Limite Inferior para Ordenação por Comparação

Prof. Daniel Kikuti

Universidade Estadual de Maringá

Introdução

Limite inferior

Dado um problema X, qual a função $f(n)$ tal que todo algoritmo consome pelo menos tempo $\Omega(f(n))$ para entradas de tamanho n?

- Queremos identificar "a maior" função $f(n)$ possível.
- Limite inferior ajuda a entender a **dificuldade intrínseca** do problema e o quão próximo estamos da melhor solução possível para o problema.

Objetivo desta aula

Mostrar que qualquer algoritmo determinístico para ordenação baseada em comparações leva no pior caso tempo $\Omega(n \lg n)$ para ordenar um vetor com n elementos.

Modelo de computação para o problema

Definição de algoritmo de ordenação baseado em comparação

- Recebe como entrada um vetor de n números $\langle a_1, a_2, \ldots, a_n \rangle$.
- Só pode obter informações sobre os elementos através da comparação em pares.
- Cada comparação $(a_i < a_j)$ devolve **sim** ou **não** e consome um tempo constante (o custo para efetuar trocas entre posições de elementos não é considerado).
- No final, o algoritmo devolve como saída uma permutação da entrada tal que todos os números estão em ordem.
- Assumiremos que todos os elementos são *distintos*.

Exemplo

Quicksort, Mergesort, Insertion-sort, Heap-sort são algoritmos de ordenação baseados em comparação.

Limite inferior para algoritmos baseados em comparação

Teorema

Qualquer algoritmo de ordenação determinístico baseado em comparação deve efetuar $\Omega(n \lg n)$ comparações para ordenar n elementos no pior caso.

Jogo de adivinhação com dois jogadores

Eu vs. Você

- Seu objetivo é determinar a ordem correta de n elementos (que você não sabe *a priori*).
- Você pode fazer perguntas comparando pares de elementos.
- **Quantas questões você precisa fazer?**

Jogo de adivinhação com dois jogadores

Eu vs. Você

- Seu objetivo é determinar a ordem correta de n elementos (que você não sabe *a priori*).
- Você pode fazer perguntas comparando pares de elementos.
- **Quantas questões você precisa fazer?**

Definindo o espaço de resposta

Considere a entrada: $\langle a_1, a_2, a_3 \rangle$. Qual seria o espaço das soluções?

$$\langle a_1, a_2, a_3 \rangle \quad \langle a_2, a_1, a_3 \rangle \quad \langle a_3, a_1, a_2 \rangle$$
$$\langle a_1, a_3, a_2 \rangle \quad \langle a_2, a_3, a_1 \rangle \quad \langle a_3, a_2, a_1 \rangle$$

Jogo de adivinhação com dois jogadores

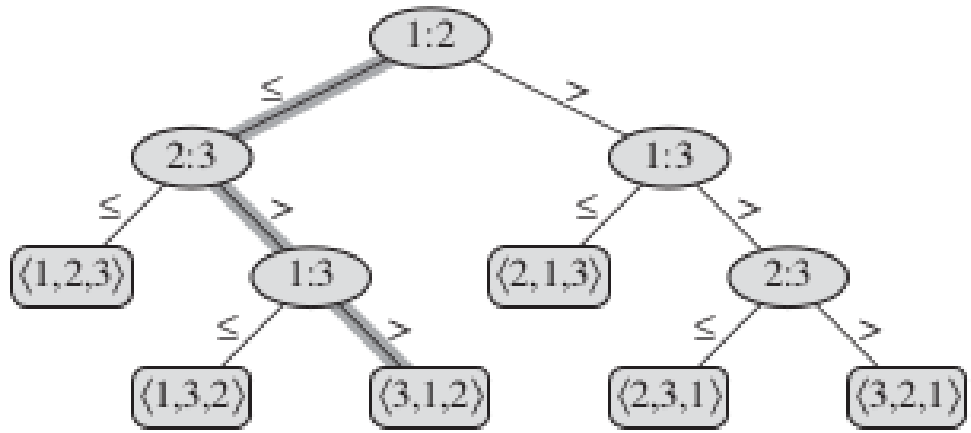
- Existem $n!$ permutações que podem ser a saída do algoritmo.
- Para cada uma destas permutações, existe uma entrada para a qual aquela permutação é a única resposta correta.
- Seja S o conjunto de permutações que são consistentes com as questões já feitas (inicialmente $|S| = n!$).
- O que ocorre quando você faz uma pergunta?
  - A comparação devolve sim ou não (divide o conjunto de permutações S em dois grupos).
  - Suponha que eu escolha a resposta que leve ao grupo com mais elementos (isto significa que você conseguirá eliminar no máximo metade dos elementos em S).
- Você irá parar quando $|S| = 1$ (você sabe qual é a permutação).
- Portanto, teremos no máximo $\lg(n!)$ comparações e, precisamos mostrar que $\lg(n!) = \Omega(n \lg n)$.

Outra maneira de visualizar o resultado

- Para um algoritmo determinístico, a permutação devolvida é somente uma função das séries de respostas recebidas (quaisquer duas entradas produzindo a mesma série de respostas gerará a mesma permutação na saída).
- Se um algoritmo sempre faz no máximo $k < \lg n!$ comparações, então existe no máximo $2^k < n!$ permutações possíveis de serem geradas (ou seja, existem algumas permutações que não podem ser produzidas).
- Portanto, o algoritmo falhará para qualquer entrada em que somente aquela permutação ausente seria a resposta correta.

Visualizando algoritmos como árvores de decisão

Árvore de decisão do Insertion sort



Árvore de decisão	**Algoritmo**
Nós internos	Decisões binárias (comparações)
Folhas	Saída do algoritmo
Caminho da raiz até um nó folha	Execução do algoritmo
Comprimento de um caminho	Tempo de execução
Altura da árvore	Tempo de execução no pior caso

Demonstração do limite inferior usando árvores de decisão

Teorema

Qualquer algoritmo de ordenação determinístico baseado em comparação deve efetuar $\Omega(n \lg n)$ comparações para ordenar n elementos no pior caso.

Demonstração

- Considere uma árvore de decisão com altura h com l folhas.
- Cada uma das $n!$ permutações da entrada aparecem como alguma folha, portanto $n! \leq l$.
- Como uma árvore binária de altura h não tem mais que 2^h folhas, temos $n! \leq l \leq 2^h$.
- Aplicando logaritmo, temos $h \geq \lg(n!)$.
- Como vimos, $\lg(n!) = \Omega(n \lg n)$.

Resultado deste teorema

Corolário

HEAPSORT e MERGESORT são algoritmos de ordenação assintoticamente ótimos.

Demonstração

O limite superior $O(n \lg n)$ dos algoritmos HEAPSORT e MERGESORT é o mesmo que o limite inferior $\Omega(n \lg n)$ no pior caso demonstrado.

Introdução à ordenação em tempo linear

Problema

Rearranjar um vetor $A[1..n]$ de modo que ele fique ordenado.

- Estudamos algoritmos que fazem isto em tempo $O(n \lg n)$.
- Existem algoritmos assintoticamente melhores?
- **NÃO** se o algoritmo for **baseado em comparações**.

SIM se o algoritmo usa outras informações sobre a entrada além das comparações

Neste caso o limite inferior não se aplica. Exemplos:

- Counting-Sort
- Radix-Sort
- Bucket-Sort

Counting-Sort

Sobre o algoritmo

- Baseado em contagem de elementos com mesma chave.
- Estável (se implementado corretamente).
- Entrada contém **inteiros no intervalo de 0 a k**.
- Quando $k = O(n)$, a ordenação é executada no tempo $\Theta(n)$

Exemplo de funcionamento

Considere os seguintes elementos de entrada: $\langle 2, 5, 3, 0, 2, 3, 0, 3 \rangle$.

Como garantir que o algoritmo seja estável?

- A entrada do algoritmo é um array $A[1..n]$.
- A saída é dada em um array $B[1..n]$.
- O array $C[0..k]$ é utilizado para contagem.

O que precisa ser feito?

- Determinar, para cada elemento x da entrada, o número de elementos menores que x.
- Inserir o elemento diretamente em sua posição no arranjo de saída.
- Começar pelo último elemento do array A até o primeiro.

O algoritmo Counting Sort

COUNTING-SORT(A, B, k)

```
1  n ← A.length
2  for i ← 0 to k do
3  |   C[i] ← 0
4  for j ← 1 to n do
5  |   C[A[j]] ← C[A[j]] + 1
6  for i ← 1 to k do
7  |   C[i] ← C[i] + C[i − 1]
8  for j ← n to 1 do
9  |   B[C[A[j]]] ← A[j]
10 |   C[A[j]] ← C[A[j]] − 1
```

Exemplo

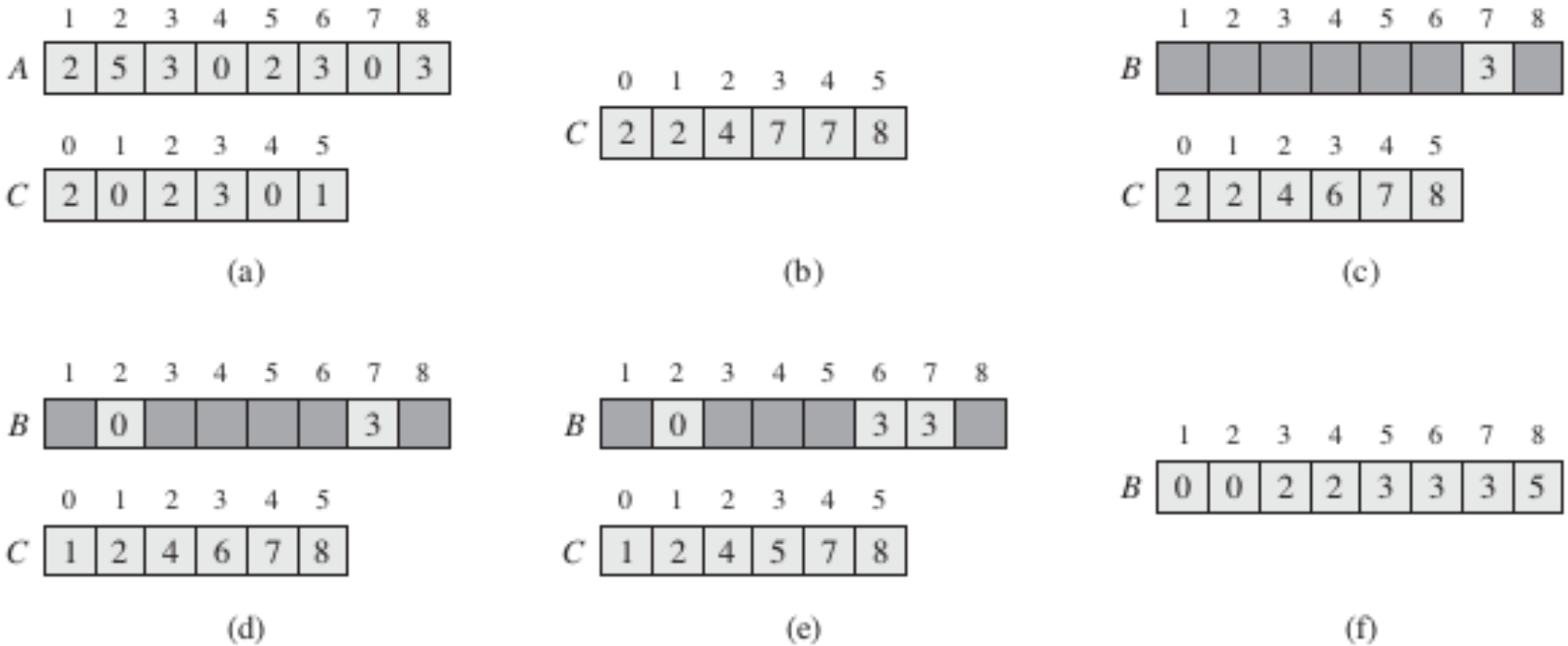


Figure 8.2 The operation of COUNTING-SORT on an input array $A[1 \ldots 8]$, where each element of A is a nonnegative integer no larger than $k = 5$. **(a)** The array A and the auxiliary array C after line 5. **(b)** The array C after line 8. **(c)–(e)** The output array B and the auxiliary array C after one, two, and three iterations of the loop in lines 10–12, respectively. Only the lightly shaded elements of array B have been filled in. **(f)** The final sorted output array B.

Análise de complexidade do Counting-Sort

Quanto tempo a ordenação por contagem exige?

- O loop das linhas 2 e 3 consome tempo $\Theta(k)$.
- O loop das linhas 4 e 5 consome tempo $\Theta(n)$.
- O loop das linhas 6 e 7 consome tempo $\Theta(k)$.
- O loop das linhas 8 e 10 consome tempo $\Theta(n)$.

Portanto, o tempo total é $\Theta(k+n)$. Quando $k = O(n)$, o tempo de execução é $\Theta(n)$.

Algum ponto negativo no algoritmo?

Análise de complexidade do COUNTING-SORT

Quanto tempo a ordenação por contagem exige?

- O loop das linhas 2 e 3 consome tempo $\Theta(k)$.
- O loop das linhas 4 e 5 consome tempo $\Theta(n)$.
- O loop das linhas 6 e 7 consome tempo $\Theta(k)$.
- O loop das linhas 8 e 10 consome tempo $\Theta(n)$.

Portanto, o tempo total é $\Theta(k+n)$. Quando $k = O(n)$, o tempo de execução é $\Theta(n)$.

Algum ponto negativo no algoritmo?

- Uso de memória para grandes intervalos.
- Somente inteiros.

Counting-Sort usando listas

Counting-Sort(A, k)

```
1 L ← array of k empty lists
2 for j in range n do
3 |   L[A[j]].append(A[j])
4 output ← []
5 for i in range k do
6 |   output.extend(L[i])
```

Análise de complexidade

Linhas	**Tempo**
1	$O(k)$
2 e 3	$O(n)$
4	$O(1)$
5 e 6	$O(k+n)$.

Radix-Sort

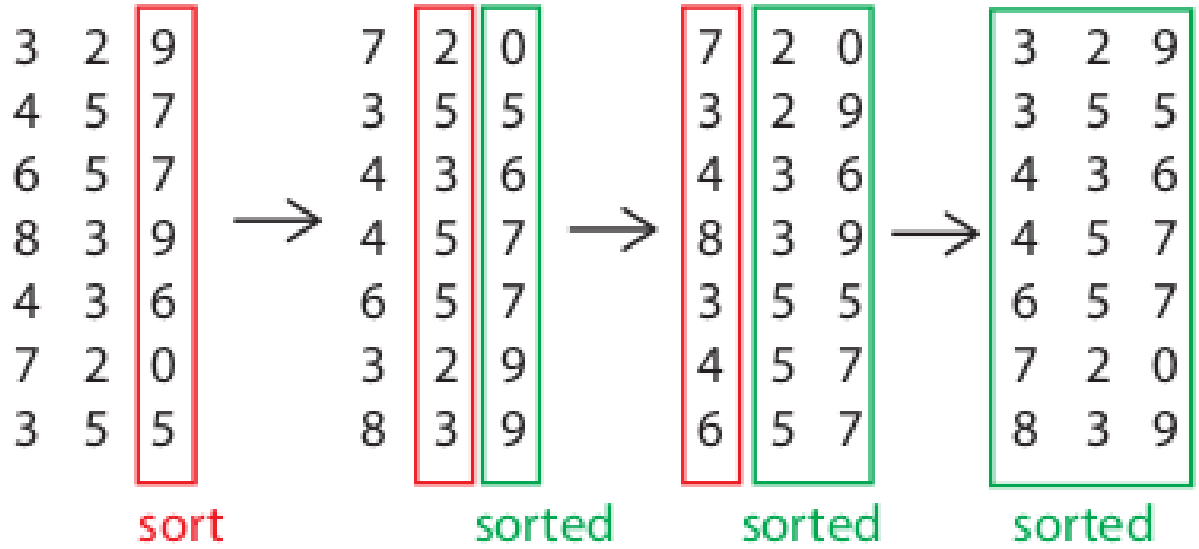
- Ordenar as chaves pelos dígitos, começando com o menos significativo.
- O algoritmo de ordenação de dígitos deve ser estável.
- É utilizado para ordenar registros cuja a chave é constituída de vários campos (exemplo, planilha do Excel).
- O algoritmo assume que os números possuem d dígitos, com k possíveis valores para os dígitos.

O algoritmo

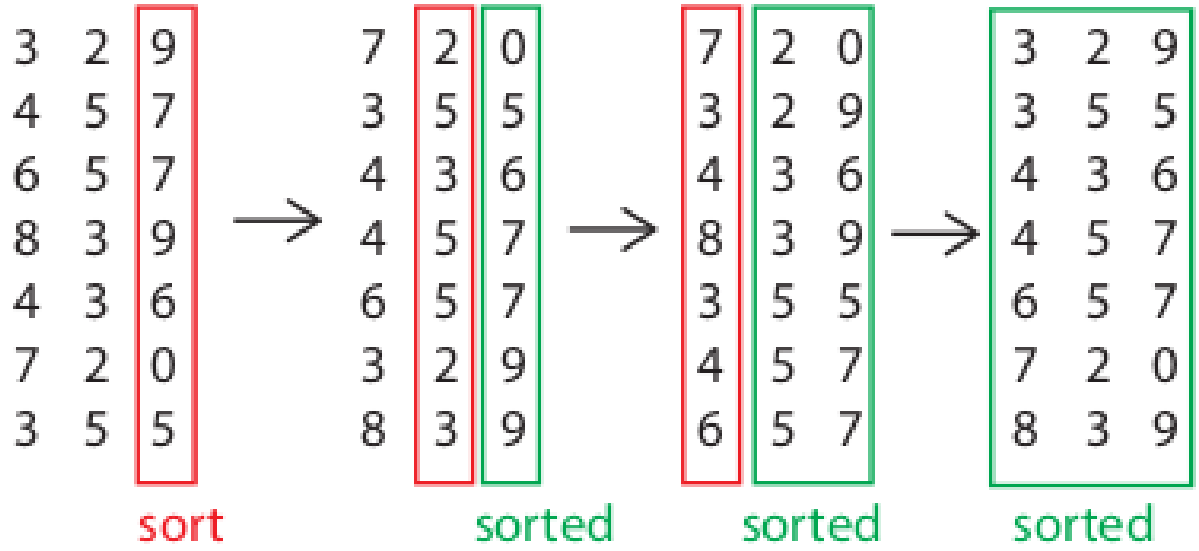
Radix-Sort(A, d)

1 **for** $i \leftarrow 1$ **to** d **do**
2 | Usar Counting-Sort para ordenar o arranjo A considerando o dígito i

Exemplo e complexidade



Exemplo e complexidade



Complexidade?

- Ordenar n números de um dígito, podendo assumir k valores distintos: $\Theta(n + k)$ (usando Counting-Sort).
- Cada dígito possui no máximo d dígitos, portanto $\Theta(d(n + k))$.
- Se $k = O(n)$ e d é constante, então $\Theta(n)$.

Bucket-Sort

- Assume que os n números da entrada estão **uniformemente distribuídos** no intervalo $[0, 1)$.
- A ideia é dividir o intervalo $[0, 1)$ em n segmentos de mesmo tamanho (**buckets**) e distribuir os n elementos nos seus respectivos segmentos.
- Como os elementos estão distribuídos uniformemente, espera-se que o número de elementos seja aproximadamente o mesmo em todos os segmentos.
- Ordenamos os elementos nos *buckets* usando um algoritmo de ordenação qualquer e percorremos os *buckets* listando-os.

Exemplo do bucket

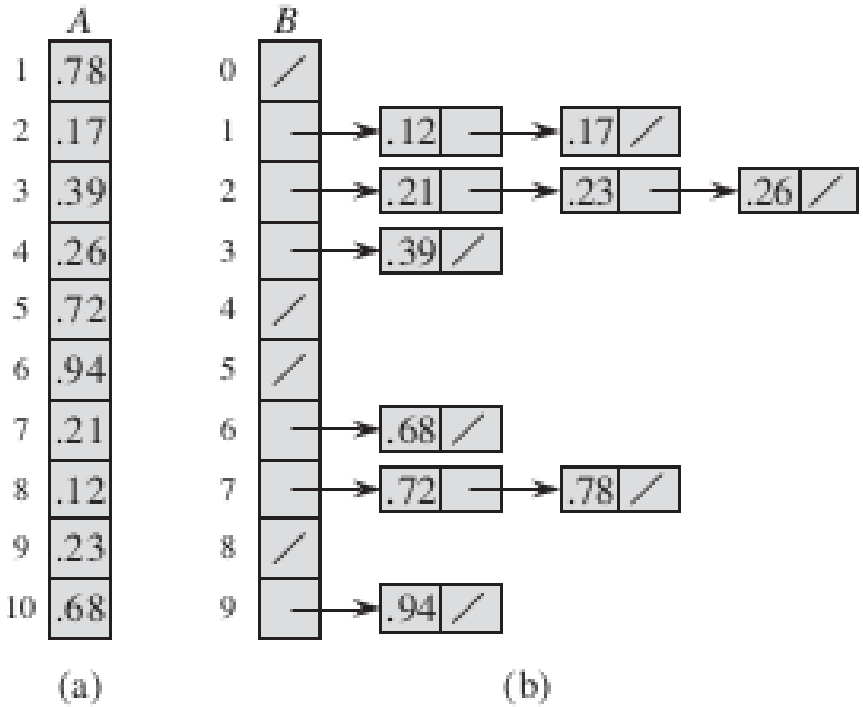


Figure 8.4 The operation of BUCKET-SORT for $n = 10$. (**a**) The input array $A[1..10]$. (**b**) The array $B[0..9]$ of sorted lists (buckets) after line 8 of the algorithm. Bucket i holds values in the half-open interval $[i/10, (i+1)/10)$. The sorted output consists of a concatenation in order of the lists $B[0], B[1], \ldots, B[9]$.

Bucket-Sort

- Entrada: $A[1..n]$, onde $0 \leq A[i] < 1$ para todo i.
- Array de listas ligadas $B[0..n-1]$ inicialmente vazias.

Bucket-Sort(A)

1 $n \leftarrow A.length$
2 **for** $i \leftarrow 1$ **to** n **do**
3 $\quad$ insert $A[i]$ into $B[\lfloor n * A[i] \rfloor]$
4 **for** $i \leftarrow 0$ **to** $n-1$ **do**
5 $\quad$ sort list $B[i]$ with Insertion sort
6 concatenate the lists $B[0], B[1], \ldots, B[n-1]$ together in order

Análise do Bucket

Correção

Dois elementos x e y de A, $x < y$, ou terminam na mesma lista ou são colocados em listas diferentes.

- Mesma lista: x aparecera antes de y na concatenação final, visto que cada lista é ordenada.
- Listas diferentes: como $x < y \implies \lfloor n * x \rfloor < \lfloor n * y \rfloor$, assim, a lista de x será processada antes da lista de y na concatenação da lista final, portanto x aparecerá antes de y.

Análise do Bucket

Correção

Dois elementos x e y de A, $x < y$, ou terminam na mesma lista ou são colocados em listas diferentes.

- Mesma lista: x aparecera antes de y na concatenação final, visto que cada lista é ordenada.
- Listas diferentes: como $x < y \implies \lfloor n * x \rfloor < \lfloor n * y \rfloor$, assim, a lista de x será processada antes da lista de y na concatenação da lista final, portanto x aparecerá antes de y.

Complexidade

- Pior caso: $O(n^2)$ (usando o INSERTION SORT)
- Tempo esperado: $\Theta(n)$ (ver CLRS).

Referências

- Cormen, et al.. Introduction to algorithms. Chapter 8 (Sorting in linear time).
- Notas de aula do prof. Erick Domaine: `http://ocw.mit.edu/courses/.../MIT6_006F11_lec07.pdf`