MC202 - Estruturas de Dados

Guilherme P. Telles

IC

5 de Novembro de 2019

Avisos

- Estes slides contêm erros.
- Estes slides são incompletos.
- Estes slides foram escritos usando português anterior à reforma ortográfica de 2009.

Parte I

Ordenação

Ordenação

- O problema da ordenação é rearranjar um conjunto de registros $R_1, R_2, \ldots, R_n$ em função de uma chave secundária k_i, de tal forma que um $k_1, k_2, \ldots, k_n$ fiquem em ordem não-decrescente.
- Uma chave secundária é um campo que não é necessariamente único mas que é comparável. P.ex, nome, data de nascimento, idade.

Algoritmos de ordenação

- Há vários algoritmos.
- Alguns são eficientes em geral.
- Outros são mais eficientes apenas quando as chaves têm alguma característica particular.

in-place

- Alguns algoritmos usam apenas uma quantidade constante de memória de trabalho. São chamados de *in-place*.

Estável

- Um algoritmo de ordenação é *estável* se preserva a ordem relativa de chaves repetidas.
- Essa característica é relevante para ordenações sucessivas (por exemplo, ordenar por data e depois por valor) e para outras aplicações.

Operações na comparação de desempenho

- Comparações de chaves.
- Trocas de chaves. Cada troca faz três operações de leitura e três de escrita. Duas envolvem a memória e uma envolve um registrador.

```
temp = A[j];
A[j] = A[i];
A[i] = temp;
```

- Movimentação de chaves. Faz uma operação de leitura e uma de escrita na memória.

```
A[j] = A[i];
```

Notação

- Vamos supor que as n chaves que serão ordenadas estejam em um vetor A nas posições entre 1 e n.

Ordenação por comparação de chaves

- Há muitos algoritmos que funcionam fazendo comparações de chaves.
- Algoritmos desse tipo fazem $\Omega(n \log_2 n)$ comparações de chaves.
- Existe uma prova desse fato.

Bubble-sort

```
BUBBLE-SORT(A[1..n])
for i = 2 to n
    for j = n downto i
        if A[j] < A[j − 1]
            exchange A[j] with A[j − 1]
```

Bubble-sort

- No pior caso faz

$$(n-1)+(n-2)+\ldots+1 = n(n-1)/2 = O(n^2)$$

  comparações de chaves.
- No pior caso, faz o mesmo número de trocas de chaves.

Selection-sort

```
SELECTION-SORT(A[1..n])
for i = n downto 2
    max = 1
    for j = 2 to i
        if A[j] > A[max]
            max = j
    exchange A[i] and A[max]
```

Selection-sort

- No pior caso faz

$$(n-1) + (n-2) + \ldots + 1 = n(n-1)/2 = O(n^2)$$

  comparações de chaves.
- O número de trocas de chaves é igual a $n-1$.

Insertion-sort

$\text{Insertion-Sort}(A[1..n])$

for $j = 2$ **to** n
    $key = A[j]$
    $i = j - 1$
    while $i > 0$ and $A[i] > key$
        $A[i+1] = A[i]$
        $i = i - 1$
    $A[i+1] = key$

Insertion-sort

- No pior caso, faz

$$1 + 2 + \ldots + (n-1) = n(n-1)/2 = O(n^2)$$

  comparações de chaves.

- No pior caso também faz

$$n - 2 + n(n+1)/2 = O(n^2)$$

  movimentações de chaves.

Insertion-sort

- No pior caso, faz

$$1 + 2 + \ldots + (n-1) = n(n-1)/2 = O(n^2)$$

  comparações de chaves.

- No pior caso também faz

$$n - 2 + n(n+1)/2 = O(n^2)$$

  movimentações de chaves.

- Em casos em as chaves estão quase ordenadas faz $O(n)$ comparações e movimentações.

Shell-sort[1]

```
SHELL-SORT(A[1..n])
incs = {..., 209, 109, 41, 19, 5, 1}
for t = 1 to |incs|
    for j = incs[t] + 1 to n
        key = A[j]
        i = j - incs[t]
        while i > 0 and A[i] > key
            A[i + incs[t]] = A[i]
            i = i - incs[t]
        A[i + incs[t]] = key
```

[1] D.L. Shell, 1959

Shell-sort

- O número de comparações e movimentações depende muito da seqüência de incrementos.
- Há várias, escolhidas teórica e empiricamente.
- Para a seqüência de Sedgewick o número de operações é $O(n^{7/6})$ no caso médio.

$$h_s = \begin{cases} 9 \times 2^s - 9 \times 2^{s/2} + 1 & \text{se } s \text{ é par} \\ 8 \times 2^s - 6 \times 2^{(s+1)/2} + 1 & \text{se } s \text{ é impar} \end{cases}$$

Merge-sort[2]

```
MERGE-SORT(A, ℓ, r)
if ℓ < r
    m = ⌊(ℓ + r)/2⌋
    MERGE-SORT(A, ℓ, m)
    MERGE-SORT(A, m + 1, r)
    MERGE(A, ℓ, m, r)
```

[2] J. von Neumann, 1945

Intercalação

```
MERGE(A, ℓ, m, r)
n_1 = m − ℓ + 1
n_2 = r − m
for i = 1 to n_1
        L[i] = A[ℓ + i − 1]
L[n_1 + 1] = +∞
for j = 1 to n_2
        R[j] = A[m + j]
R[n_2 + 1] = +∞
i = 1
j = 1
for k = ℓ to r
        if L[i] ≤ R[j]
                A[k] = L[i]
                i = i + 1
        else
                A[k] = R[j]
                j = j + 1
```

Merge-sort

- Vamos supor que n é uma potência de 2 para ter contas exatas, isso não afeta nossa conclusão.
- Na primeira chamada, $\textsc{Merge}$ faz n comparações de chaves. Na duas segundas faz $(n/2)$ em cada. Nas quatro terceiras faz $(n/4)$ em cada e assim por diante. Nas $n/2$ últimas faz 2 em cada.
- Somando ao longo de $\lg n$ níveis, temos $n \lg n$ comparações de chave.
- O número de movimentações de chaves é sempre $2n$ em cada nível. Então são $2n \lg n$ trocas no total.
- O número total de operações no pior caso é $O(n \lg n)$.

Merge-sort

- O $\textsc{Merge}$ anterior usa memória adicional igual a $n + 2$.
- Existem algoritmos de intercalação que usam tempo $O(n)$ e memória constante[3].

[3] Um exemplo é B.C. Huang e M.A. Langston. Practical In-Place Merging. Comm. ACM, v. 31, p. 348, 1988.

Heapsort[4]

```
Heapsort(A)
Build-Max-Heap(A)
for i = A.length downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size − 1
    Max-Heapify(A, 1)
```

[4] J.W.J. Williams, 1964.

Heapsort

- A construção do heap faz menos que $2n$ comparações de chaves e menos que $2n$ trocas de chaves.
- No pior caso possível, cada remoção de máximo tem que mover uma chave da raiz até alguma folha, fazendo 2 comparações e 1 troca de chaves a cada passo. Para n remoções de máximo o número total de comparações é $O(n \lg n)$. O número total de trocas também.

QUICKSORT[5]

```
QUICKSORT(A, ℓ, r)
// Sort A[ℓ, r].
if ℓ < r
      p = PARTITION(A, ℓ, r)
      QUICKSORT(A, ℓ, p − 1)
      QUICKSORT(A, p + 1, r)
```

[5] C.A.R. Hoare, 1961.

Partição

```
PARTITION(A, l, r)
p = A[l]
i = l + 1
j = r
while true
    while A[i] < p and i ≤ r
        i++
    while A[j] > p
        j--
    if j ≤ i
        break
    else
        SWAP(A[i], A[j])
        i++
        j--
SWAP(A[j], A[l])
return j
```

QUICKSORT

- Suponha que A já esteja ordenado e que as chaves sejam distintas. Então p é o mínimo em A. PARTITION faz $i = l$ e $j = l$, depois troca p com ele mesmo. Faz $n + 1$ comparações de chaves e 1 troca.
- Isso reduz o problema a dois outros, um de tamanho 0 e outro de tamanho $n - 1$.
- No pior caso temos então $n + 1$ comparações, depois n comparações, depois $n - 1$ comparações e assim por diante.
- O número total de comparações no pior caso é $O(n^2)$. O número de trocas é $O(n)$.

Quicksort

- Agora suponha que A é um vetor tal que as escolhas do pivô sempre usam a mediana.
- Com a mediana como pivô, Partition faz aproximadamente n comparações de chaves e no máximo $n/2$ trocas.
- Isso reduz o problema a dois outros, de tamanho aproximadamente $(n-1)/2$.
- Nesse caso temos aproximadamente n comparações, depois $2n/2$ comparações, depois $4n/4$ comparações e assim por diante. A divisão acontece $\lg n - 1$ vezes.
- O número total de comparações é $O(n \lg n)$. O número de trocas também.

Quicksort

- Apesar de um pior caso ruim, o caso médio é bom, $O(n \lg n)$.
- Tipicamente o pivô é escolhido aleatoriamente ou como a mediana entre o primeiro, o último e o elemento mediano do vetor.
- O Quicksorté considerado o melhor algoritmo prático, em parte porque os elementos do vetor são comparados contra o pivô (um na memória, outro em registrador), ao contrário dos outros algoritmos que comparam dois elementos do vetor entre si (dois na memória).
- Está implementado em várias bibliotecas.

Quicksort aleatorizado

Randomized-Partition(A, ℓ, r)

$i = \text{Random}(\ell, r)$
exchange $A[\ell]$ with $A[i]$
return Partition(A, ℓ, r)

Randomized-Quicksort(A, ℓ, r)

if $\ell < r$
    $p = \text{Randomized-Partition}(A, \ell, r)$
    Randomized-Quicksort$(A, \ell, p-1)$
    Randomized-Quicksort$(A, p+1, r)$

Memória

- Se o maior sub-problema for chamado primeiro na recursão, podemos ter $n-1$ registros de ativação na pilha.
- O $\textsc{Quicksort}$pode ser modificado para deixar na recursão de cauda sempre o maior subproblema.
- Dessa forma, o subproblema resolvido recursivamente sempre tem tamanho menor que $n/2$ e o tamanho da pilha não excede $\lceil \lg n \rceil$.

Melhorias práticas

- Além da recursão de cauda, outras melhorias foram propostas para as implementações de QUICKSORT, como:
  - Evitar os testes dos limites do array durante o particionamento.
  - Usar um algoritmo não-recursivo, como Insertion-Sort, mais rápido que o QUICKSORTpara subproblemas pequenos.
  - Alternativamente, não resolver os subproblemas pequenos e executar Insertion-Sort para o array inteiro no fim.

```
QSORT(A, l, r)
while r − l + 1 > n_0
    j = PICK-PIVOT(a, l, r)
    SWAP(A[l], A[j])
    p = A[l]
    i = l
    j = r
    repeat
        while A[i] < p do i++
        while A[j] > p do j−−
        if i ≤ j
            SWAP(A[i], A[j])
            i++
            j−−
    until i > j
    if i < (l + r)/2
        QSORT(A, l, j)
        l = i
    else
        QSORT(A, i, r)
        r = j
INSERTIONSORT(A[l..r])
```

Ordenação sem comparação de chaves

- Se baseiam em alguma característica dos dados para levarem tempo menor que $O(n \lg n)$.

Bucket-sort

- Supõe que as chaves estão no intervalo $[0, k]$.
- Usa uma lista encadeada para cada valor de chave, cada chave é colocada na lista correspondente ao seu valor e depois recolhe as listas em ordem.

Bucket-sort

```
BUCKET-SORT(A, k)
let B[0..k] be a new array of lists
for i = 0 to k
    B[i] = empty list
for i = 0 to n − 1
    insert A[i] at the end of list B[A[i]]
copy lists B[0], B[1], ..., B[k] in order into A
```

Bucket-sort

- É $O(k+n)$.
- Usa memória para $k+n$ apontadores e n registros.

Counting-sort

- Supõe que as chaves estão no intervalo $[0, k]$.
- Constrói um vetor de freqüências acumuladas das chaves e depois redistribui as chaves em ordem.

Counting-sort

COUNTING-SORT(A, B, k)

let $C[0\,..\,k]$ be a new array
for $i = 0$ **to** k
    $C[i] = 0$
for $j = 0$ **to** $n - 1$
    $C[A[j]] = C[A[j]] + 1$
for $i = 1$ **to** k
    $C[i] = C[i] + C[i-1]$
for $j = n - 1$ **downto** 0
    $C[A[j]] = C[A[j]] - 1$
    $B[C[A[j]]] = A[j]$

Counting-sort

- É $O(k + n)$.
- Usa memória para k inteiros e n registros.

Radix-sort

- Supõe que as cada chave tem d dígitos.
- Ordena as chaves dígito-a-dígito do menos significativo para o mais significativo.
- Vamos supor que o dígito mais significativo ocupa a posição 1 e que o menos significativo ocupa a posição d.

Radix-sort

```
RADIX-SORT(A, d)
for i = d downto 1
        use a stable sort to sort array A on digit i
```

Radix-sort

- Faz trabalho proporcional a d vezes o trabalho do algoritmo aplicado a cada dígito.
- Com bucket-sort é $O(d(k + n))$.