MO417 — Complexidade de Algoritmos I

Cid Carvalho de Souza Cândida Nunes da Silva
Orlando Lee

5 de setembro de 2011

Revisado por Zanoni Dias

Ordenação em Tempo Linear

Algoritmos lineares para ordenação

Os seguintes algoritmos de ordenação têm complexidade $O(n)$:

- Counting Sort: Elementos são números inteiros "pequenos"; mais precisamente, inteiros $x \in O(n)$.
- Radix Sort: Elementos são números inteiros de comprimento máximo constante, isto é, independente de n.
- Bucket Sort: Elementos são números reais uniformemente distribuídos no intervalo $[0..1)$.

Counting Sort

- Considere o problema de ordenar um vetor $A[1 \ldots n]$ de inteiros quando se sabe que todos os inteiros estão no intervalo entre 0 e k.
- Podemos ordenar o vetor simplesmente contando, para cada inteiro i no vetor, quantos elementos do vetor são menores que i.
- É exatamente o que faz o algoritmo *Counting Sort*.

Counting Sort

```
COUNTING-SORT(A, B, n, k)
1  para i ← 0 até k faça
2      C[i] ← 0
3  para j ← 1 até n faça
4      C[A[j]] ← C[A[j]] + 1
   ▷ C[i] é o número de js tais que A[j] = i
5  para i ← 1 até k faça
6      C[i] ← C[i] + C[i − 1]
   ▷ C[i] é o número de js tais que A[j] ≤ i
7  para j ← n decrescendo até 1 faça
8      B[C[A[j]]] ← A[j]
9      C[A[j]] ← C[A[j]] − 1
```


Counting Sort - Complexidade

- Qual a complexidade do algoritmo COUNTING-SORT?
- O algoritmo não faz comparações entre elementos de A!
- Sua complexidade deve ser medida em função do número das outras operações, aritméticas, atribuições, etc.
- Claramente, a complexidade de COUNTING-SORT é $O(n+k)$. Quando $k \in O(n)$, ele tem complexidade $O(n)$.

Há algo de errado com o limite inferior de $\Omega(n \log n)$ para ordenação?

Algoritmos *in-place* e *estáveis*

- Algoritmos de ordenação podem ser ou não *in-place* ou *estáveis*.
- Um algoritmo de ordenação é in-place se a memória adicional requerida é independente do tamanho do vetor que está sendo ordenado.
- Exemplos: QUICKSORT e HEAPSORT são métodos de ordenação *in-place*, já MERGESORT e COUNTING-SORT não são.
- Um método de ordenação é estável se elementos iguais ocorrem no vetor ordenado na mesma ordem em que são passados na entrada.
- Exemplos: COUNTING-SORT e QUICKSORT são exemplos de métodos estáveis (desde que certos cuidados sejam tomados na implementação). HEAPSORT não é.

Radix Sort

- Considere agora o problema de ordenar um vetor $A[1 \ldots n]$ inteiros quando se sabe que todos os inteiros podem ser representados com apenas d dígitos, onde d é uma constante.
- Por exemplo, os elementos de A podem ser CEPs, ou seja, inteiros de 8 dígitos.

Radix Sort

- Poderíamos ordenar os elementos do vetor dígito a dígito da seguinte forma:
  - Separamos os elementos do vetor em grupos que compartilham o mesmo dígito mais significativo.
  - Em seguida, ordenamos os elementos em cada grupo pelo mesmo método, levando em consideração apenas os $d - 1$ dígitos menos significativos.
- Esse método funciona, mas requer o uso de bastante memória adicional para a organização dos grupos e subgrupos.

Radix Sort

- Podemos evitar o uso excessivo de memória adicional começando pelo dígito menos significativo.
- É isso o que faz o algoritmo Radix Sort.
- Para que Radix Sort funcione corretamente, ele deve usar um método de ordenação *estável*.
- Por exemplo, o COUNTING-SORT.

Radix Sort

Suponha que os elementos do vetor A a ser ordenado sejam números inteiros de até d dígitos. O *Radix Sort* é simplesmente:

```
RADIX-SORT(A, n, d)
1  para i ← 1 até d faça
2      Ordene A[1...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	$\rightarrow$	457	$\rightarrow$	839	$\rightarrow$	457	
436		657		355		657	
720		329		457		720	
355		839		657		839	
		$\uparrow$		$\uparrow$		$\uparrow$	

Radix Sort - Corretude

O seguinte argumento indutivo garante a corretude do algoritmo:

- Hipótese de indução: os números estão ordenados com relação aos $i-1$ dígitos menos significativos.
- O que acontece ao ordenarmos pelo i-ésimo dígito?
- Se dois números têm i-ésimo dígitos distintos, o de menor i-ésimo dígito aparece antes do de maior i-ésimo dígito.
- Se ambos possuem o mesmo i-ésimo dígito, então a ordem dos dois também estará correta pois o método de ordenação é estável e, pela HI, os dois elementos já estavam ordenados segundo os $i-1$ dígitos menos significativos.

Radix Sort - Complexidade

- Qual é a complexidade de RADIX-SORT?
- Depende da complexidade do algoritmo estável usado para ordenar cada dígito.
- Se essa complexidade for $\Theta(f(n))$, obtemos uma complexidade total de $\Theta(d\ f(n))$.
- Se o algoritmo estável for, por exemplo, o COUNTING-SORT, obtemos a complexidade $\Theta(d(n+k))$.
- Se $k \in O(n)$ e $d \in O(1)$, então o RADIX-SORT possui uma complexidade linear em n.

E o limite inferior de $\Omega(n \log n)$ para ordenação?

Radix Sort - Complexidade

- Em contraste, um algoritmo por comparação como o MERGESORT teria complexidade $\Theta(n \lg n)$.
- Assim, RADIX-SORT é mais vantajoso que MERGESORT quando $d < \lg n$, ou seja, o número de dígitos for menor que $\lg n$.
- Se n for um limite superior para o maior valor a ser ordenado, então $O(\log n)$ é uma estimativa para a quantidade de dígitos dos números.
- Isso significa que não há diferença significativa entre o desempenho do MERGESORT e do RADIX-SORT?

Radix Sort - Complexidade

- O nome *Radix Sort* vem da **base** (em inglês *radix*) em que intepretamos os dígitos.
- A vantagem de se usar RADIX-SORT fica evidente quando interpretamos os dígitos de forma mais geral que a base decimal ([0..9]).
- Tomemos o seguinte exemplo: suponha que desejemos ordenar um conjunto de $n = 2^{20}$ números de 64 bits. Então, MERGESORT faria cerca de $n \lg n = 20 \times 2^{20}$ comparações e usaria um vetor auxiliar de tamanho 2^{20}.

Radix Sort - Complexidade

- Agora suponha que interpretamos cada número do como tendo $d = 4$ dígitos em base $k = 2^{16}$, e usarmos RADIX-SORT com o *Counting Sort* como método estável.

  Então a complexidade de tempo seria da ordem de $d(n+k) = 4(2^{20} + 2^{16})$ operações, bem menor que 20×2^{20} do MERGESORT. Mas, note que utilizamos dois vetores auxiliares, de tamanhos 2^{16} e 2^{20}.
- Se o uso de memória auxiliar for muito limitado, então o melhor mesmo é usar um algoritmo de ordenação por comparação *in-place*.
- Note que é possível usar o *Radix Sort* para ordenar outros tipos de elementos, como datas, palavras em ordem lexicográfica e qualquer outro tipo que possa ser visto como uma d-tupla ordenada de itens comparáveis.

Bucket Sort

- Supõe que os n elementos da entrada estão distribuídos uniformemente no intervalo $[0, 1)$.
- A idéia 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.
- Em seguida, os elementos de cada segmento são ordenados por um método qualquer. Finalmente, os segmentos ordenados são concatenados em ordem crescente.

Bucket Sort - Pseudocódigo

```
BUCKETSORT(A, n)
1  para i ← 1 até n faça
2      insira A[i] na lista ligada B[⌊n A[i]⌋]
3  para i ← 0 até n − 1 faça
4      ordene a lista B[i] com INSERTION-SORT
5  Concatene as listas B[0], B[1], ..., B[n − 1]
```


Bucket Sort - Exemplo

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

Bucket Sort - Corretude

- Dois elementos x e y de A, $x < y$, ou terminam na mesma lista ou são colocados em listas diferentes $B[i]$ e $B[j]$.
- A primeira possibilidade implica que x aparecerá antes de y na concatenação final, já que cada lista é ordenada.
- No segundo caso, como $x < y$, segue que $i = \lfloor nx \rfloor \leq \lfloor ny \rfloor = j$. Como $i \neq j$, temos $i < j$. Assim, x aparecerá antes de y na lista final.

Bucket Sort - Complexidade

- É claro que o pior caso do *Bucket Sort* é quadrático, supondo-se que as ordenações das listas seja feita com ordenação por inserção.
- Entretanto, o tempo esperado é linear.
  Intuitivamente, a idéia da demonstração é que, como os n elementos estão distribuídos uniformemente no intervalo $[0, 1)$, então o tamanho esperado das listas é pequeno.
- Portanto, as ordenações das n listas $B[i]$ leva tempo total esperado $\Theta(n)$.
- Os detalhes podem ser vistos no livro de CLRS.