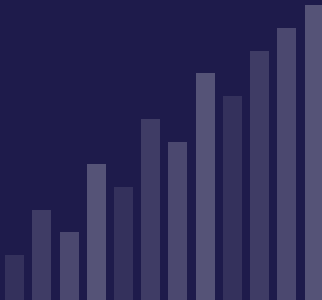

COMPLEMENTO 1 · BLOCO COMPLEMENTAR: ORDENAÇÃO AVANÇADA (OPCIONAL)

Quicksort

Seções 7.1 e 7.2

Estruturas de dados e seus algoritmos · José Dihego · IFBA



Roteiro da aula

Ao final, você será capaz de

- ▶ Executar PARTITION e provar sua corretude pelo invariante das quatro regiões
- ▶ Analisar o pior caso ($\Theta(n^2)$) e o melhor caso ($\Theta(n \lg n)$)
- ▶ Entender por que partições desbalanceadas em proporção constante ainda dão $n \lg n$
- ▶ Medir $\approx 2n \ln n$ comparações em entradas aleatórias e $n^2/2$ em entradas ordenadas



Teoria

Partição de Lomuto, invariante, desempenho do quicksort



Prática em C

Quicksort em C



Experimento

Aleatório contra ordenado: $1,39 n \lg n$ contra $n^2/2$



Conexões

Bibliotecas padrão, bancos de dados, segurança e paralelismo

Dividir e conquistar, de outro jeito

- ▶ **Dividir:** reorganiza $A[p..r]$ em $A[p..q-1] \leq A[q] < A[q+1..r]$. O trabalho está aqui.
- ▶ **Conquistar:** ordena as duas partes recursivamente.
- ▶ **Combinar:** nada a fazer: o vetor já está ordenado.

Ao contrário do merge sort (divisão trivial, combinação cara), o quicksort tem divisão cara e combinação trivial. É **no lugar** e, na prática, o mais rápido em média.

Inventado por C. A. R. Hoare em 1959, ao traduzir dicionários de russo; está nas bibliotecas de C, C++, Java e Go.

A partição

Invariante (início de cada volta):

1. $A[p..i] \leq x$
2. $A[i + 1..j - 1] > x$
3. $A[r] = x$

Custo: $\Theta(n)$, exatamente $n - 1$ comparações.

PARTITION(A, p, r)

```

1  $x \leftarrow A[r]$  // pivô
2  $i \leftarrow p - 1$ 
3 para  $j \leftarrow p$  até  $r - 1$  faça
4     se  $A[j] \leq x$  então
5          $i \leftarrow i + 1$ 
6     troque  $A[i]$  com  $A[j]$ 
7 troque  $A[i + 1]$  com  $A[r]$ 
8 retorne  $i + 1$ 
```

A partição (exemplo)

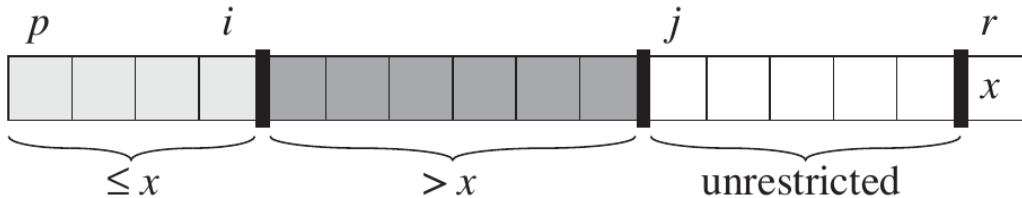


Figura 7.2 de Cormen et al. (2009) — as quatro regiões: $\leq x$, $> x$, ainda não vista e o pivô.

Partição passo a passo

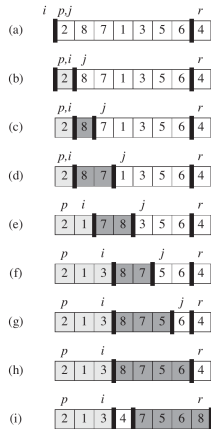


Figura 7.1 de Cormen et al. (2009) — PARTITION em $\langle 2, 8, 7, 1, 3, 5, 6, 4 \rangle$ com pivô 4: cinza claro $\leq x$, cinza escuro

Pior caso, melhor caso

Pior caso

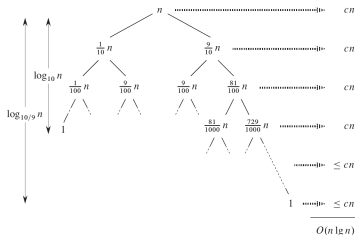
Partição com 0 e $n - 1$ elementos (vetor já ordenado!):

$$T(n) = T(n - 1) + \Theta(n) = \Theta(n^2)$$

Melhor caso

Partição ao meio:

$$T(n) = 2T(n/2) + \Theta(n) = \Theta(n \lg n)$$





Prática em C

- ▶ Partição de Lomuto, como no livro
- ▶ Recursão nas duas partes; nada a combinar
- ▶ Teste com o vetor da Figura 7.1

Cuidados práticos

- ▶ **Pilha de recursão.** No pior caso, a recursão tem profundidade n : estoura a pilha para n grande. Solução: recursão só no lado **menor** e laço no maior, garantindo profundidade $O(\lg n)$ (problema 7-4). O experimento faz isso.
- ▶ **Chaves repetidas.** Com muitos iguais, Lomuto degenera; a partição de Hoare (problema 7-1) ou a de três vias resolve.
- ▶ **Subvetores pequenos.** Bibliotecas trocam para a inserção abaixo de ≈ 16 elementos (exercício 7.4-5).

Quicksort em C (1/3)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void troca(int A[], int i, int j) { int t = A[i]; A[i] = A[j]; A[j] = t; }
5
6  /* PARTITION: pivô x = A[r]. Invariante: A[p..i] <= x, A[i+1..j-1] > x. */
7  int particiona(int A[], int p, int r) {
8      int x = A[r], i = p - 1;
9      for (int j = p; j < r; j++)
10         if (A[j] <= x) troca(A, ++i, j);           // cresce a região dos menores
11     troca(A, i + 1, r);                          // o pivô vai para o meio
12     return i + 1;
13 }
```

Quicksort em C (2/3)

```
14
15 void quicksort(int A[], int p, int r) {
16     if (p < r) {
17         int q = particiona(A, p, r);           // A[p..q-1] <= A[q] < A[q+1..r]
18         quicksort(A, p, q - 1);
19         quicksort(A, q + 1, r);               // nada a combinar
20     }
21 }
```

Quicksort em C (3/3)

```
23 int main(void) {
24     int n;
25     if (scanf("%d", &n) != 1 || n <= 0) return 1;
26     int *A = malloc(n * sizeof(int));
27     for (int k = 0; k < n; k++)
28         if (scanf("%d", &A[k]) != 1) return 1;
29     quicksort(A, 0, n - 1);
30     for (int k = 0; k < n; k++) printf("%d%c", A[k], k < n - 1 ? ' ' : '\n');
31     free(A);
32     return 0;
33 }
```



Experimento

- ▶ Aleatório: $2(n+1)H_n - 4n \approx 1,39 n \lg n$
- ▶ Ordenado: exatamente $n(n-1)/2$
- ▶ n até 2^{20} no caso aleatório

Caso médio e pior caso, medidos

Hipótese. Em vetores aleatórios, o número esperado de comparações é

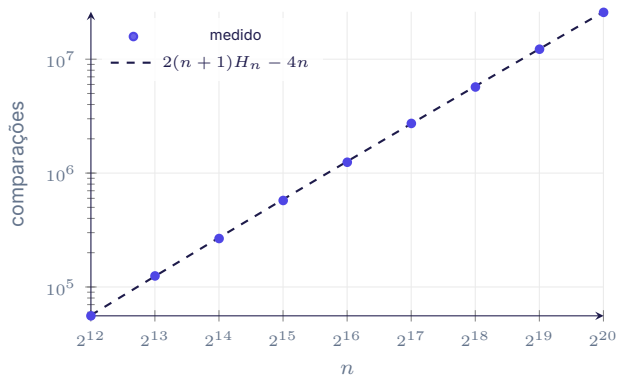
$$2(n+1)H_n - 4n \approx 2n \ln n \approx 1,39 n \lg n.$$

Num vetor já ordenado, o pivô $A[r]$ é sempre o máximo: partições $n-1$ e 0 , e exatamente $n(n-1)/2$ comparações.

Método

1. Contador na comparação $A[j] \leq x$
2. Aleatório: $n = 2^{12}$ a 2^{20} , média de 5 vetores
3. Ordenado: n de 1.000 a 12.000
4. Recursão no lado menor (pilha $O(\lg n)$)

Vetores aleatórios



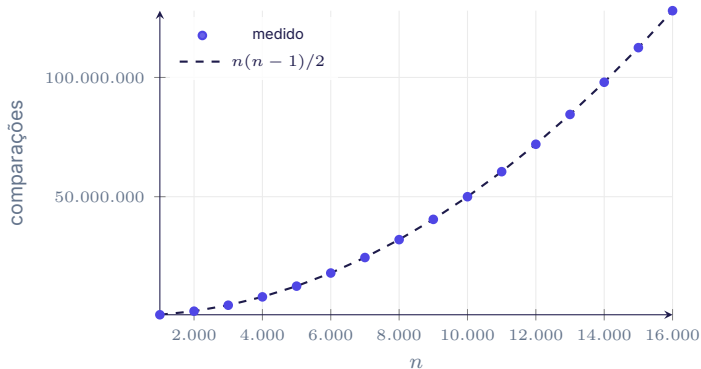
n	medido	razão
4.096	55.803	0,988
8.192	124.989	1,005
16.384	266.356	0,981
32.768	574.775	0,977
65.536	1.245.052	0,983
1.048.576	25.775.316	0,988

razão = medido $\div$ $2(n+1)H_n - 4n$

✓ desvio máx. 1,7%

$H_n \approx \ln n + 0,5772$ (constante de Euler). O medido fica a cerca de 1% da previsão.

Vetor já ordenado: o pior caso



n	medido	razão
1.000	499.500	1,000
4.000	7.998.000	1,000
7.000	24.496.500	1,000
10.000	49.995.000	1,000
13.000	84.493.500	1,000
16.000	127.992.000	1,000

razão = medido $\div$ $n(n-1)/2$

✓ desvio máx. 0,0%

Exatamente $n(n-1)/2$: com 12.000 elementos já ordenados, 72 milhões de comparações. Aleatório com 2^{20} : 26 milhões.

EXPERIMENTO

O que o experimento mostrou

A previsão de caso médio, $2(n+1)H_n - 4n \approx 1,39 n \lg n$, se confirma com desvio de cerca de 1%. E o pior caso é exatamente $n(n-1)/2$, justamente na entrada mais “fácil”, o vetor ordenado. A próxima aula elimina esse risco com aleatoriedade.

➤ Reproduza: `gcc -O2 compl1_experimento.c -o exp -lm && ./exp`



Conexões

- ▶ Bibliotecas padrão
- ▶ Bancos de dados
- ▶ Segurança
- ▶ Computação paralela

Onde isso aparece na computação

Bibliotecas padrão

qsort (C), `std::sort` (introsort em C++), `Arrays.sort` para primitivos (quicksort de dois pivôs no Java) e `sort` do Go (pdqsort).

Bancos de dados

Ordenar resultados em memória (ORDER BY) e preparar *joins* usa quicksort ou variações; a partição também aparece na seleção do k -ésimo (LIMIT).

Segurança

Um adversário que conhece o pivô pode montar entradas $\Theta(n^2)$ ("McIlroy killer): por isso as bibliotecas aleatorizam ou usam introsort.

Computação paralela

Depois de particionar, as duas partes são independentes; GPUs ordenam com variações paralelas da partição.

Para casa

Exercícios

1. (7.1-1) Ilustre PARTITION em $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 21, 2, 6, 11 \rangle$.
2. (7.1-2) Que valor q PARTITION devolve quando todos são iguais? Modifique-a para devolver o meio.
3. (7.2-3) Mostre que o quicksort é $\Theta(n^2)$ quando A está em ordem decrescente.
4. (Problema 7-1) Implemente a partição de Hoare e compare o número de trocas com o de Lomuto.
5. Meça o experimento com 90% de chaves repetidas. O que acontece com Lomuto?

Leitura

Cormen et al., seções 7.1 e 7.2; problema 7-1.

Código

josedihego.net/estruturas