Algoritmos de Ordenação Parte 2

Algoritmos e Programação 2

Prof. Dr. Anderson Bessa da Costa
Universidade Federal de Mato Grosso do Sul

Ordenação Rápida (*Quicksort*)

Quicksort

- Um dos mais eficientes
- **Ideia:** Dada tabela $\mathscr{L}$ com n elementos, procedimento recursivo para ordenar $\mathscr{L}$:
  - se $n = 0$ ou $n = 1$ então a tabela está ordenada
  - escolha elemento x em $\mathscr{L}$ - este elemento é chamado **pivô**
  - separe $\mathscr{L} - \{x\}$ em dois conjuntos de elementos disjuntos: $S_1 = \{w \in \mathscr{L} - \{x\} \mid w < x\}$ e $S_2 = \{w \in \mathscr{L} - \{x\} \mid w \geq x\}$; o procedimento de ordenação é chamado recursivamente para S_1 e S_2
  - $\mathscr{L}$ recebe a concatenação de S_1, seguido de x, seguido de S_2

Quicksort: Pontos importantes

- Dois pontos decisivos para bom desempenho:
  1. Escolha do pivô;
  2. Particionamento da tabela.

Quicksort: Escolha do pivô

- Estratégia comum é selecionar primeiro elemento como **pivô**
    - Se tabela ordenada na ordem inversa à desejada, provoca o pior desempenho

0	1	2	3	4	5	6
95	**60**	**51**	**42**	**40**	**39**	**37**

primeiro elemento

Quicksort: Escolha do pivô 2

- Estratégia alternativa é a escolha aleatória do **pivô**
  - Mas a geração de números aleatórios poderia pesar no tempo de execução

0	1	2	3	4	5	6
40	37	95	42	39	51	60

aleatório

Quicksort: Escolha do pivô 2 (cont.)

- Estratégia alternativa é a escolha aleatória do **pivô**
  - Mas a geração de números aleatórios poderia pesar no tempo de execução

0	1	2	3	4	5	6
40	37	95	42	39	51	60

aleatório

Quicksort: Escolha do pivô 2 (cont.)

- Estratégia alternativa é a escolha aleatória do **pivô**
  - Mas a geração de números aleatórios poderia pesar no tempo de execução

0	1	2	3	4	5	6
40	37	95	42	39	51	60

aleatório

Quicksort: Escolha do pivô 3

- Idealmente escolher **pivô** tal que a partição da tabela em dois subconjuntos de dimensão equivalente
  - A mediana da tabela deveria ser selecionada .. Infelizmente, o cálculo custoso

0	1	2	3	4	5	6
40	37	95	42	39	51	60

mediana

Quicksort: Escolha do pivô 4

- Uma solução utilizada com bons resultados é a escolha da mediana dentre três elementos: o primeiro, o último e o central

0	1	2	3	4	5	6
40	37	95	42	39	51	60

mediana

Quicksort: Particionamento

- **Pivô** é afastado da tabela a ser percorrida
  - Coloque-o na última posição e considere somente o restante da tabela
- Em seguida, dois ponteiros são utilizados:
  - i apontando para primeiro elemento
    - Percorra enquanto valores apontados são menores do que o **pivô**
  - j inicializado na penúltima posição
    - Percorra enquanto valores apontados são maiores do que o **pivô**

Quicksort: Particionamento (cont.)

- Percurso interrompido quando elemento i > **pivô** e elemento j < **pivô**
  - Se $i < j$, troca-se os elementos e continue
  - Se $i > j$, partição está determinada
- **Pivô**, que se encontra na última posição da tabela, deve ser trocado com o elemento de índice i
- Após troca, elementos < i são menores do que o **pivô**; o índice i indica a posição definitiva do **pivô**, e os elementos > i formam o conjunto de elementos maiores do que o **pivô**

Exemplo *Quicksort*

40	37	95	42	23	51	27
27	37	95	42	23	51	40
↑					↑	
i					*j*	
27	37	95	42	23	51	40
		↑		↑		
		i		*j*		
27	37	23	42	95	51	40
		↑		↑		
		i		*j*		
27	37	23	42	95	51	40
		↑	↑			
		j	*i*			
[27	37	23]	40	[95	51	42]

FIGURA 7.4 Ordenação rápida.

Algoritmo *Quicksort*

```
void quicksort(int v[], int ini, int fim) {
    if (fim - ini < 2) {
        if (fim - ini == 1) {
            if (v[ini] > v[fim]) {
                troque(&v[ini], &v[fim]);
            }
        }
    } else {
        int mediana = pivo(v, ini, fim);
        troque(&v[mediana], &v[fim]);

        int i = ini, j = fim - 1;
        int chave = v[fim];

        while (j >= i) {
            while (v[i] < chave) i++;
            while (v[j] > chave) j--;
            if (j >= i) {
                troque(&v[i], &v[j]);
                i++; j--;
            }
        }
        troque(&v[i], &v[fim]);
        quicksort(v, ini, i - 1);
        quicksort(v, i + 1, fim);
    }
}
```

Complexidade *Quicksort*

- $O(n \log n)$ no **melhor caso** e no **caso médio**
- Embora o **pior caso** não é bom $O(n^2)$, o seu **caso médio** é muito melhor que o **pior caso**
  - Então, *quicksort* é um dos melhores algoritmos de ordenação utilizando comparação entre os elementos

Ordenação por Balde (*Bucket Sort*)

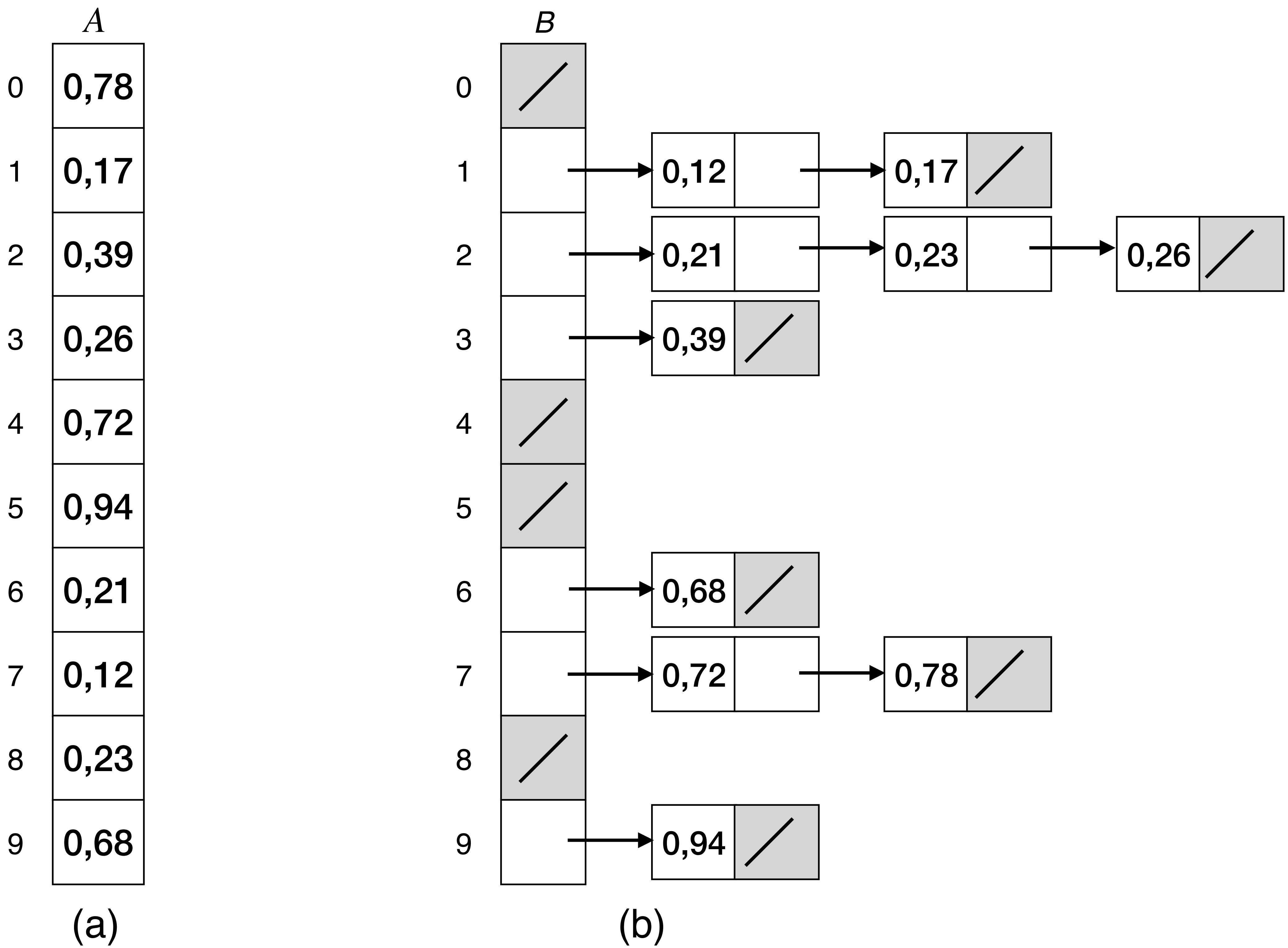
Bucket Sort

- Tempo linear $O(n)$
  - Quando entrada é gerada por uma **distribuição uniforme**
- Rápida porque pressupõe algo sobre a entrada
  - Presume que a entrada é gerada por um processo aleatório que distribui elementos uniformemente sobre o intervalo $[0,1)$

Ideia

- Divida o intervalo $[0,1)$ em n subintervalos de igual tamanho (**baldes**)
- Distribua n entradas entre os **baldes**
  - Como supomos que a entrada é uniformemente distribuída, não esperamos que muitos caiam em cada **balde**
- Ordenar os números em cada **balde**
- Percorra os **baldes** em ordem, listando os elementos contidos em cada um

Figura: *Bucket sort*



(a) O arranjo de entrada $A[0..9]$. (b) O arranjo $B[0..9]$ de listas ordenadas.

Algoritmo: *Bucket sort*

```c
void bucket_sort(int l[], int n) {
    Bucket buckets[NUM_BUCKETS];
    inicialize_buckets(buckets);

    // Distribuir elementos no bucket adequado
    for (int i = 0; i < n; i++) {
        int index = l[i] * NUM_BUCKETS;  // Determinar o bucket com base no valor
        insira_no_bucket(&buckets[index], l[i]);
    }

    // Ordenar cada bucket individualmente com ordenação por inserção
    for (int i = 0; i < NUM_BUCKETS; i++) {
        ordene_bucket(buckets[i], buckets[i].count);
    }

    // Concatenar todos os elementos ordenados de cada bucket no array original
    int pos = 0;
    for (int i = 0; i < NUM_BUCKETS; i++) {
        for (int j = 0; j < buckets[i].count; j++) {
            l[pos++] = // j-ésimo elemento do i-ésimo bucket
        }
    }
}
```

Considerações

- O tempo esperado do ***bucket sort*** é $\Theta(n) + n \cdot O(2 - 1/n) = \Theta(n)$
- Mesmo que a entrada não seja obtida a partir de uma distribuição uniforme, ***bucket sort*** ainda pode ser executado em tempo linear

Referências

- CORMEN, T. H.[et al]. Algoritmos: teoria e prática. 3ª ed. Rio de Janeiro: Elsevier, 2012.
- SZWARCFITER, Jayme Luiz; MARKENZON, Lilian. Estruturas de Dados e seus Algoritmos. Edição: 3a. Editora: LTC. 2010;