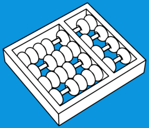
Cota inferior para ordenação. Ordenação linear

MO417 - Complexidade de Algoritmos I

11

04/24



Santiago Valdés Ravelo
https://ic.unicamp.br/~santiago/
ravelo@unicamp.br

“O conhecimento de valores extremos transforma a incerteza ilimitada em um domínio viável.”

Anônimo.

Cotas e modelo de árvore de decisão

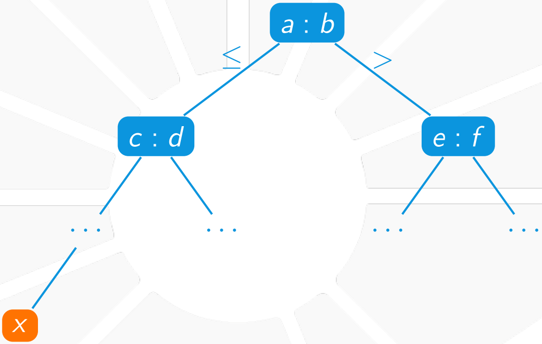
Cotas de um problema

Dados um problema P e um modelo computacional M (no qual seja possível resolver P):

- **Cota superior**. Se um algoritmo A que soluciona P em M com eficiência $T(n)$, então $T(n)$ é uma cota superior para P em M.
- **Cota inferior**. $I(n)$ é uma cota inferior de P em M, se ela expressa a eficiência mínima de qualquer algoritmo para resolver P em M.

Um modelo (abstrato)

para algoritmos baseados em comparações de elementos



Modelo de árvore (binária) de decisão

Árvore binária (cheia) e enraizada que representa as comparações entre elementos executadas por um determinado algoritmo aplicado a uma entrada de um dado tamanho:

- Cada vértice interno representa uma comparação entre dois elementos realizada pelo algoritmo.
- Cada subárvore de um nó, representa o caminho a seguir conforme o resultado (verdadeiro ou falso) da comparação.
- Cada folha representa uma possível resposta do algoritmo, obtida pelas decisões tomadas da raiz até ela.

Modelo de árvore (binária) de decisão

A eficiência de um algoritmo A no modelo de árvore de decisão está dada pela altura da árvore que o representa.

Observação. Neste modelo só são consideradas operações de comparação entre elementos, os outros aspectos do algoritmo são ignorados.

Altura de árvore binária

Teorema

Se uma árvore binária tem $\mathbf{n}$ *folhas então sua altura é no mínimo* $\log \mathbf{n}$.

Prova:

- Primeiro, provamos que se uma árvore binária tem altura $\mathbf{h}$, então ela tem no máximo $\mathbf{2^h}$ folhas.
- Esse resultado é provado por indução na altura $\mathbf{h}$.
- Se o resultado é válido, então se o número de folhas for $\mathbf{n}$ e a altura for $\mathbf{h}$, temos:

$$n \leq 2^h \quad \Longrightarrow \quad \log n \leq h.$$

Concluindo a prova do teorema.

Altura de árvore binária

Provemos que se uma árvore binária tem altura $\mathbf{h}$, então ela tem no máximo $\mathbf{2^h}$ folhas.

- **Caso básico**. $\mathbf{h} = 0$. A árvore tem só uma folha, onde $1 \leq 2^0 = 2^h$.
- **Passo**. $\mathbf{h} > 0$.
  - A árvore tem pelo menos dois vértices e a raiz tem pelo menos um filho.
  - As folhas da árvore são a soma das folhas da subárvore esquerda da raiz mais as da subárvore direita.
  - Cada uma dessas subárvores tem no máximo altura $\mathbf{h-1}$ e, por h.i., cada uma tem no máximo 2^{h-1} folhas.
  - Portanto, a árvore tem no máximo $2^{h-1} + 2^{h-1} = 2^h$ folhas.

Ordenação de um vetor

Definição do problema

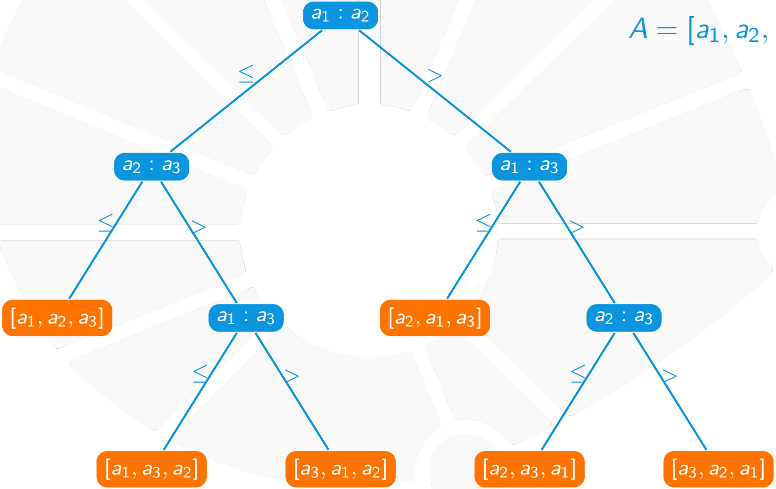
Problema

Entrada: *Um vetor* $A[1..n]$ *de* n *números.*

Saída *Uma permutação (reordenação)* $A'[1..n]$ *de* $A[1..n]$*, tal que* $A'[1], A'[2], \ldots, A'[n]$ *é uma sequência ordenada.*

Exemplo
Árvore de decisão do Insertion-Sort para um vetor de 3 elementos

$A = [a_1, a_2, a_3]$



Cota inferior

Seja A um algoritmo qualquer que resolve o problema de ordenação via comparações, então:

- A pode ser representado por uma árvore de decisão T.
- Por cada possível resposta deve existir pelo menos uma folha.
- Para um vetor de tamanho n, existem $n!$ possíveis respostas.

$\Rightarrow$ A altura de T é pelo menos $\Omega(\log(n!))$.

Cota inferior

É possível provar que $\log(n!) = \Omega(n \log n)$.

$\Rightarrow$ A altura de T é pelo menos $\Omega(n \log n)$.

$\Rightarrow$ A no pior caso realiza pelo menos $\Omega(n \log n)$ comparações.

Cota inferior

Provemos que $\log(n!) = \Omega(n \log n)$:

$$
\begin{aligned}
\log(n!) &= \log\left(\prod_{i=1}^{n} i\right) \geq \log\left(\prod_{i=\frac{n}{2}}^{n} i\right) \geq \log\left(\prod_{i=\frac{n}{2}}^{n} \frac{n}{2}\right) = \log\left(\frac{n}{2}^{\frac{n}{2}}\right) \\
&\geq \frac{n}{2}\log\left(\frac{n}{2}\right) = \frac{n/2}{\log} n - \frac{n}{2}\log 2 = \frac{n/2}{\log} n - \frac{n}{2} \\
&\geq \frac{n}{2}\log n - \frac{n}{4}\log n \\
&\geq \frac{n}{4}\log n.
\end{aligned}
$$

- A penúltima desigualdade se obtém porque $\frac{n}{2} \leq \frac{n}{4}\log n$ para $n \geq 4$.
- Logo, encontramos $c = \frac{1}{4}$ e $n_0 = 4$ tais que: $\log(n!) \geq cn\log n,\ \forall n \geq n_0$.

Cota inferior

Teorema

Qualquer algoritmo que ordene um vetor via comparações, no pior caso realiza $\Omega(n \log n)$ *comparações.*

Busca binária

Definição do problema

Problema

Entrada: *Um vetor de números ordenado A com n elementos e um número x.*

Saída: *Algum índice i tal que $A[i] = x$ se x estiver no vetor, ou* NIL *se x não estiver no vetor.*

Ideia de solução

Comparamos x com $A[\frac{n}{2}]$

$A[1]$	$A[2]$	$\cdots$	$\cdots$	$\cdots$	$A[\frac{n}{2}]$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$A[n]$
1	2			$\frac{n}{2}-1$	$\frac{n}{2}$	$\frac{n}{2}+1$				n

Ideia de solução

Se $x < A[\frac{n}{2}]$ então só precisamos buscar na primeira metade

$A[1]$	$A[2]$	$\cdots$	$\cdots$	$\cdots$	$A[\frac{n}{2}]$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$A[n]$
1	2			$\frac{n}{2}-1$	$\frac{n}{2}$	$\frac{n}{2}+1$				n

Ideia de solução

Se $x > A[\frac{n}{2}]$ então só precisamos buscar na segunda metade

$A[1]$	$A[2]$	$\cdots$	$\cdots$	$\cdots$	$A[\frac{n}{2}]$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$A[n]$
1	2			$\frac{n}{2}-1$	$\frac{n}{2}$	$\frac{n}{2}+1$				n

Ideia de solução

Se $x = A[\frac{n}{2}]$ encontramos o elemento!!

$A[1]$	$A[2]$	$\cdots$	$\cdots$	$\cdots$	$A[\frac{n}{2}]$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$A[n]$
1	2			$\frac{n}{2}-1$	$\frac{n}{2}$	$\frac{n}{2}+1$				n

Algoritmo de busca binária

Algoritmo: $\textsc{Busca-Binaria}(A, e, d, x)$

se $e > d$
  $i \leftarrow \text{NIL}$
senão
  $i \leftarrow \lfloor \frac{e+d}{2} \rfloor$
  se $A[i] > x$
    $i \leftarrow \textsc{Busca-Binaria}(A, e, i-1, x)$
  se $A[i] < x$
    $i \leftarrow \textsc{Busca-Binaria}(A, i+1, d, x)$
devolva i

Algoritmo de busca binária

O algoritmo de busca binária realiza até $O(\log n)$ comparações.

Seria possível projetar um algoritmo baseado em comparações mais eficiente ($o(\log n)$)?

Cota inferior

Seja A um algoritmo qualquer que resolve o problema de busca num vetor ordenado via comparações, então:

- A pode ser representado por uma árvore de decisão T.
- Como visto anteriormente, por cada possível resposta deve existir pelo menos uma folha.
- Para um vetor de tamanho n, existem $n+1$ possíveis respostas.

$\Rightarrow$ A altura de T é pelo menos $\Omega(\log n)$.

$\Rightarrow$ A exige $\Omega(\log n)$ comparações no pior caso.

Considerações finais sobre cotas

Conclusão

Considerando o modelo de árvore de decisão:

- Os algoritmos Heap-Sort e Merge-Sort são assintoticamente ótimos para o problema de ordenar um vetor.
- A busca binária é assintoticamente ótima para o problema de procurar um elemento em um vetor ordenado.

Observação: Os resultados obtidos só valem para algoritmos por comparações!!

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.

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

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

para $i \leftarrow 0$ **até** k
  $C[i] \leftarrow 0$
para $j \leftarrow 1$ **até** n
  $C[A[j]] \leftarrow C[A[j]] + 1$
para $i \leftarrow 1$ **até** k
  $C[i] \leftarrow C[i] + C[i-1]$
para $j \leftarrow n$ **até** 1
  $B[C[A[j]]] \leftarrow A[j]$
  $C[A[j]] \leftarrow C[A[j]] - 1$

Counting Sort - Exemplo

A

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

C

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

B

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

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, Quick-Sort, 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, Quick-Sort, Counting-Sort.
- O Heap-Sort não é estável.

Radix Sort

Ideia:

- A entrada é um vetor $A[1 \dots n]$ de inteiros.
- Cada inteiro é representado na **base** k-ária
- Supomos que um inteiro tem 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

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

1 **para** $i \leftarrow 1$ **até** d

2 Ordene $A[1 \dots n]$ pelo i-ésimo dígito usando um método estável

Radix Sort - Exemplo

329		720		720		329
457		355		329		355
657		436		436		436
839		457		839		457
436	$\rightarrow$	657	$\rightarrow$	355	$\rightarrow$	657
720		329		457		720
355		839		657		839
		$\uparrow$		$\uparrow$		$\uparrow$

Radix Sort - Correção

Demonstramos a correção do algoritmo por indução:

- Suponha que o vetor de números esteja ordenado em relação aos $i-1$ dígitos menos significativos.
- Ordene o vetor pelo i-ésimo dígito com um método estável.
- Considere números a e b no vetor resultante com $a < b$:
  1. Se os i-ésimos dígitos forem distintos, então eles estão em ordem em relação a este dígito, independentemente dos $i-1$ primeiros dígitos. Portanto, a aparece antes de b.
  2. Se os i-ésimos dígitos forem iguais, então por hipótese de indução, eles estavam em ordem pelos $i-1$ primeiros dígitos antes da i-ésima iteração. Como usamos um algoritmo estável nesta iteração, a ordem é mantida de forma que a aparecia antes de b e continua assim.

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 $\Theta(n + k)$.
  - Lembre-se de que $k - 1$ é o maior valor do número na entrada do Counting-Sort (e.g., $k = 9$ na base decimal).
  - Se k é constante, 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 $k = 2^{16}$ (cada 16 bits corresponde a um dígito).
- Cada número tem $d = 4$ dígitos.

1. Ordenando com Merge-Sort:
   - Executamos cerca de $n \log_2 n = \mathbf{20 \times 2^{20}}$ **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 $d(n+k) = \mathbf{4(2^{20} + 2^{16})}$ **operações**.
   - Usamos dois vetores auxiliares de tamanhos 2^{16} e 2^{20}.

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

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.
- No final, concatenamos as listas já ordenadas.

Bucket Sort - Algoritmo

Algoritmo: $\textsc{Bucket-Sort}(A, n)$

para $i \leftarrow 0$ **até** $n - 1$
    crie uma lista ligada vazia $B[i]$
para $i \leftarrow 1$ **até** n
    insira $A[i]$ na lista ligada $B[\lfloor n\, A[i] \rfloor]$
para $i \leftarrow 0$ **até** $n - 1$
    ordene a lista $B[i]$ com $\textsc{Insertion-Sort}$
concatene as listas $B[0], B[1], \ldots, B[n-1]$

Bucket Sort - Exemplo

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

Bucket Sort - 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.
  - Se manterão ordenados após a concatenação.
- Se 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 $\Theta(n)$.

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

Bucket Sort - Pior caso

Pior caso

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

- 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:

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

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 $n_i = \sum_{j=1}^{n} X_{ij}$.

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

Bucket Sort - Tempo esperado (cont)

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

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

Bucket Sort - Tempo esperado (cont)

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

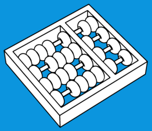
$$\begin{aligned} E[T(n)] &= \sum_{i=0}^{n-1} cE[n_i^2] + \Theta(n) \\ &= \sum_{i=0}^{n-1} cO(1) + \Theta(n) \\ &= \Theta(n). \end{aligned}$$

Cota inferior para ordenação. Ordenação linear

MO417 - Complexidade de Algoritmos I

11

04/24



Santiago Valdés Ravelo
https://ic.unicamp.br/~santiago/
ravelo@unicamp.br

UNICAMP