Projeto e Análise de Algoritmos

Cotas inferiores

Lehilton Pedrosa

Primeiro Semestre de 2023

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
- executam $\Theta(n \log n)$ comparações no pior caso
- QUICKSORT também tem cota $\Theta(n \log n)$ no caso médio

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
- executam $\Theta(n \log n)$ comparações no pior caso
- QUICKSORT também tem cota $\Theta(n \log n)$ no caso médio

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
- executam $\Theta(n \log n)$ comparações no pior caso
- QUICKSORT também tem cota $\Theta(n \log n)$ no caso médio

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
- executam $\Theta(n \log n)$ comparações no pior caso
- QUICKSORT também tem cota $\Theta(n \log n)$ no caso médio

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
- executam $\Theta(n \log n)$ comparações no pior caso
- QUICKSORT também tem cota $\Theta(n \log n)$ no caso médio

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
- executam $\Theta(n \log n)$ comparações no pior caso
- QUICKSORT também tem cota $\Theta(n \log n)$ no caso médio

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
- executam $\Theta(n \log n)$ comparações no pior caso
- QUICKSORT também tem cota $\Theta(n \log n)$ no caso médio

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
- executam $\Theta(n \log n)$ comparações no pior caso
- QUICKSORT também tem cota $\Theta(n \log n)$ no caso médio

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
- executam $\Theta(n \log n)$ comparações no pior caso
- QUICKSORT também tem cota $\Theta(n \log n)$ no caso médio

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
- executam $\Theta(n \log n)$ comparações no pior caso
- QUICKSORT também tem cota $\Theta(n \log n)$ no caso médio

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

- só temos informação sobre os 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

- só temos informação sobre os 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

- só temos informação sobre os 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

- só temos informação sobre os 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

- só temos informação sobre os 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

- só temos informação sobre os 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 permutação 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 permutação 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 permutação 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 permutação 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 permutação 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 permutação 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 permutação 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 permutação da entrada

Ordenação como problema de permutação

Podemos redefinir o problema da ordenação

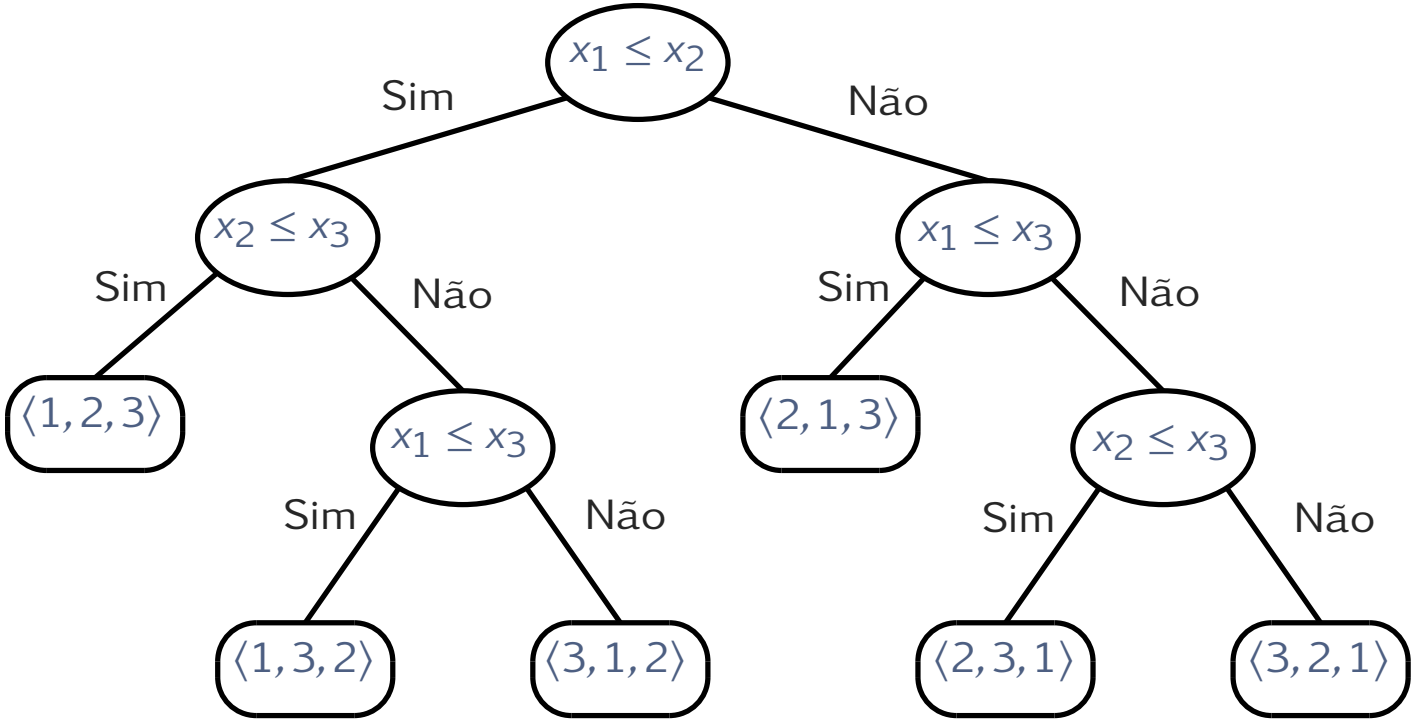
- **Entrada:** vetor de n inteiros $x_1, x_2, \ldots, x_n$
- **Saída:** permutação p tal que $x_{p(1)} \leq x_{p(2)} \leq \ldots \leq x_{p(n)}$

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 permutação 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, $h \geq \log_2 n!$

Desenvolvendo o limitante inferior

$$\log_2 n! = \sum_{i=1}^{n} \log_2 i \geq \sum_{i=\lceil n/2 \rceil}^{n} \log_2 n/2 \geq n/2 \cdot \log_2 n/2$$

Então $h \in \Omega(n \log n)$.

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, $h \geq \log_2 n!$

Desenvolvendo o limitante inferior

$$\log_2 n! = \sum_{i=1}^{n} \log_2 i \geq \sum_{i=\lceil n/2 \rceil}^{n} \log_2 n/2 \geq n/2 \cdot \log_2 n/2$$

Então $h \in \Omega(n \log n)$.

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, $h \geq \log_2 n!$

Desenvolvendo o limitante inferior

$$\log_2 n! = \sum_{i=1}^{n} \log_2 i \geq \sum_{i=\lceil n/2 \rceil}^{n} \log_2 n/2 \geq n/2 \cdot \log_2 n/2$$

Então $h \in \Omega(n \log n)$.

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, $h \geq \log_2 n!$

Desenvolvendo o limitante inferior

$$\log_2 n! = \sum_{i=1}^{n} \log_2 i \geq \sum_{i=\lceil n/2 \rceil}^{n} \log_2 n/2 \geq n/2 \cdot \log_2 n/2$$

Então $h \in \Omega(n \log n)$.

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, $h \geq \log_2 n!$

Desenvolvendo o limitante inferior

$$\log_2 n! = \sum_{i=1}^{n} \log_2 i \geq \sum_{i=\lceil n/2 \rceil}^{n} \log_2 n/2 \geq n/2 \cdot \log_2 n/2$$

Então $h \in \Omega(n \log n)$.

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, $h \geq \log_2 n!$

Desenvolvendo o limitante inferior

$$\log_2 n! = \sum_{i=1}^{n} \log_2 i \geq \sum_{i=\lceil n/2 \rceil}^{n} \log_2 n/2 \geq n/2 \cdot \log_2 n/2$$

Então $h \in \Omega(n \log n)$.

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, $h \geq \log_2 n!$

Desenvolvendo o limitante inferior

$$\log_2 n! = \sum_{i=1}^{n} \log_2 i \geq \sum_{i=\lceil n/2 \rceil}^{n} \log_2 n/2 \geq n/2 \cdot \log_2 n/2$$

Então $h \in \Omega(n \log n)$.

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

Número de comparações: $O(\log n)$.

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

Número de comparações: $O(\log n)$.

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

Número de comparações: $O(\log n)$.

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

Número de comparações: $O(\log n)$.

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

Número de comparações: $O(\log n)$.

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 $\Omega(\log n)$ 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 $\Omega(\log n)$ 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 $\Omega(\log n)$ 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 $\Omega(\log n)$ 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 $\Omega(\log n)$

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 $\Omega(\log n)$

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 $\Omega(\log n)$

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 $\Omega(\log n)$

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 $\Omega(\log n)$

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 $\Omega(\log n)$

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 $\Omega(\log n)$

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 $\Omega(\log n)$

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) representa uma $A[i] \leq A[j]$
- dizemos que i perdeu a disputa
- adicionamos um par para cada comparação executada

```
MÁXIMO(A, n)
1  A ← ∅
2  enquanto algoritmo não encontrou o máximo
3      escolha índices i e j em {1, ..., n}
4      se A[i] ≤ A[j]
5          A ← A ∪ {(i, j)}
6      senão
7          A ← A ∪ {(j, i)}
8  devolva o máximo encontrado
```

Algoritmo genérico para o problema do máximo

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

- um par (i, j) representa uma $A[i] \leq A[j]$
- dizemos que i perdeu a disputa
- adicionamos um par para cada comparação executada

```
MÁXIMO(A, n)
1  A ← ∅
2  enquanto algoritmo não encontrou o máximo
3      escolha índices i e j em {1, ..., n}
4      se A[i] ≤ A[j]
5          A ← A ∪ {(i, j)}
6      senão
7          A ← A ∪ {(j, i)}
8  devolva o máximo encontrado
```

Algoritmo genérico para o problema do máximo

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

- um par (i, j) representa uma $A[i] \leq A[j]$
- dizemos que i perdeu a disputa
- adicionamos um par para cada comparação executada

```
MÁXIMO(A, n)
1  A ← ∅
2  enquanto algoritmo não encontrou o máximo
3      escolha índices i e j em {1, ..., n}
4      se A[i] ≤ A[j]
5          A ← A ∪ {(i, j)}
6      senão
7          A ← A ∪ {(j, i)}
8  devolva o máximo encontrado
```

Algoritmo genérico para o problema do máximo

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

- um par (i, j) representa uma $A[i] \leq A[j]$
- dizemos que i perdeu a disputa
- adicionamos um par para cada comparação executada

```
MÁXIMO(A, n)
1  A ← ∅
2  enquanto algoritmo não encontrou o máximo
3      escolha índices i e j em {1, ..., n}
4      se A[i] ≤ A[j]
5          A ← A ∪ {(i, j)}
6      senão
7          A ← A ∪ {(j, i)}
8  devolva o máximo encontrado
```

Algoritmo genérico para o problema do máximo

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

- um par (i, j) representa uma $A[i] \leq A[j]$
- dizemos que i perdeu a disputa
- adicionamos um par para cada comparação executada

```
MÁXIMO(A, n)
1  A ← ∅
2  enquanto algoritmo não encontrou o máximo
3      escolha índices i e j em {1, ..., n}
4      se A[i] ≤ A[j]
5          A ← A ∪ {(i, j)}
6      senão
7          A ← A ∪ {(j, i)}
8  devolva o máximo encontrado
```

Cota inferior para problema do máximo

Considere a coleção de pares $\mathcal{A}$

- o máximo nunca perde uma disputa
- cada um dos outros deve perder pelo menos uma vez
- assim $\mathcal{A}$ tem pelo menos $n-1$ pares

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 de pares $\mathcal{A}$

- o máximo nunca perde uma disputa
- cada um dos outros deve perder pelo menos uma vez
- assim $\mathcal{A}$ tem pelo menos $n - 1$ pares

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 de pares $\mathcal{A}$

- o máximo nunca perde uma disputa
- cada um dos outros deve perder pelo menos uma vez
- assim $\mathcal{A}$ tem pelo menos $n - 1$ pares

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 de pares $\mathcal{A}$

- o máximo nunca perde uma disputa
- cada um dos outros deve perder pelo menos uma vez
- assim $\mathcal{A}$ tem pelo menos $n - 1$ pares

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 de pares $\mathcal{A}$

- o máximo nunca perde uma disputa
- cada um dos outros deve perder pelo menos uma vez
- assim $\mathcal{A}$ tem pelo menos $n - 1$ pares

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 de pares $\mathcal{A}$

- o máximo nunca perde uma disputa
- cada um dos outros deve perder pelo menos uma vez
- assim $\mathcal{A}$ tem pelo menos $n - 1$ pares

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 de pares $\mathcal{A}$

- o máximo nunca perde uma disputa
- cada um dos outros deve perder pelo menos uma vez
- assim $\mathcal{A}$ tem pelo menos $n - 1$ pares

Concluímos

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

Cotas inferiores

- Cotas inferiores a partir de outras cotas inferiores

Cotas inferiores a partir de cotas inferiores conhecidas

É possível mostrar que um problema P tem uma cota inferior $\Omega(T_P(n))$ usando uma cota inferior $\Omega(T_Q(n))$ já conhecida para algum problema Q.
A estratégia é a seguinte:

1. mostre que com um algoritmo A_P para resolver P pode-se criar um algoritmo A_Q para resolver Q
2. a complexidade de A_Q dependerá da de A_P
3. mostre que se A_P tem complexidade $o(T_P(n))$, então A_Q tem complexidade $o(T_Q(n))$
4. mas já sabemos que A_Q roda em $\Omega(T_Q(n))$, logo, A_P não pode estar em $o(T_P(n))$
5. ou seja, A_P deve ter complexidade $\Omega(T_P(n))$ (QED)

Cotas inferiores a partir de cotas inferiores conhecidas

É possível mostrar que um problema P tem uma cota inferior $\Omega(T_P(n))$ usando uma cota inferior $\Omega(T_Q(n))$ já conhecida para algum problema Q.
A estratégia é a seguinte:

1. mostre que com um algoritmo A_P para resolver P pode-se criar um algoritmo A_Q para resolver Q
2. a complexidade de A_Q dependerá da de A_P
3. mostre que se A_P tem complexidade $o(T_P(n))$, então A_Q tem complexidade $o(T_Q(n))$
4. mas já sabemos que A_Q roda em $\Omega(T_Q(n))$, logo, A_P não pode estar em $o(T_P(n))$
5. ou seja, A_P deve ter complexidade $\Omega(T_P(n))$ (QED)

Cotas inferiores a partir de cotas inferiores conhecidas

É possível mostrar que um problema P tem uma cota inferior $\Omega(T_P(n))$ usando uma cota inferior $\Omega(T_Q(n))$ já conhecida para algum problema Q.
A estratégia é a seguinte:

1. mostre que com um algoritmo A_P para resolver P pode-se criar um algoritmo A_Q para resolver Q
2. a complexidade de A_Q dependerá da de A_P
3. mostre que se A_P tem complexidade $o(T_P(n))$, então A_Q tem complexidade $o(T_Q(n))$
4. mas já sabemos que A_Q roda em $\Omega(T_Q(n))$, logo, A_P não pode estar em $o(T_P(n))$
5. ou seja, A_P deve ter complexidade $\Omega(T_P(n))$ (QED)

Cotas inferiores a partir de cotas inferiores conhecidas

É possível mostrar que um problema P tem uma cota inferior $\Omega(T_P(n))$ usando uma cota inferior $\Omega(T_Q(n))$ já conhecida para algum problema Q.
A estratégia é a seguinte:

1. mostre que com um algoritmo A_P para resolver P pode-se criar um algoritmo A_Q para resolver Q
2. a complexidade de A_Q dependerá da de A_P
3. mostre que se A_P tem complexidade $o(T_P(n))$, então A_Q tem complexidade $o(T_Q(n))$
4. mas já sabemos que A_Q roda em $\Omega(T_Q(n))$, logo, A_P não pode estar em $o(T_P(n))$
5. ou seja, A_P deve ter complexidade $\Omega(T_P(n))$ (QED)

Cotas inferiores a partir de cotas inferiores conhecidas

É possível mostrar que um problema P tem uma cota inferior $\Omega(T_P(n))$ usando uma cota inferior $\Omega(T_Q(n))$ já conhecida para algum problema Q.
A estratégia é a seguinte:

1. mostre que com um algoritmo A_P para resolver P pode-se criar um algoritmo A_Q para resolver Q
2. a complexidade de A_Q dependerá da de A_P
3. mostre que se A_P tem complexidade $o(T_P(n))$, então A_Q tem complexidade $o(T_Q(n))$
4. mas já sabemos que A_Q roda em $\Omega(T_Q(n))$, logo, A_P não pode estar em $o(T_P(n))$
5. ou seja, A_P deve ter complexidade $\Omega(T_P(n))$ (QED)

Cotas inferiores a partir de cotas inferiores conhecidas

É possível mostrar que um problema P tem uma cota inferior $\Omega(T_P(n))$ usando uma cota inferior $\Omega(T_Q(n))$ já conhecida para algum problema Q.
A estratégia é a seguinte:

1. mostre que com um algoritmo A_P para resolver P pode-se criar um algoritmo A_Q para resolver Q
2. a complexidade de A_Q dependerá da de A_P
3. mostre que se A_P tem complexidade $o(T_P(n))$, então A_Q tem complexidade $o(T_Q(n))$
4. mas já sabemos que A_Q roda em $\Omega(T_Q(n))$, logo, A_P não pode estar em $o(T_P(n))$
5. ou seja, A_P deve ter complexidade $\Omega(T_P(n))$ (QED)

Isso nos dá uma prova do tipo "se o problema Q tem cota inferior $\Omega(T_Q(n))$, então o problema P tem cota inferior $\Omega(T_P(n))$"

- quando já conhecemos cotas inferiores para Q, obtemos cotas inferiores para P
- mas mesmo quando não conhecemos tais cotas, elas podem ser conjecturadas e essa prova continua sendo útil
- nesse caso, podemos interpretá-la como "acredita-se que Q tem cota inferior $\Omega(T_Q(n))$, logo, P deve ter cota inferior $\Omega(T_P(n))$"

Isso nos dá uma prova do tipo "se o problema Q tem cota inferior $\Omega(T_Q(n))$, então o problema P tem cota inferior $\Omega(T_P(n))$"

- quando já conhecemos cotas inferiores para Q, obtemos cotas inferiores para P
- mas mesmo quando não conhecemos tais cotas, elas podem ser conjecturadas e essa prova continua sendo útil
- nesse caso, podemos interpretá-la como "acredita-se que Q tem cota inferior $\Omega(T_Q(n))$, logo, P deve ter cota inferior $\Omega(T_P(n))$"

Isso nos dá uma prova do tipo "se o problema Q tem cota inferior $\Omega(T_Q(n))$, então o problema P tem cota inferior $\Omega(T_P(n))$"

- quando já conhecemos cotas inferiores para Q, obtemos cotas inferiores para P
- mas mesmo quando não conhecemos tais cotas, elas podem ser conjecturadas e essa prova continua sendo útil
- nesse caso, podemos interpretá-la como "acredita-se que Q tem cota inferior $\Omega(T_Q(n))$, logo, P deve ter cota inferior $\Omega(T_P(n))$"

Outra situação útil para esse tipo de prova:

- podemos pensar que instâncias com alguma estrutura particular podem ser mais fáceis, mas elas acabam não sendo
- por exemplo, calcular o quadrado é o mesmo que multiplicar dois inteiros iguais
- temos o problema geral Q e definimos o problema "restrito" P
- mostramos que P tem a mesma cota inferior que Q

Outra situação útil para esse tipo de prova:

- podemos pensar que instâncias com alguma estrutura particular podem ser mais fáceis, mas elas acabam não sendo
- por exemplo, calcular o quadrado é o mesmo que multiplicar dois inteiros iguais
- temos o problema geral Q e definimos o problema "restrito" P
- mostramos que P tem a mesma cota inferior que Q

Outra situação útil para esse tipo de prova:

- podemos pensar que instâncias com alguma estrutura particular podem ser mais fáceis, mas elas acabam não sendo
- por exemplo, calcular o quadrado é o mesmo que multiplicar dois inteiros iguais
- temos o problema geral Q e definimos o problema "restrito" P
- mostramos que P tem a mesma cota inferior que Q

Outra situação útil para esse tipo de prova:

- podemos pensar que instâncias com alguma estrutura particular podem ser mais fáceis, mas elas acabam não sendo
- por exemplo, calcular o quadrado é o mesmo que multiplicar dois inteiros iguais
- temos o problema geral Q e definimos o problema "restrito" P
- mostramos que P tem a mesma cota inferior que Q

Exemplo

MMV

Dados um vetor $x \in \mathbb{Z}^n$ e uma matriz $A \in \mathbb{Z}^{n \times n}$, calcular o vetor $y = A \cdot x \in \mathbb{Z}^n$.

Mostre que se $\Omega(n^{2+\epsilon})$ é uma cota inferior para o problema de multiplicação de matrizes inteiras $n \times n$, então $\Omega(n^{1+\epsilon})$ é uma cota inferior para MMV.

Exemplo

Inversão de matrizes (IM)

Dada uma matriz não singular $A \in \mathbb{R}^{n \times n}$, calcular $A^{-1} \in \mathbb{R}^{n \times n}$.

Resolução de sistemas lineares (SL)

Dados uma matriz $A \in \mathbb{R}^{n \times n}$ e um vetor $y \in \mathbb{R}^n$, calcular $x \in \mathbb{R}^n$ tal que $A \cdot x = y$.

Mostre que se $\Omega(n^{2+\epsilon})$ é uma cota inferior SL, então $\Omega(n^{2+\epsilon})$ é uma cota inferior para IM.

(Por contrapositiva: assuma um algoritmo $\mathcal{A}_I$ que calcula A^{-1} em $o(n^{2+\epsilon})$ e compute $x = A^{-1} \cdot y$. Calculou-se x em tempo $o(n^{2+\epsilon}) + O(n^2) = o(n^{2+\epsilon})$, então, $\Omega(n^{2+\epsilon})$ não é uma cota inferior de SL)

Exemplo

Inversão de matrizes (IM)

Dada uma matriz não singular $A \in \mathbb{R}^{n \times n}$, calcular $A^{-1} \in \mathbb{R}^{n \times n}$.

Resolução de sistemas lineares (SL)

Dados uma matriz $A \in \mathbb{R}^{n \times n}$ e um vetor $y \in \mathbb{R}^n$, calcular $x \in \mathbb{R}^n$ tal que $A \cdot x = y$.

Mostre que se $\Omega(n^{2+\epsilon})$ é uma cota inferior SL, então $\Omega(n^{2+\epsilon})$ é uma cota inferior para IM.

(Por contrapositiva: assuma um algoritmo $\mathcal{A}_I$ que calcula A^{-1} em $o(n^{2+\epsilon})$ e compute $x = A^{-1} \cdot y$. Calculou-se x em tempo $o(n^{2+\epsilon}) + O(n^2) = o(n^{2+\epsilon})$, então, $\Omega(n^{2+\epsilon})$ não é uma cota inferior de SL)

Exemplo

Multiplicação de matrizes (MM)

Dadas matrizes $A, B \in \mathbb{R}^{n \times n}$, calcular $C = A \cdot B \in \mathbb{R}^{n \times n}$.

Multiplicação de matrizes simétricas (MMS)

Dadas matrizes simétricas $A, B \in \mathbb{R}^{n \times n}$, calcular $C = A \cdot B \in \mathbb{R}^{n \times n}$.

Mostre que se $\Omega(n^{2+\epsilon})$ é uma cota inferior MM, então $\Omega(n^{2+\epsilon})$ é uma cota inferior para MMS.

$$\begin{pmatrix} 0 & A \\ A^T & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & B^T \\ B & 0 \end{pmatrix} = \begin{pmatrix} AB & 0 \\ 0 & A^T B^T \end{pmatrix}$$

Exemplo

Multiplicação de matrizes (MM)

Dadas matrizes $A, B \in \mathbb{R}^{n \times n}$, calcular $C = A \cdot B \in \mathbb{R}^{n \times n}$.

Multiplicação de matrizes simétricas (MMS)

Dadas matrizes simétricas $A, B \in \mathbb{R}^{n \times n}$, calcular $C = A \cdot B \in \mathbb{R}^{n \times n}$.

Mostre que se $\Omega(n^{2+\epsilon})$ é uma cota inferior MM, então $\Omega(n^{2+\epsilon})$ é uma cota inferior para MMS.

$$\begin{pmatrix} 0 & A \\ A^T & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & B^T \\ B & 0 \end{pmatrix} = \begin{pmatrix} AB & 0 \\ 0 & A^T B^T \end{pmatrix}$$

Um exemplo com polinômis

Vamos denotar por $\mathbb{C}[X]$ o conjunto de polinômios com coeficientes complexos. Por exemplo,

- $(2+i)\cdot X + 3,5\cdot X^6 + i\cdot X^7 \in \mathbb{C}[X]$
- $2\cdot X + 3\cdot X^5 \in \mathbb{C}[X]$
- $1 \in \mathbb{C}[X]$
- $(X-3)\cdot(X^2+(3+2i)\cdot X+1) \in \mathbb{C}[X]$

Uma raíz de um polinômio $p(X) \in \mathbb{C}[X]$ é qualquer valor $r \in \mathbb{C}$ tal que $p(r) = 0$. Por exemplo,

- $r = i$ é raiz de $p(X) = X^2 - 2i\cdot X - 1$
- $1, 2$ e 5 são raízes de $X^3 + 2\cdot X^2 - 13\cdot X + 10$

Um exemplo com polinômis

Vamos denotar por $\mathbb{C}[X]$ o conjunto de polinômios com coeficientes complexos. Por exemplo,

- $(2+i)\cdot X + 3{,}5\cdot X^6 + i\cdot X^7 \in \mathbb{C}[X]$
- $2\cdot X + 3\cdot X^5 \in \mathbb{C}[X]$
- $1 \in \mathbb{C}[X]$
- $(X-3)\cdot(X^2+(3+2i)\cdot X+1) \in \mathbb{C}[X]$

Uma raíz de um polinômio $p(X) \in \mathbb{C}[X]$ é qualquer valor $r \in \mathbb{C}$ tal que $p(r) = 0$. Por exemplo,

- $r = i$ é raiz de $p(X) = X^2 - 2i\cdot X - 1$
- $1, 2$ e 5 são raízes de $X^3 + 2\cdot X^2 - 13\cdot X + 10$

Um exemplo com polinômis

Vamos denotar por $\mathbb{C}[X]$ o conjunto de polinômios com coeficientes complexos. Por exemplo,

- $(2+i)\cdot X + 3{,}5 \cdot X^6 + i \cdot X^7 \in \mathbb{C}[X]$
- $2 \cdot X + 3 \cdot X^5 \in \mathbb{C}[X]$
- $1 \in \mathbb{C}[X]$
- $(X-3)\cdot(X^2+(3+2i)\cdot X+1) \in \mathbb{C}[X]$

Uma raíz de um polinômio $p(X) \in \mathbb{C}[X]$ é qualquer valor $r \in \mathbb{C}$ tal que $p(r) = 0$. Por exemplo,

- $r = i$ é raiz de $p(X) = X^2 - 2i \cdot X - 1$
- $1, 2$ e 5 são raízes de $X^3 + 2 \cdot X^2 - 13 \cdot X + 10$

Um exemplo com polinômis

Vamos denotar por $\mathbb{C}[X]$ o conjunto de polinômios com coeficientes complexos. Por exemplo,

- $(2+i)\cdot X + 3,5\cdot X^6 + i\cdot X^7 \in \mathbb{C}[X]$
- $2\cdot X + 3\cdot X^5 \in \mathbb{C}[X]$
- $1 \in \mathbb{C}[X]$
- $(X-3)\cdot(X^2+(3+2i)\cdot X+1) \in \mathbb{C}[X]$

Uma raíz de um polinômio $p(X) \in \mathbb{C}[X]$ é qualquer valor $r \in \mathbb{C}$ tal que $p(r) = 0$. Por exemplo,

- $r = i$ é raiz de $p(X) = X^2 - 2i\cdot X - 1$
- $1, 2$ e 5 são raízes de $X^3 + 2\cdot X^2 - 13\cdot X + 10$

Um exemplo com polinômis

Vamos denotar por $\mathbb{C}[X]$ o conjunto de polinômios com coeficientes complexos. Por exemplo,

- $(2+i)\cdot X + 3{,}5 \cdot X^6 + i \cdot X^7 \in \mathbb{C}[X]$
- $2 \cdot X + 3 \cdot X^5 \in \mathbb{C}[X]$
- $1 \in \mathbb{C}[X]$
- $(X-3)\cdot(X^2+(3+2i)\cdot X+1) \in \mathbb{C}[X]$

Uma raíz de um polinômio $p(X) \in \mathbb{C}[X]$ é qualquer valor $r \in \mathbb{C}$ tal que $p(r) = 0$. Por exemplo,

- $r = i$ é raiz de $p(X) = X^2 - 2i \cdot X - 1$
- $1, 2$ e 5 são raízes de $X^3 + 2 \cdot X^2 - 13 \cdot X + 10$

Um exemplo com polinômis

Vamos denotar por $\mathbb{C}[X]$ o conjunto de polinômios com coeficientes complexos. Por exemplo,

- $(2+i)\cdot X + 3,5\cdot X^6 + i\cdot X^7 \in \mathbb{C}[X]$
- $2\cdot X + 3\cdot X^5 \in \mathbb{C}[X]$
- $1 \in \mathbb{C}[X]$
- $(X-3)\cdot(X^2+(3+2i)\cdot X+1) \in \mathbb{C}[X]$

Uma raíz de um polinômio $p(X) \in \mathbb{C}[X]$ é qualquer valor $r \in \mathbb{C}$ tal que $p(r) = 0$. Por exemplo,

- $r = i$ é raiz de $p(X) = X^2 - 2i\cdot X - 1$
- $1, 2$ e 5 são raízes de $X^3 + 2\cdot X^2 - 13\cdot X + 10$

Note que assim como avaliamos um polinômio $p(X)$ num valor qualquer, por exemplo, $p(3)$ ou $p(1)$, podemos avaliá-lo em X^2.

$$a(X) = \sum_{i=0}^{n} a_i X^i \Rightarrow a(X^2) = \sum_{i=0}^{2n} a_i X^{2i} = a_0 + a_1 X^2 + a_2 X^4 + \ldots + a_n X^{2n}$$

Por exemplo

- $a(X) = 2X + 5X^3 \Rightarrow a(X^2) = 2X^2 + 5X^6$
- $a(X) = 9 - 3X^2 + iX^5 \Rightarrow a(X^2) = 9 - 3X^4 + iX^{10}$

Vamos chamar tais polinômios de *barril*.

Definição

Um polinômio $b(X)$ de grau $2n$ é barril se existir um polinômio $a(X) = \sum_{i=0}^{n} a_i X^i$ tal que $b(X) = a(X^2)$.

Note que assim como avaliamos um polinômio $p(X)$ num valor qualquer, por exemplo, $p(3)$ ou $p(1)$, podemos avaliá-lo em X^2.

$$a(X) = \sum_{i=0}^{n} a_i X^i \Rightarrow a(X^2) = \sum_{i=0}^{2n} a_i X^{2i} = a_0 + a_1 X^2 + a_2 X^4 + \ldots + a_n X^{2n}$$

Por exemplo

- $a(X) = 2X + 5X^3 \Rightarrow a(X^2) = 2X^2 + 5X^6$
- $a(X) = 9 - 3X^2 + iX^5 \Rightarrow a(X^2) = 9 - 3X^4 + iX^{10}$

Vamos chamar tais polinômios de *barril*.

Definição

Um polinômio $b(X)$ de grau $2n$ é barril se existir um polinômio $a(X) = \sum_{i=0}^{n} a_i X^i$ tal que $b(X) = a(X^2)$.

Note que assim como avaliamos um polinômio $p(X)$ num valor qualquer, por exemplo, $p(3)$ ou $p(1)$, podemos avaliá-lo em X^2.

$$a(X) = \sum_{i=0}^{n} a_i X^i \Rightarrow a(X^2) = \sum_{i=0}^{2n} a_i X^{2i} = a_0 + a_1 X^2 + a_2 X^4 + \ldots + a_n X^{2n}$$

Por exemplo

- $a(X) = 2X + 5X^3 \Rightarrow a(X^2) = 2X^2 + 5X^6$
- $a(X) = 9 - 3X^2 + iX^5 \Rightarrow a(X^2) = 9 - 3X^4 + iX^{10}$

Vamos chamar tais polinômios de *barril*.

Definição

Um polinômio $b(X)$ de grau $2n$ é barril se existir um polinômio $a(X) = \sum_{i=0}^{n} a_i X^i$ tal que $b(X) = a(X^2)$.

Note que assim como avaliamos um polinômio $p(X)$ num valor qualquer, por exemplo, $p(3)$ ou $p(1)$, podemos avaliá-lo em X^2.

$$a(X) = \sum_{i=0}^{n} a_i X^i \Rightarrow a(X^2) = \sum_{i=0}^{2n} a_i X^{2i} = a_0 + a_1 X^2 + a_2 X^4 + \ldots + a_n X^{2n}$$

Por exemplo

- $a(X) = 2X + 5X^3 \Rightarrow a(X^2) = 2X^2 + 5X^6$
- $a(X) = 9 - 3X^2 + iX^5 \Rightarrow a(X^2) = 9 - 3X^4 + iX^{10}$

Vamos chamar tais polinômios de *barril*.

Definição

Um polinômio $b(X)$ de grau $2n$ é barril se existir um polinômio $a(X) = \sum_{i=0}^{n} a_i X^i$ tal que $b(X) = a(X^2)$.

Raiz de um polinômio qualquer (RQ)

- **Entrada:** $a(X) \in \mathbb{C}[X]$ de grau n.
- **Saída:** qualquer raiz $r \in \mathbb{C}$ de $a(X)$.

Raiz de polinômio barril (RB)

- **Entrada:** $b(X) \in \mathbb{C}[X]$ tal que $b(X)$ é barril.
- **Saída:** qualquer raiz $r \in \mathbb{C}$ de $b(X)$.

Considerando a complexidade temporal com respeito ao número de operações aritméticas em $\mathbb{C}$, vamos mostrar que se, para qualquer $\epsilon > 0$, RQ tem cota inferior $\Omega(n^{1+\epsilon})$, então RB também tem cota inferior $\Omega(n^{1+\epsilon})$.