

Análise da Complexidade de Algoritmos Recursivos II

11/10/2023

Sumário

- Recap
- Generalização: algoritmos exponenciais
- Procura binária – versão recursiva
- The Master Theorem
- The Smoothness Rule
- Merge-Sort
- Sugestões de leitura

Let's
RECAP

Recapitulação

Decrease-And-Conquer

- Explorar a relação entre
 - A solução de uma dada **instância de um problema**
 - A solução de uma **instância menor do mesmo problema**
- Estratégia
 - Identificar **UMA** instância menor do mesmo problema
 - A menor instância é resolvida **recursivamente**
 - As soluções de instâncias mais pequenas são processadas para se obter a solução da instância original, **se necessário**

Decrease-And-Conquer

- Como **decrece** o tamanho de cada instância ?
- Divisão por um **factor constante**
 - $n ; n / 2 ; n / 4 ; \dots$
 - $n ; n / 3 ; n / 9 ; \dots$
- Subtração de um valor **constante**
 - $n ; n - 1 ; n - 2 ; \dots$
- Decréscimo **variável**
 - O padrão de decréscimo varia de iteração para iteração

D&C – Divisão por um factor constante

- Redução do tamanho da instância através da divisão por um factor constante em cada passo
 - Habitualmente, **dividir por 2** !

$$T(1) = c$$

$$T(n) = T(n / b) + f(n)$$

- Exemplos ?

D&C – Subtração de um valor constante

- Redução do tamanho da instância por subtração de um valor constante em cada passo
 - Habitualmente, **subtrair uma unidade** !

$$T(1) = c$$

$$T(n) = T(n - 1) + f(n)$$

- Exemplos ?

Divide-And-Conquer

- A estratégia algorítmica mais conhecida
- Estratégia
 - Subdividir uma instância de um problema em (**duas** ou **mais**) instâncias semelhantes e de menor dimensão
 - As instâncias mais pequenas são resolvidas **recursivamente**
 - As soluções de instâncias mais pequenas são combinadas para se obter a solução da instância original, **se necessário**

Divide-And-Conquer

- Em cada passo de subdivisão, as instâncias de menor dimensão deverão ter **aprox. o mesmo tamanho** !
- **Todas** as instâncias de menor dimensão têm de ser resolvidas **!!**
- Quando **termina** o processo de subdivisão ?
 - Casos de base ? Um só ou mais do que um ?
 - Instâncias mais pequenas podem ser resolvidas por outro algoritmo

Divide-And-Conquer

- Esta estratégia recursiva pode ser implementada
 - Usando funções recursivas (solução óbvia !)
 - Iterativamente, usando uma **estrutura de dados auxiliar**
 - STACK, QUEUE, etc.
 - **Escolher** que instância resolver de seguida !!
- Problemas ?
 - A recursividade é **lenta** !
 - Resolver instâncias mais pequenas usando outros algoritmos
 - Pode não ser a melhor estratégia para problemas simples !
 - Instâncias de menor dimensão podem **sobrepôr-se** !
 - Reutilizar resultados / soluções anteriores – **Prog. Dinâmica** !

Divide-And-Conquer

- Fizem este **exemplo** da nossa aula anterior ?
- Calcular **b^n** usando $b^n = b^{n \text{ div } 2} \times b^{(n+1) \text{ div } 2}$
- Número de **multiplicações** ?

$$M(n) = M(n \text{ div } 2) + M((n+1) \text{ div } 2) + 1$$

Divide-And-Conquer – Caso particular

- Se n for uma potência de 2 : $n = 2^k$, $k = \log_2 n$

$$M(n) = M(n / 2) + M(n / 2) + 1 = 2 M(n / 2) + 1 = \dots$$

- Expressão final ? Ordem de Complexidade ?
- É melhor do que o algoritmo direto ?

Tarefa 1 – Procura num array

- Dado um array de n elementos inteiros
- Encontrar o **maior valor**
- Divide-And-Conquer
 - Obter o **maior valor** do **1º sub-array** : $((n+1) \text{ div } 2)$ elementos
 - Obter o **maior valor** do **2º sub-array** : $(n \text{ div } 2)$ elementos
 - **Comparar** os dois valores e devolver o maior
- Quantas **comparações** ?

Tarefa 2 – Decrease-And-Conquer

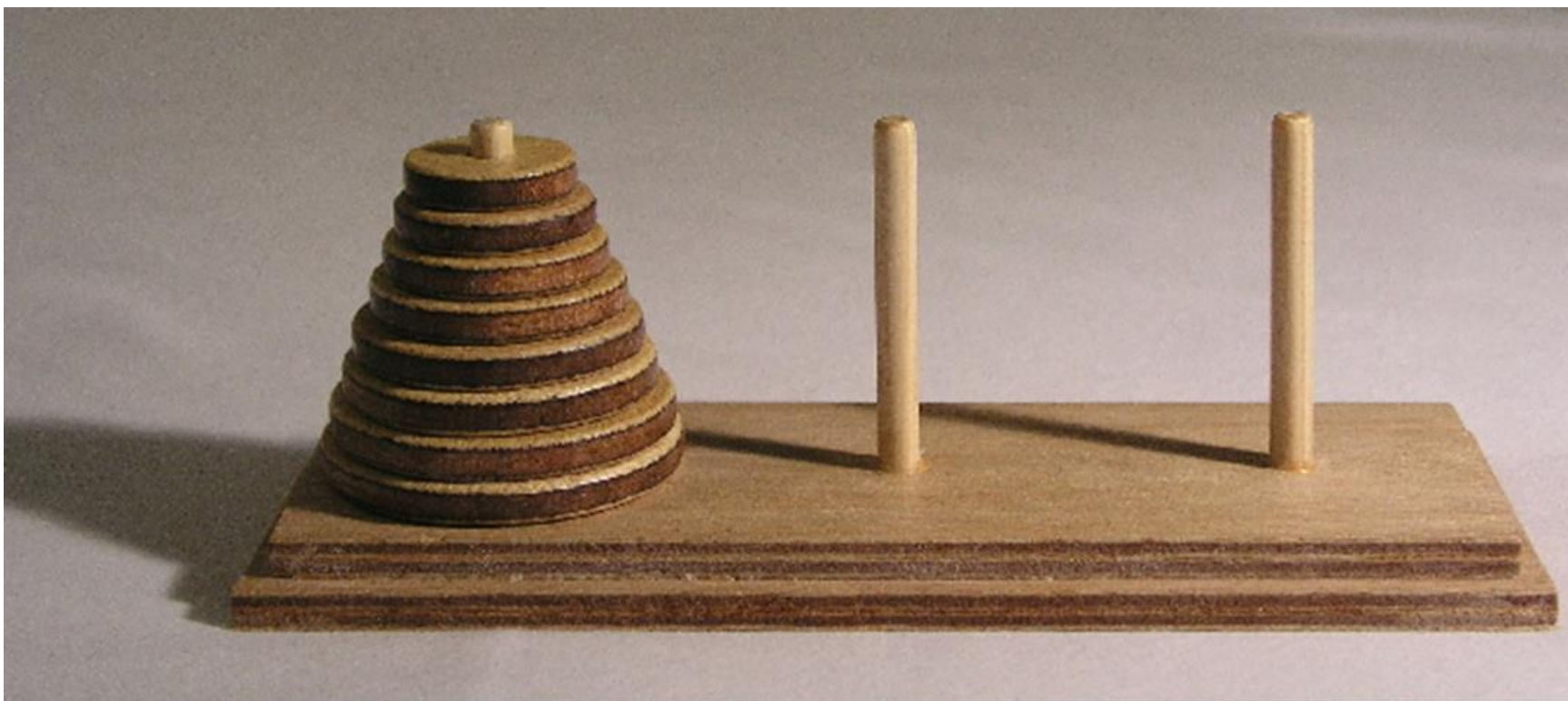
- Desenvolva uma função para calcular b^n usando

$$b^n = b^{n \text{ div } 2} \times b^{n \text{ div } 2}, \text{ se } n \text{ é par}$$
$$b^n = b \times b^{(n-1) \text{ div } 2} \times b^{(n-1) \text{ div } 2}, \text{ se } n \text{ é impar}$$

- Fazer **uma só chamada recursiva** em cada passo !!
- Quais são os **casos de base** ?
- Quantas **multiplicações** são efetuadas ?
- Qual é a **ordem de complexidade** ?

Generalização: Algoritmos Exponenciais

As Torres de Hanói



[Wikipedia]

Função recursiva

```
// torresDeHanoi('A', 'B', 'C', 8);

void torresDeHanoi(char origem, char auxiliar, char destino, int n) {
    if (n == 1) {
        contadorGlobalMovs++;
        moverDisco(origem, destino); // Imprime o movimento
        return;
    }
    // Divide-and-Conquer
    torresDeHanoi(origem, destino, auxiliar, n - 1);

    contadorGlobalMovs++;
    moverDisco(origem, destino);

    torresDeHanoi(auxiliar, origem, destino, n - 1);
}
```

Nº de movimentos realizados

$$M(1) = 1$$

$$M(n) = M(n-1) + 1 + M(n-1) = 1 + 2 M(n-1)$$

1 chamada de recorrência e o problema diminui em 1

$$M(n) = 1 + 2 M(n-1) = 1 + 2 \times (1 + 2 M(n-2)) = 1 + 2 + 4 M(n-2) = \dots$$

$$M(n) = 2^0 + 2^1 + 2^2 + \dots + 2^{k-1} + 2^k M(n-k)$$

Caso de base: $M(1) = 1$; $k = n - 1$

$$M(n) = 2^0 + 2^1 + 2^2 + \dots + 2^{n-2} + 2^{n-1} = 2^n - 1$$

$$M(n) \in \mathcal{O}(2^n)$$

Padrão de comportamento

$$T(1) = b$$

$$T(n) = a \times T(n - c) + d$$

- a : nº de **subproblemas** a resolver em cada passo
- b : nº de operações / tempo para o caso de base
- c : diminuição do tamanho do problema
- d : nº de operações / tempo de processamento de cada passo

Decrease-and-Conquer

Qual algoritmo que obedeça este padrão é linear

$$T(1) = b$$

$$T(n) = a \times T(n - c) + d$$

- $a = 1$

$$T(n) = b + d \times (n - 1) / c$$

$$T(n) \in \mathcal{O}(n)$$

- Aplica-se a algum exemplo anterior ?
- Sugestão: fazer o desenvolvimento

Divide-and-Conquer

$$T(1) = b$$

$$T(n) = a \times T(n - c) + d \quad \leftarrow \text{O que interessa é que, se nesta fórmula o } a > 1, \text{ ent é exponencial}$$

- $a > 1$

Exponencial

$$T(n) = d/(1 - a) + (b - d/(1 - a)) \times a^{(n-1)/c} \quad \leftarrow \text{Isto não interessa tanto}$$

$$T(n) \in \mathcal{O}(a^{\frac{n}{c}})$$

Nas torres de Hanói, o a é 2, pois as torres de Hanói é $2^{(n)-1}$

- Aplica-se às Torres de Hanói ? Verificar !
- Sugestão: fazer o desenvolvimento

Procura Binária

– Versão recursiva

Procura Binária

- Dado um array **ordenado** com n elementos : $A[\text{left}..\text{right}]$
- Procurar valor / chave X : índice ?
- Estratégia
 - Comparar $A[\text{middle}]$ com X
 - Se **iguais**, devolver middle
 - Se **maior**, procura recursiva em $A[\text{left}..\text{middle} - 1]$
 - Se **menor**, procura recursiva em $A[\text{middle} + 1..\text{right}]$

Procura Binária

- Como calcular o índice **middle** ?
 - Evitar overflow ! Shifting !
- Quantas **comparações** em cada passo ?
 - Variante : Tentar usar apenas uma comparação !
- Como assinalar um **valor / chave não existente** ?
 - Inteiros com sinal vs. sem sinal !

Procura Binária

```
int pesqBinRec(int* v, int esq, int dir, int valor) {  
    unsigned int meio;  
  
    if (esq > dir) return -1;  
  
    meio = (esq + dir) / 2;  
  
    contadorComps++;  
    if (v[meio] == valor) {  
        return meio;  
    }  
    contadorComps++;  
    if (v[meio] > valor) {  
        return pesqBinRec(v, esq, meio - 1, valor);  
    }  
  
    return pesqBinRec(v, meio + 1, dir, valor);  
}
```

Procura Binária

- Melhor Caso ?
 - 1 só comparação
- Pior Caso ?
 - Selecionar sempre a maior partição !
 - N^o impar vs. n^o par de elementos ?
 - Em que casos temos sempre partições com o mesmo tamanho ?
- Expressão para o n^o de comparações realizadas ?

Procura Binária – Caso particular

- $n = 2^k$
- $\text{esq} = 0$ $\text{dir} = 2^k - 1$ $\text{meio} = 2^{k-1} - 1$
- **Pior caso:** escolher sempre a **partição da direita**
 - É a maior das duas !!

$$W(1) = 2$$

$$W(n) = 2 + W(n/2) = 4 + W(n/4) = 6 + W(n/8) = \dots$$

$$W(n) = 2 \times k + W(1) = 2 + 2 \log n$$

$$W(n) \in \mathcal{O}(\log_2 n)$$

Tarefa 3 – O Problema da Moeda Falsa

- Dadas n moedas aparentemente idênticas, em que uma delas é uma moeda falsa
- Encontrar a moeda falsa !
- Usando apenas uma balança !
- A moeda falsa é mais leve do que uma moeda genuína !
- Algoritmo eficiente ?



[Wikipedia]

The Master Theorem

The Master Theorem

- Dada uma recorrência, para $n = b^k$, $k \geq 1$

$$T(1) = c \quad \text{e} \quad T(n) = a T(n / b) + f(n)$$

em que $a \geq 1$, $b \geq 2$, $c > 0$

- **Teorema** : Se $f(n)$ em $\Theta(n^d)$, em que $d \geq 0$, então

$$T(n) \text{ em } \Theta(n^d), \text{ se } a < b^d$$

$$T(n) \text{ em } \Theta(n^d \log n), \text{ se } a = b^d$$

$$T(n) \text{ em } \Theta(n^{\log_b a}), \text{ se } a > b^d$$

Na procura binária:

$a = 1$

$b = 2$

$d = 0$

Temos que $a = b^d$

$1 = 2^0$

The Master Theorem

- Permite obter diretamente a **ordem de complexidade**, dada uma recorrência
 - MAS **não a expressão final** para o nº de operações !
- Resultados válidos para as notações **$O(n)$** e **$\Omega(n)$**
- Exemplo

$$M(n) = 2 M(n / 2) + 1$$

$$f(n) = 1, f(n) \in \Theta(n^0), d = 0$$

$$a = 2, b = 2, a > b^d$$

$$M(n) \in \Theta(n)$$

The Smoothness Rule

Smooth Functions

- Função **eventualmente não-decrescente**

$$f(n_1) \leq f(n_2), \text{ para qualquer } n_2 > n_1 \geq n_0$$

- Função **“smooth”**

- 1) $f(n)$ é eventualmente não-decrescente
- 2) $f(2n) \in \Theta(f(n))$

- Exemplos

- $\log n$, n , $n \log n$ e n^k são funções “smooth”
- a^n não é !!

The Smoothness Rule

- Seja $T(n)$ uma função eventualmente não-decrescente
- E seja $f(n)$ uma função “smooth”
- Se $T(n) \in \Theta(f(n))$ para valores de n que sejam potências de b , com $b \geq 2$
- Então $T(n) \in \Theta(f(n))$
- Resultados análogos para $O(n)$ e $\Omega(n)$!!
- Boas notícias !!

Exemplo – Decrease by a Constant Factor

- Redução da dimensão de cada instância através da divisão por um fator constante

$$T(1) = c$$

$$T(n) = T(n / b) + f(n)$$

- Complexidade ?
 - $T(n)$ in $\Theta(\log n)$, se $f(n) = \text{constante}$
 - $T(n)$ in $\Theta(n)$, se $f(n)$ in $\Theta(n)$
- Exemplos ?

Mergesort

– Ordenação por Fusão

Mergesort

- Ordenar um array / lista
 - Se o tamanho é **0** ou **1**, já está ordenada
 - Caso contrário, **subdividir** em duas “metades”
 - Aprox. do mesmo tamanho **!!**
 - **Ordenar recursivamente** cada “metade”
 - **Fundir** as duas “metades” ordenadas num só array / lista
- Questão : usar ou não um **array / lista adicional** ?

Mergesort

0	1	2	3	4
7	2	6	4	3

Mergesort – Subdivisão

0	1	2	3	4
7	2	6	4	3

7	2	6
----------	----------	----------

Mergesort – Subdivisão

0	1	2	3	4
7	2	6	4	3

7	2
----------	----------

Mergesort – Subdivisão

0	1	2	3	4
7	2	6	4	3

7

Mergesort – Subdivisão

0	1	2	3	4
7	2	6	4	3

7	2
----------	----------

Mergesort – Fusão

0	1	2	3	4
7	2	6	4	3

7	2
----------	----------

Mergesort – Fusão

0	1	2	3	4
7	2	6	4	3

7
2

Mergesort – Fusão

0	1	2	3	4
7	2	6	4	3

2	7
----------	----------

Mergesort – Subdivisão

0	1	2	3	4
7	2	6	4	3

2	7	6
---	---	---

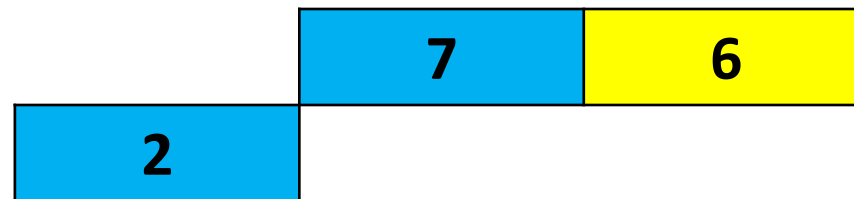
Mergesort – Fusão

0	1	2	3	4
7	2	6	4	3

2	7	6
----------	----------	----------

Mergesort – Fusão

0	1	2	3	4
7	2	6	4	3



Mergesort – Fusão

0	1	2	3	4
7	2	6	4	3

	7
2	6

Mergesort – Fusão

0	1	2	3	4
7	2	6	4	3

2	6	7
----------	----------	----------

Mergesort – Subdivisão

0	1	2	3	4
7	2	6	4	3

4	3
---	---

2	6	7
---	---	---

Mergesort – Subdivisão

0	1	2	3	4
7	2	6	4	3

4

2	6	7
---	---	---

Mergesort – Subdivisão

0	1	2	3	4
7	2	6	4	3

4	3
---	---

2	6	7
---	---	---

Mergesort – Fusão

0	1	2	3	4
7	2	6	4	3

4	3
---	---

2	6	7
---	---	---

Mergesort – Fusão

0	1	2	3	4
7	2	6	4	3

2	6	7	4	3
---	---	---	---	---

Mergesort – Fusão

0	1	2	3	4
7	2	6	4	3

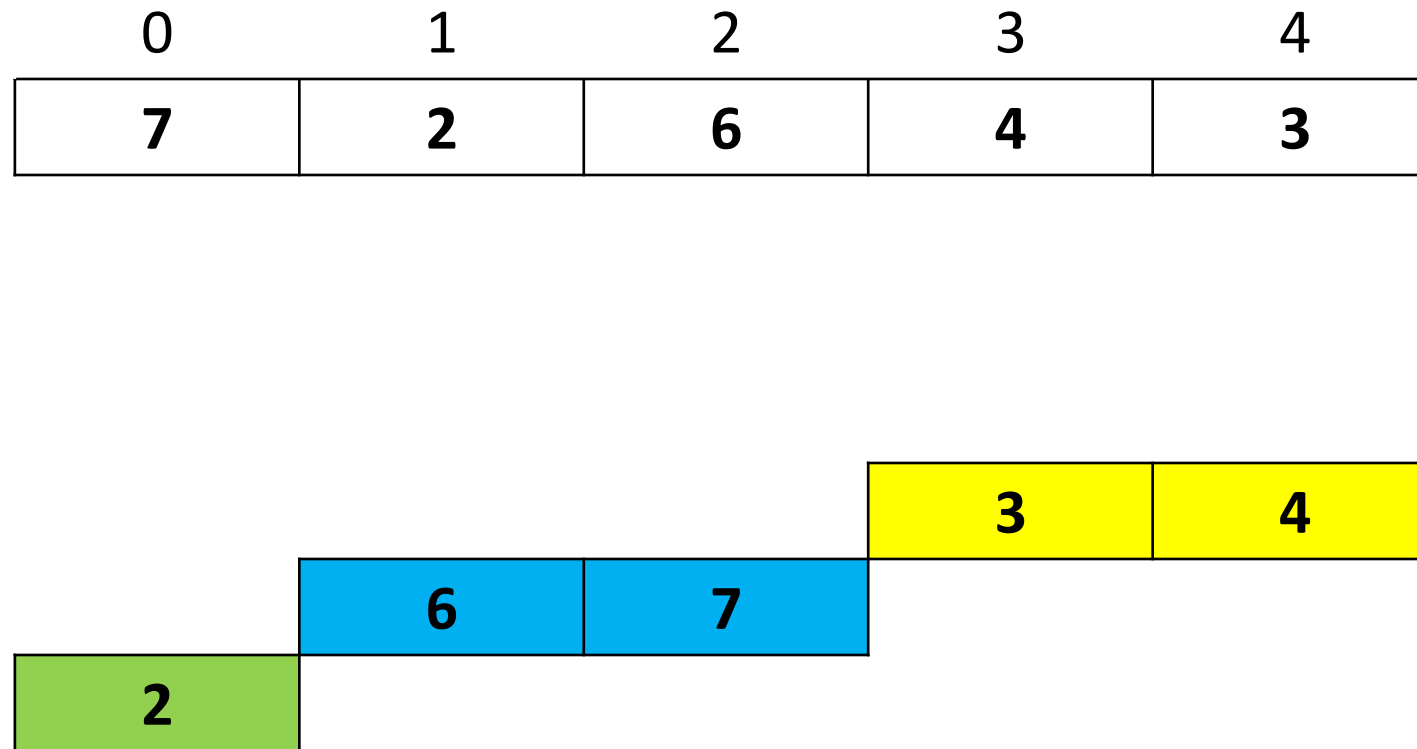
2	6	7	3	4
---	---	---	---	---

Mergesort

0	1	2	3	4
7	2	6	4	3

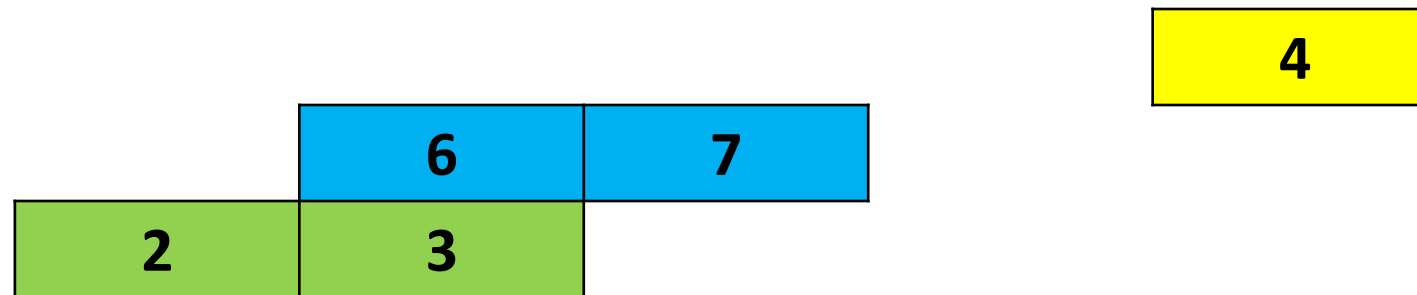
2	6	7	3	4
---	---	---	---	---

Mergesort – Fusão



Mergesort – Fusão

0	1	2	3	4
7	2	6	4	3



Mergesort – Fusão

0	1	2	3	4
7	2	6	4	3

	6	7
2	3	4

Mergesort – Fusão

0	1	2	3	4
7	2	6	4	3

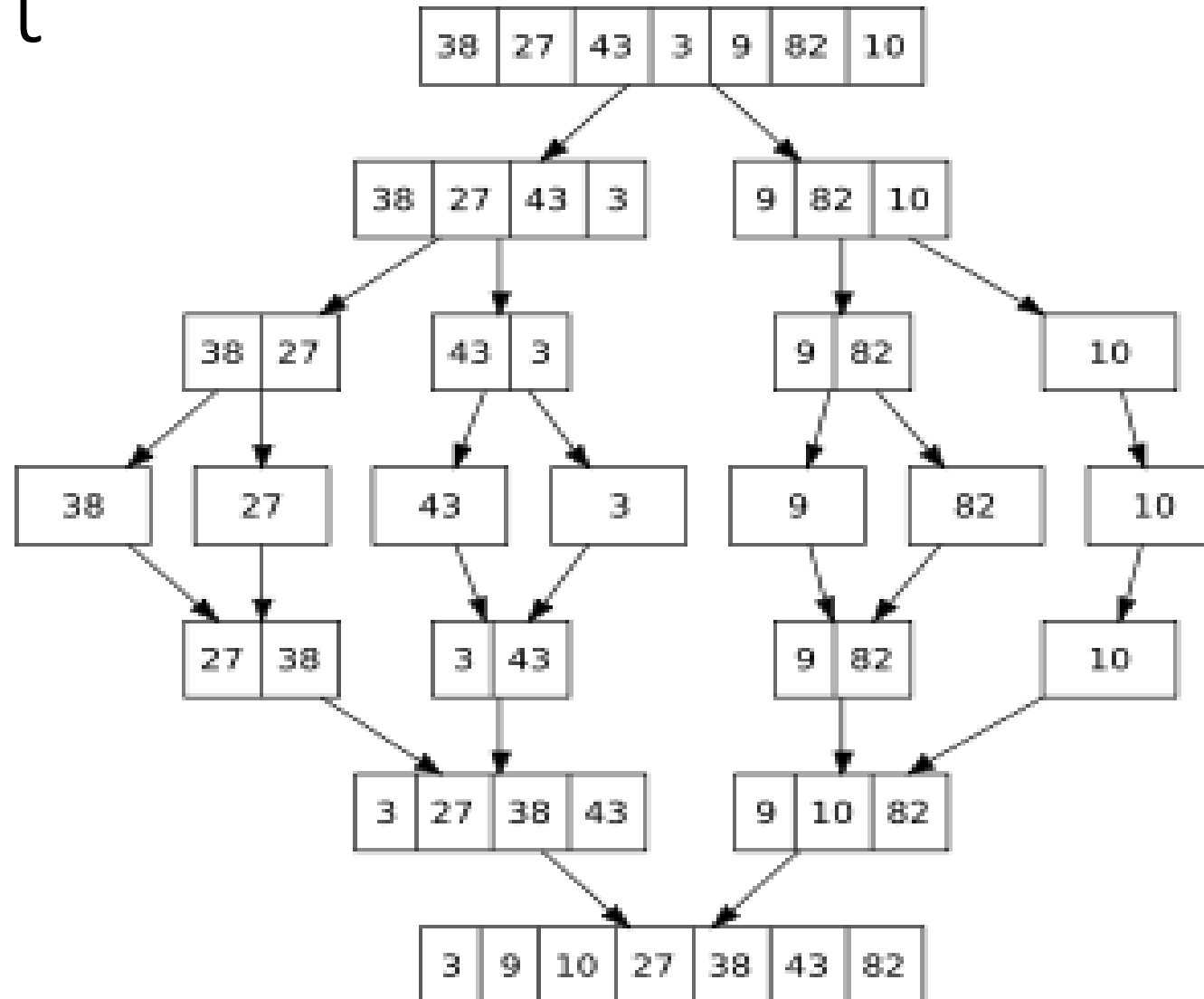
		7	
2	3	4	6

Mergesort – Array ordenado

0	1	2	3	4
7	2	6	4	3

2	3	4	6	7
----------	----------	----------	----------	----------

Mergesort



SUBDIVISÃO

FUSÃO

Tarefa: associar a cada seta um rótulo que identifica a sequência pela qual as chamadas são executadas

[Wikipedia]

Mergesort

```
// mergeSort(A, tmpA, 0, n - 1);


void mergeSort(int* A, int* tmpA, int left, int right) {
    // Mais do que 1 elemento ?

    if (left < right) {
        int center = (left + right) / 2;

        mergeSort(A, tmpA, left, center);
        mergeSort(A, tmpA, center + 1, right);
        merge(A, tmpA, left, center + 1, right);
    }
}
```


Mergesort

```
void merge(int* A, int* tmpA, int lPos, int rPos, int rEnd) {  
    int lEnd = rPos - 1;  
    int tmpPos = lPos;  
    int nElements = rEnd - lPos + 1;  
  
    // COMPARAR O 1o ELEMENTO DE CADA METADE  
    // E COPIAR ORDENADAMENTE PARA O ARRAY TEMPORÁRIO  
  
    while (lPos <= lEnd && rPos <= rEnd) {  
        if (A[lPos] <= A[rPos])  
            tmpA[tmpPos++] = A[lPos++];  
        else  
            tmpA[tmpPos++] = A[rPos++];  
    }  
}
```



Mergesort

```
// SOBRA, PELO MENOS, 1 ELEMENTO NUMA DAS METADES

while (lPos <= lEnd) { ...
}

while (rPos < rEnd) { ...
}

// COPIAR DE VOLTA PARA O ARRAY ORIGINAL

for (int i = 0; i < nElements; i++, rEnd--) {
    A[rEnd] = tmpA[rEnd];
}
}
```

Mergesort – Tarefas

- Eficiência ?
- Todas as comparações são feitas pela função de fusão
- Escrever a recorrência !
- Melhor caso / Pior caso / Caso médio ?
- $O(n \log n)$!!

Sugestões de leitura

Sugestões de leitura

- A. Levitin, Introduction to the Design and Analysis of Algorithms, 3rd Edition, 2012
 - Capítulo 4: secção 4.4
 - Capítulo 5: secção 5.1
 - Apêndice B