# 第2章 分治法

## 《人工智能算法》

清华大学出版社
2022年7月

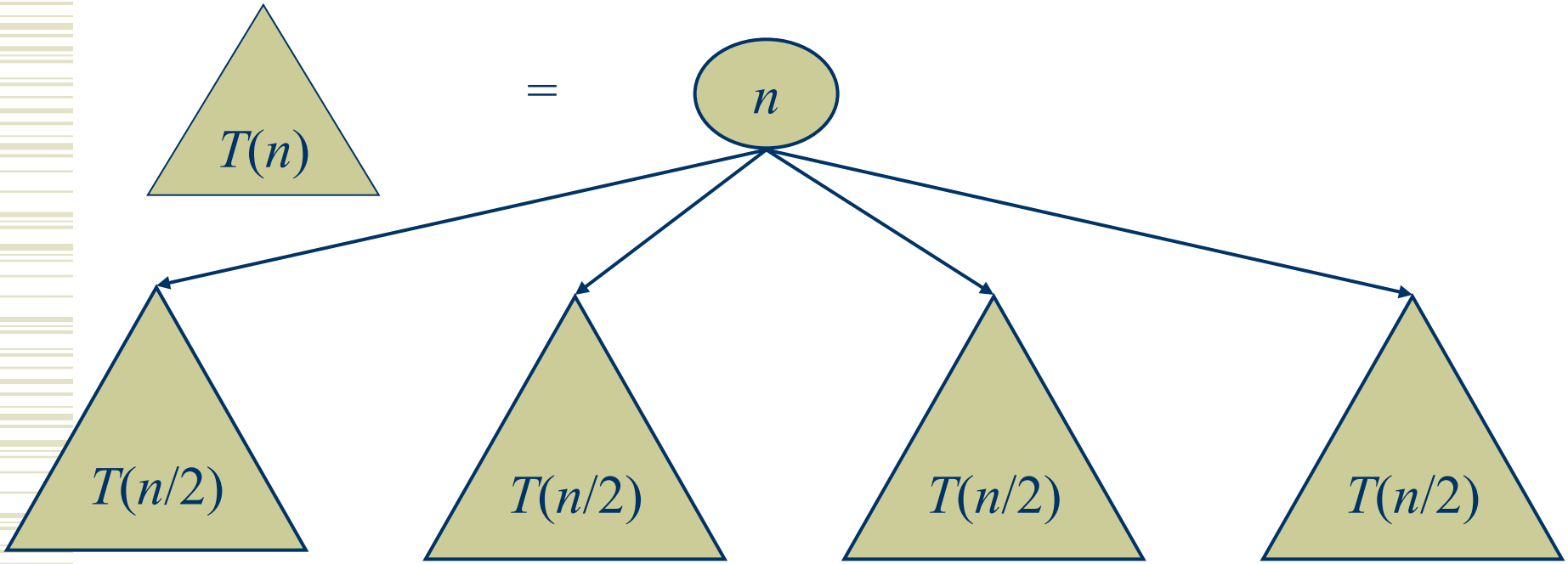
# 提纲

- 应用背景和动机
- 分治法的基本思想和一般步骤
- 分治法的适用条件
- 分治法的复杂度分析方法
- 合并排序
- 总结

# 应用背景和动机 (1)

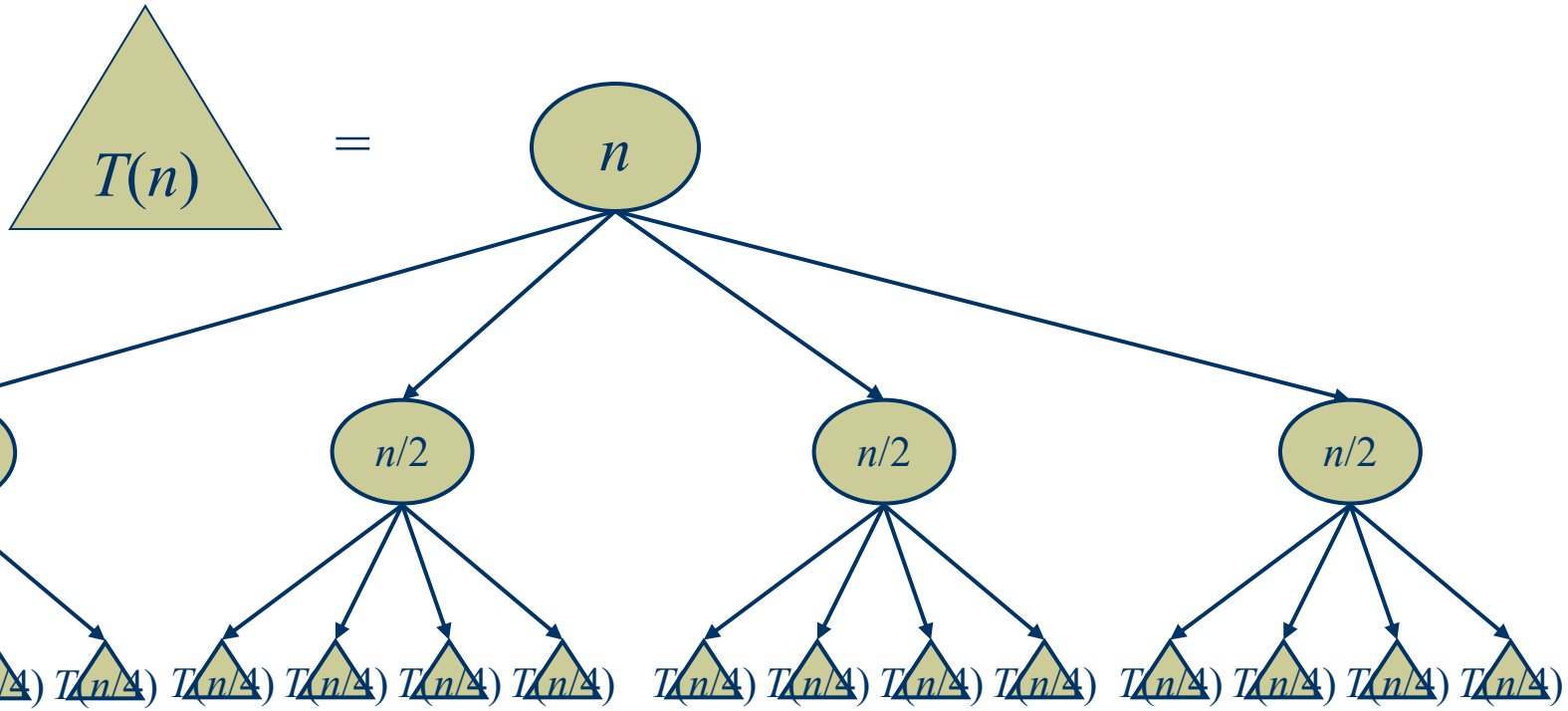
对这$k$个子问题分别求解

- 如果子问题的规模仍然不够小，再划分为$k$个子问题
- 如此递归地进行下去，直到问题规模足够小，很容易求出其解为止



# 应用背景和动机 (2)

- 对这$k$个子问题分别求解，其中分解直到问题规模足够小，很容易求出其解为止
- 合并小规模问题的解，自底向上求出原来问题的解



# 提纲

- 应用背景和动机
- 分治法的基本思想和一般步骤
- 分治法的适用条件
- 分治法的复杂度分析方法
- 合并排序
- 总结

# 分治法的基本思想和一般步骤 (1)

**分治法（Divide-and-Conquer）：**

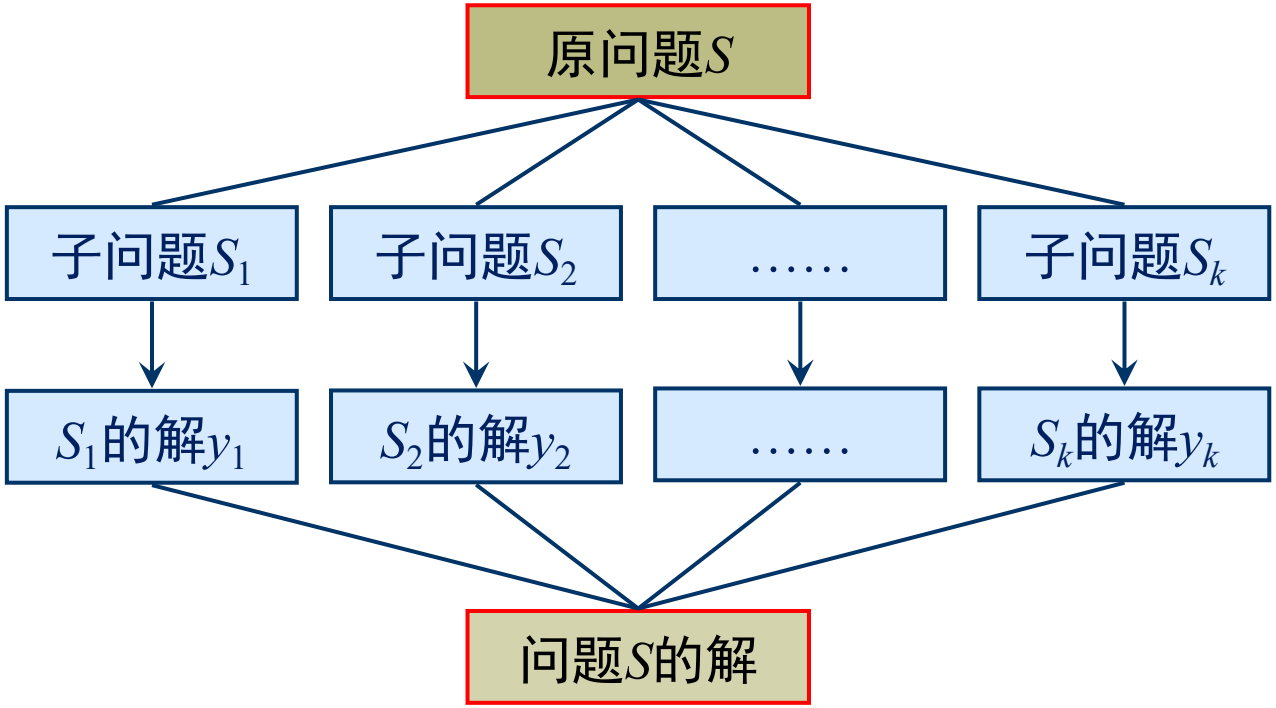
将一个难以直接解决的复杂问题，将其从大到小逐步分解，进而将较易求解的小问题解合并得到原问题的解。

凡治众如治寡，分数是也。

—— 孙子兵法

divide-and-conquer($S$)
  if ($|S|<=n_0$) adhoc($S$)  //解决小规模的问题
  else
   divide $S$ into smaller subinstances $S_1$, $S_2$, ..., $S_k$ //分解为子问题
   for $i$=1 to $k$ do
    $y_i \leftarrow$ divide-and-conquer($S_i$)  //递归求解各子问题
   end for
   return merge($y_1$, ..., $y_k$)  //合并各子问题的解
  end if

# 分治法的基本思想和一般步骤 (2)



# 分治法的基本思想和一般步骤 (3)

**注意：**

- 分解得到的子问题之间相互独立
- 子问题使用相同的方法求解
- 尽可能使子问题规模均等（平衡子问题）

**问题：**

- DAG算法要被执行多少次？
- 算法中的基本操作要被执行多少次？
- 如何分析该类算法的时间复杂度？
- 分治法适合求解什么样的问题?

# 提纲

- 应用背景和动机
- 分治法的基本思想和一般步骤
- 分治法的适用条件
- 分治法的复杂度分析方法
- 合并排序
- 总结

# 分治法的适用条件

- 该问题的规模缩小到一定的程度就可以容易地解决
- 该问题具有最优子结构性质:
  - 该问题可以分解为若干个规模较小的相同问题
  - 该问题的最优解包含着其子问题的最优解
  - 利用该问题分解出的子问题的解可以合并为该问题的解
- 该问题所分解出的各个子问题是相互独立的，即子问题之间不包含公共的子问题，并不重复计算公共子问题

若子问题不独立，如何处理？

# 提纲

- 应用背景和动机
- 分治法的基本思想和一般步骤
- 分治法的适用条件
- 分治法的复杂度分析方法
- 合并排序
- 总结

# 分治法的复杂度分析方法 (1)

- 分治算法时间复杂度分析不直观
- 若将分治法映射到四个步骤，且已知每个步骤的计算时间，则分治法的时间复杂度可使用递推关系（**Recurrence relation**）进行分析
- $$DAC(S)=\begin{cases} Adhoc(S), & |S|<n_0 \\ DIV(S)+\sum_{i=1}^{k} DAC(S_i)+Merge(S) \quad , & \text{else} \end{cases}$$

```
int factorial(int n)
{
  if(n=0) return 1
  return n*factorial(n−1)
}
```

以乘法（*）作为基本操作

$$T(n)=\begin{cases} 0 & n=0 \\ T(n-1)+1 & n\geq 1 \end{cases}$$

# 分治法的复杂度分析方法 (2)

- **平衡子问题**

  - 子问题规模大致相同

  - 若$|S|=n$, 分解为个$k$个规模为$n/m$的子问题

  - $f(n)$时间将$k$个子问题合并为原问题的解

  计算时间$T(n)$用递推关系表示为：

$$T(n)=\begin{cases}O(1) & n=n_0\\ kT(n/m)+f(n) & n>n_0\end{cases}$$

# 分治法的复杂度分析方法 (3)

- **分治法运算时间的通用分治递推式：**

  一个规模$n$的问题，每次被分为$a$个子问题，每个子问题规模 $n/b$

  （为简化分析，假设$n=b^k$，$k=1, 2, 3, ...$）

$$T(n)=\begin{cases} c, & n \le t_r \\ aT(n/b)+f(n), & n > t_r \end{cases}$$

  $c$：直接求解子问题（规模为 $t_r$）时间（常量）

  $f(n)$：子问题分解和子问题解合并的时间

# 提纲

- 应用背景和动机
- 分治法的基本思想和一般步骤
- 分治法的适用条件
- 分治法的复杂度分析方法
- 合并排序
- 总结

# 合并排序 (1)

- **引例：**合并两个有序子列表

{179, 285, 351}, {310, 312, 652, 800}

(1) 179<310: {179}

(2) 285<310: {179, 285}

(3) 351>310: {179, 285, 310}

{179, 285, 351}, {310, 312, 652, 800}

(4) 312<351: {179, 285, 310, 312}

(5) 652>351: {179, 285, 310, 312, 351, 652}

{179, 285, 351}, {310, 312, 652,800}

# 合并排序 (2)

- 合并两个有序子列表能高效地完成
- 只包含一个元素的列表是有序的
- 自顶向下将列表划分为只含一个元素的片段

  自底向上将有序子列表两两合并起来
- 将列表{*first*, …, *last*}中两个有序子列表合并的递归思想：

  - 若*first*小于*last*，则将其从中间位置划分为两个子列表
  - 当*first*=*last*时, 子列表中只含一个元素
  - 将子列表合并起来，大小分别为1、2、4、 …,

# 合并排序 (3)

**算法**MergeSort(*list*, *first*, *last*)

```
if first<last then
  middle ← (first + last)/2
  MergeSort(list, first, middle)
  MergeSort(list, middle+1, last)
  MergeLists(list, first, middle, middle+1, last)
end if
```

递归

A

B

MergeLists什么时候首次执行?

# 合并排序 (4)

**示例：**列给定表{310, 285, 179, 254, 351, 423, 861, 139, 450, 520}

将列表从中间划分为两个子列表

310 | 285 | 179 | 254 351 | 423 861 254 450 520

合并两个有序子列表

(1) A={310}, B={285}

C={285, 310}

(2) A={285, 310}, B={179}

C={179} ⇒ C={179, 285, 310}

- 当列表B中已无元素
- 将剩余元素放到C的末尾

。。。

下一步执行什么操作?

# 合并排序 (5)

划分：

310 | 285 | 179 | 254 | 351 | 423 861 139 450 520

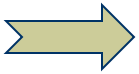
合并：

(3) A={254}, B={351}

C={254, 351}

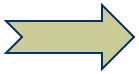
之前的结果 {179, 285, 310}

(4) A={179, 285, 310}, B={254, 351}



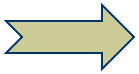
C={179}

A={179, 285, 310}, B={254, 351}



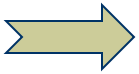
C={179, 254}

A={179, 285, 310}, B={254, 351}



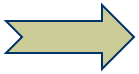
C={179, 254, 285}

A={179, 285, 310}, B={254, 351}



C={179, 254, 285, 310}

A={179, 285, 310}, B={254, 351}



C={179, 254, 285, 310, 351}

列表A中已无元素

- 一个隐含二叉树所表示的一系列递归调用
- 栈存储每次调用过程的局部数据

# 合并排序 (6)

MergeLists(*list*, *start*1, *end*1, *start*2, *end*2)

(1)

```
indexC←1, finalStart←start1, finalEnd←end2
while (start1≤end1) and (start2≤end2) do
  if list[start1] < list[start2] then
    result[indexC]←list[start1]
    start1←start1+1
  else
    result[indexC]←list[start2]
    start2←start2+1
  end if
  indexC←indexC+1
end while
```

列表B中元素值更小

列表A中元素值更小

**While循环什么时候停止执行?**

# 合并排序 (7)

(2) 移动剩余元素

```
if start1≤end1 then
  for i←start1 to end1 do
    result[indexC]←list[i]
    indexC←indexC+1
  end for
else
  for i←start2 to end2 do
    result[indexC]←list[i]
    indexC←indexC+1
  end for
```

(3) 将结果从C中填回原列表

```
indexC←1
for i←finalStart to finalEnd do
  list[i]←result[indexC]
  indexC←indexC+1
end for
```

无需比较操作!

# 合并排序 (8)

- **MergeLists的最优情况**
  - 列表A中所有元素都不大于B中最小
  - $n_A$ 次比较 ($n/2$)
  - 例如：A={1, 2, 3}, B={4, 5 ,6}
- **MergeLists的最坏情况**
  - 列表A列表 B中元素交叉排列时
  - 每执行一次比较操作，将A或B中的一个元素移到列表C中
  - $n_A + n_B - 1$次比较 ($n-1$)
  - 例如：A={1, 3, 5}, B={2, 4, 6}

- **MergeSort算法的执行时间**

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

$$O(n \log n)$$

- 以元素比较为基本操作的排序问题下界为$\Theta(n\log n)$
- 合并排序最坏情况下比较次数接近以上下界，最优的排序算法

# 合并排序 (9)

- 上述的递推式对于$n$是2的幂的时候成立，如果$n$是任意的整数（不是2的幂）呢？

- **递推关系：**

$$T(n)=\begin{cases}O(1) & n\le 1\\ T\lfloor n/2\rfloor+T\lceil n/2\rceil+O(n) & n>1\end{cases} \qquad O(n\log n)$$

- **结论：**

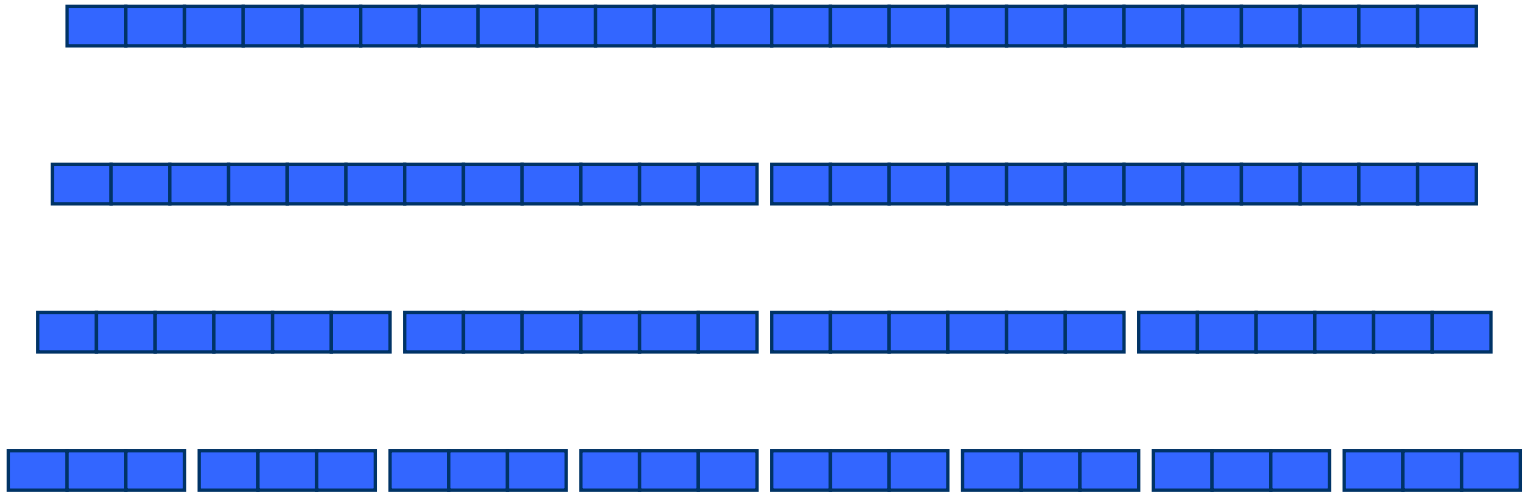
- 算法MergeSort对一个$n$个元素的数组排序所需的时间是$O(n\log n)$，空间是$O(n)$。
- 合并排序的计算时间开销仅来自合并，划分本身无需时间开销。

# 合并排序 (10)

- **合并排序算法的主要缺点：**需要$O(n)$的额外空间
- **空间开销：**额外的result（列表C）空间，递归算法栈的空间
- **改进思路：**

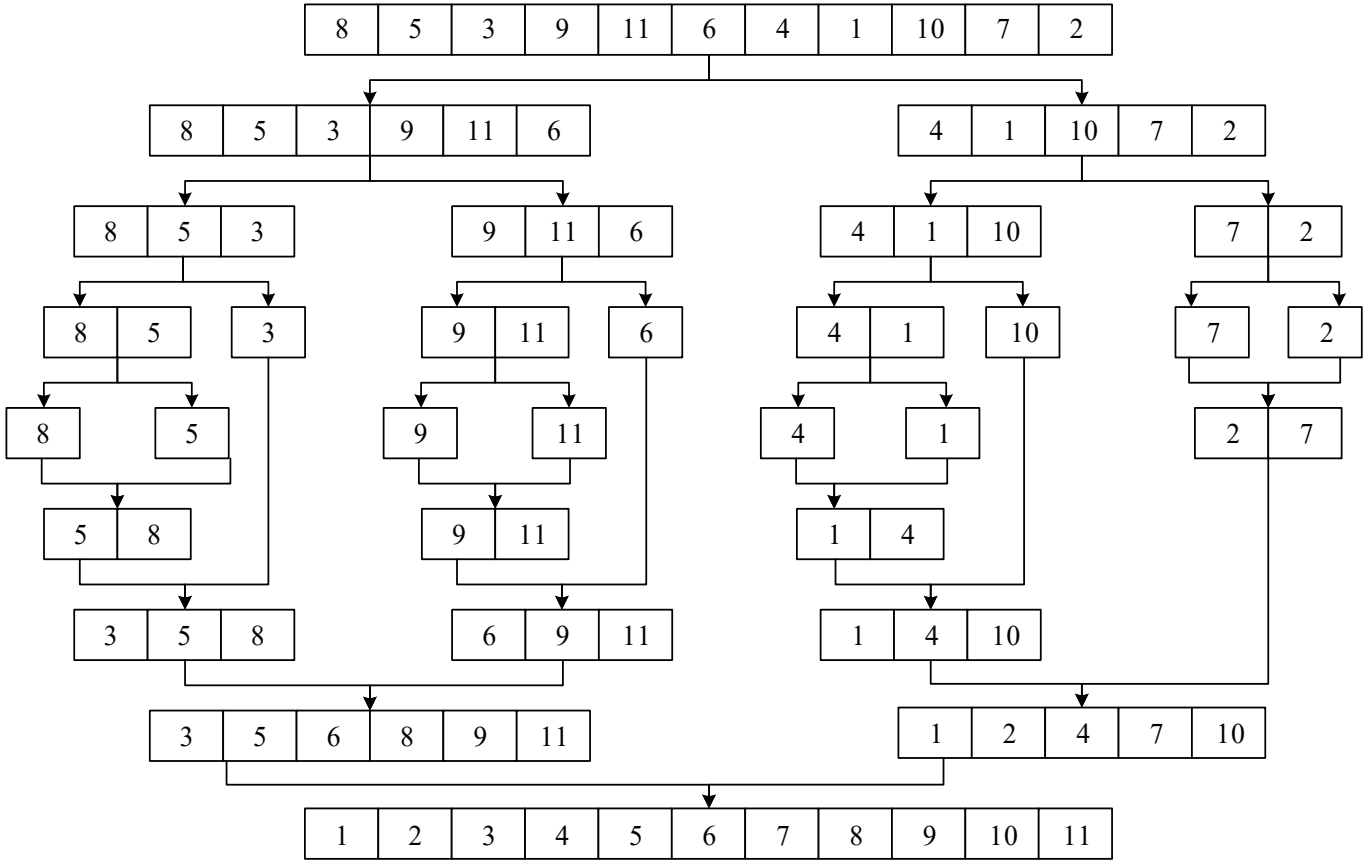
（1）“在位”的MergeLists，不需要额外的结果数组result；算法过于复杂，只具有理论上的意义

（2）非递归合并排序算法；算法没有递归算法直观、容易理解



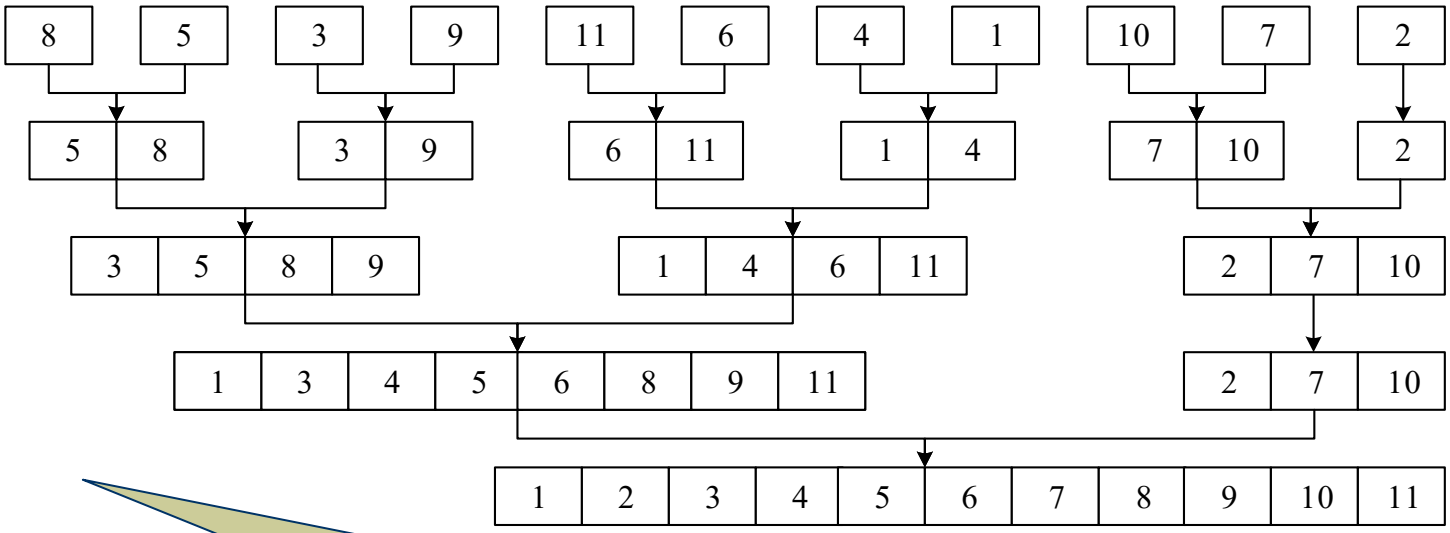
# 合并排序 (11)

- **对比：**递归的合并排序算法



# 合并排序 (12)

- **对比：**非递归的合并排序算法




如何选择或设计递归、非递归的合并排序算法？

# 提纲

- 应用背景和动机
- 分治法的基本思想和一般步骤
- 分治法的适用条件
- 分治法的复杂度分析方法
- 合并排序
- 总结

# 总结

- 分治法的适用条件——子问题独立
- 分治法的基本思想——四个步骤
- 分治法的复杂度分析方法——递推式
- 合并排序（递归的思想，递归的算法）

  ——以元素比较为基本操作，源于合并步骤

# 结语

谢谢!