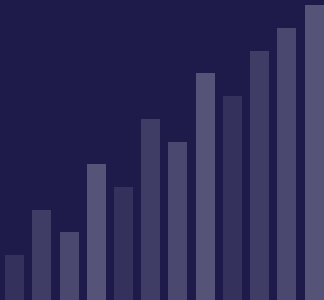

AULA 4 · BLOCO 1: FUNDAMENTOS, ORDENAÇÃO E CUSTO

Dividir e conquistar: merge sort

Seção 2.3

Estruturas de dados e seus algoritmos · José Dihego · IFBA



Roteiro da aula

Ao final, você será capaz de

- ▶ Aplicar o paradigma dividir, conquistar e combinar
- ▶ Executar Merge e provar sua corretude por invariante
- ▶ Escrever a recorrência do merge sort e resolvê-la pela árvore de recursão
- ▶ Medir $n \lg n$ e achar o ponto em que a inserção deixa de valer a pena



Teoria

Paradigma, Merge com sentinelas, recorrência e árvore de recursão



Prática em C

Merge sort em C com vetores auxiliares



Experimento

Comparações $\approx n \lg n - 1,26 n$ e o cruzamento com a inserção



Conexões

Big Data, bancos de dados, LSM-trees e paralelismo

Dividir, conquistar, combinar

- ▶ **Dividir** o problema em subproblemas menores do mesmo tipo.
- ▶ **Conquistar** os subproblemas recursivamente; se forem pequenos, resolver direto.
- ▶ **Combinar** as soluções dos subproblemas na solução do problema.

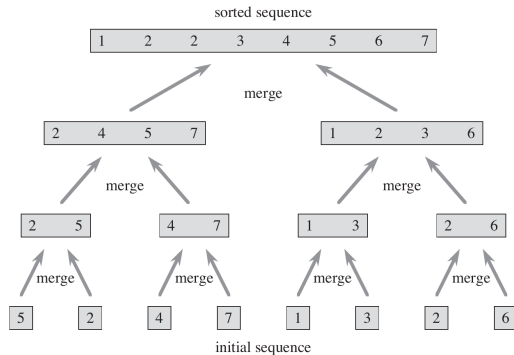


Figura 2.4 de Cormen et al. (2009) — merge sort de baixo para cima em $\langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$.

No merge sort: dividir o vetor ao meio ($\Theta(1)$), ordenar cada metade recursivamente, **intercalar** as duas metades ordenadas ($\Theta(n)$).

Intercalação com sentinelas

- ▶ Duas pilhas de cartas ordenadas, viradas para cima: a menor do topo vai para a saída.
- ▶ As sentinelas ∞ evitam testar se uma pilha acabou.
- ▶ Exatamente $n = r - p + 1$ passos: $\Theta(n)$.
- ▶ Invariante: $A[p..k - 1]$ tem os $k - p$ menores de L e R , em ordem.

Merge(A, p, q, r)

```

1  $n_1 \leftarrow q - p + 1$ ;  $n_2 \leftarrow r - q$ 
2 crie  $L[1..n_1 + 1]$  e  $R[1..n_2 + 1]$ 
3 para  $i \leftarrow 1$  até  $n_1$  faça
4    $L[i] \leftarrow A[p + i - 1]$ 
5 para  $j \leftarrow 1$  até  $n_2$  faça
6    $R[j] \leftarrow A[q + j]$ 
7  $L[n_1 + 1] \leftarrow \infty$ ;  $R[n_2 + 1] \leftarrow \infty$ 
8  $i \leftarrow 1$ ;  $j \leftarrow 1$ 
9 para  $k \leftarrow p$  até  $r$  faça
10   se  $L[i] \leq R[j]$  então
11      $A[k] \leftarrow L[i]$ ;  $i \leftarrow i + 1$ 
12   senão
13      $A[k] \leftarrow R[j]$ ;  $j \leftarrow j + 1$ 

```

Intercalação passo a passo

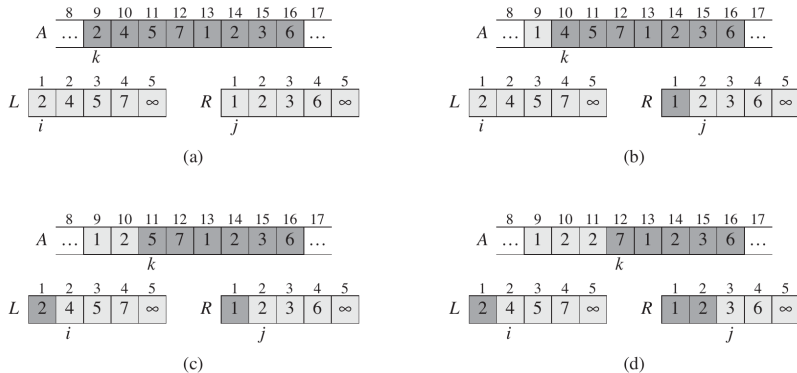


Figura 2.3 de Cormen et al. (2009) — (a)–(d) as primeiras iterações de Merge($A, 9, 12, 16$); em cinza claro, o que já foi copiado.

O algoritmo e sua recorrência

Chamada inicial: $\text{MERGE-SORT}(A, 1, n)$.

Para n potência de 2:

$$T(n) = \begin{cases} c & n = 1 \\ 2T(n/2) + cn & n > 1 \end{cases}$$

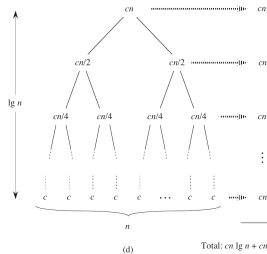
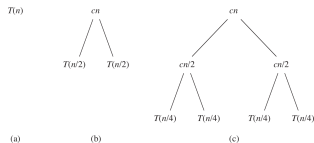
Dividir custa $\Theta(1)$, conquistar custa $2T(n/2)$, combinar custa $\Theta(n)$.

Merge-Sort(A, p, r)

```
1 se  $p < r$  então  
2    $q \leftarrow \lfloor (p + r)/2 \rfloor$   
3   Merge-Sort( $A, p, q$ )  
4   Merge-Sort( $A, q + 1, r$ )  
5   Merge( $A, p, q, r$ )
```

Árvore de recursão: $T(n) = cn \lg n + cn$

Cada nível da árvore custa cn no total; com $\lg n + 1$ níveis, $T(n) = cn \lg n + cn = \Theta(n \lg n)$.





Prática em C

- ▶ intercala sem sentinelas: testa o fim de L e R
- ▶ $\leq$ na intercalação mantém o algoritmo **estável**
- ▶ Teste com a entrada da Figura 2.4

Sentinelas ∞ em C?

- ▶ O livro usa ∞ no fim de L e R . Em C, INT_MAX funciona só se nenhum dado for igual a INT_MAX.
- ▶ Solução robusta: testar o fim de cada lado.
- ▶ `if (j >= n2 || (i < n1 && L[i] <= R[j]))`: se R acabou, ou se $L[i]$ é o menor, copia de L .

Memória: o merge sort usa $\Theta(n)$ de memória extra; não é *in place*. No experimento usamos um único vetor auxiliar para não chamar malloc a cada intercalação.

Merge sort em C (1/3)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  /* MERGE(A, p, q, r): intercala A[p..q] e A[q+1..r], ambos ordenados.
5     No livro, sentinelas  $\infty$  marcam o fim de L e R; aqui testamos os limites. */
6  void intercala(int A[], int p, int q, int r) {
7      int n1 = q - p + 1, n2 = r - q;
8      int *L = malloc(n1 * sizeof(int)), *R = malloc(n2 * sizeof(int));
9      for (int i = 0; i < n1; i++) L[i] = A[p + i];
10     for (int j = 0; j < n2; j++) R[j] = A[q + 1 + j];
11     int i = 0, j = 0;
12     for (int k = p; k <= r; k++) {
13         if (j >= n2 || (i < n1 && L[i] <= R[j])) // <= mantém a ordem dos iguais
14             A[k] = L[i++];
```

Merge sort em C (2/3)

```
15  ∞
16      else
17          A[k] = R[j++];
18      }
19      free(L);
20      free(R);
21  }
22
23  void merge_sort(int A[], int p, int r) {
24      if (p < r) {
25          int q = (p + r) / 2;           // dividir
26          merge_sort(A, p, q);           // conquistar a metade esquerda
27          merge_sort(A, q + 1, r);       // conquistar a metade direita
28          intercala(A, p, q, r);         // combinar
29      }
30  }
```

Merge sort em C (3/3)

```
30  ∞
31
32  int main(void) {
33      int n;
34      if (scanf("%d", &n) != 1 || n <= 0) return 1;
35      int *A = malloc(n * sizeof(int));
36      for (int k = 0; k < n; k++)
37          if (scanf("%d", &A[k]) != 1) return 1;
38      merge_sort(A, 0, n - 1);
39      for (int k = 0; k < n; k++) printf("%d%c", A[k], k < n - 1 ? ' ' : '\n');
40      free(A);
41      return 0;
42  }
```



Experimento

- ▶ Hipótese: comparações $\approx n \lg n - 1,26 n$
- ▶ n de 2^{12} a 2^{20} , média de 5 vetores
- ▶ Tempo de inserção e merge sort para $n \leq 500$

EXPERIMENTO

$n \lg n$ na prática e o ponto de cruzamento

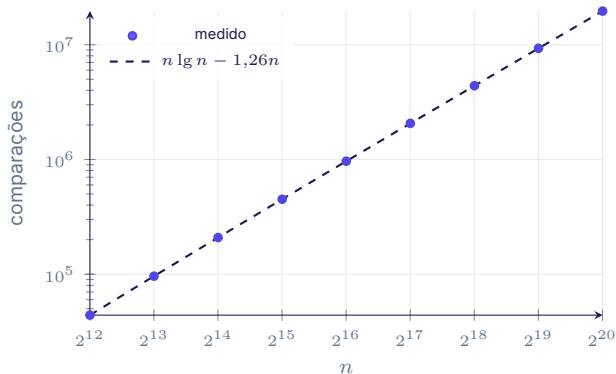
Hipótese. Cada intercalação de m elementos faz entre $m/2$ e $m - 1$ comparações; em média, um pouco menos que m . Somando os $\lg n$ níveis, o merge sort faz cerca de $n \lg n - 1,26 n$ comparações em vetores aleatórios. Para n pequeno, porém, a inserção ($\approx n^2/4$, sem recursão nem cópias) deve ser **mais rápida**.

Método

1. Contador de comparações entre chaves na intercalação
2. $n = 2^{12}, 2^{13}, \dots, 2^{20}$; média de 5 vetores aleatórios
3. Tempo: 2.000 ordenações por ponto, melhor de 5 lotes
4. n de 10 a 500 para localizar o cruzamento

🧪 EXPERIMENTO

Comparações do merge sort



n	medido	razão
4.096	43.981	1,000
8.192	96.151	1,000
16.384	208.670	1,000
32.768	450.119	1,000
65.536	965.759	1,000
1.048.576	19.645.257	1,000

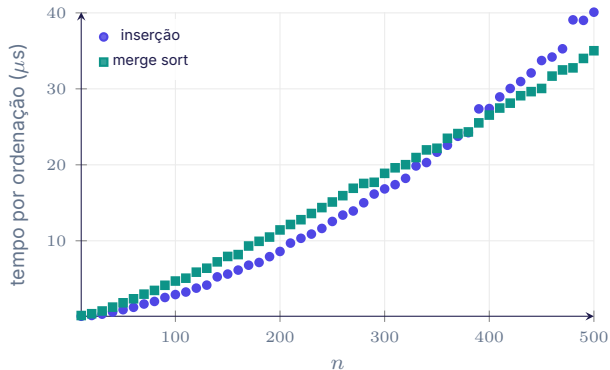
razão = medido $\div$ $n \lg n - 1,26n$

✓ desvio máx. 0,0%

Desvio abaixo de 1% em toda a faixa: a constante 1,26 vem da análise de caso médio da intercalação.

🧪 EXPERIMENTO

Para n pequeno, a inserção vence



As curvas se cruzam perto de $n \approx 400$ nesta máquina. Abaixo disso, a constante pequena da inserção compensa o n^2 .

EXPERIMENTO

O que o experimento mostrou

O merge sort faz $n \lg n - 1,26 n$ comparações, com desvio menor que 1%. Mas a ordem de crescimento não diz tudo: para n até algumas centenas, a inserção é mais rápida.

Algoritmos híbridos (Timsort, introsort) usam inserção abaixo de um limiar e dividir e conquistar acima dele.

Desafio (problema 2-1): use inserção para subvetores com até k elementos e encontre o melhor k na sua máquina.

➤ Reproduza: `gcc -O2 aula04_experimento.c -o exp -lm && ./exp`



Conexões

- ▶ Bancos de dados
- ▶ Armazenamento (LSM-trees)
- ▶ Big Data
- ▶ Computação paralela

Onde isso aparece na computação

Bancos de dados

O *sort-merge join* ordena duas tabelas pela chave e as intercala. A ordenação externa ordena arquivos maiores que a RAM em blocos e depois os intercala.

Armazenamento (LSM-trees)

RocksDB, Cassandra e LevelDB escrevem em blocos ordenados e os *compactam* por intercalação: o Merge em escala de terabytes.

Big Data

MapReduce e Spark dividem os dados entre máquinas, processam em paralelo e combinam os resultados: o mesmo paradigma, num cluster.

Computação paralela

As duas metades são independentes: o merge sort paraleliza naturalmente em CPUs com vários núcleos e em GPUs.

Para casa

Exercícios

1. (2.3-1) Ilustre o merge sort em $A = \langle 3, 41, 52, 26, 38, 57, 9, 49 \rangle$.
2. (2.3-2) Reescreva MERGE sem sentinelas.
3. (2.3-5) Escreva a busca binária e mostre que seu pior caso é $\Theta(\lg n)$.
4. (Problema 2-1) Implemente o híbrido inserção + merge e meça o melhor limiar k .
5. Implemente o merge sort de baixo para cima (iterativo), como na Figura 2.4.

Leitura

Cormen et al., seção 2.3 e problema 2-1.

Código

josedihego.net/estruturas