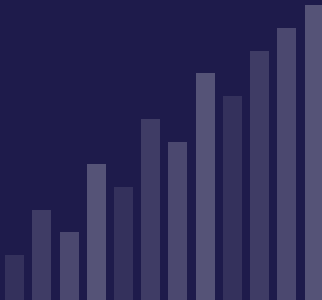

COMPLEMENTO 4 · BLOCO COMPLEMENTAR: ORDENAÇÃO AVANÇADA (OPCIONAL)

Ordenação em tempo linear: contagem e radix

Seções 8.2 e 8.3

Estruturas de dados e seus algoritmos · José Dihego · IFBA



Roteiro da aula

Ao final, você será capaz de

- ▶ Ordenar inteiros em $0..k$ por contagem em $\Theta(n + k)$
- ▶ Explicar por que a estabilidade é essencial para o radix sort
- ▶ Ordenar d dígitos em $\Theta(d(n + k))$ e escolher a base
- ▶ Medir tempo por elemento: constante no radix, $\lg n$ no qsort



Teoria

Counting sort, estabilidade, radix sort LSD e escolha da base



Prática em C

Contagem e radix em C



Experimento

Radix (base 256) contra qsort até 4 milhões de elementos



Conexões

GPUs, bancos de dados, bioinformática e processamento de imagens

Ordenação por contagem

- ▶ Nenhuma comparação entre chaves: o limite $\Omega(n \lg n)$ não se aplica.
- ▶ Custo $\Theta(n + k)$: linear quando $k = O(n)$.
- ▶ O último laço anda **de trás para frente**: iguais saem na ordem original. O algoritmo é **estável**.

COUNTING-SORT(A, B, k)

```

1 crie  $C[0..k]$  com zeros
2 para  $j \leftarrow 1$  até  $n$  faça
3    $C[A[j]] \leftarrow C[A[j]] + 1$ 
4                                     //  $C[i]$  = quantos iguais a  $i$ 
5 para  $i \leftarrow 1$  até  $k$  faça
6    $C[i] \leftarrow C[i] + C[i - 1]$ 
7                                     //  $C[i]$  = quantos  $\leq i$ 
8 para  $j \leftarrow n$  descendo até 1 faça
9    $B[C[A[j]]] \leftarrow A[j]$ 
10   $C[A[j]] \leftarrow C[A[j]] - 1$ 
  
```

Contagem passo a passo

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	2	0	2	3	0	1

(a)

	0	1	2	3	4	5
C	2	2	4	7	7	8

(b)

	1	2	3	4	5	6	7	8
B							3	

	0	1	2	3	4	5
C	2	2	4	6	7	8

(c)

	1	2	3	4	5	6	7	8
B		0					3	

	0	1	2	3	4	5
C	1	2	4	6	7	8

(d)

	1	2	3	4	5	6	7	8
B		0				3	3	

	0	1	2	3	4	5
C	1	2	4	5	7	8

(e)

	1	2	3	4	5	6	7	8
B	0	0	2	2	3	3	3	5

(f)

Figura 8.2 de Cormen et al. (2009) — (a) contagens; (b) somas acumuladas; (c)–(e) os três últimos elementos vão para B ; (f) resultado

Radix sort: dígito por dígito

RADIX-SORT(A, d): para $i = 1$ até d , ordene A pelo dígito i com uma ordenação **estável**.

Lema 8.3: d dígitos em base k : $\Theta(d(n + k))$.

Lema 8.4: chaves de b bits em grupos de r bits: $\Theta(\frac{b}{r}(n + 2^r))$. Com $r \approx \lg n$: $\Theta(bn / \lg n)$.

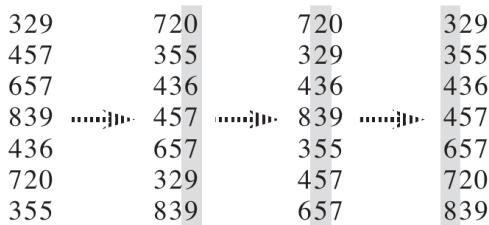


Figura 8.3 de Cormen et al. (2009) — do dígito menos significativo para o mais significativo.

Por que a estabilidade importa

- ▶ Depois de ordenar pelo dígito i , os empates nesse dígito devem **manter** a ordem dada pelos dígitos anteriores (menos significativos).
- ▶ Invariante: após a passada i , o vetor está ordenado pelos i dígitos menos significativos.
- ▶ Uma ordenação instável (quicksort, heapsort) quebraria o invariante.
- ▶ **Base na prática:** inteiros de 32 bits em base 256 ($r = 8$): 4 passadas de contagem com vetor C de 256 posições, que cabe no cache.

Radix sort é linear, mas não está sempre à frente: precisa de memória extra $\Theta(n)$, e cada passada percorre o vetor todo.



Prática em C

- ▶ Contagem para chaves pequenas; radix em base 10 para as maiores
- ▶ Laço final de trás para frente: estabilidade
- ▶ Testes com as entradas das Figuras 8.2 e 8.3

Contagem e radix sort em C (1/4)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  /* COUNTING-SORT: chaves em 0..k. C[v] conta; depois vira "quantos são <= v".
5   Percorrer A de trás para frente torna o algoritmo estável. */
6  void contagem(const int A[], int B[], int n, int k) {
7      int *C = calloc(k + 1, sizeof(int));
8      for (int j = 0; j < n; j++) C[A[j]]++;
9      for (int v = 1; v <= k; v++) C[v] += C[v - 1];
10     for (int j = n - 1; j >= 0; j--) B[--C[A[j]]] = A[j];
11     free(C);
12 }
```

Contagem e radix sort em C (2/4)

```
14  /* RADIX-SORT (LSD) em base 10: contagem estável dígito a dígito, do menos  
15  significativo para o mais significativo. */  
16  void radix(int A[], int n, int digitos) {  
17      int *B = malloc(n * sizeof(int));  
18      for (int d = 0, pot = 1; d < digitos; d++, pot *= 10) {  
19          int C[10] = {0};  
20          for (int j = 0; j < n; j++) C[A[j] / pot % 10]++;  
21          for (int v = 1; v < 10; v++) C[v] += C[v - 1];  
22          for (int j = n - 1; j >= 0; j--) B[--C[A[j] / pot % 10]] = A[j];  
23          for (int j = 0; j < n; j++) A[j] = B[j];  
24      }  
25      free(B);  
26  }
```

Contagem e radix sort em C (3/4)

```
27
28 int main(void) {
29     int n;
30     if (scanf("%d", &n) != 1 || n <= 0) return 1;
31     int *A = malloc(n * sizeof(int)), *B = malloc(n * sizeof(int)), max = 0;
32     for (int k = 0; k < n; k++) {
33         if (scanf("%d", &A[k]) != 1 || A[k] < 0) return 1;
34         if (A[k] > max) max = A[k];
35     }
36     if (max < 100) {                                     // chaves pequenas: contagem direta
37         contagem(A, B, n, max);
```

Contagem e radix sort em C (4/4)

```
38     printf("contagem:");
39 } else {                                     // chaves maiores: radix nos dígitos
40     int digitos = 0;
41     for (int m = max; m > 0; m /= 10) digitos++;
42     radix(A, n, digitos);
43     for (int k = 0; k < n; k++) B[k] = A[k];
44     printf("radix:");
45 }
46 for (int k = 0; k < n; k++) printf(" %d", B[k]);
47 printf("\n");
48 free(A); free(B);
49 return 0;
50 }
```



Experimento

- ▶ Radix LSD em base 256 (4 passadas)
- ▶ qsort da biblioteca C
- ▶ Tempo por elemento, $n = 2^{16}$ a 2^{22}

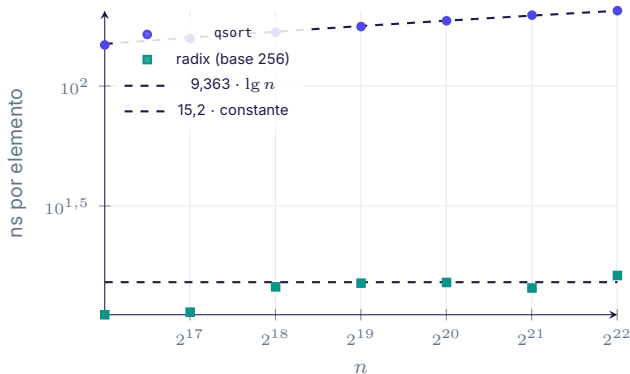
Linear contra $n \lg n$, em tempo real

Hipótese. Se $T(n) = cn$, então $T(n)/n = c$: o tempo **por elemento** do radix sort deve ser **constante**. Para o qsort, $T(n)/n = c' \lg n$: deve crescer devagar, como o logaritmo.

Método

1. Inteiros de 32 bits aleatórios
2. $n = 2^{16}, \dots, 2^{22}$ (até 4 milhões)
3. Tempo: melhor de 3, dividido por n (ns por elemento)
4. Mesmos dados para os dois algoritmos

Tempo por elemento



n	medido	razão
65.536	148,3	9,269
131.072	157,8	9,283
262.144	167,4	9,301
524.288	177,0	9,315
1.048.576	187,1	9,354
4.194.304	206,2	9,371

razão = medido $\div$ $\lg n$

✓ desvio máx. 0,5%

O radix fica em torno de 8 ns por elemento; o qsort passa de 100 ns, e cresce com $\lg n$.

EXPERIMENTO

O que o experimento mostrou

O tempo por elemento do radix sort é praticamente **constante** (linear no total) e mais de 10 vezes menor que o do qsort, cujo custo por elemento cresce como $\lg n$. O limite $\Omega(n \lg n)$ não se aplica porque o radix sort **não compara chaves**: usa os bits delas.

Parte da diferença é constante: qsort chama uma função de comparação por ponteiro a cada passo. Compare também com o seu quicksort da complemento 1.

➤ Reproduza: `gcc -O2 compl4_experimento.c -o exp -lm && ./exp`



Conexões

- ▶ GPUs
- ▶ Bancos de dados
- ▶ Bioinformática
- ▶ Processamento de imagens

Onde isso aparece na computação

 GPUs

Radix sort é o padrão para ordenar em GPU (CUB, Thrust): as contagens e somas de prefixo paralelizam muito bem.

 Bancos de dados

Ordenar por chaves inteiras (datas, códigos) com radix e agrupar por contagem (GROUP BY) aparecem em bancos colunares como DuckDB.

 Bioinformática

Arranjos de sufixos de genomas são construídos com radix sort sobre o alfabeto $\{A, C, G, T\}$: base 4, chaves curtas.

 Processamento de imagens

O histograma de uma imagem é a primeira etapa da contagem; a equalização de histograma usa exatamente as somas acumuladas de C .

Para casa

Exercícios

1. (8.2-1) Ilustre a contagem em $A = \langle 6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2 \rangle$.
2. (8.2-3) Se o último laço andar de 1 até n , o algoritmo ainda funciona? Ainda é estável?
3. (8.3-1) Ilustre o radix em COW, DOG, SEA, RUG, ROW, MOB, BOX, TAB, BAR, EAR, TAR, DIG, BIG, TEA, NOW, FOX.
4. (8.3-4) Ordene n inteiros em $0..n^3 - 1$ em tempo $O(n)$.
5. Varie a base do experimento (16, 256, 65.536) e explique o resultado pelo lema 8.4 e pelo cache.

Leitura

Cormen et al., seções 8.2 e 8.3.

Código

josedihego.net/estruturas