

Algoritmos de Ordenação: Tempo Linear

ACH2002 - Introdução à Ciência da Computação II

Delano M. Beder

Escola de Artes, Ciências e Humanidades (EACH)

Universidade de São Paulo

`dbeder@usp.br`

10/2008

Material baseado em slides dos professores Cid de Souza e Cândida da Silva

Algoritmos lineares para ordenação

Os seguintes algoritmos de ordenação têm complexidade $O(n)$, onde n é o tamanho do vetor A a ser ordenado:

- *Counting Sort*: Elementos são números inteiros **pequenos**; mais precisamente, inteiros x onde $x \in O(n)$.
 - Exemplo: Ordenar um conjunto S de n elementos x_i (para todo i , $1 \leq i \leq n$, $x_i \leq n$).
- *Radix Sort*: Elementos são números inteiros de comprimento máximo constante, isto é, independente de n .
 - Exemplo: Ordenar um conjunto S de n elementos x_i (para todo i , $1 \leq i \leq n$, x_i tem no máximo d dígitos).

Counting Sort

- Considere o problema de ordenar um arranjo A de n inteiros quando é sabido que todos os inteiros i estão no intervalo entre 0 e k ($0 \leq i < k$), ou seja, $k \in O(n)$.
- 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*, em tempo $O(n + k)$, isto é $O(n)$.
- O algoritmo usa dois vetores auxiliares:
 - C de tamanho k que guarda em $C[i]$ o número de ocorrências de elementos $\leq i$ em A .
 - B de tamanho n onde se constrói o vetor ordenado.

Counting Sort – Algoritmo

CountingSort(A, k)

Entrada – Vetor A de tamanho n e um inteiro k , o valor do maior inteiro em A .

Saída – Os elementos do Vetor A ordenados em ordem crescente.

1. **para** $i = 0$ **até** $k-1$ **faça** $C[i] = 0$;
2. **para** $j = 0$ **até** $n-1$ **faça** $C[A[j]] = C[A[j]] + 1$;
3. **para** $i = 1$ **até** $k-1$ **faça** $C[i] = C[i] + C[i-1]$;
4. **para** $j = n-1$ **até** 0 **faça**
 5. $B[C[A[j]] - 1] = A[j]$;
 6. $C[A[j]] = C[A[j]] - 1$;
7. **retorne** (B)

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, o número de tais operações é uma função em $O(n + k)$, já que temos dois laços simples com n iterações e dois com k iterações.
- Assim, quando $k \in O(n)$, este algoritmo tem complexidade $O(n)$.

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: o *QuickSort* e o *HeapSort* são métodos de ordenação *in-place*, já o *MergeSort* e o *Counting Sort* não.
- Um método de ordenação é estável se elementos iguais ocorrem no vetor ordenado na mesma ordem em que são passados na entrada.
- Exemplos: o *Counting Sort* e o *QuickSort* são exemplos de métodos estáveis (desde que certos cuidados sejam tomados na implementação). O *HeapSort* não é.

- Considere agora o problema de ordenar um vetor A de n inteiros quando é sabido 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.
- Poderíamos ordenar os elementos do vetor dígito a dígito começando pelo dígito menos significativo.
 - É isso que faz o algoritmo *Radix Sort*.
- Para que o algoritmo *Radix Sort* funcione corretamente é necessário utilizar um método estável de ordenação para a ordenação de cada dígito.
- Para isso podemos usar, por exemplo, o *Counting Sort*.

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

RadixSort(A, d)

Entrada – Vetor A e um inteiro d , o número máximo de dígitos dos elementos em A .

Saída – Os elementos do Vetor A ordenados em ordem crescente.

1. **para** $i = 1$ **até** d **faça**
2. ordene os elementos de A 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		657		355		657
720		329		457		720
355		839		657		839

Radix Sort – Complexidade

- Depende da complexidade do algoritmo estável usado para ordenar cada dígito dos elementos.
- Se essa complexidade estiver em $\Theta(f(n))$, obtemos uma complexidade total de $\Theta(d f(n))$ para o Radix Sort.
- Como supomos d constante, a complexidade reduz-se para $\Theta(f(n))$.
- Se o algoritmo estável for, por exemplo, o *Counting Sort*, obtemos a complexidade $\Theta(n + k)$.
- Supondo $k \in O(n)$, resulta numa complexidade linear em n .

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

Radix Sort – Complexidade

- O nome *Radix Sort* vem da base (em inglês *radix*) em que interpretamos os dígitos.
- Veja que se o uso de memória auxiliar for muito limitado, então o melhor é usar um algoritmo de ordenação de 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 vista como uma d -upla ordenada de itens comparáveis.

Discussão:

Suponha que desejemos ordenar um conjunto de 2^{20} números de 64 bits. Qual seria o melhor algoritmo ? *MergeSort* ou *Radix Sort* ?