Ordenação

Rômulo César Silva

Unioeste

Fevereiro de 2020

Sumário

Revisão de métodos de ordenação

Os métodos de ordenação já estudados anteriormente com a complexidade de tempo para o pior caso:

Algoritmo	**Pior caso**
SelectionSort	$O(n^2)$
InsertionSort	$O(n^2)$
BubbleSort	$O(n^2)$
MergeSort	$O(n\, lg\ n)$
QuickSort	$O(n^2)$
HeapSort	$O(n\, lg\ n)$

Observações:

- o QuickSort no caso médio tem desempenho $O(n\, lg\ n)$, se as sucessivas partições é tal que os elementos são separados próximos da metade
- todos os métodos anteriores são baseados em comparações de valores

Revisão de métodos de ordenação

Observações:

- pode-se demonstrar que a cota inferior do pior caso para ordenação usando comparação de valores é $O(n \, lg \, n)$ (veja [Cormen 1997])
- ou seja, não é possível gastar menos tempo que $O(n \, lg \, n)$ no pior caso.
- isto significa que os métodos MergeSort e HeapSort são algoritmos ótimos para o problema da ordenação usando comparação

Será possível fazer **ordenação sem usar comparação de valores**??

Counting Sort

Counting Sort

Ordena um vetor A de n **inteiros** quando se sabe que os valores estão no intervalo $[1, k]$, sendo k conhecido e $k \in O(n)$

Restrições:

- todos valores são inteiros e menores que k conhecido

Counting Sort

- algoritmo usa 2 vetores auxiliares:
  - vetor B: usado para construir o vetor ordenado
  - vetor C: de tamanho k, usado para contar o número de ocorrências de cada elemento em A

Counting Sort

```
function CountingSort(A[1..n], k)
    ▷ entrada: A é vetor de n elementos a ser ordenado e k o maior valor em A
    ▷ saída: vetor B contendo os elementos A em ordem crescente
    for i ← 1, ..., k do
        C[i] ← 0
    end for
    for j ← 1, ..., n do
        C[A[j]] ← C[A[j]] + 1
    end for
    for i ← 2, ..., k do
        C[i] ← C[i] + C[i − 1]
    end for
    for j ← n downto 1 do
        B[C[A[j]]] ← A[j]
        C[A[j]] ← C[A[j]] − 1
    end for
    return B
end function
```

Counting Sort

Exemplo:

(a)

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

	1	2	3	4	5	6
C	2	0	2	3	0	1

(b)

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

(c)

	1	2	3	4	5	6	7	8
B							4	

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

(d)

	1	2	3	4	5	6	7	8
B		1					4	

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

(e)

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

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

(f)

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

- (a): valores dos vetores A e C após a execução das linhas 04-09
- (b): valores do vetor C após a execução das linhas 10-12
- (c)-(e): valores do vetor B e C após 1, 2 e 3 iterações do laço da linha 13
- (f): vetor de saída B final

Counting Sort

- complexidade: $O(n+k)$. Quando $k \in O(n)$ então a complexidade é $O(n)$, sendo portanto linear.
- CountingSort é método estável
- CountingSort não é método *in place*, já que faz uso de vetores auxiliares

Radix Sort

Radix Sort

Ordena um vetor de n elementos **inteiros** quando todos elementos são representados por d dígitos, onde d é constante

Restrições

- todos os valores são inteiros
- todos os valores são representados com d dígitos

Exemplo de aplicação: ordenação de códigos postais (CEP).

Radix Sort

- faz ordenação dos elementos do vetor dígito a dígito:
  - separa elementos em grupos que tenhma o mesmo dígito mais significativo
  - repete-se o processo considerando apenas d - 1 dígitos menos significativos
- necessita de um método estável para fazer a ordenação dentro de cada dígito

Radix Sort

procedure RadixSort($A[1..n]$, d)
    ▷ entrada: A é vetor de n elementos a ser ordenado e d o número de dígitos de cada valor em A
    ▷ saída: vetor A ordenado
    for $i \leftarrow 1, \ldots, d$ **do**
        ordene os elementos de A pelo i-ésimo dígito usando método estável
    end for
end procedure

Radix Sort

- complexidade: $\Theta(d.f(n))$, onde $f(n)$ é a complexidade do método de ordenação estável usado. Como d é considerado uma constante, então a complexidade é $\Theta(f(n))$
- Se o algoritmo de ordenação estável empregado for o CountingSort, por exemplo, então a complexidade do RadixSort será $\Theta(n)$

Radix Sort

Exemplo:

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

Bucket Sort

Bucket Sort

Ordena um vetor A de n elementos distribuídos uniformemente no intervalo semi-aberto $[0, 1)$.

Restrições:

- os valores são números reais no intervalo $[0, 1)$
- a eficiência do algoritmo depende da distribuição dos elementos no intervalo $[0, 1)$

Bucket Sort

- ideia: dividir o intervalo $[0, 1)$ em n segmentos de mesmo tamanho (*buckets*) e distribuir os n elementos nos respectivos segmentos
- se os elementos estão distribuídos uniformemente no intervalo $[0, 1)$, espera-se que os segmentos tenha aproximadamente o mesmo número de elementos
- ordena-se os segmentos usando algum método de ordenação e em seguida eles são concatenados para se obter todos os elementos em ordem crescente

Bucket Sort

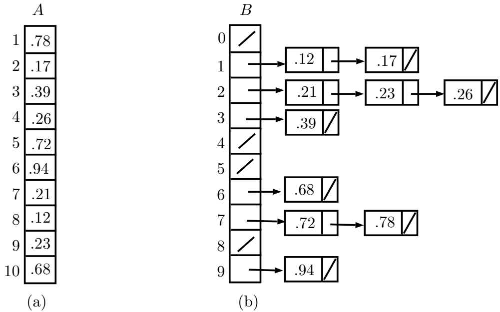
procedure BUCKETSORT($A[1..n]$)
    ▷ entrada: A é vetor de n elementos no intervalor $[0, 1)$
    ▷ saída: vetor A ordenado
    for $i \leftarrow 1, \ldots, n$ **do**
        insere $A[i]$ na lista encadeada $B[\lfloor nA[i] \rfloor]$
    end for
    for $i \leftarrow 0, \ldots, n-1$ **do**
        ordene $B[i]$ usando InsertionSort
    end for
    concatene as listas $B[0], B[1], \ldots, B[n-1]$ nessa ordem
end procedure

Bucket Sort

- o pior caso do algoritmo BucketSort é quadrático devido ao usdo método InsertionSort. Porém, se n elementos estão ditribuídos uniformemente no intervalo $[0, 1)$ então cada seguimento deve ter aproxidamente 1 elemento, e logo as listas devem ter tamanho $\Theta(1)$. Assim a concatenação das n listas leva $\Theta(n)$

Bucket Sort

Exemplo:



- (a): vetor A a ser ordenado
- (b): listas $B[0], B[1], \ldots, B[9]$ geradas, antes da concatenação

Bibliografia I

[Cormen 1997] Cormen,T.; Leiserson, C.; Rivest, R.
Introduction to Algorithms. McGrawHill, New York, 1997.

[Feofiloff 2009] Paulo Feofiloff.
Algoritmos em linguagem C. Elsivier, Rio de Janeiro, 2009.