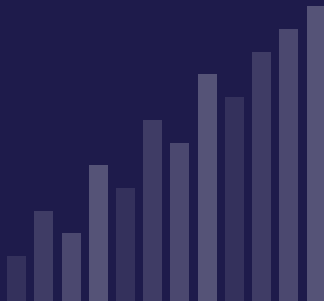


COMPLEMENTO 2 · BLOCO COMPLEMENTAR: ORDENAÇÃO AVANÇADA (OPCIONAL)

Quicksort aleatorizado e a análise do caso médio

Seções 7.3 e 7.4

Estruturas de dados e seus algoritmos · José Dihego · IFBA



Roteiro da aula

Ao final, você será capaz de

- ▶ Aleatorizar a escolha do pivô e entender o que isso garante
- ▶ Calcular o número esperado de comparações com variáveis indicadoras
- ▶ Distinguir caso médio (sobre entradas) de tempo esperado (sobre sorteios)
- ▶ Medir a concentração das comparações em torno da esperança



Teoria

Pivô sorteado, análise por pares comparados,
 $E[X] = O(n \lg n)$



Prática em C

Quicksort aleatorizado em C



Experimento

Entrada ordenada: pivô fixo contra pivô sorteado;
histograma



Conexões

Algoritmos aleatorizados em redes, segurança, IA e
sistemas distribuídos

Aleatoriedade contra o adversário

- ▶ Nenhuma entrada específica provoca o pior caso: ele depende só do **sorteio**.
- ▶ O **tempo esperado** vale para **toda** entrada, inclusive a ordenada.
- ▶ Caso médio (complemento 1) supunha entradas aleatórias; aqui a aleatoriedade está no algoritmo.

Randomized-PARTITION(A, p, r)

- 1 $i \leftarrow \text{RANDOM}(p, r)$
- 2 troque $A[r]$ com $A[i]$
- 3 **retorne** PARTITION(A, p, r)

Quem é comparado com quem?

Renomeie os elementos em ordem: $z_1 < z_2 < \dots < z_n$, e seja $Z_{ij} = \{z_i, \dots, z_j\}$.

- ▶ Dois elementos são comparados **no máximo uma vez**: só com o pivô, que depois sai do jogo.
- ▶ z_i e z_j são comparados **se e somente se** o primeiro pivô escolhido em Z_{ij} é z_i ou z_j . Se for um elemento do meio, eles vão para lados diferentes e nunca se encontram.
- ▶ Todos os $j - i + 1$ elementos de Z_{ij} têm a mesma chance de ser o primeiro pivô:

$$\Pr\{z_i \text{ comparado com } z_j\} = \frac{2}{j - i + 1}$$

Somando as probabilidades

Com X_{ij} indicadora de " z_i é comparado com z_j " e $X = \sum X_{ij}$:

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} < \sum_{i=1}^{n-1} \sum_{k=1}^n \frac{2}{k} = \sum_{i=1}^{n-1} O(\lg n) = O(n \lg n)$$

Fazendo a conta exata: $E[X] = 2(n+1)H_n - 4n \approx 1,39 n \lg n$.

Linearidade da esperança: a esperança da soma é a soma das esperanças, mesmo que os X_{ij} sejam dependentes.

Partições boas são a regra

Com pivô aleatório, a chance de a partição ser pior que 1 para 9 é só 20%. Partições ruins ocasionais são absorvidas pelas boas: a recursão continua com profundidade $O(\lg n)$.

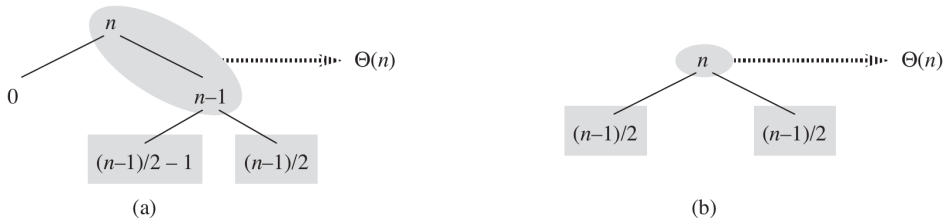


Figura 7.5 de Cormen et al. (2009) — (a) uma partição ruim seguida de uma boa custa o mesmo que (b) uma boa: o custo da ruim é absorvido.



Prática em C

- ▶ `rand()` sorteia o pivô em $[p, r]$
- ▶ `srand(time(NULL))`: sorteios diferentes a cada execução
- ▶ Entrada já ordenada deixa de ser problema

Quicksort aleatorizado em C (1/3)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4
5  void troca(int A[], int i, int j) { int t = A[i]; A[i] = A[j]; A[j] = t; }
6
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);
11     troca(A, i + 1, r);
12     return i + 1;
13 }
```

Quicksort aleatorizado em C (2/3)

```
15  /* RANDOMIZED-PARTITION: sorteia o pivô em A[p..r] e o leva para o fim. */
16  int particiona_aleatorio(int A[], int p, int r) {
17      int k = p + rand() % (r - p + 1);
18      troca(A, k, r);
19      return particiona(A, p, r);
20  }
21
22  void quicksort_aleatorio(int A[], int p, int r) {
23      if (p < r) {
24          int q = particiona_aleatorio(A, p, r);
25          quicksort_aleatorio(A, p, q - 1);
26          quicksort_aleatorio(A, q + 1, r);
27      }
28  }
```

Quicksort aleatorizado em C (3/3)

```
29
30 int main(void) {
31     srand((unsigned)time(NULL));           // cada execução sorteia pivôs diferentes
32     int n;
33     if (scanf("%d", &n) != 1 || n <= 0) return 1;
34     int *A = malloc(n * sizeof(int));
35     for (int k = 0; k < n; k++)
36         if (scanf("%d", &A[k]) != 1) return 1;
37     quicksort_aleatorio(A, 0, n - 1);
38     for (int k = 0; k < n; k++) printf("%d%c", A[k], k < n - 1 ? ' ' : '\n');
39     free(A);
40     return 0;
41 }
```



Experimento

- ▶ Entrada ordenada: pivô fixo contra sorteado
- ▶ Pivô sorteado: média de 20 execuções por n
- ▶ Histograma de 20.000 execuções com $n = 1000$

O pior caso desaparece

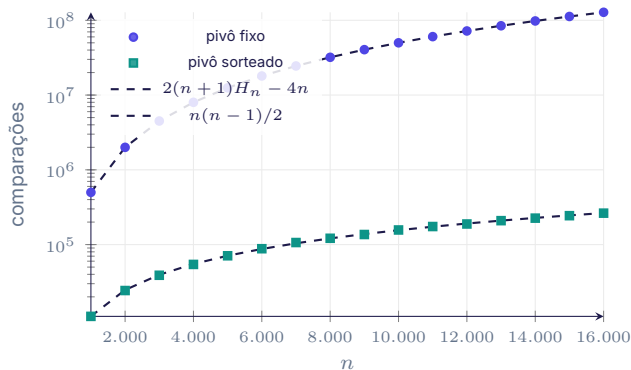
Hipótese. Na entrada já ordenada, o pivô fixo ($A[r]$) faz $n(n - 1)/2$ comparações. Com pivô sorteado, a esperança é $2(n + 1)H_n - 4n$ para **qualquer** entrada, inclusive esta. E as execuções devem se concentrar perto da esperança.

Método

1. Mesmo vetor ordenado $0, 1, \dots, n - 1$
2. n de 1.000 a 16.000
3. Histograma: 20.000 execuções, $n = 1000$, classes de 250
4. Semente fixa (`srand(12)`): reproduzível

🔪 EXPERIMENTO

Pivô fixo contra pivô sorteado (entrada ordenada)



<i>n</i>	medido	razão
1.000	10.960	0,998
4.000	54.288	0,987
7.000	106.516	1,024
10.000	156.841	1,007
13.000	208.975	0,998
16.000	263.305	0,996

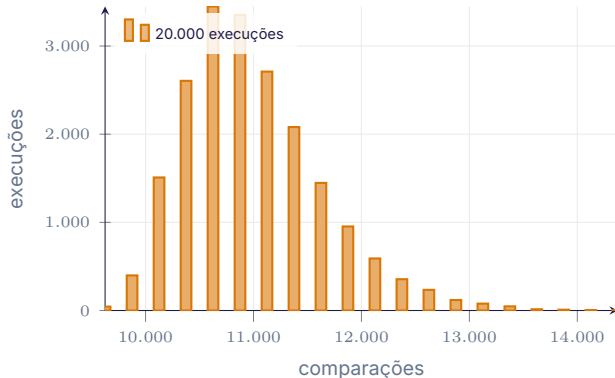
razão = medido $\div$ $2(n+1)H_n - 4n$

✓ desvio máx. 1,4%

Eixo y logarítmico: para $n = 16.000$, o pivô sorteado faz cerca de 500 vezes menos comparações.

🧪 EXPERIMENTO

Como as execuções se distribuem ($n = 1000$)



Média perto de

$$2 \cdot 1001 \cdot H_{1000} - 4000 \approx 10.986.$$

Nenhuma das 20.000 execuções
chegou perto do pior caso, 499.500.

🧪 EXPERIMENTO

O que o experimento mostrou

Na entrada que destrói o pivô fixo ($n(n-1)/2$ comparações), o pivô sorteado volta à esperança $2(n+1)H_n - 4n$. O histograma mostra execuções concentradas: a probabilidade de algo perto do pior caso é desprezível. **Aleatoriedade troca uma entrada ruim por uma sorte ruim, e sorte ruim é rara.**

➤ Reproduza: `gcc -O2 compl2_experimento.c -o exp -lm && ./exp`



Conexões

- ▶ Segurança
- ▶ Redes
- ▶ Aprendizado de máquina
- ▶ Sistemas distribuídos

Onde isso aparece na computação

Segurança

Tabelas hash de linguagens (Python, Rust) sorteiam a semente da função hash a cada execução pelo mesmo motivo: impedir que um atacante escolha a entrada do pior caso.

Redes

O Ethernet usa recuo exponencial aleatório após colisões; balanceadores de carga usam "a melhor de duas escolhas aleatórias para distribuir requisições.

Aprendizado de máquina

Gradiente estocástico, *random forests* e *dropout*: a aleatoriedade torna algoritmos mais rápidos e mais robustos, com garantias em esperança.

Sistemas distribuídos

Protocolos de fofoca (*gossip*) e eleição de líder no Raft usam tempos aleatórios para evitar empates e espalhar informação em $O(\lg n)$ rodadas.

Para casa

Exercícios

1. (7.3-1) Por que analisamos o tempo *esperado* e não o de pior caso do quicksort aleatorizado?
2. (7.3-2) Quantas chamadas a RANDOM ocorrem no pior e no melhor caso?
3. (7.4-2) Mostre que o melhor caso do quicksort é $\Omega(n \lg n)$.
4. (Problema 7-5) Implemente a partição pela mediana de 3 e meça a diferença.
5. No experimento, troque a semente e verifique que a média muda menos de 1%.

Leitura

Cormen et al., seções 7.3 e 7.4; apêndice C.3 (variáveis indicadoras).

Código

josedihego.net/estruturas