprendemos sobre o problema da ordenação?

- qualquer algoritmo executa $\Omega(n \log n)$ comparações
- assim, $\Omega(n \log n)$ é uma cota inferior para o problema
- portanto, Merge-Sort e Heap-Sort são algoritmos assintoticamente ótimos

É preciso atentar-se ao problema considerado!

- veremos algoritmos lineares para ordenação depois
- mas eles não são baseados em comparação
- e têm restrições adicionais sobre a entrada

Conclusão

O que aprendemos sobre o problema da ordenação?

- qualquer algoritmo executa $\Omega(n \log n)$ comparações
- assim, $\Omega(n \log n)$ é uma cota inferior para o problema
- portanto, Merge-Sort e Heap-Sort são algoritmos assintoticamente ótimos

É preciso atentar-se ao problema considerado!

- veremos algoritmos lineares para ordenação depois
- mas eles não são baseados em comparação
- e têm restrições adicionais sobre a entrada

Cotas inferiores

- Cota inferior para busca em vetor ordenado

Busca em vetor ordenado

Problema:

- **Entrada:** vetor crescente $A[p \ldots r]$ e elemento x
- **Saída:** índice i com $A[i] = x$, ou -1 se não existir.

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

Número de comparações: $O(\lg n)$.

Busca em vetor ordenado

Problema:

- **Entrada:** vetor crescente $A[p \ldots r]$ e elemento x
- **Saída:** índice i com $A[i] = x$, ou -1 se não existir.

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

Número de comparações: $O(\lg n)$.

Busca em vetor ordenado

Problema:

- **Entrada:** vetor crescente $A[p \ldots r]$ e elemento x
- **Saída:** índice i com $A[i] = x$, ou -1 se não existir.

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

Número de comparações: $O(\lg n)$.

Busca em vetor ordenado

Problema:

- **Entrada:** vetor crescente $A[p \ldots r]$ e elemento x
- **Saída:** índice i com $A[i] = x$, ou -1 se não existir.

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

Número de comparações: $O(\lg n)$.

Busca em vetor ordenado

Problema:

- **Entrada:** vetor crescente $A[p \ldots r]$ e elemento x
- **Saída:** índice i com $A[i] = x$, ou -1 se não existir.

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

Número de comparações: $O(\lg n)$.

Cota inferior para busca em vetor ordenado

É possível projetar um algoritmo mais rápido?

- não, se baseado em comparações $A[i] < x$ ou $A[i] > x$
- um algoritmo deve executar $\Omega(\lg n)$ comparações
- provamos isso usando uma árvore de decisão

Cota inferior para busca em vetor ordenado

É possível projetar um algoritmo mais rápido?

- não, se baseado em comparações $A[i] < x$ ou $A[i] > x$
- um algoritmo deve executar $\Omega(\lg n)$ comparações
- provamos isso usando uma árvore de decisão

Cota inferior para busca em vetor ordenado

É possível projetar um algoritmo mais rápido?

- não, se baseado em comparações $A[i] < x$ ou $A[i] > x$
- um algoritmo deve executar $\Omega(\lg n)$ comparações
- provamos isso usando uma árvore de decisão

Cota inferior para busca em vetor ordenado

É possível projetar um algoritmo mais rápido?

- não, se baseado em comparações $A[i] < x$ ou $A[i] > x$
- um algoritmo deve executar $\Omega(\lg n)$ comparações
- provamos isso usando uma árvore de decisão

Árvore de decisão para busca em vetor ordenado

Para cada algoritmo, definimos uma árvore de decisão

- **nós internos** correspondem comparações com x
- **subárvores** correspondem às saídas
- **folhas** correspondem às possíveis respostas

Obtendo uma cota inferior

- existem $n + 1$ respostas possíveis
- a árvore de decisão tem pelo menos $n + 1$ folhas
- daí, a altura é pelo menos $\Omega(\lg n)$

Árvore de decisão para busca em vetor ordenado

Para cada algoritmo, definimos uma árvore de decisão

- **nós internos** correspondem comparações com x
- **subárvores** correspondem às saídas
- **folhas** correspondem às possíveis respostas

Obtendo uma cota inferior

- existem $n + 1$ respostas possíveis
- a árvore de decisão tem pelo menos $n + 1$ folhas
- daí, a altura é pelo menos $\Omega(\lg n)$

Árvore de decisão para busca em vetor ordenado

Para cada algoritmo, definimos uma árvore de decisão

- **nós internos** correspondem comparações com x
- **subárvores** correspondem às saídas
- **folhas** correspondem às possíveis respostas

Obtendo uma cota inferior

- existem $n + 1$ respostas possíveis
- a árvore de decisão tem pelo menos $n + 1$ folhas
- daí, a altura é pelo menos $\Omega(\lg n)$

Árvore de decisão para busca em vetor ordenado

Para cada algoritmo, definimos uma árvore de decisão

- **nós internos** correspondem comparações com x
- **subárvores** correspondem às saídas
- **folhas** correspondem às possíveis respostas

Obtendo uma cota inferior

- existem $n + 1$ respostas possíveis
- a árvore de decisão tem pelo menos $n + 1$ folhas
- daí, a altura é pelo menos $\Omega(\lg n)$

Árvore de decisão para busca em vetor ordenado

Para cada algoritmo, definimos uma árvore de decisão

- **nós internos** correspondem comparações com x
- **subárvores** correspondem às saídas
- **folhas** correspondem às possíveis respostas

Obtendo uma cota inferior

- existem $n + 1$ respostas possíveis
- a árvore de decisão tem pelo menos $n + 1$ folhas
- daí, a altura é pelo menos $\Omega(\lg n)$

Árvore de decisão para busca em vetor ordenado

Para cada algoritmo, definimos uma árvore de decisão

- **nós internos** correspondem comparações com x
- **subárvores** correspondem às saídas
- **folhas** correspondem às possíveis respostas

Obtendo uma cota inferior

- existem $n + 1$ respostas possíveis
- a árvore de decisão tem pelo menos $n + 1$ folhas
- daí, a altura é pelo menos $\Omega(\lg n)$

Árvore de decisão para busca em vetor ordenado

Para cada algoritmo, definimos uma árvore de decisão

- **nós internos** correspondem comparações com x
- **subárvores** correspondem às saídas
- **folhas** correspondem às possíveis respostas

Obtendo uma cota inferior

- existem $n + 1$ respostas possíveis
- a árvore de decisão tem pelo menos $n + 1$ folhas
- daí, a altura é pelo menos $\Omega(\lg n)$

Árvore de decisão para busca em vetor ordenado

Para cada algoritmo, definimos uma árvore de decisão

- **nós internos** correspondem comparações com x
- **subárvores** correspondem às saídas
- **folhas** correspondem às possíveis respostas

Obtendo uma cota inferior

- existem $n + 1$ respostas possíveis
- a árvore de decisão tem pelo menos $n + 1$ folhas
- daí, a altura é pelo menos $\Omega(\lg n)$

Cotas inferiores

- Cota inferior para problema do máximo

Problema do máximo

Problema:

- **Entrada:** um vetor $A[1 \ldots n]$
- **Saída:** o maior elemento de A

O algoritmo trivial executa $n - 1$ comparações

- é o melhor algoritmo baseado em comparações
- para ver isso, vamos investigar um algoritmo genérico

Problema do máximo

Problema:

- **Entrada:** um vetor $A[1 \ldots n]$
- **Saída:** o maior elemento de A

O algoritmo trivial executa $n - 1$ comparações

- é o melhor algoritmo baseado em comparações
- para ver isso, vamos investigar um algoritmo genérico

Problema do máximo

Problema:

- **Entrada:** um vetor $A[1 \ldots n]$
- **Saída:** o maior elemento de A

O algoritmo trivial executa $n - 1$ comparações

- é o melhor algoritmo baseado em comparações
- para ver isso, vamos investigar um algoritmo genérico

Problema do máximo

Problema:

- **Entrada:** um vetor $A[1 \ldots n]$
- **Saída:** o maior elemento de A

O algoritmo trivial executa $n - 1$ comparações

- é o melhor algoritmo baseado em comparações
- para ver isso, vamos investigar um algoritmo genérico

Problema do máximo

Problema:

- **Entrada:** um vetor $A[1 \ldots n]$
- **Saída:** o maior elemento de A

O algoritmo trivial executa $n - 1$ comparações

- é o melhor algoritmo baseado em comparações
- para ver isso, vamos investigar um algoritmo genérico

Problema do máximo

Problema:

- **Entrada:** um vetor $A[1 \ldots n]$
- **Saída:** o maior elemento de A

O algoritmo trivial executa $n - 1$ comparações

- é o melhor algoritmo baseado em comparações
- para ver isso, vamos investigar um algoritmo genérico

Algoritmo genérico para o problema do máximo

Para uma algoritmo, defina uma coleção $\mathcal{A}$ de pares (i,j)

- um par (i,j) é um arco tal que $A[i] \leq A[j]$
- adicionamos um par para cada comparação executada

Algumas implicações da relação de comparação

- cada índice sem arco de saída é um sorvedouro
- se houver dois ou mais deles, não conhecemos o máximo

Assim, um algoritmo deve ter a seguinte forma

```
MÁXIMO(A, n)
1  A ← ∅
2  enquanto A não possui sorvedouro único faça
3      escolha índices 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
```

Algoritmo genérico para o problema do máximo

Para uma algoritmo, defina uma coleção $\mathcal{A}$ de pares (i,j)

- um par (i,j) é um arco tal que $A[i] \leq A[j]$
- adicionamos um par para cada comparação executada

Algumas implicações da relação de comparação

- cada índice sem arco de saída é um sorvedouro
- se houver dois ou mais deles, não conhecemos o máximo

Assim, um algoritmo deve ter a seguinte forma

```
MÁXIMO(A, n)
1  A ← ∅
2  enquanto A não possui sorvedouro único faça
3      escolha índices 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
```

Algoritmo genérico para o problema do máximo

Para uma algoritmo, defina uma coleção $\mathcal{A}$ de pares (i,j)

- um par (i,j) é um arco tal que $A[i] \leq A[j]$
- adicionamos um par para cada comparação executada

Algumas implicações da relação de comparação

- cada índice sem arco de saída é um sorvedouro
- se houver dois ou mais deles, não conhecemos o máximo

Assim, um algoritmo deve ter a seguinte forma

```
MÁXIMO(A, n)
1  A ← ∅
2  enquanto A não possui sorvedouro único faça
3      escolha índices 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
```

Algoritmo genérico para o problema do máximo

Para uma algoritmo, defina uma coleção $\mathcal{A}$ de pares (i,j)

- um par (i,j) é um arco tal que $A[i] \leq A[j]$
- adicionamos um par para cada comparação executada

Algumas implicações da relação de comparação

- cada índice sem arco de saída é um sorvedouro
- se houver dois ou mais deles, não conhecemos o máximo

Assim, um algoritmo deve ter a seguinte forma

```
MÁXIMO(A, n)
1  A ← ∅
2  enquanto A não possui sorvedouro único faça
3      escolha índices 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
```

Algoritmo genérico para o problema do máximo

Para uma algoritmo, defina uma coleção $\mathcal{A}$ de pares (i,j)

- um par (i,j) é um arco tal que $A[i] \leq A[j]$
- adicionamos um par para cada comparação executada

Algumas implicações da relação de comparação

- cada índice sem arco de saída é um sorvedouro
- se houver dois ou mais deles, não conhecemos o máximo

Assim, um algoritmo deve ter a seguinte forma

```
MÁXIMO(A, n)
1  A ← ∅
2  enquanto A não possui sorvedouro único faça
3      escolha índices 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
```

Algoritmo genérico para o problema do máximo

Para uma algoritmo, defina uma coleção $\mathcal{A}$ de pares (i,j)

- um par (i,j) é um arco tal que $A[i] \leq A[j]$
- adicionamos um par para cada comparação executada

Algumas implicações da relação de comparação

- cada índice sem arco de saída é um sorvedouro
- se houver dois ou mais deles, não conhecemos o máximo

Assim, um algoritmo deve ter a seguinte forma

```
MÁXIMO(A, n)
1  A ← ∅
2  enquanto A não possui sorvedouro único faça
3      escolha índices 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
```

Algoritmo genérico para o problema do máximo

Para uma algoritmo, defina uma coleção $\mathcal{A}$ de pares (i,j)

- um par (i,j) é um arco tal que $A[i] \leq A[j]$
- adicionamos um par para cada comparação executada

Algumas implicações da relação de comparação

- cada índice sem arco de saída é um sorvedouro
- se houver dois ou mais deles, não conhecemos o máximo

Assim, um algoritmo deve ter a seguinte forma

```
MÁXIMO(A, n)
1  A ← ∅
2  enquanto A não possui sorvedouro único faça
3      escolha índices 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
```

Algoritmo genérico para o problema do máximo

Para uma algoritmo, defina uma coleção $\mathcal{A}$ de pares (i,j)

- um par (i,j) é um arco tal que $A[i] \leq A[j]$
- adicionamos um par para cada comparação executada

Algumas implicações da relação de comparação

- cada índice sem arco de saída é um sorvedouro
- se houver dois ou mais deles, não conhecemos o máximo

Assim, um algoritmo deve ter a seguinte forma

```
MÁXIMO(A, n)
1  A ← ∅
2  enquanto A não possui sorvedouro único faça
3      escolha índices 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
```

Cota inferior para problema do máximo

Considere a coleção $\mathcal{A}$ devolvido

- $\mathcal{A}$ contém exatamente um sorvedouro
- isso induz uma árvore enraizada no índice do máximo
- uma árvore contém pelo menos $n - 1$ arcos

Concluímos

- todo algoritmo para o problema do máximo baseado em comparações executa pelo menos $n - 1$ comparações
- o algoritmo trivial dado é ótimo

Cota inferior para problema do máximo

Considere a coleção $\mathcal{A}$ devolvido

- $\mathcal{A}$ contém exatamente um sorvedouro
- isso induz uma árvore enraizada no índice do máximo
- uma árvore contém pelo menos $n - 1$ arcos

Concluímos

- todo algoritmo para o problema do máximo baseado em comparações executa pelo menos $n - 1$ comparações
- o algoritmo trivial dado é ótimo

Cota inferior para problema do máximo

Considere a coleção $\mathcal{A}$ devolvido

- $\mathcal{A}$ contém exatamente um sorvedouro
- isso induz uma árvore enraizada no índice do máximo
- uma árvore contém pelo menos $n - 1$ arcos

Concluímos

- todo algoritmo para o problema do máximo baseado em comparações executa pelo menos $n - 1$ comparações
- o algoritmo trivial dado é ótimo

Cota inferior para problema do máximo

Considere a coleção $\mathcal{A}$ devolvido

- $\mathcal{A}$ contém exatamente um sorvedouro
- isso induz uma árvore enraizada no índice do máximo
- uma árvore contém pelo menos $n - 1$ arcos

Concluímos

- todo algoritmo para o problema do máximo baseado em comparações executa pelo menos $n - 1$ comparações
- o algoritmo trivial dado é ótimo

Cota inferior para problema do máximo

Considere a coleção $\mathcal{A}$ devolvido

- $\mathcal{A}$ contém exatamente um sorvedouro
- isso induz uma árvore enraizada no índice do máximo
- uma árvore contém pelo menos $n - 1$ arcos

Concluímos

- todo algoritmo para o problema do máximo baseado em comparações executa pelo menos $n - 1$ comparações
- o algoritmo trivial dado é ótimo

Cota inferior para problema do máximo

Considere a coleção $\mathcal{A}$ devolvido

- $\mathcal{A}$ contém exatamente um sorvedouro
- isso induz uma árvore enraizada no índice do máximo
- uma árvore contém pelo menos $n - 1$ arcos

Concluímos

- todo algoritmo para o problema do máximo baseado em comparações executa pelo menos $n - 1$ comparações
- o algoritmo trivial dado é ótimo

Ordenação em tempo linear

Algoritmos lineares para ordenação

Estudaremos algoritmos de ordenação de tempo linear:

- Counting Sort
  - os elementos são inteiros pequenos
  - os valores são limitados por $O(n)$
- Radix Sort
  - representação numérica com comprimento constante
  - os valores dos elementos independente de n
- Bucket Sort
  - elementos do vetor são sorteados no intervalo $[0..1)$
  - os valores são distribuídos uniformemente

Algoritmos lineares para ordenação

Estudaremos algoritmos de ordenação de tempo linear:

- Counting Sort
  - os elementos são inteiros pequenos
  - os valores são limitados por $O(n)$
- Radix Sort
  - representação numérica com comprimento constante
  - os valores dos elementos independente de n
- Bucket Sort
  - elementos do vetor são sorteados no intervalo $[0..1)$
  - os valores são distribuídos uniformemente

Algoritmos lineares para ordenação

Estudaremos algoritmos de ordenação de tempo linear:

- Counting Sort
  - os elementos são inteiros pequenos
  - os valores são limitados por $O(n)$
- Radix Sort
  - representação numérica com comprimento constante
  - os valores dos elementos independente de n
- Bucket Sort
  - elementos do vetor são sorteados no intervalo $[0..1)$
  - os valores são distribuídos uniformemente

Algoritmos lineares para ordenação

Estudaremos algoritmos de ordenação de tempo linear:

- Counting Sort
  - os elementos são inteiros pequenos
  - os valores são limitados por $O(n)$
- Radix Sort
  - representação numérica com comprimento constante
  - os valores dos elementos independente de n
- Bucket Sort
  - elementos do vetor são sorteados no intervalo $[0..1)$
  - os valores são distribuídos uniformemente

Algoritmos lineares para ordenação

Estudaremos algoritmos de ordenação de tempo linear:

- Counting Sort
  - os elementos são inteiros pequenos
  - os valores são limitados por $O(n)$
- Radix Sort
  - representação numérica com comprimento constante
  - os valores dos elementos independente de n
- Bucket Sort
  - elementos do vetor são sorteados no intervalo $[0..1)$
  - os valores são distribuídos uniformemente

Algoritmos lineares para ordenação

Estudaremos algoritmos de ordenação de tempo linear:

- Counting Sort
  - os elementos são inteiros pequenos
  - os valores são limitados por $O(n)$
- Radix Sort
  - representação numérica com comprimento constante
  - os valores dos elementos independente de n
- Bucket Sort
  - elementos do vetor são sorteados no intervalo $[0..1)$
  - os valores são distribuídos uniformemente

Algoritmos lineares para ordenação

Estudaremos algoritmos de ordenação de tempo linear:

- Counting Sort
  - os elementos são inteiros pequenos
  - os valores são limitados por $O(n)$
- Radix Sort
  - representação numérica com comprimento constante
  - os valores dos elementos independente de n
- Bucket Sort
  - elementos do vetor são sorteados no intervalo $[0..1)$
  - os valores são distribuídos uniformemente

Algoritmos lineares para ordenação

Estudaremos algoritmos de ordenação de tempo linear:

- Counting Sort
  - os elementos são inteiros pequenos
  - os valores são limitados por $O(n)$
- Radix Sort
  - representação numérica com comprimento constante
  - os valores dos elementos independente de n
- Bucket Sort
  - elementos do vetor são sorteados no intervalo $[0..1)$
  - os valores são distribuídos uniformemente

Algoritmos lineares para ordenação

Estudaremos algoritmos de ordenação de tempo linear:

- Counting Sort
  - os elementos são inteiros pequenos
  - os valores são limitados por $O(n)$
- Radix Sort
  - representação numérica com comprimento constante
  - os valores dos elementos independente de n
- Bucket Sort
  - elementos do vetor são sorteados no intervalo $[0..1)$
  - os valores são distribuídos uniformemente

Algoritmos lineares para ordenação

Estudaremos algoritmos de ordenação de tempo linear:

- Counting Sort
  - os elementos são inteiros pequenos
  - os valores são limitados por $O(n)$
- Radix Sort
  - representação numérica com comprimento constante
  - os valores dos elementos independente de n
- Bucket Sort
  - elementos do vetor são sorteados no intervalo $[0..1)$
  - os valores são distribuídos uniformemente

Ordenação em tempo linear

- Counting sort

Counting Sort

Ideia:

- entrada é um vetor $A[1 \ldots n]$ de inteiros
- saída será outro vetor $B[1 \ldots n]$ de inteiros
- supomos que o valores estão no intervalo entre 0 e k
- para cada inteiro i no vetor, contamos o número $C[i]$ de elementos menores ou iguais i em A
- então, na posição $C[i]$ do vetor B, deve haver o elemento i

Counting Sort

Ideia:

- entrada é um vetor $A[1 \ldots n]$ de inteiros
- saída será outro vetor $B[1 \ldots n]$ de inteiros
- supomos que o valores estão no intervalo entre 0 e k
- para cada inteiro i no vetor, contamos o número $C[i]$ de elementos menores ou iguais i em A
- então, na posição $C[i]$ do vetor B, deve haver o elemento i

Counting Sort

Ideia:

- entrada é um vetor $A[1 \ldots n]$ de inteiros
- saída será outro vetor $B[1 \ldots n]$ de inteiros
- supomos que o valores estão no intervalo entre 0 e k
- para cada inteiro i no vetor, contamos o número $C[i]$ de elementos menores ou iguais i em A
- então, na posição $C[i]$ do vetor B, deve haver o elemento i

Counting Sort

Ideia:

- entrada é um vetor $A[1 \ldots n]$ de inteiros
- saída será outro vetor $B[1 \ldots n]$ de inteiros
- supomos que o valores estão no intervalo entre 0 e k
- para cada inteiro i no vetor, contamos o número $C[i]$ de elementos menores ou iguais i em A
- então, na posição $C[i]$ do vetor B, deve haver o elemento i

Counting Sort

Ideia:

- entrada é um vetor $A[1 \ldots n]$ de inteiros
- saída será outro vetor $B[1 \ldots n]$ de inteiros
- supomos que o valores estão no intervalo entre 0 e k
- para cada inteiro i no vetor, contamos o número $C[i]$ de elementos menores ou iguais i em A
- então, na posição $C[i]$ do vetor B, deve haver o elemento i

Counting Sort

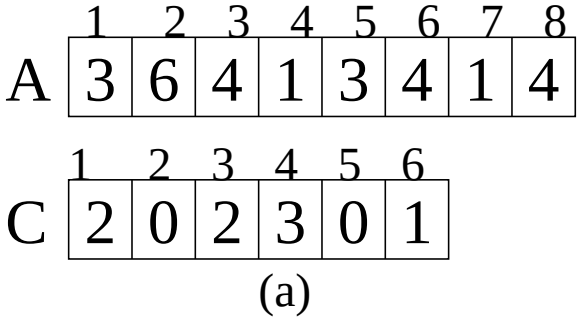
Ideia:

- entrada é um vetor $A[1 \ldots n]$ de inteiros
- saída será outro vetor $B[1 \ldots n]$ de inteiros
- supomos que o valores estão no intervalo entre 0 e k
- para cada inteiro i no vetor, contamos o número $C[i]$ de elementos menores ou iguais i em A
- então, na posição $C[i]$ do vetor B, deve haver o elemento i

Counting Sort - Algoritmo

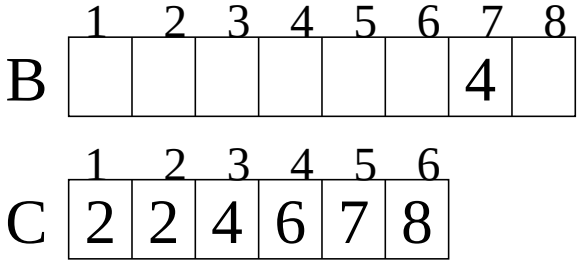
```
COUNTING-SORT(A, B, n, k)
1   para i ← 0 até k faça
2       C[i] ← 0

3   para j ← 1 até n faça
4       C[A[j]] ← C[A[j]] + 1

5   para i ← 1 até k faça
6       C[i] ← C[i] + C[i − 1]

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 - Exemplo

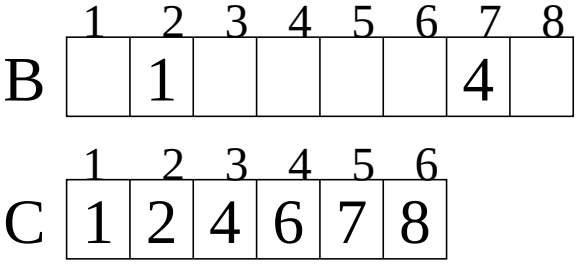


(a)

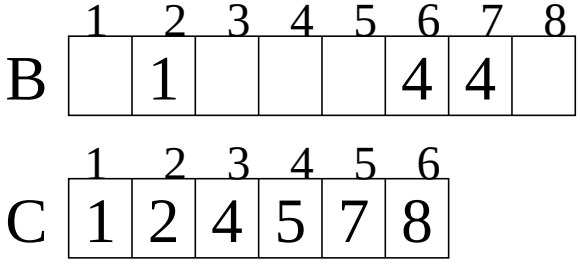
(b)



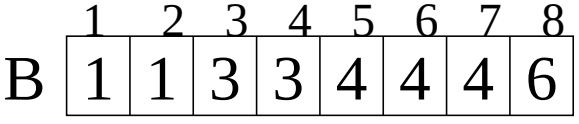
(c)



(d)



(e)



(f)

Counting Sort - Complexidade

Qual a complexidade de Counting-Sort?

- Counting-Sort realiza $O(n+k)$ instruções elementares
- quando $k \in O(n)$, ele tem complexidade $O(n)$

E a cota inferior de $\Omega(n \log n)$ para ordenação?

- não há comparações entre elementos de A
- a cota só vale para algoritmos baseados em comparação

Counting Sort - Complexidade

Qual a complexidade de Counting-Sort?

- Counting-Sort realiza $O(n+k)$ instruções elementares
- quando $k \in O(n)$, ele tem complexidade $O(n)$

E a cota inferior de $\Omega(n \log n)$ para ordenação?

- não há comparações entre elementos de A
- a cota só vale para algoritmos baseados em comparação

Counting Sort - Complexidade

Qual a complexidade de Counting-Sort?

- Counting-Sort realiza $O(n+k)$ instruções elementares
- quando $k \in O(n)$, ele tem complexidade $O(n)$

E a cota inferior de $\Omega(n \log n)$ para ordenação?

- não há comparações entre elementos de A
- a cota só vale para algoritmos baseados em comparação

Counting Sort - Complexidade

Qual a complexidade de Counting-Sort?

- Counting-Sort realiza $O(n+k)$ instruções elementares
- quando $k \in O(n)$, ele tem complexidade $O(n)$

E a cota inferior de $\Omega(n \log n)$ para ordenação?

- não há comparações entre elementos de A
- a cota só vale para algoritmos baseados em comparação

Counting Sort - Complexidade

Qual a complexidade de Counting-Sort?

- Counting-Sort realiza $O(n+k)$ instruções elementares
- quando $k \in O(n)$, ele tem complexidade $O(n)$

E a cota inferior de $\Omega(n \log n)$ para ordenação?

- não há comparações entre elementos de A
- a cota só vale para algoritmos baseados em comparação

Counting Sort - Complexidade

Qual a complexidade de Counting-Sort?

- Counting-Sort realiza $O(n+k)$ instruções elementares
- quando $k \in O(n)$, ele tem complexidade $O(n)$

E a cota inferior de $\Omega(n \log n)$ para ordenação?

- não há comparações entre elementos de A
- a cota só vale para algoritmos baseados em comparação

Algoritmos in-place e estáveis

Algoritmos de ordenação in-place

- um algoritmo é in-place se a quantidade de memória adicional requerida independe de n
- são in-place Insertion-Sort, Heap-Sort
- não são in-place Merge-Sort, QuickSort, Counting-Sort

Algoritmos de ordenação estáveis

- um algoritmo é estável se elementos com chaves iguais mantêm-se na ordem passada na entrada.
- são estáveis Insertion-Sort, Merge-Sort, QuickSort, Counting-Sort
- o Heap-Sort não é estável

Algoritmos in-place e estáveis

Algoritmos de ordenação in-place

- um algoritmo é in-place se a quantidade de memória adicional requerida independe de n
- são in-place Insertion-Sort, Heap-Sort
- não são in-place Merge-Sort, QuickSort, Counting-Sort

Algoritmos de ordenação estáveis

- um algoritmo é estável se elementos com chaves iguais mantêm-se na ordem passada na entrada.
- são estáveis Insertion-Sort, Merge-Sort, QuickSort, Counting-Sort
- o Heap-Sort não é estável

Algoritmos in-place e estáveis

Algoritmos de ordenação in-place

- um algoritmo é in-place se a quantidade de memória adicional requerida independe de n
- são in-place Insertion-Sort, Heap-Sort
- não são in-place Merge-Sort, QuickSort, Counting-Sort

Algoritmos de ordenação estáveis

- um algoritmo é estável se elementos com chaves iguais mantêm-se na ordem passada na entrada.
- são estáveis Insertion-Sort, Merge-Sort, QuickSort, Counting-Sort
- o Heap-Sort não é estável

Algoritmos in-place e estáveis

Algoritmos de ordenação in-place

- um algoritmo é in-place se a quantidade de memória adicional requerida independe de n
- são in-place Insertion-Sort, Heap-Sort
- não são in-place Merge-Sort, QuickSort, Counting-Sort

Algoritmos de ordenação estáveis

- um algoritmo é estável se elementos com chaves iguais mantêm-se na ordem passada na entrada.
- são estáveis Insertion-Sort, Merge-Sort, QuickSort, Counting-Sort
- o Heap-Sort não é estável

Algoritmos in-place e estáveis

Algoritmos de ordenação in-place

- um algoritmo é in-place se a quantidade de memória adicional requerida independe de n
- são in-place Insertion-Sort, Heap-Sort
- não são in-place Merge-Sort, QuickSort, Counting-Sort

Algoritmos de ordenação estáveis

- um algoritmo é estável se elementos com chaves iguais mantêm-se na ordem passada na entrada.
- são estáveis Insertion-Sort, Merge-Sort, QuickSort, Counting-Sort
- o Heap-Sort não é estável

Algoritmos in-place e estáveis

Algoritmos de ordenação in-place

- um algoritmo é in-place se a quantidade de memória adicional requerida independe de n
- são in-place Insertion-Sort, Heap-Sort
- não são in-place Merge-Sort, QuickSort, Counting-Sort

Algoritmos de ordenação estáveis

- um algoritmo é estável se elementos com chaves iguais mantêm-se na ordem passada na entrada.
- são estáveis Insertion-Sort, Merge-Sort, QuickSort, Counting-Sort
- o Heap-Sort não é estável

Algoritmos in-place e estáveis

Algoritmos de ordenação in-place

- um algoritmo é in-place se a quantidade de memória adicional requerida independe de n
- são in-place Insertion-Sort, Heap-Sort
- não são in-place Merge-Sort, QuickSort, Counting-Sort

Algoritmos de ordenação estáveis

- um algoritmo é estável se elementos com chaves iguais mantêm-se na ordem passada na entrada.
- são estáveis Insertion-Sort, Merge-Sort, QuickSort, Counting-Sort
- o Heap-Sort não é estável

Algoritmos in-place e estáveis

Algoritmos de ordenação in-place

- um algoritmo é in-place se a quantidade de memória adicional requerida independe de n
- são in-place Insertion-Sort, Heap-Sort
- não são in-place Merge-Sort, QuickSort, Counting-Sort

Algoritmos de ordenação estáveis

- um algoritmo é estável se elementos com chaves iguais mantêm-se na ordem passada na entrada.
- são estáveis Insertion-Sort, Merge-Sort, QuickSort, Counting-Sort
- o Heap-Sort não é estável

Ordenação em tempo linear

- Radix sort

Radix Sort

Ideia:

- a entrada é um vetor $A[1 \ldots n]$ inteiros
- cada inteiro é representado em na base k-ária
- supomos que um inteiro tem apenas d dígitos
- por exemplo, CEPs são inteiros de 8 dígitos na base 10
- ordenaremos um dígito por vez com um algoritmo estável
- ordenamos primeiro os dígitos menos significativos

Radix Sort

Ideia:

- a entrada é um vetor $A[1 \ldots n]$ inteiros
- cada inteiro é representado em na base k-ária
- supomos que um inteiro tem apenas d dígitos
- por exemplo, CEPs são inteiros de 8 dígitos na base 10
- ordenaremos um dígito por vez com um algoritmo estável
- ordenamos primeiro os dígitos menos significativos

Radix Sort

Ideia:

- a entrada é um vetor $A[1 \ldots n]$ inteiros
- cada inteiro é representado em na base k-ária
- supomos que um inteiro tem apenas d dígitos
- por exemplo, CEPs são inteiros de 8 dígitos na base 10
- ordenaremos um dígito por vez com um algoritmo estável
- ordenamos primeiro os dígitos menos significativos

Radix Sort

Ideia:

- a entrada é um vetor $A[1 \ldots n]$ inteiros
- cada inteiro é representado em na base k-ária
- supomos que um inteiro tem apenas d dígitos
- por exemplo, CEPs são inteiros de 8 dígitos na base 10
- ordenaremos um dígito por vez com um algoritmo estável
- ordenamos primeiro os dígitos menos significativos

Radix Sort

Ideia:

- a entrada é um vetor $A[1 \ldots n]$ inteiros
- cada inteiro é representado em na base k-ária
- supomos que um inteiro tem apenas d dígitos
- por exemplo, CEPs são inteiros de 8 dígitos na base 10
- ordenaremos um dígito por vez com um algoritmo estável
- ordenamos primeiro os dígitos menos significativos

Radix Sort

Ideia:

- a entrada é um vetor $A[1 \ldots n]$ inteiros
- cada inteiro é representado em na base k-ária
- supomos que um inteiro tem apenas d dígitos
- por exemplo, CEPs são inteiros de 8 dígitos na base 10
- ordenaremos um dígito por vez com um algoritmo estável
- ordenamos primeiro os dígitos menos significativos

Radix Sort

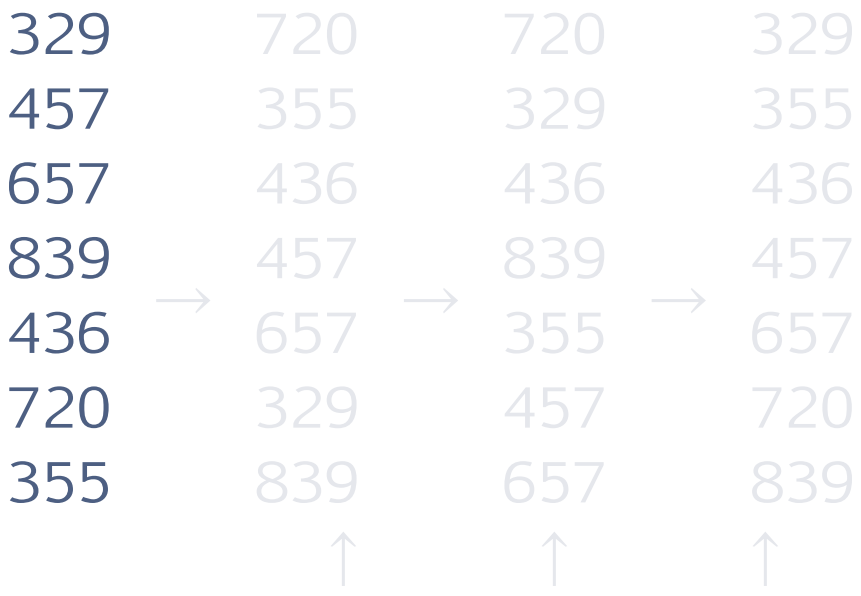
Ideia:

- a entrada é um vetor $A[1 \ldots n]$ inteiros
- cada inteiro é representado em na base k-ária
- supomos que um inteiro tem apenas d dígitos
- por exemplo, CEPs são inteiros de 8 dígitos na base 10
- ordenaremos um dígito por vez com um algoritmo estável
- ordenamos primeiro os dígitos menos significativos

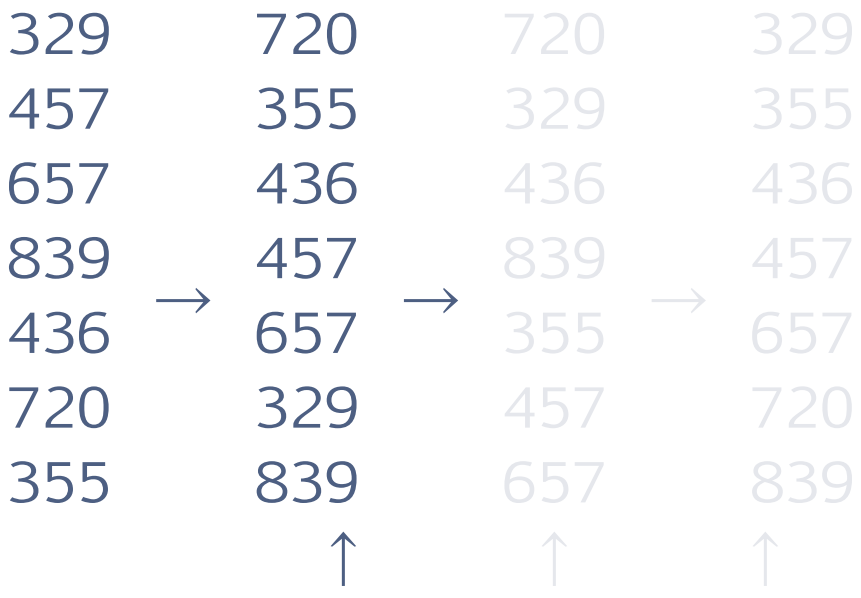
Radix Sort - Algoritmo

```
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
```

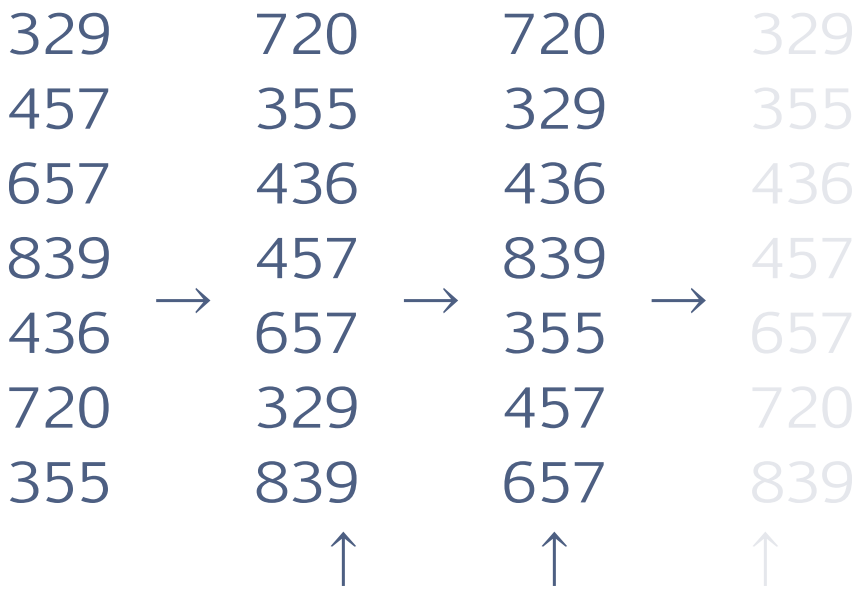
Radix Sort - Exemplo



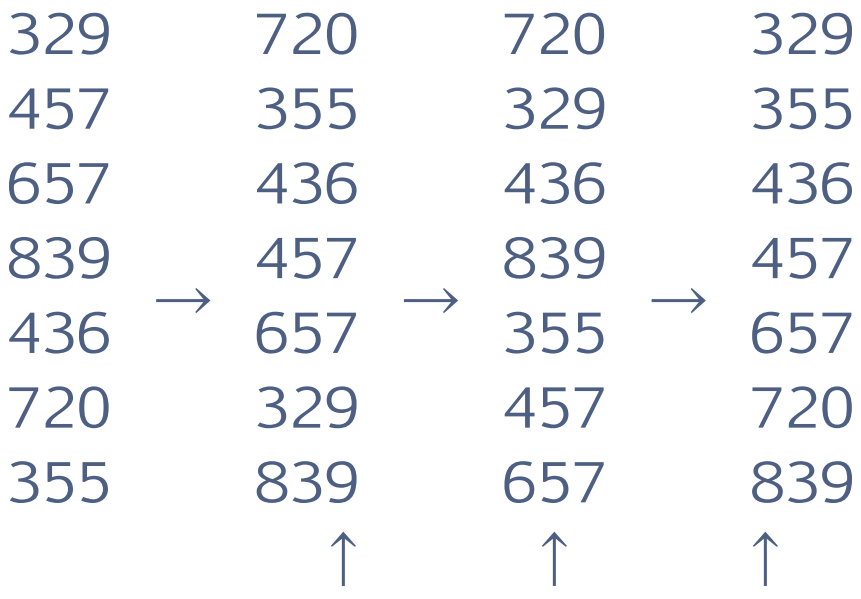
Radix Sort - Exemplo



Radix Sort - Exemplo



Radix Sort - Exemplo



Radix Sort - Correção

Demonstramos a correção do algoritmo por indução

- suponha que o vetor de números está ordenado relação aos $i - 1$ dígitos menos significativos
- ordene o vetor pelo i-ésimo dígito com um método estável
- considere dois números adjacentes no vetor resultante:
  1. se os i-ésimos dígitos forem distintos, então eles estão em ordem com relação aos i dígitos menos significativos, independentemente dos $i - 1$ primeiros dígitos
  2. se os i-ésimos dígitos forem iguais, então eles estão em ordem com relação aos i dígitos menos significativos, já que estão em ordem pelos $i - 1$ primeiros dígitos, pois o método é estável

Radix Sort - Correção

Demonstramos a correção do algoritmo por indução

- suponha que o vetor de números está ordenado relação aos $i - 1$ dígitos menos significativos
- ordene o vetor pelo i-ésimo dígito com um método estável
- considere dois números adjacentes no vetor resultante:
  1. se os i-ésimos dígitos forem distintos, então eles estão em ordem com relação aos i dígitos menos significativos, independentemente dos $i - 1$ primeiros dígitos
  2. se os i-ésimos dígitos forem iguais, então eles estão em ordem com relação aos i dígitos menos significativos, já que estão em ordem pelos $i - 1$ primeiros dígitos, pois o método é estável

Radix Sort - Correção

Demonstramos a correção do algoritmo por indução

- suponha que o vetor de números está ordenado relação aos $i - 1$ dígitos menos significativos
- ordene o vetor pelo i-ésimo dígito com um método estável
- considere dois números adjacentes no vetor resultante:
  1. se os i-ésimos dígitos forem distintos, então eles estão em ordem com relação aos i dígitos menos significativos, independentemente dos $i - 1$ primeiros dígitos
  2. se os i-ésimos dígitos forem iguais, então eles estão em ordem com relação aos i dígitos menos significativos, já que estão em ordem pelos $i - 1$ primeiros dígitos, pois o método é estável

Radix Sort - Correção

Demonstramos a correção do algoritmo por indução

- suponha que o vetor de números está ordenado relação aos $i - 1$ dígitos menos significativos
- ordene o vetor pelo i-ésimo dígito com um método estável
- considere dois números adjacentes no vetor resultante:
  1. se os i-ésimos dígitos forem distintos, então eles estão em ordem com relação aos i dígitos menos significativos, independentemente dos $i - 1$ primeiros dígitos
  2. se os i-ésimos dígitos forem iguais, então eles estão em ordem com relação aos i dígitos menos significativos, já que estão em ordem pelos $i - 1$ primeiros dígitos, pois o método é estável

Radix Sort - Correção

Demonstramos a correção do algoritmo por indução

- suponha que o vetor de números está ordenado relação aos $i - 1$ dígitos menos significativos
- ordene o vetor pelo i-ésimo dígito com um método estável
- considere dois números adjacentes no vetor resultante:
  1. se os i-ésimos dígitos forem distintos, então eles estão em ordem com relação aos i dígitos menos significativos, independentemente dos $i - 1$ primeiros dígitos
  2. se os i-ésimos dígitos forem iguais, então eles estão em ordem com relação aos i dígitos menos significativos, já que estão em ordem pelos $i - 1$ primeiros dígitos, pois o método é estável

Radix Sort - Correção

Demonstramos a correção do algoritmo por indução

- suponha que o vetor de números está ordenado relação aos $i - 1$ dígitos menos significativos
- ordene o vetor pelo i-ésimo dígito com um método estável
- considere dois números adjacentes no vetor resultante:
  1. se os i-ésimos dígitos forem distintos, então eles estão em ordem com relação aos i dígitos menos significativos, independentemente dos $i - 1$ primeiros dígitos
  2. se os i-ésimos dígitos forem iguais, então eles estão em ordem com relação aos i dígitos menos significativos, já que estão em ordem pelos $i - 1$ primeiros dígitos, pois o método é estável

Radix Sort - Complexidade

Qual é a complexidade do RADIX-SORT?

- o algoritmo de ordenação usado tem tempo $\Theta(f(n))$
- então RADIX-SORT tem tempo total $\Theta(d\,f(n))$
- quando d é constante, a complexidade é $\Theta(f(n))$
- se usarmos COUNTING-SORT, a complexidade é $\Theta(n+k)$
- se k é contante, então o tempo resultante é $\Theta(n)$

Radix Sort - Complexidade

Qual é a complexidade do RADIX-SORT?

- o algoritmo de ordenação usado tem tempo $\Theta(f(n))$
- então RADIX-SORT tem tempo total $\Theta(d\,f(n))$
- quando d é constante, a complexidade é $\Theta(f(n))$
- se usarmos COUNTING-SORT, a complexidade é $\Theta(n+k)$
- se k é contante, então o tempo resultante é $\Theta(n)$

Radix Sort - Complexidade

Qual é a complexidade do RADIX-SORT?

- o algoritmo de ordenação usado tem tempo $\Theta(f(n))$
- então RADIX-SORT tem tempo total $\Theta(d\,f(n))$
- quando d é constante, a complexidade é $\Theta(f(n))$
- se usarmos COUNTING-SORT, a complexidade é $\Theta(n+k)$
- se k é contante, então o tempo resultante é $\Theta(n)$

Radix Sort - Complexidade

Qual é a complexidade do RADIX-SORT?

- o algoritmo de ordenação usado tem tempo $\Theta(f(n))$
- então RADIX-SORT tem tempo total $\Theta(d\,f(n))$
- quando d é constante, a complexidade é $\Theta(f(n))$
- se usarmos COUNTING-SORT, a complexidade é $\Theta(n+k)$
- se k é contante, então o tempo resultante é $\Theta(n)$

Radix Sort - Complexidade

Qual é a complexidade do RADIX-SORT?

- o algoritmo de ordenação usado tem tempo $\Theta(f(n))$
- então RADIX-SORT tem tempo total $\Theta(d\,f(n))$
- quando d é constante, a complexidade é $\Theta(f(n))$
- se usarmos COUNTING-SORT, a complexidade é $\Theta(n+k)$
- se k é contante, então o tempo resultante é $\Theta(n)$

Radix Sort - Complexidade

Qual é a complexidade do RADIX-SORT?

- o algoritmo de ordenação usado tem tempo $\Theta(f(n))$
- então RADIX-SORT tem tempo total $\Theta(d\,f(n))$
- quando d é constante, a complexidade é $\Theta(f(n))$
- se usarmos COUNTING-SORT, a complexidade é $\Theta(n+k)$
- se k é contante, então o tempo resultante é $\Theta(n)$

Radix Sort - Comparação

Vamos comparar Radix-Sort e Merge-Sort

- temos $n = 2^{20}$ números de 64 bits
- interpretamos os números na base $k = 2^{16}$
- então cada um tem até $d = 4$ dígitos

1. Ordenando com Merge-Sort
   - executamos cerca de $n \lg n = 20 \times 2^{20}$ comparações
   - usamos um vetor auxiliar de tamanho 2^{20}

2. Ordenando com Radix-Sort
   - utilizamos Counting-Sort como sub-rotina de ordenação
   - executamos cerca de $d(n + k) = 4(2^{20} + 2^{16})$ operações
   - usamos dois vetores auxiliares de tamanhos 2^{16} e 2^{20}

Conclusão: Radix-Sort foi mais rápido, mas usa mais memória

Radix Sort - Comparação

Vamos comparar Radix-Sort e Merge-Sort

- temos $n = 2^{20}$ números de 64 bits
- interpretamos os números na base $k = 2^{16}$
- então cada um tem até $d = 4$ dígitos

1. Ordenando com Merge-Sort
   - executamos cerca de $n \lg n = 20 \times 2^{20}$ comparações
   - usamos um vetor auxiliar de tamanho 2^{20}
2. Ordenando com Radix-Sort
   - utilizamos Counting-Sort como sub-rotina de ordenação
   - executamos cerca de $d(n + k) = 4(2^{20} + 2^{16})$ operações
   - usamos dois vetores auxiliares de tamanhos 2^{16} e 2^{20}

Conclusão: Radix-Sort foi mais rápido, mas usa mais memória

Radix Sort - Comparação

Vamos comparar Radix-Sort e Merge-Sort

- temos $n = 2^{20}$ números de 64 bits
- interpretamos os números na base $k = 2^{16}$
- então cada um tem até $d = 4$ dígitos

1. Ordenando com Merge-Sort
   - executamos cerca de $n \lg n = 20 \times 2^{20}$ comparações
   - usamos um vetor auxiliar de tamanho 2^{20}

2. Ordenando com Radix-Sort
   - utilizamos Counting-Sort como sub-rotina de ordenação
   - executamos cerca de $d(n + k) = 4(2^{20} + 2^{16})$ operações
   - usamos dois vetores auxiliares de tamanhos 2^{16} e 2^{20}

Conclusão: Radix-Sort foi mais rápido, mas usa mais memória

Radix Sort - Comparação

Vamos comparar Radix-Sort e Merge-Sort

- temos $n = 2^{20}$ números de 64 bits
- interpretamos os números na base $k = 2^{16}$
- então cada um tem até $d = 4$ dígitos

1. Ordenando com Merge-Sort
   - executamos cerca de $n \lg n = 20 \times 2^{20}$ comparações
   - usamos um vetor auxiliar de tamanho 2^{20}

2. Ordenando com Radix-Sort
   - utilizamos Counting-Sort como sub-rotina de ordenação
   - executamos cerca de $d(n + k) = 4(2^{20} + 2^{16})$ operações
   - usamos dois vetores auxiliares de tamanhos 2^{16} e 2^{20}

Conclusão: Radix-Sort foi mais rápido, mas usa mais memória

Radix Sort - Comparação

Vamos comparar Radix-Sort e Merge-Sort

- temos $n = 2^{20}$ números de 64 bits
- interpretamos os números na base $k = 2^{16}$
- então cada um tem até $d = 4$ dígitos

1. Ordenando com Merge-Sort
   - executamos cerca de $n \lg n = 20 \times 2^{20}$ comparações
   - usamos um vetor auxiliar de tamanho 2^{20}
2. Ordenando com Radix-Sort
   - utilizamos Counting-Sort como sub-rotina de ordenação
   - executamos cerca de $d(n + k) = 4(2^{20} + 2^{16})$ operações
   - usamos dois vetores auxiliares de tamanhos 2^{16} e 2^{20}

Conclusão: Radix-Sort foi mais rápido, mas usa mais memória

Radix Sort - Comparação

Vamos comparar Radix-Sort e Merge-Sort

- temos $n = 2^{20}$ números de 64 bits
- interpretamos os números na base $k = 2^{16}$
- então cada um tem até $d = 4$ dígitos

1. Ordenando com Merge-Sort
   - executamos cerca de $n \lg n = 20 \times 2^{20}$ comparações
   - usamos um vetor auxiliar de tamanho 2^{20}
2. Ordenando com Radix-Sort
   - utilizamos Counting-Sort como sub-rotina de ordenação
   - executamos cerca de $d(n + k) = 4(2^{20} + 2^{16})$ operações
   - usamos dois vetores auxiliares de tamanhos 2^{16} e 2^{20}

Conclusão: Radix-Sort foi mais rápido, mas usa mais memória

Radix Sort - Comparação

Vamos comparar Radix-Sort e Merge-Sort

- temos $n = 2^{20}$ números de 64 bits
- interpretamos os números na base $k = 2^{16}$
- então cada um tem até $d = 4$ dígitos

1. Ordenando com Merge-Sort
   - executamos cerca de $n \lg n = 20 \times 2^{20}$ comparações
   - usamos um vetor auxiliar de tamanho 2^{20}
2. Ordenando com Radix-Sort
   - utilizamos Counting-Sort como sub-rotina de ordenação
   - executamos cerca de $d(n + k) = 4(2^{20} + 2^{16})$ operações
   - usamos dois vetores auxiliares de tamanhos 2^{16} e 2^{20}

Conclusão: Radix-Sort foi mais rápido, mas usa mais memória

Radix Sort - Comparação

Vamos comparar Radix-Sort e Merge-Sort

- temos $n = 2^{20}$ números de 64 bits
- interpretamos os números na base $k = 2^{16}$
- então cada um tem até $d = 4$ dígitos

1. Ordenando com Merge-Sort
   - executamos cerca de $n \lg n = 20 \times 2^{20}$ comparações
   - usamos um vetor auxiliar de tamanho 2^{20}
2. Ordenando com Radix-Sort
   - utilizamos Counting-Sort como sub-rotina de ordenação
   - executamos cerca de $d(n + k) = 4(2^{20} + 2^{16})$ operações
   - usamos dois vetores auxiliares de tamanhos 2^{16} e 2^{20}

Conclusão: Radix-Sort foi mais rápido, mas usa mais memória

Radix Sort - Comparação

Vamos comparar Radix-Sort e Merge-Sort

- temos $n = 2^{20}$ números de 64 bits
- interpretamos os números na base $k = 2^{16}$
- então cada um tem até $d = 4$ dígitos

1. Ordenando com Merge-Sort
   - executamos cerca de $n \lg n = 20 \times 2^{20}$ comparações
   - usamos um vetor auxiliar de tamanho 2^{20}
2. Ordenando com Radix-Sort
   - utilizamos Counting-Sort como sub-rotina de ordenação
   - executamos cerca de $d(n + k) = 4(2^{20} + 2^{16})$ operações
   - usamos dois vetores auxiliares de tamanhos 2^{16} e 2^{20}

Conclusão: Radix-Sort foi mais rápido, mas usa mais memória

Radix Sort - Comparação

Vamos comparar Radix-Sort e Merge-Sort

- temos $n = 2^{20}$ números de 64 bits
- interpretamos os números na base $k = 2^{16}$
- então cada um tem até $d = 4$ dígitos

1. Ordenando com Merge-Sort
   - executamos cerca de $n \lg n = 20 \times 2^{20}$ comparações
   - usamos um vetor auxiliar de tamanho 2^{20}
2. Ordenando com Radix-Sort
   - utilizamos Counting-Sort como sub-rotina de ordenação
   - executamos cerca de $d(n + k) = 4(2^{20} + 2^{16})$ operações
   - usamos dois vetores auxiliares de tamanhos 2^{16} e 2^{20}

Conclusão: Radix-Sort foi mais rápido, mas usa mais memória

Radix Sort - Comparação

Vamos comparar Radix-Sort e Merge-Sort

- temos $n = 2^{20}$ números de 64 bits
- interpretamos os números na base $k = 2^{16}$
- então cada um tem até $d = 4$ dígitos

1. Ordenando com Merge-Sort
   - executamos cerca de $n \lg n = 20 \times 2^{20}$ comparações
   - usamos um vetor auxiliar de tamanho 2^{20}
2. Ordenando com Radix-Sort
   - utilizamos Counting-Sort como sub-rotina de ordenação
   - executamos cerca de $d(n + k) = 4(2^{20} + 2^{16})$ operações
   - usamos dois vetores auxiliares de tamanhos 2^{16} e 2^{20}

Conclusão: Radix-Sort foi mais rápido, mas usa mais memória

Radix Sort - Comparação

Vamos comparar Radix-Sort e Merge-Sort

- temos $n = 2^{20}$ números de 64 bits
- interpretamos os números na base $k = 2^{16}$
- então cada um tem até $d = 4$ dígitos

1. Ordenando com Merge-Sort
   - executamos cerca de $n \lg n = 20 \times 2^{20}$ comparações
   - usamos um vetor auxiliar de tamanho 2^{20}
2. Ordenando com Radix-Sort
   - utilizamos Counting-Sort como sub-rotina de ordenação
   - executamos cerca de $d(n + k) = 4(2^{20} + 2^{16})$ operações
   - usamos dois vetores auxiliares de tamanhos 2^{16} e 2^{20}

Conclusão: Radix-Sort foi mais rápido, mas usa mais memória

Ordenação em tempo linear

- Bucket sort

Bucket Sort

Ideia:

- temos n números em $[0,1)$ distribuídos uniformemente
- dividimos $[0,1)$ em n segmentos de tamanhos iguais
- particionamos os elementos em listas (buckets) correspondentes aos segmentos
- o número esperado de elementos por lista é constante
- podemos ordenar cada lista independentemente
- e concatenar os listas já ordenadas

Bucket Sort

Ideia:

- temos n números em $[0,1)$ distribuídos uniformemente
- dividimos $[0,1)$ em n segmentos de tamanhos iguais
- particionamos os elementos em listas (buckets) correspondentes aos segmentos
- o número esperado de elementos por lista é constante
- podemos ordenar cada lista independentemente
- e concatenar os listas já ordenadas

Bucket Sort

Ideia:

- temos n números em $[0, 1)$ distribuídos uniformemente
- dividimos $[0, 1)$ em n segmentos de tamanhos iguais
- particionamos os elementos em listas (buckets) correspondentes aos segmentos
- o número esperado de elementos por lista é constante
- podemos ordenar cada lista independentemente
- e concatenar os listas já ordenadas

Bucket Sort

Ideia:

- temos n números em $[0, 1)$ distribuídos uniformemente
- dividimos $[0, 1)$ em n segmentos de tamanhos iguais
- particionamos os elementos em listas (buckets) correspondentes aos segmentos
- o número esperado de elementos por lista é constante
- podemos ordenar cada lista independentemente
- e concatenar os listas já ordenadas

Bucket Sort

Ideia:

- temos n números em $[0, 1)$ distribuídos uniformemente
- dividimos $[0, 1)$ em n segmentos de tamanhos iguais
- particionamos os elementos em listas (buckets) correspondentes aos segmentos
- o número esperado de elementos por lista é constante
- podemos ordenar cada lista independentemente
- e concatenar os listas já ordenadas

Bucket Sort

Ideia:

- temos n números em $[0, 1)$ distribuídos uniformemente
- dividimos $[0, 1)$ em n segmentos de tamanhos iguais
- particionamos os elementos em listas (buckets) correspondentes aos segmentos
- o número esperado de elementos por lista é constante
- podemos ordenar cada lista independentemente
- e concatenar os listas já ordenadas

Bucket Sort

Ideia:

- temos n números em $[0, 1)$ distribuídos uniformemente
- dividimos $[0, 1)$ em n segmentos de tamanhos iguais
- particionamos os elementos em listas (buckets) correspondentes aos segmentos
- o número esperado de elementos por lista é constante
- podemos ordenar cada lista independentemente
- e concatenar os listas já ordenadas

Bucket Sort - Algoritmo

```
BUCKET-SORT(A, n)
1   para i ← 0 até n − 1 faça
2       crie uma lista ligada vazia B[i]
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

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

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

Bucket Sort - Exemplo

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

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

Bucket Sort - Correção

Considere dois elementos x e y da entrada com $x < y$:

- se ambos terminam na mesma lista
  - x aparecerá antes de y já que a lista foi ordenada
  - e se manterão ordenados após a concatenação
- sem terminam em listas $B[i]$ e $B[j]$, respectivamente
  - como $x < y$, temos $i = \lfloor nx \rfloor \leq \lfloor ny \rfloor = j$ e então $i < j$
  - assim, x aparecerá antes de y após a concatenação

Bucket Sort - Correção

Considere dois elementos x e y da entrada com $x < y$:

- se ambos terminam na mesma lista
  - x aparecerá antes de y já que a lista foi ordenada
  - e se manterão ordenados após a concatenação
- sem terminam em listas $B[i]$ e $B[j]$, respectivamente
  - como $x < y$, temos $i = \lfloor nx \rfloor \leq \lfloor ny \rfloor = j$ e então $i < j$
  - assim, x aparecerá antes de y após a concatenação

Bucket Sort - Correção

Considere dois elementos x e y da entrada com $x < y$:

- se ambos terminam na mesma lista
  - x aparecerá antes de y já que a lista foi ordenada
  - e se manterão ordenados após a concatenação
- sem terminam em listas $B[i]$ e $B[j]$, respectivamente
  - como $x < y$, temos $i = \lfloor nx \rfloor \leq \lfloor ny \rfloor = j$ e então $i < j$
  - assim, x aparecerá antes de y após a concatenação

Bucket Sort - Correção

Considere dois elementos x e y da entrada com $x < y$:

- se ambos terminam na mesma lista
  - x aparecerá antes de y já que a lista foi ordenada
  - e se manterão ordenados após a concatenação
- sem terminam em listas $B[i]$ e $B[j]$, respectivamente
  - como $x < y$, temos $i = \lfloor nx \rfloor \leq \lfloor ny \rfloor = j$ e então $i < j$
  - assim, x aparecerá antes de y após a concatenação

Bucket Sort - Correção

Considere dois elementos x e y da entrada com $x < y$:

- se ambos terminam na mesma lista
  - x aparecerá antes de y já que a lista foi ordenada
  - e se manterão ordenados após a concatenação
- sem terminam em listas $B[i]$ e $B[j]$, respectivamente
  - como $x < y$, temos $i = \lfloor nx \rfloor \leq \lfloor ny \rfloor = j$ e então $i < j$
  - assim, x aparecerá antes de y após a concatenação

Bucket Sort - Correção

Considere dois elementos x e y da entrada com $x < y$:

- se ambos terminam na mesma lista
  - x aparecerá antes de y já que a lista foi ordenada
  - e se manterão ordenados após a concatenação
- sem terminam em listas $B[i]$ e $B[j]$, respectivamente
  - como $x < y$, temos $i = \lfloor nx \rfloor \leq \lfloor ny \rfloor = j$ e então $i < j$
  - assim, x aparecerá antes de y após a concatenação

Bucket Sort - Correção

Considere dois elementos x e y da entrada com $x < y$:

- se ambos terminam na mesma lista
  - x aparecerá antes de y já que a lista foi ordenada
  - e se manterão ordenados após a concatenação
- sem terminam em listas $B[i]$ e $B[j]$, respectivamente
  - como $x < y$, temos $i = \lfloor nx \rfloor \leq \lfloor ny \rfloor = j$ e então $i < j$
  - assim, x aparecerá antes de y após a concatenação

Bucket Sort - Complexidade

O tempo de execução é uma váriável aleatória

- denote por $T(n)$ o tempo de execução do algoritmo
- denote por n_i o tamanho de $B[i]$
- observe que o número de elementos é $n = \sum_{i=1}^{n} n_i$

Somando o tempo das operações

- particionamos os elementos em tempo $\Theta(n)$
- ordenamos cada lista em tempo $\Theta(n_i^2)$
- concatenamos todas as listas em tempo $\Theta(n)$

$$T(n) = \sum_{i=0}^{n-1} \Theta(n_i^2)$$

Bucket Sort - Complexidade

O tempo de execução é uma váriável aleatória

- denote por $T(n)$ o tempo de execução do algoritmo
- denote por n_i o tamanho de $B[i]$
- observe que o número de elementos é $n = \sum_{i=1}^{n} n_i$

Somando o tempo das operações

- particionamos os elementos em tempo $\Theta(n)$
- ordenamos cada lista em tempo $\Theta(n_i^2)$
- concatenamos todas as listas em tempo $\Theta(n)$

$$T(n) = \sum_{i=0}^{n-1} \Theta(n_i^2)$$

Bucket Sort - Complexidade

O tempo de execução é uma váriável aleatória

- denote por $T(n)$ o tempo de execução do algoritmo
- denote por n_i o tamanho de $B[i]$
- observe que o número de elementos é $n = \sum_{i=1}^{n} n_i$

Somando o tempo das operações

- particionamos os elementos em tempo $\Theta(n)$
- ordenamos cada lista em tempo $\Theta(n_i^2)$
- concatenamos todas as listas em tempo $\Theta(n)$

$$T(n) = \sum_{i=0}^{n-1} \Theta(n_i^2)$$

Bucket Sort - Complexidade

O tempo de execução é uma váriável aleatória

- denote por $T(n)$ o tempo de execução do algoritmo
- denote por n_i o tamanho de $B[i]$
- observe que o número de elementos é $n = \sum_{i=1}^{n} n_i$

Somando o tempo das operações

- particionamos os elementos em tempo $\Theta(n)$
- ordenamos cada lista em tempo $\Theta(n_i^2)$
- concatenamos todas as listas em tempo $\Theta(n)$

$$T(n) = \sum_{i=0}^{n-1} \Theta(n_i^2)$$

Bucket Sort - Complexidade

O tempo de execução é uma váriável aleatória

- denote por $T(n)$ o tempo de execução do algoritmo
- denote por n_i o tamanho de $B[i]$
- observe que o número de elementos é $n = \sum_{i=1}^{n} n_i$

Somando o tempo das operações

- particionamos os elementos em tempo $\Theta(n)$
- ordenamos cada lista em tempo $\Theta(n_i^2)$
- concatenamos todas as listas em tempo $\Theta(n)$

$$T(n) = \sum_{i=0}^{n-1} \Theta(n_i^2)$$

Bucket Sort - Complexidade

O tempo de execução é uma váriável aleatória

- denote por $T(n)$ o tempo de execução do algoritmo
- denote por n_i o tamanho de $B[i]$
- observe que o número de elementos é $n = \sum_{i=1}^{n} n_i$

Somando o tempo das operações

- particionamos os elementos em tempo $\Theta(n)$
- ordenamos cada lista em tempo $\Theta(n_i^2)$
- concatenamos todas as listas em tempo $\Theta(n)$

$$T(n) = \sum_{i=0}^{n-1} \Theta(n_i^2)$$

Bucket Sort - Complexidade

O tempo de execução é uma váriável aleatória

- denote por $T(n)$ o tempo de execução do algoritmo
- denote por n_i o tamanho de $B[i]$
- observe que o número de elementos é $n = \sum_{i=1}^{n} n_i$

Somando o tempo das operações

- particionamos os elementos em tempo $\Theta(n)$
- ordenamos cada lista em tempo $\Theta(n_i^2)$
- concatenamos todas as listas em tempo $\Theta(n)$

$$T(n) = \sum_{i=0}^{n-1} \Theta(n_i^2)$$

Bucket Sort - Complexidade

O tempo de execução é uma váriável aleatória

- denote por $T(n)$ o tempo de execução do algoritmo
- denote por n_i o tamanho de $B[i]$
- observe que o número de elementos é $n = \sum_{i=1}^{n} n_i$

Somando o tempo das operações

- particionamos os elementos em tempo $\Theta(n)$
- ordenamos cada lista em tempo $\Theta(n_i^2)$
- concatenamos todas as listas em tempo $\Theta(n)$

$$T(n) = \sum_{i=0}^{n-1} \Theta(n_i^2)$$

Bucket Sort - Complexidade

O tempo de execução é uma váriável aleatória

- denote por $T(n)$ o tempo de execução do algoritmo
- denote por n_i o tamanho de $B[i]$
- observe que o número de elementos é $n = \sum_{i=1}^{n} n_i$

Somando o tempo das operações

- particionamos os elementos em tempo $\Theta(n)$
- ordenamos cada lista em tempo $\Theta(n_i^2)$
- concatenamos todas as listas em tempo $\Theta(n)$

$$T(n) = \sum_{i=0}^{n-1} \Theta(n_i^2)$$

Bucket Sort - Pior caso

Pior caso

$$T(n) \leq \sum_{i=0}^{n-1} cn_i^2 \leq cn^2 = O(n^2)$$

- um pior caso ocorre se todos números caem em uma lista
- o tempo de execução é maior se há listas muito grandes
- mas o número esperado em cada lista é pequeno

Bucket Sort - Pior caso

Pior caso

$$T(n) \leq \sum_{i=0}^{n-1} cn_i^2 \leq cn^2 = O(n^2)$$

- um pior caso ocorre se todos números caem em uma lista
- o tempo de execução é maior se há listas muito grandes
- mas o número esperado em cada lista é pequeno

Bucket Sort - Pior caso

Pior caso

$$T(n) \leq \sum_{i=0}^{n-1} cn_i^2 \leq cn^2 = O(n^2)$$

- um pior caso ocorre se todos números caem em uma lista
- o tempo de execução é maior se há listas muito grandes
- mas o número esperado em cada lista é pequeno

Bucket Sort - Pior caso

Pior caso

$$T(n) \leq \sum_{i=0}^{n-1} cn_i^2 \leq cn^2 = O(n^2)$$

- um pior caso ocorre se todos números caem em uma lista
- o tempo de execução é maior se há listas muito grandes
- mas o número esperado em cada lista é pequeno

Bucket Sort - Pior caso

Pior caso

$$T(n) \leq \sum_{i=0}^{n-1} cn_i^2 \leq cn^2 = O(n^2)$$

- um pior caso ocorre se todos números caem em uma lista
- o tempo de execução é maior se há listas muito grandes
- mas o número esperado em cada lista é pequeno

Bucket Sort - Pior caso

Pior caso

$$T(n) \leq \sum_{i=0}^{n-1} cn_i^2 \leq cn^2 = O(n^2)$$

- um pior caso ocorre se todos números caem em uma lista
- o tempo de execução é maior se há listas muito grandes
- mas o número esperado em cada lista é pequeno

Bucket Sort - Tempo esperado

Tempo esperado

$$
\begin{aligned}
E[T(n)] &= E\left[\sum_{i=0}^{n-1} cn_i^2\right] \\
&= \sum_{i=0}^{n-1} E[cn_i^2] \\
&= \sum_{i=0}^{n-1} cE[n_i^2]
\end{aligned}
$$

Bucket Sort - Tempo esperado (cont)

Queremos calcular $E[n_i^2]$

- seja X_{ij} a variável que indica se $A[j]$ está em $B[i]$
- assim, temos $n_i = \sum_{j=1}^{n} X_{ij}$

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

Bucket Sort - Tempo esperado (cont)

Queremos calcular $E[n_i^2]$

- seja X_{ij} a variável que indica se $A[j]$ está em $B[i]$
- assim, temos $n_i = \sum_{j=1}^{n} X_{ij}$

$$E[n_i^2] = E\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right]$$

$$= E\left[\left(\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij}X_{ik}\right)\right]$$

$$= E\left[\sum_{j=1}^{n} X_{ij}^2 + \sum_{j=1}^{n}\sum_{k=1,k\neq j}^{n} X_{ij}X_{ik}\right]$$

$$= \sum_{j=1}^{n} E[X_{ij}^2] + \sum_{j=1}^{n}\sum_{k=1,k\neq j}^{n} E[X_{ij}X_{ik}]$$

Bucket Sort - Tempo esperado (cont)

Queremos calcular $E[n_i^2]$

- seja X_{ij} a variável que indica se $A[j]$ está em $B[i]$
- assim, temos $n_i = \sum_{j=1}^{n} X_{ij}$

$$E[n_i^2] = E\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right]$$

$$= E\left[\left(\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij}X_{ik}\right)\right]$$

$$= E\left[\sum_{j=1}^{n} X_{ij}^2 + \sum_{j=1}^{n}\sum_{k=1,k\neq j}^{n} X_{ij}X_{ik}\right]$$

$$= \sum_{j=1}^{n} E[X_{ij}^2] + \sum_{j=1}^{n}\sum_{k=1,k\neq j}^{n} E[X_{ij}X_{ik}]$$

Bucket Sort - Tempo esperado (cont)

Queremos calcular $E[n_i^2]$

- seja X_{ij} a variável que indica se $A[j]$ está em $B[i]$
- assim, temos $n_i = \sum_{j=1}^{n} X_{ij}$

$$E[n_i^2] = E\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right]$$

$$= E\left[\left(\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij}X_{ik}\right)\right]$$

$$= E\left[\sum_{j=1}^{n} X_{ij}^2 + \sum_{j=1}^{n}\sum_{k=1,k\neq j}^{n} X_{ij}X_{ik}\right]$$

$$= \sum_{j=1}^{n} E[X_{ij}^2] + \sum_{j=1}^{n}\sum_{k=1,k\neq j}^{n} E[X_{ij}X_{ik}]$$

Bucket Sort - Tempo esperado (cont)

Queremos calcular $E[n_i^2]$

- seja X_{ij} a variável que indica se $A[j]$ está em $B[i]$
- assim, temos $n_i = \sum_{j=1}^{n} X_{ij}$

$$E[n_i^2] = E\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right]$$

$$= E\left[\left(\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij}X_{ik}\right)\right]$$

$$= E\left[\sum_{j=1}^{n} X_{ij}^2 + \sum_{j=1}^{n}\sum_{k=1,k\neq j}^{n} X_{ij}X_{ik}\right]$$

$$= \sum_{j=1}^{n} E[X_{ij}^2] + \sum_{j=1}^{n}\sum_{k=1,k\neq j}^{n} E[X_{ij}X_{ik}]$$

Bucket Sort - Tempo esperado (cont)

Queremos calcular $E[n_i^2]$

- seja X_{ij} a variável que indica se $A[j]$ está em $B[i]$
- assim, temos $n_i = \sum_{j=1}^{n} X_{ij}$

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

Bucket Sort - Tempo esperado (cont)

Queremos calcular $E[n_i^2]$

- seja X_{ij} a variável que indica se $A[j]$ está em $B[i]$
- assim, temos $n_i = \sum_{j=1}^{n} X_{ij}$

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

Bucket Sort - Tempo esperado (cont)

Como X_{ij} e X_{ik} são independentes

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

Bucket Sort - Tempo esperado (cont)

Como X_{ij} e X_{ik} são independentes

$$E[n_i^2] = \sum_{j=1}^{n} E[X_{ij}^2] + \sum_{j=1}^{n} \sum_{k=1,k\neq j}^{n} E[X_{ij}X_{ik}]$$

$$= \sum_{j=1}^{n} E[X_{ij}] + \sum_{j=1}^{n} \sum_{k=1,k\neq j}^{n} E[X_{ij}]E[X_{ik}]$$

$$= \sum_{j=1}^{n} \frac{1}{n} + \sum_{j=1}^{n} \sum_{k=1,k\neq j}^{n} \frac{1}{n}\frac{1}{n}$$

$$= 1 + n(n-1)\frac{1}{n^2}$$

$$= 2 - \frac{1}{n} = O(1)$$

Bucket Sort - Tempo esperado (cont)

Como X_{ij} e X_{ik} são independentes

$$E[n_i^2] = \sum_{j=1}^{n} E[X_{ij}^2] + \sum_{j=1}^{n} \sum_{k=1,k\neq j}^{n} E[X_{ij}X_{ik}]$$

$$= \sum_{j=1}^{n} E[X_{ij}] + \sum_{j=1}^{n} \sum_{k=1,k\neq j}^{n} E[X_{ij}]E[X_{ik}]$$

$$= \sum_{j=1}^{n} \frac{1}{n} + \sum_{j=1}^{n} \sum_{k=1,k\neq j}^{n} \frac{1}{n}\frac{1}{n}$$

$$= 1 + n(n-1)\frac{1}{n^2}$$

$$= 2 - \frac{1}{n} = O(1)$$

Bucket Sort - Tempo esperado (cont)

Como X_{ij} e X_{ik} são independentes

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

Bucket Sort - Tempo esperado (cont)

Como X_{ij} e X_{ik} são independentes

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

Bucket Sort - Tempo esperado (cont)

Como X_{ij} e X_{ik} são independentes

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

Bucket Sort - Tempo esperado (cont)

Voltando à estimativa de $E[T(n)]$, temos

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

Bucket Sort - Tempo esperado (cont)

Voltando à estimativa de $E[T(n)]$, temos

$$E[T(n)] = \sum_{i=0}^{n-1} cE[n_i^2]$$

$$= \sum_{i=0}^{n-1} cO(1)$$

$$= \Theta(n)$$

Bucket Sort - Tempo esperado (cont)

Voltando à estimativa de $E[T(n)]$, temos

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

Bucket Sort - Tempo esperado (cont)

Voltando à estimativa de $E[T(n)]$, temos

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