

1

Técnicas de Concepção de Algoritmos (1ª parte): divisão e conquista

R. Rossetti, L. Ferreira, H. L. Cardoso, F. Andrade
CAL, MIEIC, FEUP

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

2

Divisão e Conquista (*divide and conquer*)

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

3

Divisão e conquista

- ◆ Divisão: resolver recursivamente problemas mais pequenos (até caso base)
- ◆ Conquista: solução do problema original é formada com as soluções dos subproblemas
- ◆ Há divisão quando o algoritmo tem pelo menos 2 chamadas recursivas no corpo
- ◆ Subproblemas devem ser disjuntos
 - Senão, resolver de forma bottom-up com programação dinâmica
- ◆ Divisão em subproblemas de dimensão semelhante é importante para se obter uma boa eficiência temporal
- ◆ Algoritmos adequados para processamento paralelo

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

4

Divisão e conquista

- ◆ Dada uma instância do problema x , a técnica Divisão-e-Conquista funciona da seguinte maneira:

```

function DAQ(  $x$  )
  if  $x$  é suficientemente pequeno then
    resolver  $x$  directamente
  else
    dividir  $x$  em subinstâncias:  $x_1, \dots, x_k$ 
    for  $i := 1$  to  $k$  do  $y_i := \text{DAQ}( x_i )$ 
     $y := \sum y_i$ 
  return  $y$ 

```

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

5

Exemplo: cálculo de x^n

- ◆ Resolução iterativa com n multiplicações: $T(n) = O(n)$
- ◆ Resolução mais eficiente, com divisão e conquista:

$$x^n = \begin{cases} 1, & \text{se } n=0 \\ x, & \text{se } n=1 \\ x^{\frac{n}{2}} \times x^{\frac{n}{2}}, & \text{se } n \text{ par} > 1 \\ x \times x^{\frac{n-1}{2}} \times x^{\frac{n-1}{2}}, & \text{se } n \text{ ímpar} > 1 \end{cases}$$

```
double power(double x, int n) {
    if (n == 0) return 1;
    if (n == 1) return x;
    double p = power(x, n / 2);
    if (n % 2 == 0) return p * p;
    else return x * p * p;
}
```

- ◆ Divisão em subproblemas iguais, junção em tempo $O(1)$
- ◆ Nº de multiplicações reduzido para $\sim \log_2 n$
- ◆ $T(n) = O(\log n)$ mas $S(n) = O(\log n)$ (espaço)
- ◆ Nota: classificação como divisão e conquista não é consensual, por os 2 subproblemas serem idênticos (logo só há 1 chamada recursiva)

Técnicas de Conceção de Algoritmos, CAL - MIEIC/FEUP

6

Exemplo: ordenação de *arrays*

- ◆ Mergesort
 - Ordenar 2 subsequências de igual dimensão e juntá-las
 - $T(n) = O(n \log n)$, tanto no pior caso como no caso médio
- ◆ Quicksort
 - Ordenar elementos menores e maiores que *pivot*, concatenar
 - $T(n) = O(n^2)$ no pior caso (1 elemento menor, restantes maiores)
 - $T(n) = O(n \log n)$ no melhor caso e no caso médio (*)
 - (*) com escolha aleatória do pivot!

Técnicas de Conceção de Algoritmos, CAL - MIEIC/FEUP

7

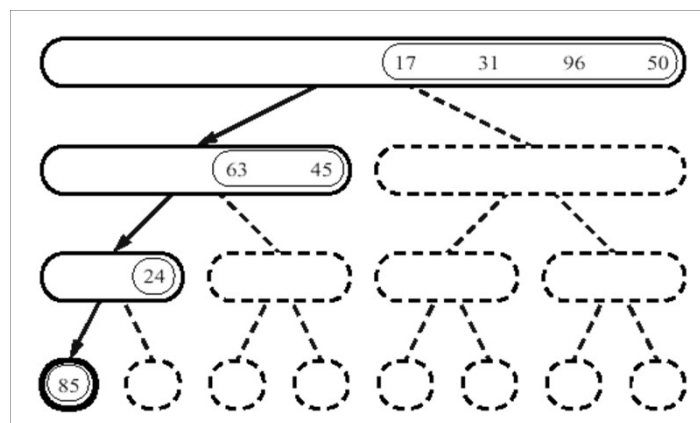
Exemplo: *Mergesort*

- ◆ Seja $S = \{s_1, \dots, s_n\}$ um conjunto que se pretenda ordenar.
- ◆ Caso base: Se $S = \{\}$ ou $S = \{s_1\}$, então nada é necessário!
- ◆ Dividir a sequência S em duas subsequências S_1 e S_2 , cada uma com $\sim n/2$ elementos
- ◆ Conquistar S_1 e S_2 , ordenando-as com mergesort (isto é, aplicando recursivamente o mesmo procedimento)
- ◆ Combinar as sequências ordenadas S_1 e S_2 numa sequência ordenada única S
- ◆ Fazer o mais possível *in-place*.

Técnicas de Conceção de Algoritmos, CAL - MIEIC/FEUP

11

Exemplo: *Mergesort*



Técnicas de Conceção de Algoritmos, CAL - MIEIC/FEUP

30

Mergesort: Pseudo-código (1/2)

```
// Sorts array A between indices p and r.
Merge-Sort(A, p, r)
  if p < r then
    q ← ⌊(p + r)/2⌋
    Merge-Sort(A, p, q)
    Merge-Sort(A, q + 1, r) } //possibly in parallel
  Merge(A, p, q, r)
```

```
Merge(A, p, q, r)
  //Take the smallest of the two topmost elements of
  //sequences A[p..q] and A[q+1..r] and put into the
  //resulting sequence. Repeat this, until both sequences
  //are empty. Copy the resulting sequence into A[p..r].
  //See next slide...
```

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

31

Mergesort: Pseudo-código (2/2)

```
//Merges sorted subarrays A[p..q] and A[q+1..r]
//into a single sorted subarray A[p..r].
Merge(A, p, q, r)
  // Copy the subarrays into auxiliary
  // memory with a sentinel
  L ← (A[p], ..., A[q], ∞), R ← (A[q+1], ..., A[r], ∞)

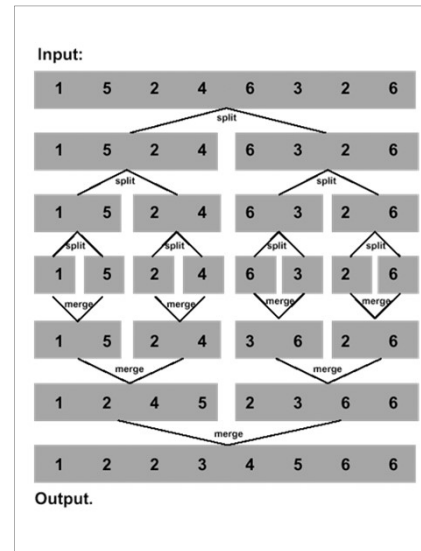
  // Repeatedly take the smallest leftmost
  // element of L and R
  i ← 1, j ← 1
  for k = p to r do
    if L[i] ≤ R[j] then A[k] ← L[i], i ← i+1
    else A[k] ← R[j], j ← j+1
```

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

33

Eficiência temporal

- ◆ Profundidade de recursão (nº de níveis) é $\lceil \log_2 n \rceil$
- ◆ Em cada nível, as várias operações de *split* podem ser efetuadas em tempo total $O(n)$
- ◆ Em cada nível, as várias operações de *merge* podem ser efetuada em tempo total $\Theta(n)$
- ◆ Logo, tempo total (em qq caso) é $T(n) = \Theta(n \log n)$



Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

37

Optimizações

Optimizações e ganhos experimentais (*speedup*) conseguidos a ordenar *arrays* aleatórios de tamanho $n=10^7$

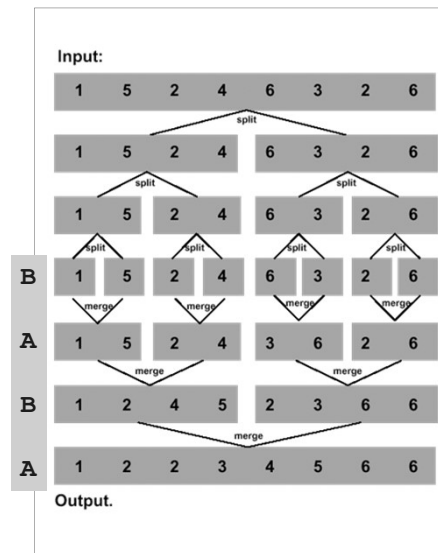
- ◆ Opção do linker: `-Wl,--stack,0xFFFFFFFF` (para array caber na stack)
- ◆ Opção do compilador: `-O3` (*optimize most*)

	Tempo (ms)	Ganho (<i>speedup</i>)
Merge sort, abordagem base (slides anteriores)	1330	-
Optimização da memória auxiliar	1226	x 1,08
Ordenação por inserção de arrays com $n < 20$	1078	x 1,14
Percorrer <i>arrays</i> com apontadores em vez de índices, usar <code>register</code> , usar <code>memcpy</code>	977	x 1,10
Processamento paralelo (4 <i>cores</i> , 8 <i>threads</i>)	398	x 2,45
Ganho total		x 3,34
<code>std::sort</code> (quick sort)	769	

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

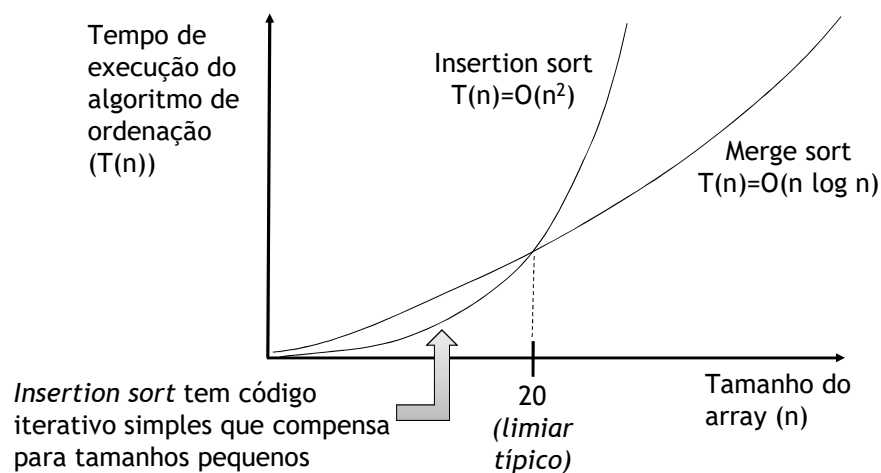
Optimização da memória auxiliar

- ◆ Em vez de fazer cópias para memória auxiliar em cada **Merge** ...
- ◆ Cria-se inicialmente uma cópia (B) de A
- ◆ As operações de **Merge** vão alternadamente colocando os resultados em A e B
- ◆ O tempo gasto nestas cópias é reduzido de $\Theta(n \log)$ para $\Theta(n)$



Técnicas de Conceção de Algoritmos, CAL - MIEIC/FEUP

Ordenação por inserção de arrays com $n < 20$



Técnicas de Conceção de Algoritmos, CAL - MIEIC/FEUP

40

Processamento paralelo

- ◆ Com k processadores ou núcleos (*cores*), executando as chamadas recursivas em paralelo, pode-se ter um ganho de desempenho de até k vezes
 - Em C++, número de núcleos é dado por

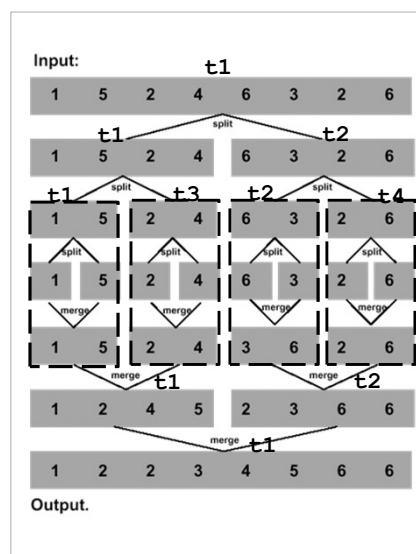

```
std::thread::hardware_concurrency()
```
- ◆ Execução paralela é conseguida usando múltiplos *threads* (pois estes executam em paralelo, partilhando o mesmo espaço de endereçamento)
 - Normalmente, desempenho ótimo com $n^\circ \text{ threads} = n^\circ \text{ cores}$
 - Em processadores com *hyper-threading*, n° ótimo é $2 \times n^\circ \text{ cores}$ (<https://en.wikipedia.org/wiki/Hyper-threading>)

Técnicas de Conceção de Algoritmos, CAL - MIEIC/FEUP

41

Ilustração

- ◆ Exemplo com 4 *cores*
- ◆ Divisão de trabalho por 4 *threads* concorrentes



Técnicas de Conceção de Algoritmos, CAL - MIEIC/FEUP

42

Implementação em C++

```
#include <thread>
template <typename T>
void MergeSort(T A[], int p, int r, int threads){
    if (p < r) {
        int q = (p + r) / 2;
        if (threads > 1) {
            std::thread t([=]() {MergeSort(A,p,q, threads/2);});
            MergeSort(A, q+1, r, threads / 2);
            t.join();
        }
        else {
            MergeSort(A, p, q, 1); MergeSort(A, q + 1, r, 1)
        }
        Merge(A, p, q, r);
    }
}
```

Técnicas de Conceção de Algoritmos, CAL - MIEIC/FEUP

43

Nota sobre funções lambda em C++

Neste caso o argumento do construtor do thread *t* é uma função lambda (função definida *on the fly*).

```
std::thread t ( [=] () {MergeSort(...);} );
```

“=” significa que o corpo da função pode usar por cópia todas as variáveis locais da função em que se insere

corpo da função definida

sem argumentos neste caso

Técnicas de Conceção de Algoritmos, CAL - MIEIC/FEUP

45


Exemplo 3: pesquisa binária

- ◆ Seja $S=(s_1, \dots, s_n)$ uma sequência ordenada de n elementos, e x um elemento que se pretende procurar em S .
- ◆ Casos bases:
 - Se $S=()$, falha!
 - Se $x=s_m$, c/ $m=\lfloor (1+n)/2 \rfloor$ (elem. médio), retorna-se a posição!
- ◆ Dividir S em duas subsequências, $L=(s_1, \dots, s_{m-1})$ e $R=(s_{m+1}, \dots, s_n)$, à esquerda e à direita do elemento médio.
- ◆ Conquistar: se $x < s_m$ ($x > s_m$) continua-se a pesquisa em L (R , resp.)
- ◆ $T(n) = O(\log n)$
- ◆ Nota: classificação como divisão e conquista não é consensual, por um dos 2 subproblemas ser vazio (logo basta 1 chamada recursiva).

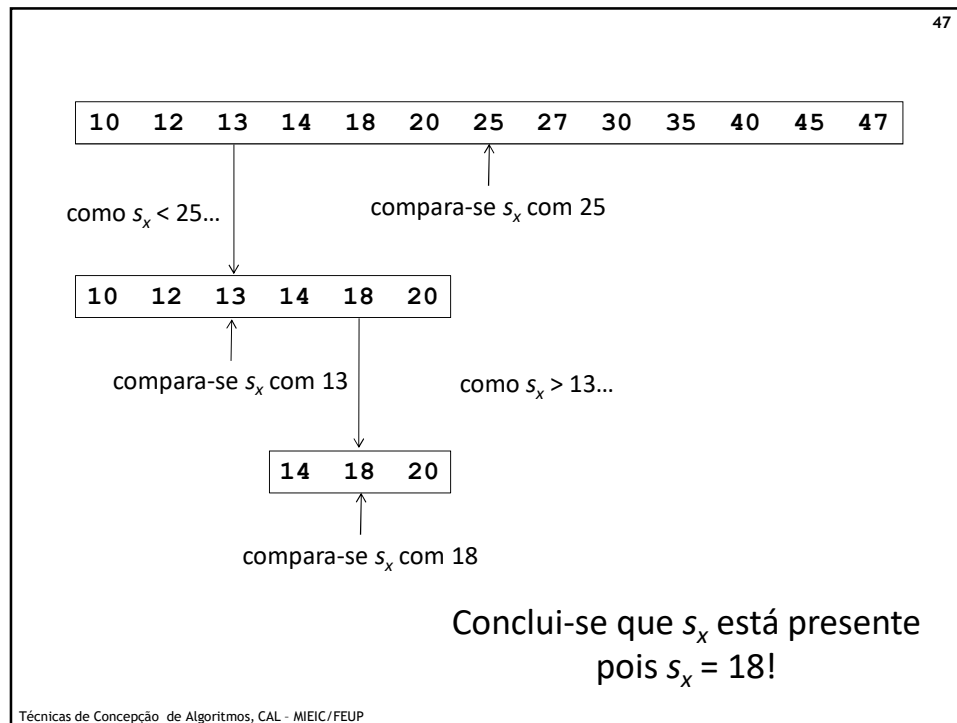
Técnicas de Conceção de Algoritmos, CAL - MIEIC/FEUP

46

Exemplo: pesquisa binária

- ◆ Suponha que $s_x = 18$ e que S é dado por:
 $\{10 \ 12 \ 13 \ 14 \ 18 \ 20 \ 25 \ 27 \ 30 \ 35 \ 40 \ 45 \ 47\}$

termo médio
- ◆ Dividir a sequência: sendo $s_x < 25$, pesquisa-se em $\{10 \ 12 \ 13 \ 14 \ 18 \ 20\}$
- ◆ Conquistar a subsequência, determinando-se se s_x está presente.
- ◆ Obtém-se a solução para a sequência S , pela solução da pesquisa nas subsequências. R: Sim! $s_x \in S$

Técnicas de Conceção de Algoritmos, CAL - MIEIC/FEUP



49

Referências

- ◆ T.H. Cormen, C. E. Leiserson, R. L. Rivest , C. Stein. Introduction to Algorithms, 3rd Edition. MIT Press, 2009
 - Capítulo 4 - Divide-and-Conquer
 - Secção 27.3 - Multithreaded Merge Sort
- ◆ Mark Allen Weiss. Data Structures & Algorithm Analysis in Java. Addison-Wesley, 1999
- ◆ Steven S. Skiena. The Algorithm Design Manual. Springer 1998
- ◆ Robert Sedgewick. Algorithms in C++. Addison-Wesley, 1992

Técnicas de Conceção de Algoritmos, CAL - MIEIC/FEUP

50

Em suma...

- ◆ Programação dinâmica (*dynamic programming*)
 - Contexto: Problemas de solução recursiva.
 - Objectivo: Minimizar tempo e espaço.
 - Forma: Induzir uma progressão iterativa de transformações sucessivas de um espaço linear de soluções.
- ◆ Algoritmos gananciosos (*greedy algorithms*)
 - Contexto: Problemas de optimização (max. ou min.)
 - Objectivo: Atingir a solução óptima, ou uma boa aproximação.
 - Forma: tomar uma decisão óptima localmente, i.e., que maximiza o ganho (ou minimiza o custo) imediato

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

51

Em suma...

- ◆ Algoritmos de retrocesso (*backtracking*)
 - Contexto: problemas sem algoritmos eficientes (convergentes) para chegar à solução.
 - Objectivo: Convergir para uma solução.
 - Forma: tentativa-erro. Gerar estados possíveis e verificar todos até encontrar solução, retrocedendo sempre que se chegar a um “beco sem saída”.
- ◆ Divisão e conquista (*divide and conquer*)
 - Contexto: Problemas passíveis de se conseguirem sub-dividir.
 - Objectivo: melhorar eficiencia temporal.
 - Forma: agregação linear da resolução de sub-problemas de dimensão semelhantes até chegar ao caso-base.

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP