计算机算法设计与分析

第二讲：分治法和排序算法

薛健

工程管理与信息技术学院

Last Modified: 2015.3.22

主要内容

主要内容

主要内容

递归 (Recursion)

- **John McCarthy** (人工智能之父)：最早认识到递归对于计算机程序设计语言的重要性，建议在 *Algol60* (Pascal、PL/I 和 C 的前身) 中增加递归特性，并自己设计了一种引入递归数据结构的语言：*Lisp*；现在，几乎所有主流的计算机程序设计语言均支持递归
- 在数学和计算机科学中，递归指在函数的定义中使用函数自身的方法，也常用于描述由一种 (或多种) 简单的基本情况定义的一类对象或方法，并规定其他所有情况都能被还原为其基本情况 (用自相似的方法描述事物的过程)
- 例如，关于某人祖先的定义：
  - 某人的双亲是他的祖先 (基本情况)
  - 某人祖先的双亲同样是某人的祖先 (递归步骤)
- 像这样的定义在数学中十分常见。例如，集合论对自然数的正式定义是：1 是一个自然数，每个自然数都有一个后继，这一个后继也是自然数

例子

Example

斐波那契数列 (Fibonacci Sequence)

- $F_0 = 0$
- $F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}$

Algorithm `Fib`(n)

```
1 if n < 2 then f ← n;
2 else
3 |   f1 ← Fib(n − 1);
4 |   f2 ← Fib(n − 2);
5 |   f ← f1 + f2;
6 end
7 return f;
```

用递归来解决问题

- 递归过程的建立：
  1. 我们已经完成了吗？如果完成了，返回结果 (如果没有这样的终止条件，递归将会永远地继续下去)
  2. 如果没有，则简化问题，解决较容易的问题，并将结果组装成原始问题的解决办法；然后返回该解决办法

- Method 99：一种更便于理解的构造方式
  - 假设要解决的问题规模为 100 (fantasy precondition)
  - 假设已经有一个名为 p99 的子程序能够解决规模为 0∼99 的问题
  - 划分不需递归的部分 (base case)
  - 构造解决问题的程序 p，写出任何不需要 p99 就可以解决的情况的代码，其他情况下在需要的时候调用 p99 (将规模太大不能直接解决的情况分解为规模在 0∼99 之间的子问题，分别用 p99 解决)

例子

Example (二分查找的递归版本)

Algorithm `BinarySearchRec`$(E[], first, last, K)$

```
1  if last < first then return −1;
2  else
3  |   mid ← (first + last)/2;
4  |   if K = E[mid] then return mid;
5  |   else if K < E[mid] then
6  |   |   return BinarySearch99(E, first, mid − 1, K);
7  |   end
8  |   else return BinarySearch99(E, mid + 1, last, K);
9  end
10 return −1;
```

例子

Example (二分查找的递归版本)

Algorithm `BinarySearchRec`$(E[], first, last, K)$

```
1  if last < first then return −1;
2  else
3  |   mid ← (first + last)/2;
4  |   if K = E[mid] then return mid;
5  |   else if K < E[mid] then
6  |   |   return BinarySearchRec(E, first, mid − 1, K);
7  |   end
8  |   else return BinarySearchRec(E, mid + 1, last, K);
9  end
10 return −1;
```

课后习题

Exercise (2)

定义文件 xx.tar.gz 的产生方式如下：

- 以 xx 为文件名的文件通过 tar 和 gzip 打包压缩产生，该文件中以字符串的方式记录了一个非负整数；
- 或者以 xx 为名的目录通过 tar 和 gzip 打包压缩产生，该目录中包含若干 xx.tar.gz。

其中，x $\in$ [0, 9]。现给定一个根据上述定义生成的文件 00.tar.gz (该文件从课程网站下载)，请确定其中包含的以 xx 为文件名的文件个数以及这些文件中所记录的非负整数之和。

deadline: 2014.03.28

课后习题 (提示)

- 编写递归程序求解
- 可使用任意你所熟悉的编程语言和工具
- 大多数高级语言或脚本语言提供了解 gzip 压缩包的工具

课后习题 (提示)

- 编写递归程序求解
- 可使用任意你所熟悉的编程语言和工具
- 大多数高级语言或脚本语言提供了解 gzip 压缩包的工具

例: 用 Python 解压缩包

```
import os, tarfile
def unpack_path_file(pathname):
    archive = tarfile.open(pathname, 'r:gz')
    for tarinfo in archive:
        archive.extract(tarinfo, os.getcwd())
    archive.close()
```

主要内容

数学归纳法

- 什么是证明？ (回顾)：在一个特定的公理系统中，由公理 (axioms) (和定理 (theorems)) 推导出某些命题 (proposition) 的过程
  - 证明方法：演绎推理，构造证明，反证法，数学归纳法，...
- 数学归纳法：用于证明某个给定命题对于一个无限集内的所有对象成立的证明方法
- 最常见的归纳在自然数集上进行，但归纳法同样适用于更一般的无限集：
  - 给定集合是偏序集 (partially ordered set)，即集合中的部分元素对之间定义了偏序关系 (满足自反性、反对称性和传递性的二元关系)
  - 给定集合中不存在无限递减的链

数学归纳法

- 证明模式：要证明关于 n 的语句 $F(n)$，数学归纳法可分两步：[1]
  1. 奠基：当 $n=0$ 时待证语句为真，即须证 $F(0)$ 真
  2. 归纳：在一定的假设下，证明情形 $n+1$ 时待证语句为真，即证明 $F(n+1)$ 真
- 完成上述两步，即可得出结论“依数学归纳法，待证语句得证”
- “一定的假设”—— 归纳假设：
  - 简单归纳假设：在假设“$F(n)$ 真”之下，去证明 $F(n+1)$ 真 $\Rightarrow$ 简单归纳证明
  - 强归纳假设：在假设“$F(0)$ 真，$F(1)$ 真，$\cdots$，$F(n)$ 真”之下去证明 $F(n+1)$ 真 $\Rightarrow$ 强归纳证明
  - 参变归纳假设：待证语句含有参数，如 $F(n,u)$，则奠基是：$F(0,u)$ 对一切 u 真；在归纳步骤中，假设“$F(n,u)$ 对一切 u 真”，从而证明 $F(n+1,u)$ 亦真 $\Rightarrow$ 参变归纳证明

[1] 莫绍揆：《递归函数论》

数学归纳法

- **证明模式**：要证明关于 n 的语句 $F(n)$，数学归纳法可分两步：[1]
  1. **奠基**：当 $n=0$ 时待证语句为真，即须证 $F(0)$ 真
  2. **归纳**：在**一定的假设**下，证明情形 $n+1$ 时待证语句为真，即证明 $F(n+1)$ 真
- 完成上述两步，即可得出结论“依数学归纳法，待证语句得证”
- “**一定的假设**”—— **归纳假设**：
  - **简单归纳假设**：在假设“$F(n)$ 真”之下，去证明 $F(n+1)$ 真 $\Rightarrow$ **简单归纳证明**
  - **强归纳假设**：在假设“$F(0)$ 真，$F(1)$ 真，$\cdots$，$F(n)$ 真”之下去证明 $F(n+1)$ 真 $\Rightarrow$ **强归纳证明** ★
  - **参变归纳假设**：待证语句含有参数，如 $F(n,u)$，则奠基是：$F(0,u)$ 对一切 u 真；在归纳步骤中，假设“$F(n,u)$ 对一切 u 真”，从而证明 $F(n+1,u)$ 亦真 $\Rightarrow$ **参变归纳证明**

[1] 莫绍揆：《递归函数论》

数学归纳法与递归

“几乎可以说：每有一种递归式，便有一种数学归纳法；反之，每有一种数学归纳法，便有一种递归式。··· 例如：简单归纳法对应于原始递归式，强归纳法对应于串值原始递归式，参变归纳法对应于参数变异递归式。”

——莫绍揆：《递归函数论》

数学归纳法与递归

“几乎可以说：每有一种递归式，便有一种数学归纳法；反之，每有一种数学归纳法，便有一种递归式。··· 例如：简单归纳法对应于原始递归式，强归纳法对应于串值原始递归式，参变归纳法对应于参数变异递归式。”

——莫绍揆：《递归函数论》

- 数学归纳法与递归有着非常紧密的联系
  - 数学归纳法证明可以看作是一种递归证明
  - 数学归纳法使得递归程序正确性证明得到极大的简化

递归程序的数学归纳法证明 — 2-tree 外路径长度

Definition (2-tree (full binary tree))

2-tree 是这样一种二叉树，它是一棵空树或仅包含下列两种结点：

1. 外结点 (external node)：没有子树 (度为 0) 的结点，即叶结点
2. 内结点 (internal node)：恰有两棵子树的结点

递归程序的数学归纳法证明 — 2-tree 外路径长度

Definition (2-tree (full binary tree))

2-tree 是这样一种二叉树，它是一棵空树或仅包含下列两种结点：

1. 外结点 (external node)：没有子树 (度为 0) 的结点，即叶结点
2. 内结点 (internal node)：恰有两棵子树的结点

Definition (外路径长度 (external path length))

一棵 2-tree T 的外路径长度是从根结点到所有外结点路径长度之和 (路径长度等于路径经过的边的数目)。其等价的递归定义：

1. 以外结点为根的 2-tree (单结点 2-tree) 其外路径长度为 0
2. 以内结点为根的 2-tree 外路径长度等于其左子树 L 的外路径长度加 L 的外结点个数 加其右子树 R 的外路径长度加 R 的外结点个数

计算外路径长度

Algorithm `CalculateEPL`(T)

Input: A 2-tree T
Output: A pair $(epl, extNum)$

```
1 if T is a leaf then return (0, 1) ;
2 else
3 |   (eplL, extNumL) ← CalculateEPL(T.left);
4 |   (eplR, extNumR) ← CalculateEPL(T.right);
5 |   epl ← eplL + eplR + extNumL + extNumR;
6 |   extNum ← extNumL + extNumR;
7 |   return (epl, extNum);
8 end
```

外路径长度引理

Lemma

T 是一棵 2-tree，e 和 m 分别对应于 `CalculateEPL` 返回值中的 epl 和 $extNum$，则有：

1. e 等于 T 的外路径长度
2. m 等于 T 的外结点个数
3. $e \geq m \lg m$

skip proof

外路径长度引理

Lemma

T 是一棵 2-tree，e 和 m 分别对应于 CalculateEPL 返回值中的 epl 和 $extNum$，则有：

1. e 等于 T 的外路径长度
2. m 等于 T 的外结点个数
3. $e \geq m \lg m$

Proof.

对 T 用数学归纳法。

奠基：T 是叶结点，根据 CalculateEPL 第 1 行，$e = 0, m = 1$ 满足 1 和 2，$0 \geq 1 \lg 1 = 0$ 满足 3。

归纳：对于非叶结点 T，假设 T 的所有子树 S 引理成立，令 L 和 R 分别为 T 的左、右子树，则根据归纳假设，对于 L 和 R 引理也成立。

外路径长度引理

Lemma

T 是一棵 2-tree，e 和 m 分别对应于 `CalculateEPL` 返回值中的 epl 和 $extNum$，则有：

1. e 等于 T 的外路径长度
2. m 等于 T 的外结点个数
3. $e \geq m \lg m$

Proof.

由 `CalculateEPL` 第 3 到 7 行有：

$$\begin{aligned} e &= e_L + e_R + m_L + m_R \\ m &= m_L + m_R \end{aligned}$$

外路径长度引理

Lemma

T 是一棵 2-tree，e 和 m 分别对应于 `CalculateEPL` 返回值中的 epl 和 $extNum$，则有：

1. e 等于 T 的外路径长度
2. m 等于 T 的外结点个数
3. $e \geq m \lg m$

Proof.

由外路径长度的递归定义可知 e 为 T 的外路径长度；又因为 T 的外结点要么在 L 中，要么在 R 中，所以 m 等于 T 的外结点个数；另外，根据归纳假设，有：

$$e \geq m_L \lg m_L + m_R \lg m_R + m$$

外路径长度引理

Lemma

T 是一棵 2-tree，e 和 m 分别对应于 `CalculateEPL` 返回值中的 epl 和 $extNum$，则有：

1. e 等于 T 的外路径长度
2. m 等于 T 的外结点个数
3. $e \geq m \lg m$

Proof.

注意到 $x \lg x$ 是凸函数，则根据凸函数性质有：

$$m_L \lg m_L + m_R \lg m_R \geq 2\left(\frac{m_L + m_R}{2}\right) \lg \left(\frac{m_L + m_R}{2}\right)$$

所以得：$e \geq m(\lg m - 1) + m = m \lg m$ □

外路径长度引理

Lemma

T 是一棵 *2-tree*，e 和 m 分别对应于 `CalculateEPL` 返回值中的 epl 和 $extNum$，则有：

1. e 等于 T 的外路径长度
2. m 等于 T 的外结点个数
3. $e \geq m \lg m$

Corollary

内结点数为 n 的 *2-tree* 外路径长度 $e \geq (n+1)\lg(n+1)$

return

主要内容

定义和术语

Definition

程序块 (block) 一段有且仅有一个入口和一个出口的程序代码

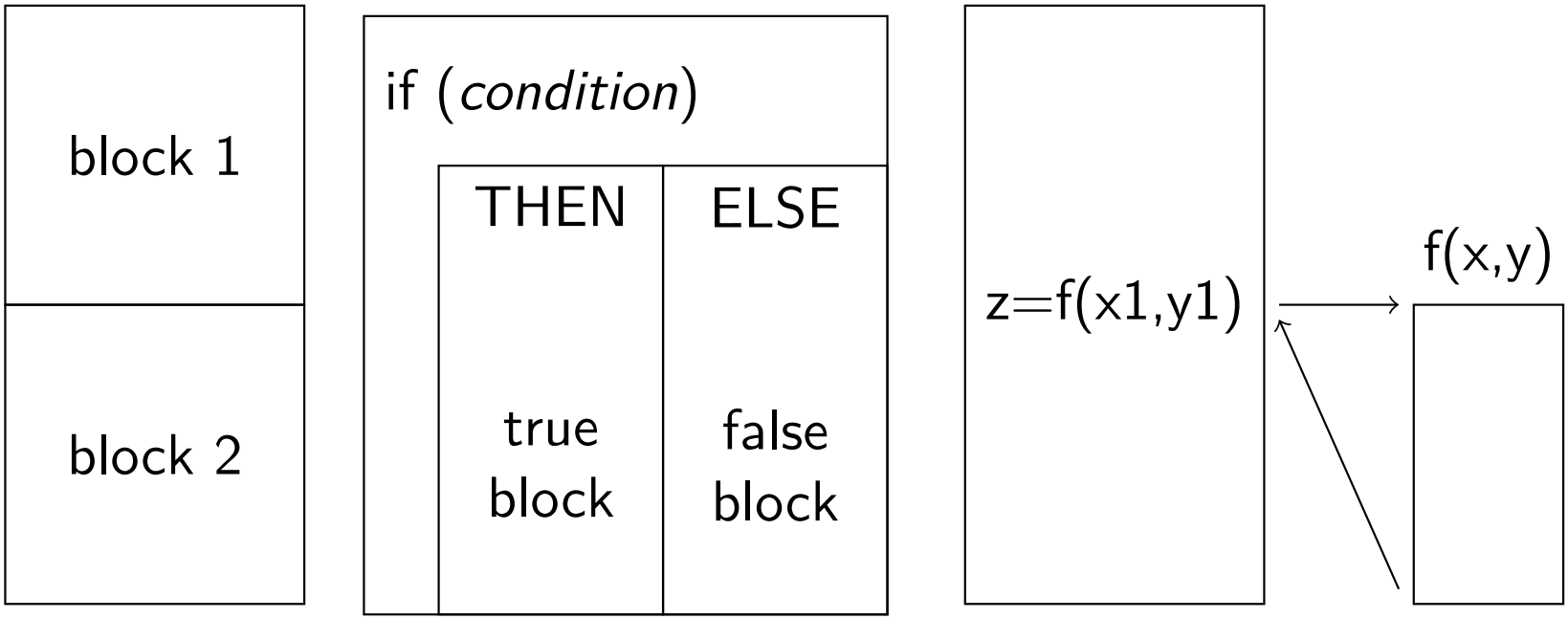
过程 (procedure) 赋予名称且可被调用的程序块，一般包含输入和输出参数

函数 (function) 有输出参数的过程

前件 (precondition) 用于描述一个程序块的输入参数和非局部数据且在进入程序块时为真的逻辑语句

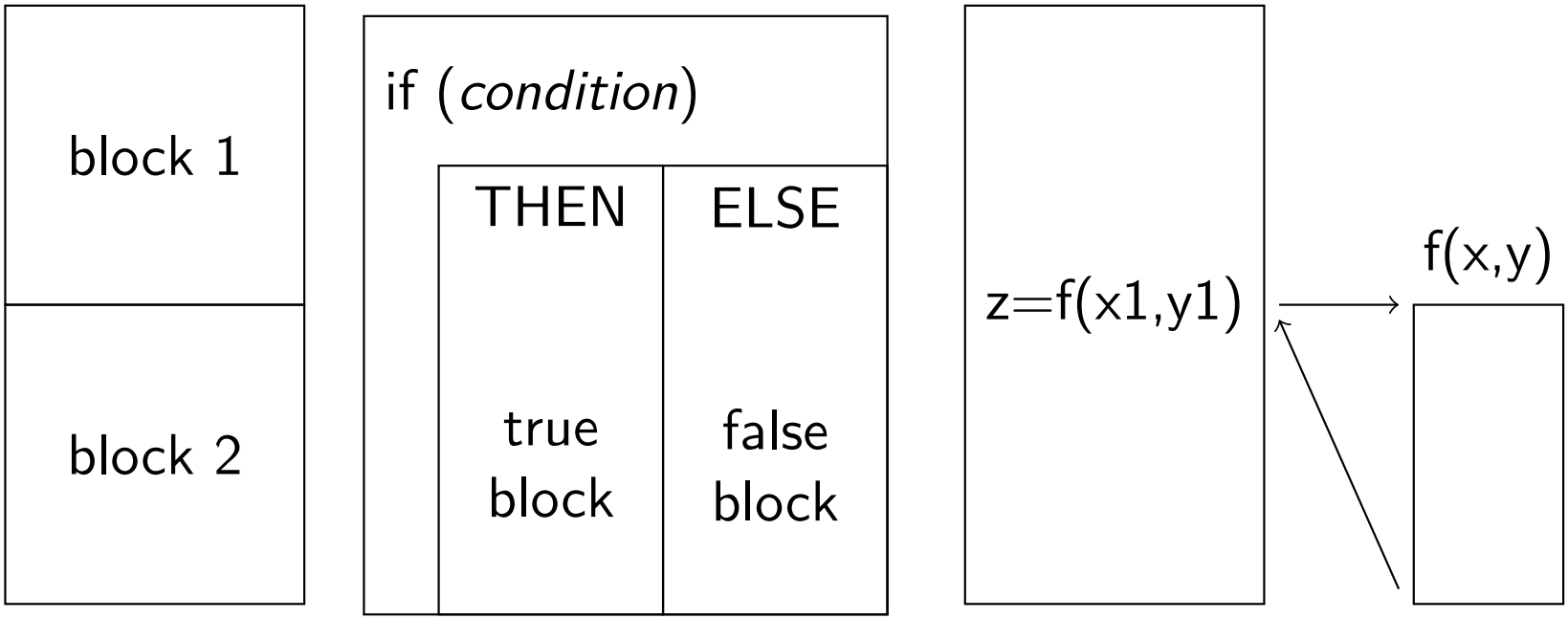
后件 (postcondition) 用于描述一个程序块的输入参数、输出参数及非局部数据且在退出程序块时为真的逻辑语句

基本控制结构



sequence *alternation* *procedure call*

基本控制结构



sequence *alternation* *procedure call*

循环结构 (**for**, **while**, ...) $\Longrightarrow$ 递归！

基本证明形式

Proposition (General correctness lemma form)

如果在进入程序块时前件成立，则在退出程序块时后件也成立。

Proposition (Sequence correctness lemma form)

1. 整个程序块的前件 $\Rightarrow$ 程序块 *1* 的前件
2. 程序块 *1* 的后件 $\Rightarrow$ 程序块 *2* 的前件
3. 程序块 *2* 的后件 $\Rightarrow$ 整个程序块的后件

Proposition (Procedure-call correctness lemma form)

1. 程序块的前件 $\Rightarrow$ 被调用过程的前件及其实际调用参数
2. 被调用过程的后件及其实际调用参数 $\Rightarrow$ 程序块的后件

基本证明形式 (cont.)

Proposition (Alternation correctness lemma form)

1. 程序块的前件以及选择分支条件为真 $\Rightarrow$ 条件为真时的分支程序块前件
2. 条件为真时的分支程序块后件以及选择分支条件为真 $\Rightarrow$ 程序块的后件
3. 程序块的前件以及选择分支条件为假 $\Rightarrow$ 条件为假时的分支程序块前件
4. 条件为假时的分支程序块后件以及选择分支条件为假 $\Rightarrow$ 程序块的后件

单赋值形式

- 证明的困难：到逻辑表达式的转换
  - 无条件跳转 (goto) 语句：可以消除
  - 赋值 (assignment) 语句："y=y+1"，"x=y+1; y=z;"，...
- **单赋值形式**：Single-Assignment Paradigm 或 Static Single Assignment Form (SSA)，如果在某个过程内赋值的每一个变量作为赋值目标只出现一次，称这个过程是 (静态) 单赋值形式。
- 在编译器设计中，单赋值形式是一种中间表示 (IR)，它能有效地将程序中的运算值和它们的存储位置分开，从而使得若干优化能具有更有效的形式
- 使用单赋值形式的语言 (编译器)：Prolog, ML, Haskell, SISAL (Streams and Iteration in a Single Assignment Language), SAC (Single Assignment C)

例子

a code fragment

```
1 if y < 0 then
2 |   y = 0;
3 end
4 x = 2 * y;
```

例子

a code fragment

```
1 if y < 0 then
2 |   y = 0;
3 end
4 x = 2 * y;
```

⇒

single assignment version

```
1 if y < 0 then
2 |   y1 = 0;
3 else
4 |   y1 = y;
5 end
6 x = 2 * y1;
```

例子

a code fragment

```
1 if y < 0 then
2 |   y = 0;
3 end
4 x = 2 * y;
```

⇒

single assignment version

```
1 if y < 0 then
2 |   y1 = 0;
3 else
4 |   y1 = y;
5 end
6 x = 2 * y1;
```

$$(y < 0 \Rightarrow y1 = 0) \wedge (y \geq 0 \Rightarrow y1 = y) \wedge (x = 2 * y1)$$

无循环过程

Example (顺序查找的递归版本)

Algorithm SeqSearchRec($E[], m, num, K$)

```
1 if m ≥ num then
2 |   ans ← −1;
3 else if E[m] = K then
4 |   ans ← m;
5 else
6 |   ans ← SeqSearchRec(E, m + 1, num, K);
7 end
8 return ans;
```

return

循环转换成递归

准备工作：

1. 将循环内部的局部变量转换为单赋值形式
2. 对所有在循环内需要更新的变量 (通常是定义在循环外的)，将所有更新工作放到循环体末尾进行

转换步骤：

1. 循环体内更新的变量变为递归过程的输入参数，其初始值对应于顶层调用递归过程时实际传入的参数值 (active parameters)
2. 在循环外定义但在循环内只访问不更新的变量也变为递归过程的输入参数，在递归调用过程中其值从不改变 (passive parameters)
3. 递归过程起始处检测循环条件，若为假则返回结果 (退出)，若为真则执行循环体内容；循环体中的 break 语句也对应于递归过程的返回
4. 当转换至循环体结束时，调用递归过程本身，其实际调用参数即为循环内更新后的变量值

例子

Example (阶乘函数)

Algorithm FactLoop(n)

```
1 k ← 1;
2 f ← 1;
3 while k ≤ n do
4 |   fnew ← f * k;
5 |   knew ← k + 1;
6 |   k ← knew;
7 |   f ← fnew;
8 end
9 return f;
```

转换结果

Example (阶乘函数的递归版本)

Algorithm `Fact`(n)

```
1 return FactRec(n, 1, 1);
```

Algorithm `FactRec`(n, k, f)

```
1 if k > n then ans ← f;
2 else
3 |   fnew ← f * k;
4 |   knew ← k + 1;
5 |   ans ← FactRec(n, knew, fnew);
6 end
7 return ans;
```

证明实例

Example (二分查找的规范化递归版本)

Algorithm `BinarySearchRec`$(E[], first, last, K)$

```
1 if last < first then index ← −1;
2 else
3 |   mid ← (first + last)/2;
4 |   if K = E[mid] then index ← mid;
5 |   else if K < E[mid] then
6 |   |   index ← BinarySearchRec(E, first, mid − 1, K);
7 |   else index ← BinarySearchRec(E, mid + 1, last, K);
8 end
9 return index;
```

return

证明

Lemma

当 $\texttt{BinarySearchRec}(E, first, last, K)$ 被调用，其问题规模为 $last - first + 1 = n$，且 $E[first], \cdots, E[last]$ 以不减序排列，则对任意 $n \geq 0$，如果 K 不在 $E[first], \cdots, E[last]$ 中，过程调用返回 -1，否则返回 $index$ 满足 $K = E[index]$.

Proof.

证明

Lemma

当 $\texttt{BinarySearchRec}(E, first, last, K)$ 被调用，其问题规模为 $last - first + 1 = n$，且 $E[first], \cdots, E[last]$ 以不减序排列，则对任意 $n \geq 0$，如果 K 不在 $E[first], \cdots, E[last]$ 中，过程调用返回 -1，否则返回 $index$ 满足 $K = E[index]$.

Proof.

对问题规模 n 归纳：

奠基：当 $n = 0$ 时，$last = first - 1$，第 1 行条件满足，返回值为 -1，结论成立；

归纳：当 $n > 0$ 时，假设对于 $0 \leq k < n$，$\texttt{BinarySearchRec}(E, f, l, K)$ 调用结论成立，其中 $l - f + 1 = k$。则当 $k = n$ 时，因为 $n > 0$，所以第 1 行条件为假，程序执行到第 3 行，这时 $mid = \lfloor (first + last)/2 \rfloor$，所以 $first \leq mid \leq last$。

证明

Lemma

当 BinarySearchRec$(E, first, last, K)$ 被调用，其问题规模为 $last - first + 1 = n$，且 $E[first], \cdots, E[last]$ 以不减序排列，则对任意 $n \geq 0$，如果 K 不在 $E[first], \cdots, E[last]$ 中，过程调用返回 -1，否则返回 $index$ 满足 $K = E[index]$.

Proof.

若第 4 行条件为真，则返回 $index = mid$ 为 K 在序列 $E[first], \cdots, E[last]$ 中的位置，结论成立；
若第 4 行条件为假，由上述条件可知

$$(mid - 1) - first + 1 < last - first + 1 = n$$
$$last - (mid + 1) + 1 < last - first + 1 = n$$

证明

Lemma

当 BinarySearchRec$(E, first, last, K)$ 被调用，其问题规模为 $last - first + 1 = n$，且 $E[first], \cdots, E[last]$ 以不减序排列，则对任意 $n \geq 0$，如果 K 不在 $E[first], \cdots, E[last]$ 中，过程调用返回 -1，否则返回 $index$ 满足 $K = E[index]$.

Proof.

因此第 6、7 行的递归调用满足归纳假设，可以返回正确结果。
当第 5 行条件为真时，第 6 行递归调用被执行，若返回非负整数值，则问题解决，结论成立；若返回 -1，则表示 K 不在 $E[first], \cdots, E[mid-1]$ 中，而第 5 行条件为真表示 K 也不在 $E[mid], \cdots, E[last]$ 中，因此返回 -1 对当前调用也是正确的。
与之类似，可证明第 5 行条件为假时结论也成立。 □

主要内容

递推方程

- 递推方程 (Recurrence Equation)：一种递推地定义一个序列的方程式：序列的每一项定义为前一项的函数。
- 递推方程可以很自然地用来描述递归程序执行过程的资源 (时间、空间...) 使用情况 (统一称其为“开销 (cost) ”)
- 最坏情况下程序开销递推方程的建立 $T(n) =?$：
  1. 对于顺序执行的程序块，直接加上它的开销；
  2. 对于有选择分支的程序块 (非递归终止分支，nonbase cases)，加上分支中最大开销；
  3. 对于子程序调用，加上所调用子程序的开销 $T_s(n_s(n))$，其中 $n_s(n)$ 为所调用子程序的输入规模，是主程序输入规模 n 的函数；
  4. 对于递归调用，加上递归调用开销 $T(n_r(n))$，其中 $n_r(n)$ 为递归调用的规模，是主程序输入规模 n 的函数
- 如何求 $T(n)$?

例子

- 递归版的顺序查找： recall

$$\big(0 + (1 + \max(0, T(n-1))\big) + 0$$

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

- 递归版的二分查找： recall

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

常用递推方程形式

- Divide and Conquer：规模为 n 的问题被分解为 b 个规模为 n/c $(c > 1)$ 的子问题，$f(n)$ 为非递归部分 (分解和合并) 的开销，$b \geq 1$ 称为分支因子 (branching factor)

$$T(n) = bT(n/c) + f(n)$$

- Chip and Conquer：规模为 n 的问题被“裁剪”成规模为 $n - c$ 的子问题

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

- Chip and Be Conquered：规模为 n 的问题被分解并“裁剪”成 b 个规模为 $n - c$ 的子问题

$$T(n) = bT(n - c) + f(n)$$

递推方程求解

Example ($T(n) = 2T(n/2) + n \lg n$)

递推方程求解

Example ($T(n) = 2T(n/2) + n \lg n$)

$$
\begin{aligned}
T(n) &= 2T(\frac{n}{2}) + n\lg n = 2(2T(\frac{n}{4}) + \frac{n}{2}\lg\frac{n}{2}) + n\lg n \\
&= 2^2T(\frac{n}{2^2}) + n\lg\frac{n}{2} + n\lg n \\
&= 2^3T(\frac{n}{2^3}) + n\lg\frac{n}{2^2} + n\lg\frac{n}{2} + n\lg n \\
&= \cdots \\
&= 2^dT(\frac{n}{2^d}) + n(\lg\frac{n}{2^{d-1}} + \cdots + \lg\frac{n}{2^0}) \\
&= nT(1) + n(\lg n - (d-1) + \lg n - (d-2) + \cdots + \lg n - 0) \\
&= nT(1) + nd\lg n - n\frac{d(d-1)}{2} \\
&= nT(1) + \frac{n}{2}\lg^2 n + \frac{n}{2}\lg n \in \Theta(n\lg^2 n)
\end{aligned}
$$

递归树 (Recursion Tree)

Divide and Conquer

$$T(n) = 2T(n/2) + n$$

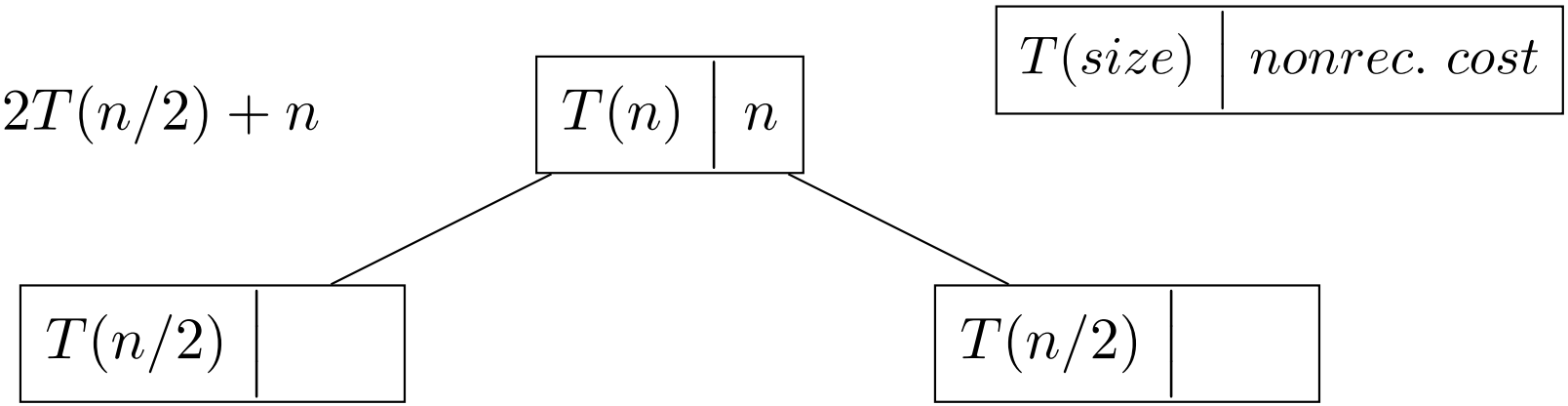
$T(n)$	

$T(size)$	$nonrec.\ cost$

递归树 (Recursion Tree)

Divide and Conquer

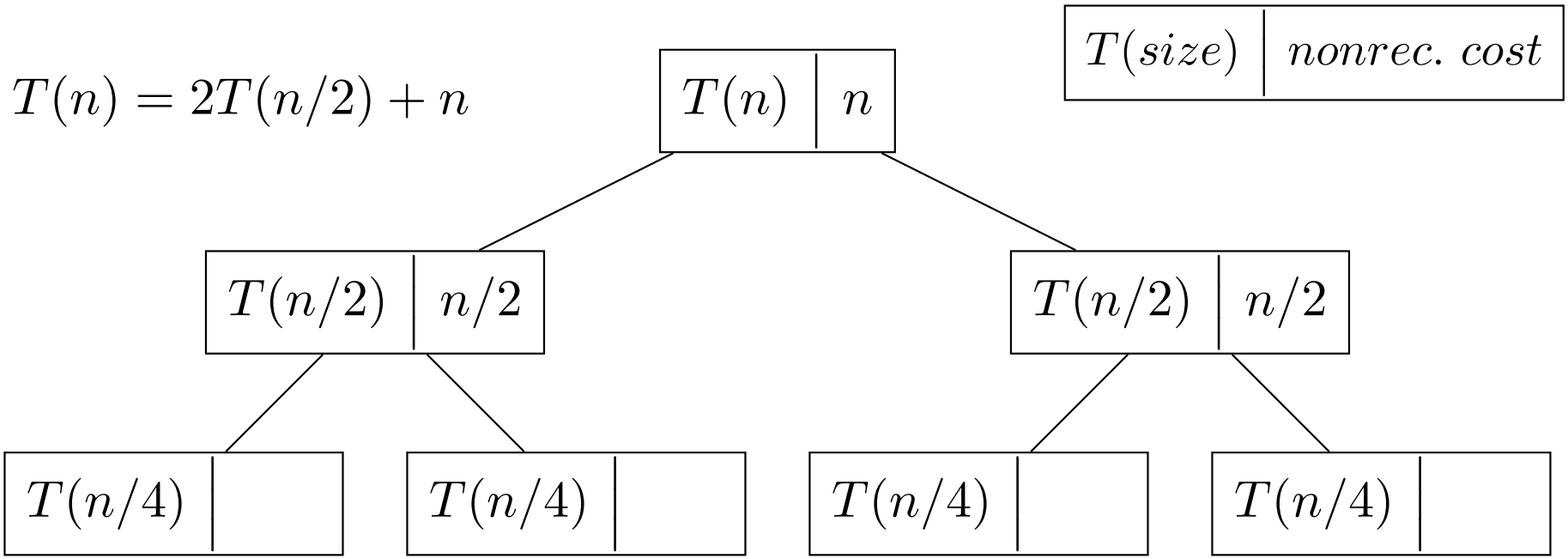
$$T(n) = 2T(n/2) + n$$



递归树 (Recursion Tree)

Divide and Conquer

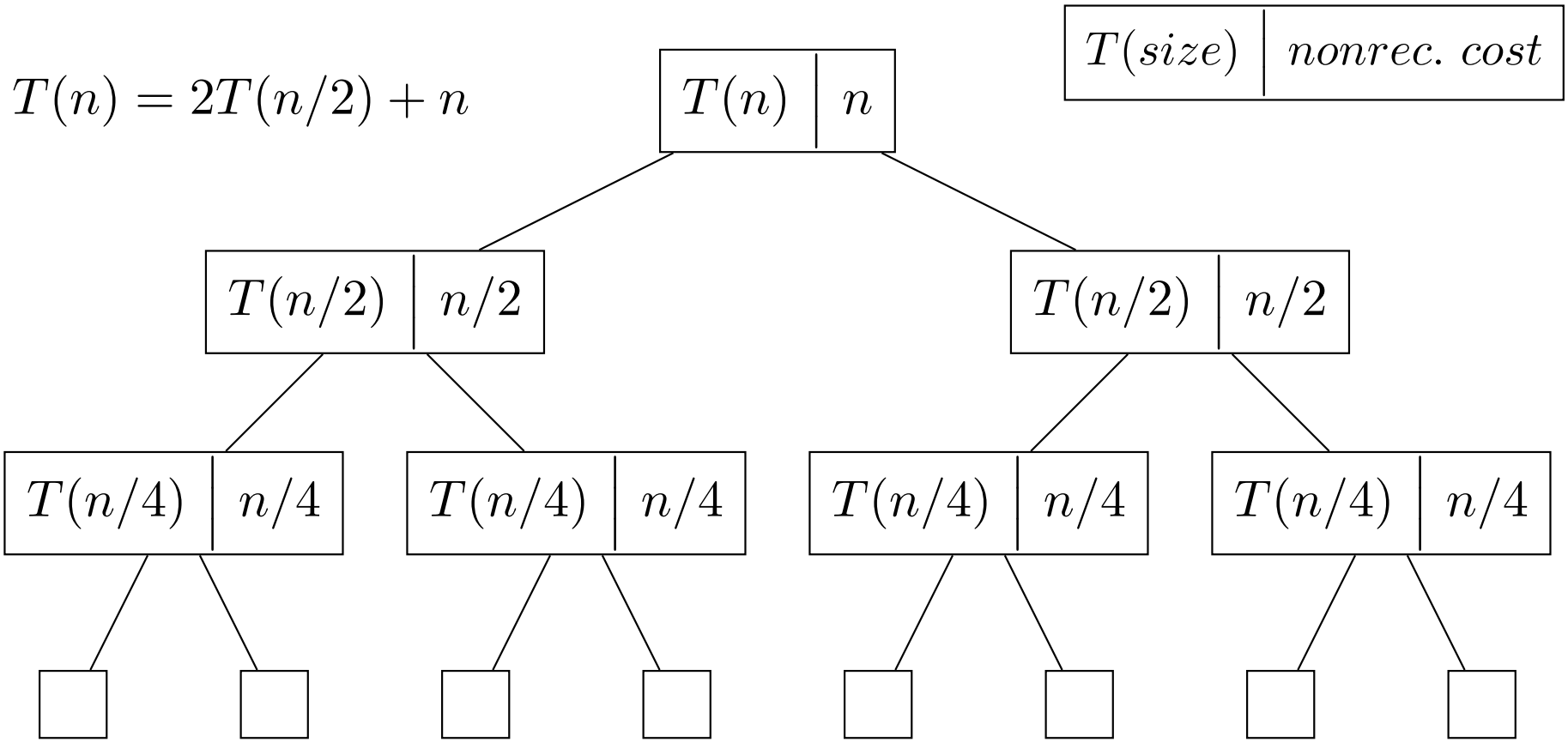
$T(n) = 2T(n/2) + n$



递归树 (Recursion Tree)

Divide and Conquer

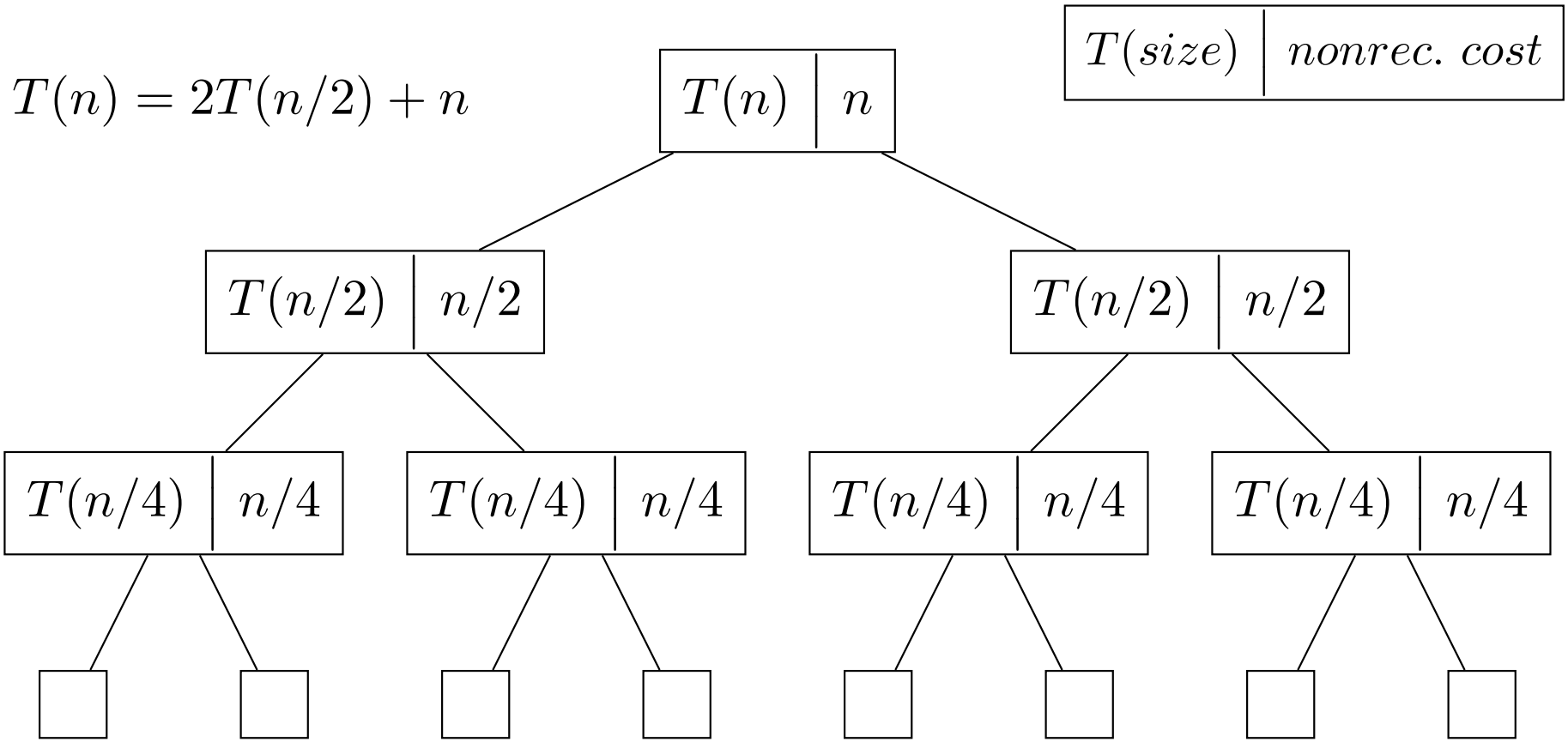
$T(n) = 2T(n/2) + n$



递归树 (Recursion Tree)

Divide and Conquer

$$T(n) = 2T(n/2) + n$$



$$n/2^d = 1 \Rightarrow T(n) \approx n \lg n$$

Divid and Conquer 一般情况

$$T(n) = bT(n/c) + f(n)$$

定义 $E = \log_c b = \dfrac{\lg b}{\lg c}$ 称为关键指数 (critical exponent)

Lemma

Divid and Conquer 递归树的叶结点个数 $L \approx n^E$

Lemma

1. 递归树的深度 (高度) $D \approx \lg n / \lg c = \log_c n$
2. 第 0 层的开销和为 $f(n)$
3. 假设递归基础的开销为 1，第 D 层的开销和为 n^E
4. $T(n)$ 等于递归树中所有结点的非递归开销总和，也就是递归树每一层“开销和”的总和

主定理

Theorem (Little Master Theorem)

对于递推方程 $T(n) = bT(n/c) + f(n)$，若 $T(1) \in \Theta(1)$ 及 $E = \log_c b$

1. 若递归树每一层的开销和呈递增几何级数，则 $T(n) \in \Theta(n^E)$
2. 若递归树每一层的开销和保持不变，则 $T(n) \in \Theta(f(n) \log n)$
3. 若递归树每一层的开销和呈递减几何级数，则 $T(n) \in \Theta(f(n))$

主定理

Theorem (Little Master Theorem)

对于递推方程 $T(n) = bT(n/c) + f(n)$，若 $T(1) \in \Theta(1)$ 及 $E = \log_c b$

1. 若递归树每一层的开销和呈递增几何级数，则 $T(n) \in \Theta(n^E)$
2. 若递归树每一层的开销和保持不变，则 $T(n) \in \Theta(f(n) \log n)$
3. 若递归树每一层的开销和呈递减几何级数，则 $T(n) \in \Theta(f(n))$

Theorem (Master Theorem)

对于递推方程 $T(n) = bT(n/c) + f(n)$，若 $T(1) \in \Theta(1)$ 及 $E = \log_c b$

1. 若对常数 $\epsilon > 0$，$f(n) \in O(n^{E-\epsilon})$，则 $T(n) \in \Theta(n^E)$
2. 若 $f(n) \in \Theta(n^E)$，则 $T(n) \in \Theta(n^E \log n)$
3. 若对常数 $\epsilon > 0$，$f(n) \in \Omega(n^{E+\epsilon})$，并且存在常数 $\delta < 1$ 以及整数 n_0，使得当 $n > n_0$ 时有 $bf(n/c) \leq \delta f(n)$，则 $T(n) \in \Theta(f(n))$

主定理的证明

证明要点.

递归树中在第 d 层 (深度为 d) 有 b^d 个结点，每个非叶结点的非递归开销为 $f(n/c^d)$，因此有

$$T(n) = n^{\log_c b}\Theta(1) + \sum_{d=0}^{\log_c n-1} b^d f\left(\frac{n}{c^d}\right)$$

令 $g(n) = \sum_{d=0}^{\log_c n-1} b^d f\left(\frac{n}{c^d}\right)$，则：

1. 在条件 1 下，有 $g(n) \in O(n^E)$
2. 在条件 2 下，有 $g(n) \in \Theta(n^E \log n)$
3. 在条件 3 下，有 $g(n) \in \Theta(f(n))$

主定理的应用

Example $(T(n) = 9T(n/3) + n)$

$\log_c b = \log_3 9 = 2$，取 $\epsilon = 0.5$，则有 $f(n) = n \in O(n^{2-0.5}) = O(2^{1.5})$，所以，$T(n) \in \Theta(n^2)$

主定理的应用

Example $(T(n) = 9T(n/3) + n)$

$\log_c b = \log_3 9 = 2$，取 $\epsilon = 0.5$，则有 $f(n) = n \in O(n^{2-0.5}) = O(2^{1.5})$，
所以，$T(n) \in \Theta(n^2)$

Example $(T(n) = T(2n/3) + 1)$

$\log_c b = \log_{3/2} 1 = 0$，故 $f(n) = 1 \in \Theta(n^0)$，
所以 $T(n) \in \Theta(n^0 \log n) = \Theta(\log n)$

主定理的应用

Example $(T(n) = 9T(n/3) + n)$

$\log_c b = \log_3 9 = 2$，取 $\epsilon = 0.5$，则有 $f(n) = n \in O(n^{2-0.5}) = O(2^{1.5})$，
所以，$T(n) \in \Theta(n^2)$

Example $(T(n) = T(2n/3) + 1)$

$\log_c b = \log_{3/2} 1 = 0$，故 $f(n) = 1 \in \Theta(n^0)$，
所以 $T(n) \in \Theta(n^0 \log n) = \Theta(\log n)$

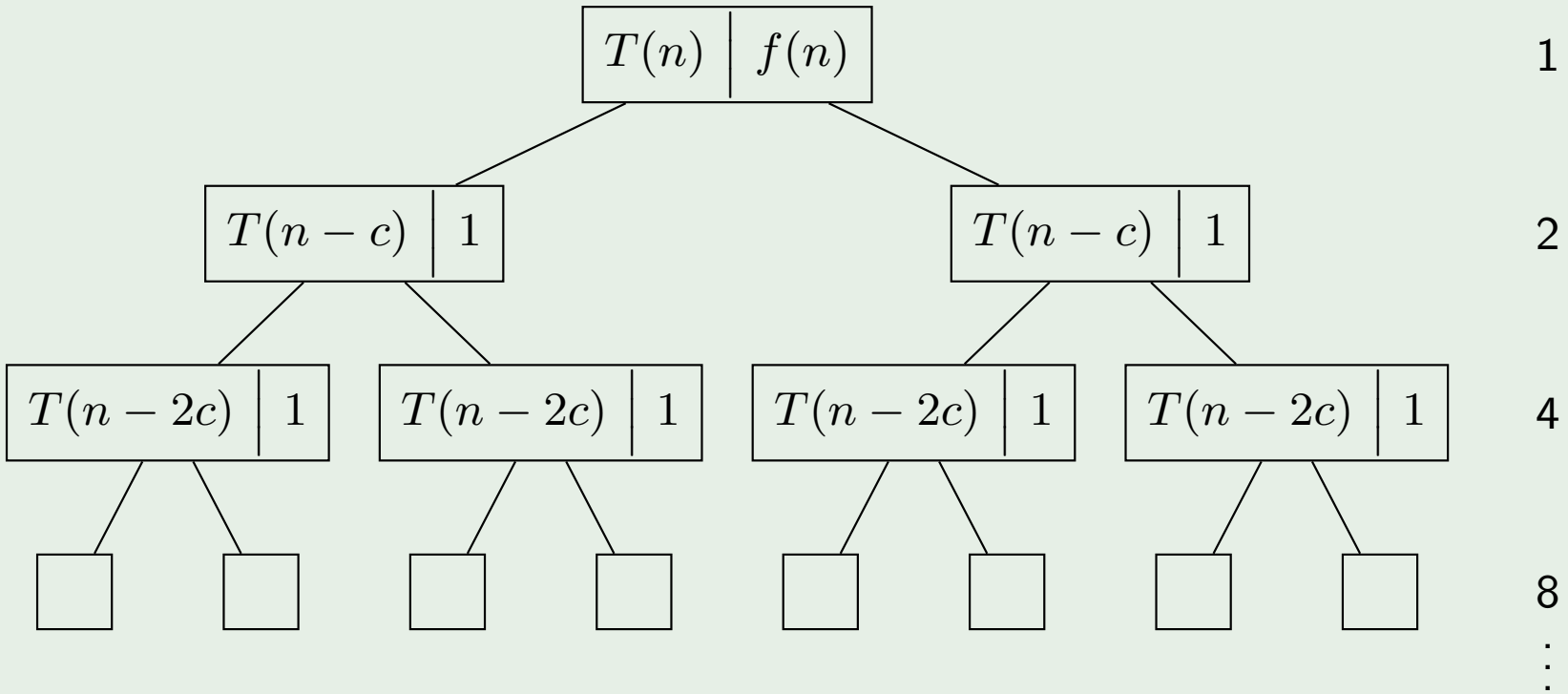
Example $(T(n) = 3T(n/4) + n \lg n)$

$\log_c b = \log_4 3 < 1$，故存在 $\epsilon > 0$ 使 $\log_4 3 + \epsilon < 1$，
则 $f(n) = n \lg n \in \Omega(n) \subset \Omega(n^{\log_4 3 + \epsilon})$；
另外，$bf(n/c) = 3(n/4)\lg(n/4) = \frac{3n}{4}(\lg n - 2) \leq \frac{3}{4} f(n)$，
所以 $T(n) \in \Theta(n \lg n)$

递归树

Chip and Conquer (Be Conquered)

Example ($T(n) = 2T(n-c) + 1$)



Chip and Conquer (Be Conquered)

- $T(n) = bT(n-c) + f(n)$
- 递归树深度大约为 $d = n/c$
- $T(n) \doteq \sum_{d=0}^{n/c} b^d f(n-cd) = b^{n/c} \sum_{h=0}^{n/c} \frac{f(ch)}{b^h} = b^{n/c} g(n), \quad h = (n/c) - d$
- 在大多数实际情况下，$g(n) \in \Theta(1)$，这时 $T(n) \in \Theta(b^{n/c})$，是一个指数阶的函数！
- 若 $b = 1$：$T(n) \doteq \sum_{h=0}^{n/c} f(ch) \approx \frac{1}{c} \int_0^n f(x)dx$
  - $f(n) = n^\alpha$：$T(n) \in \Theta(n^{\alpha+1})$
  - $f(n) = \log n$：$T(n) \in \Theta(n \log n)$

主要内容

分治法 (Divide and Conquer)

- 工作原理：“解决几个小问题通常比解决一个大问题来得简单”
- 设计思想：将一个直接难以解决的大问题分成较小规模的数个子问题；分别解决 (治) 这些子问题；将子问题结果合成原问题的结果
  - 由分治法产生的子问题往往是原问题的较小模式，在这种情况下，反复应用分治手段，可以使子问题与原问题类型一致而其规模却不断缩小，最终使子问题缩小到很容易直接求出其解，这自然导致递归过程的产生
- 适用条件：
  1. 问题的规模缩小到一定的程度就可以容易地解决；
  2. 问题可以分解为若干个规模较小的相同问题；
  3. 利用该问题分解出的子问题的解可以合并为该问题的解；
  4. 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子问题 (效率)
- 使用分治法的例子：`BinarySearchRec`

算法基本结构

Algorithm `Solve`(I)

```
1 n ← Size(I);
2 if n ≤ smallSize then
3 |   solution ← DirectlySolve(I);
4 else
5 |   divide I into I_1, ··· , I_k;
6 |   foreach i ∈ {1, ··· , k} do  S_i ← Solve(I_i) ;
7 |   solution ← Combine(S_1, ··· , S_k);
8 end
9 return solution
```

时间复杂度：$T(n) = D(n) + \sum_{i=1}^{k} T(\texttt{Size}(I_i)) + C(n)$

主要内容

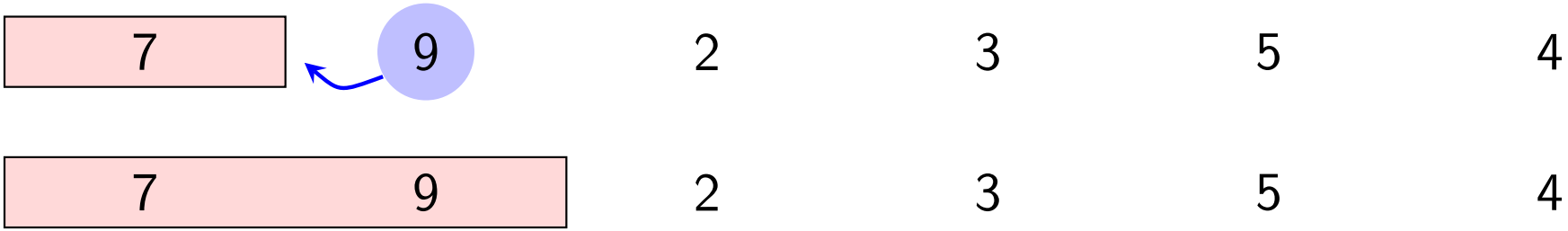
主要内容

基本策略

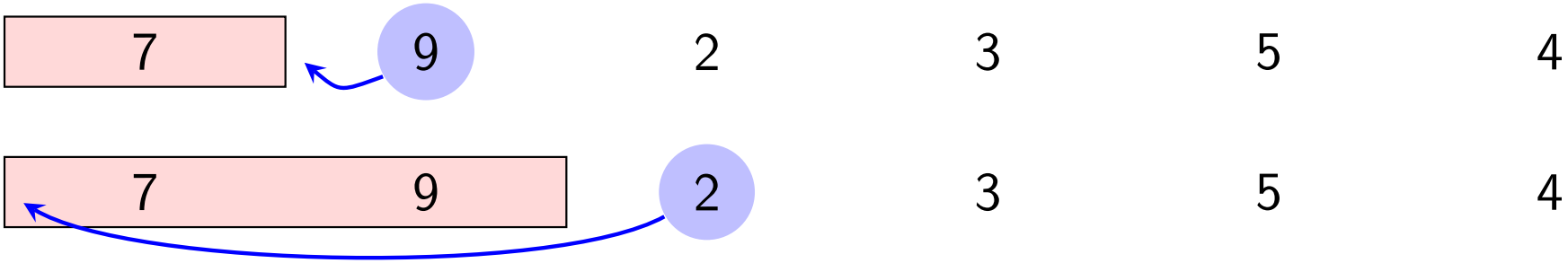
7	9	2	3	5	4

基本策略

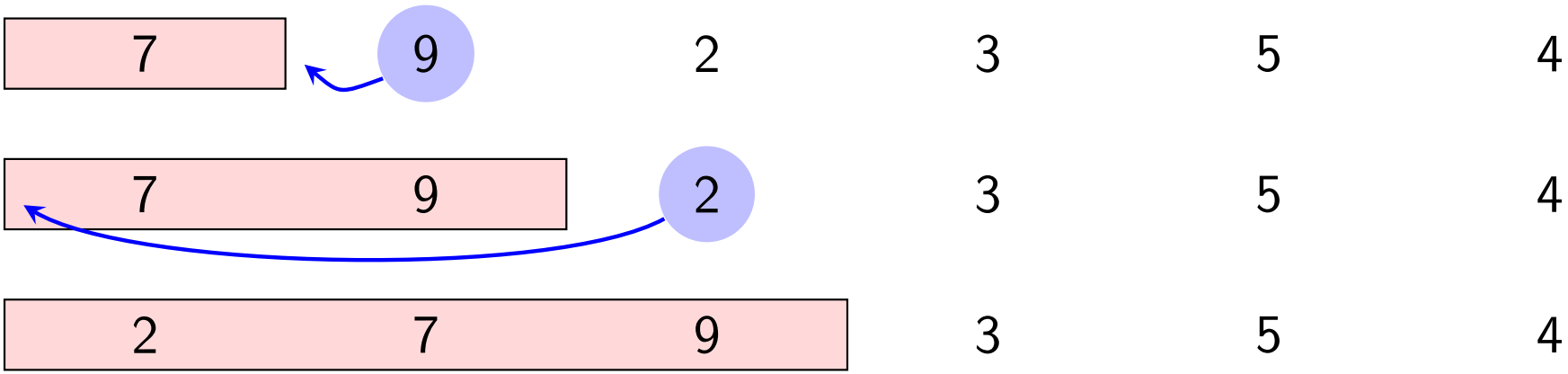
基本策略



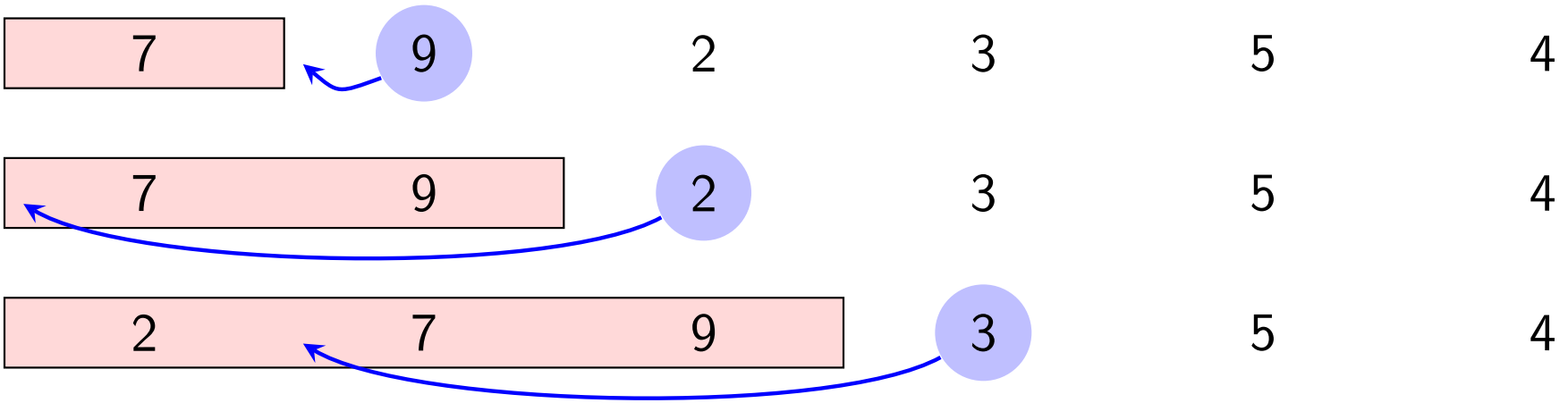
基本策略



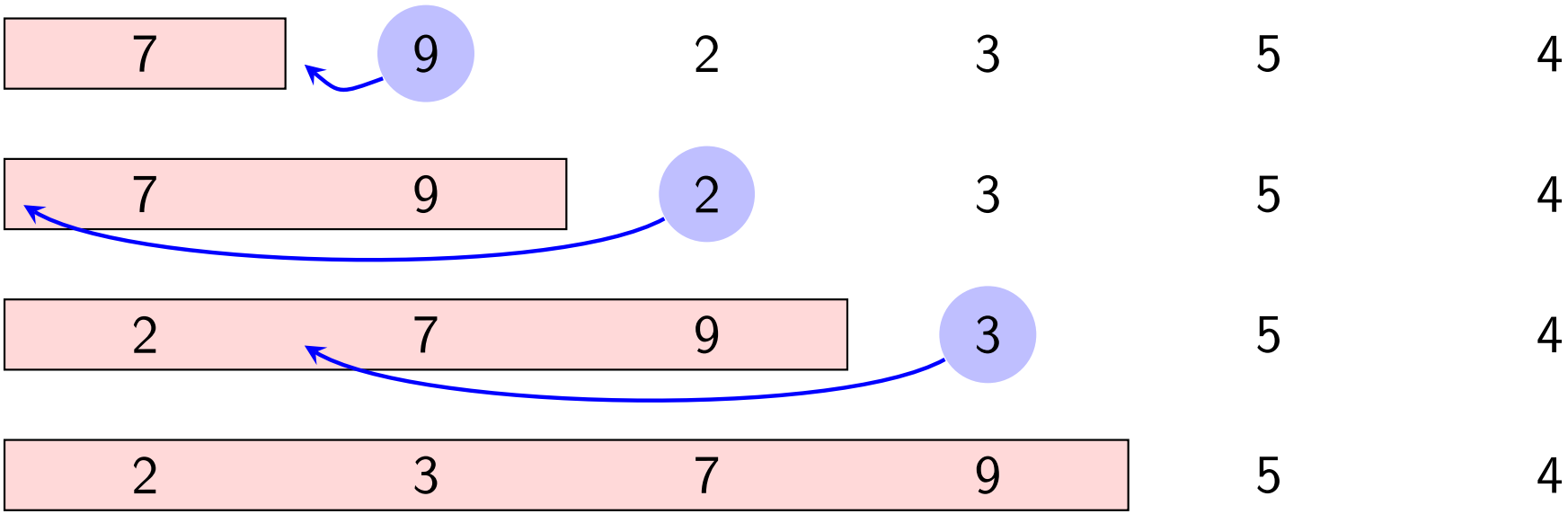
基本策略



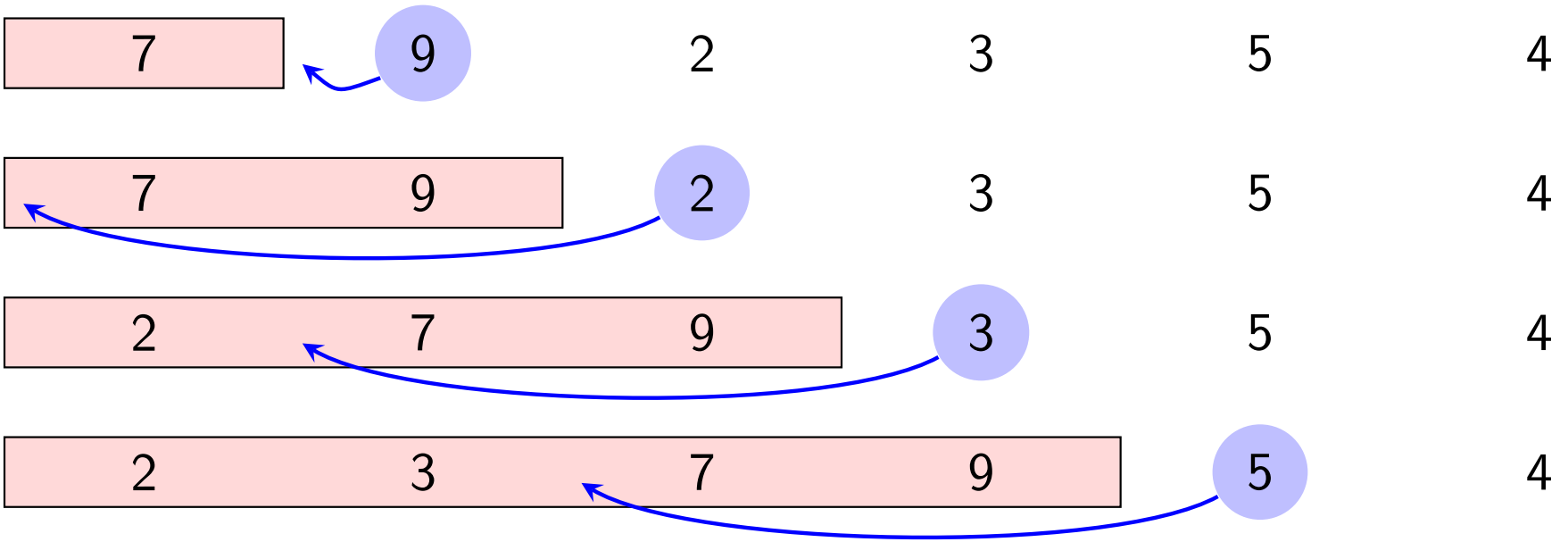
基本策略



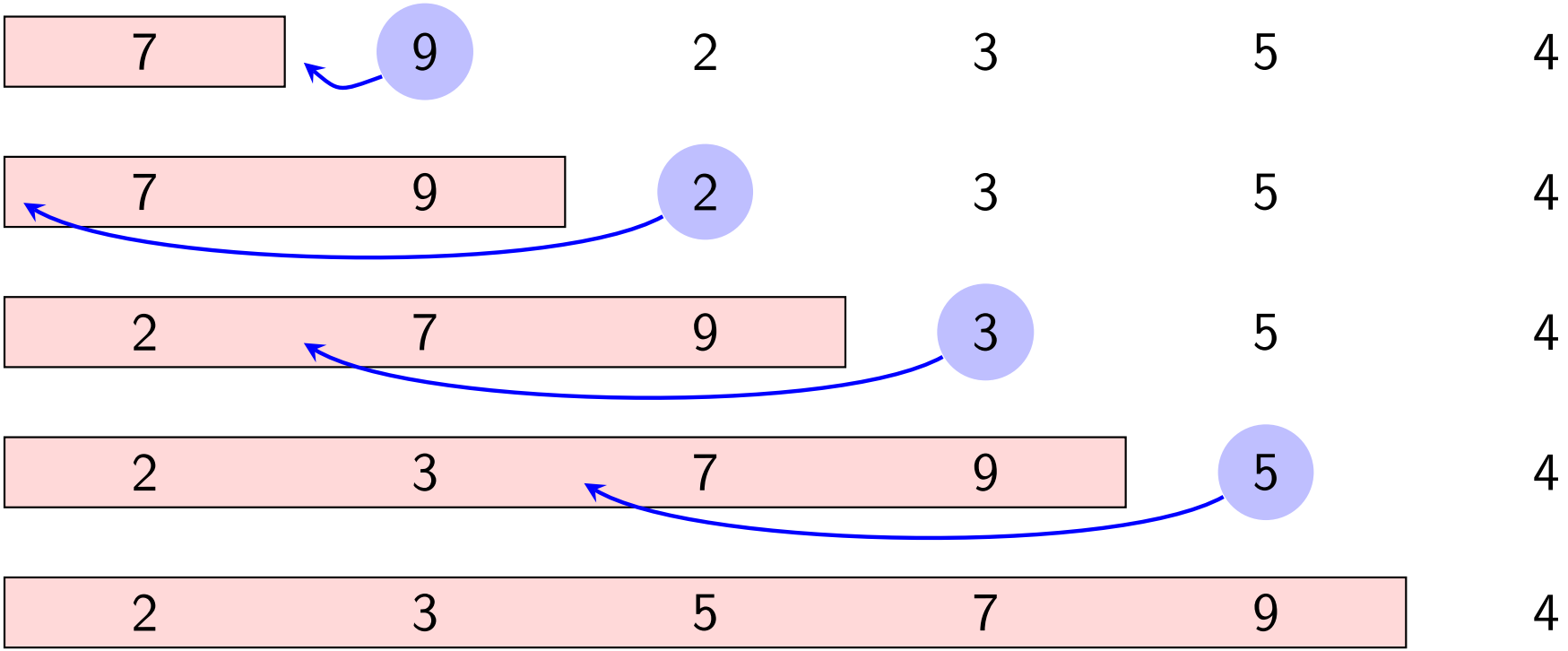
基本策略



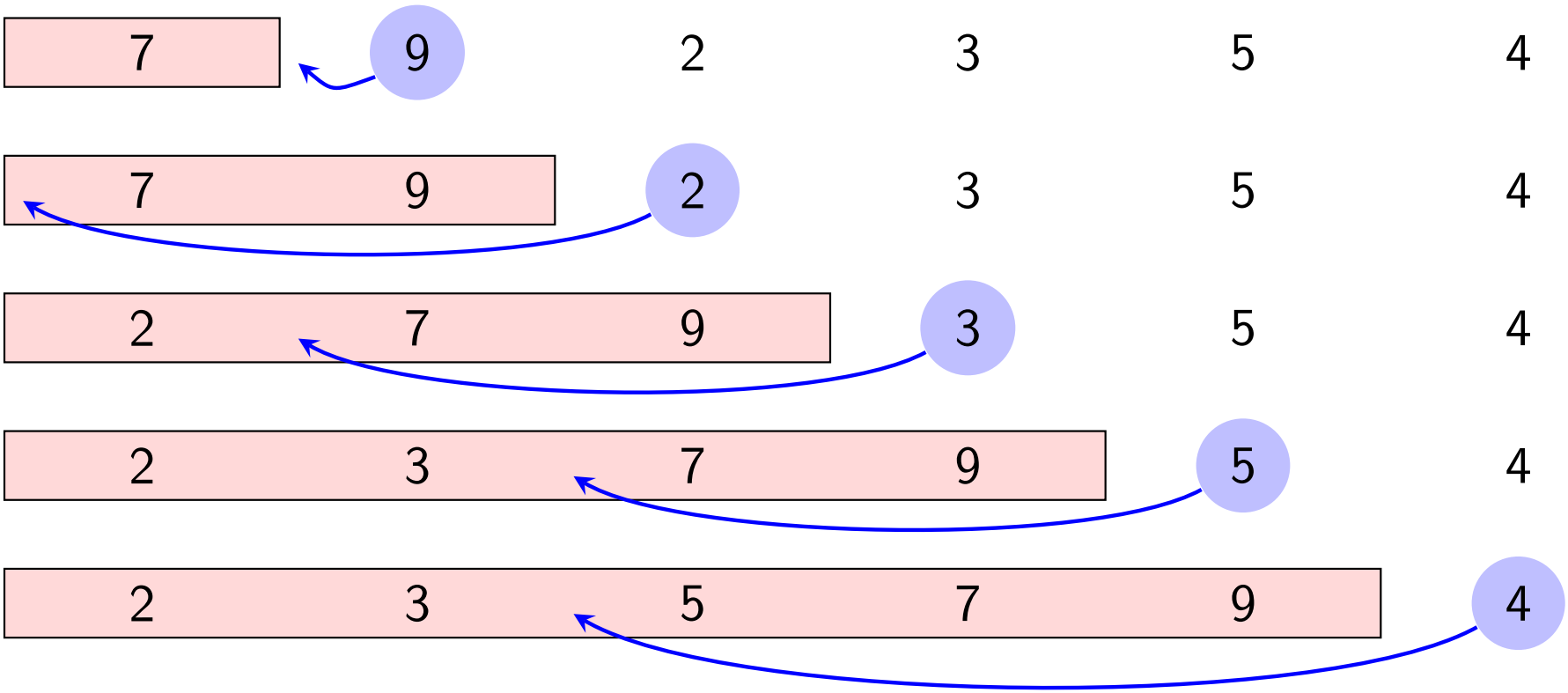
基本策略



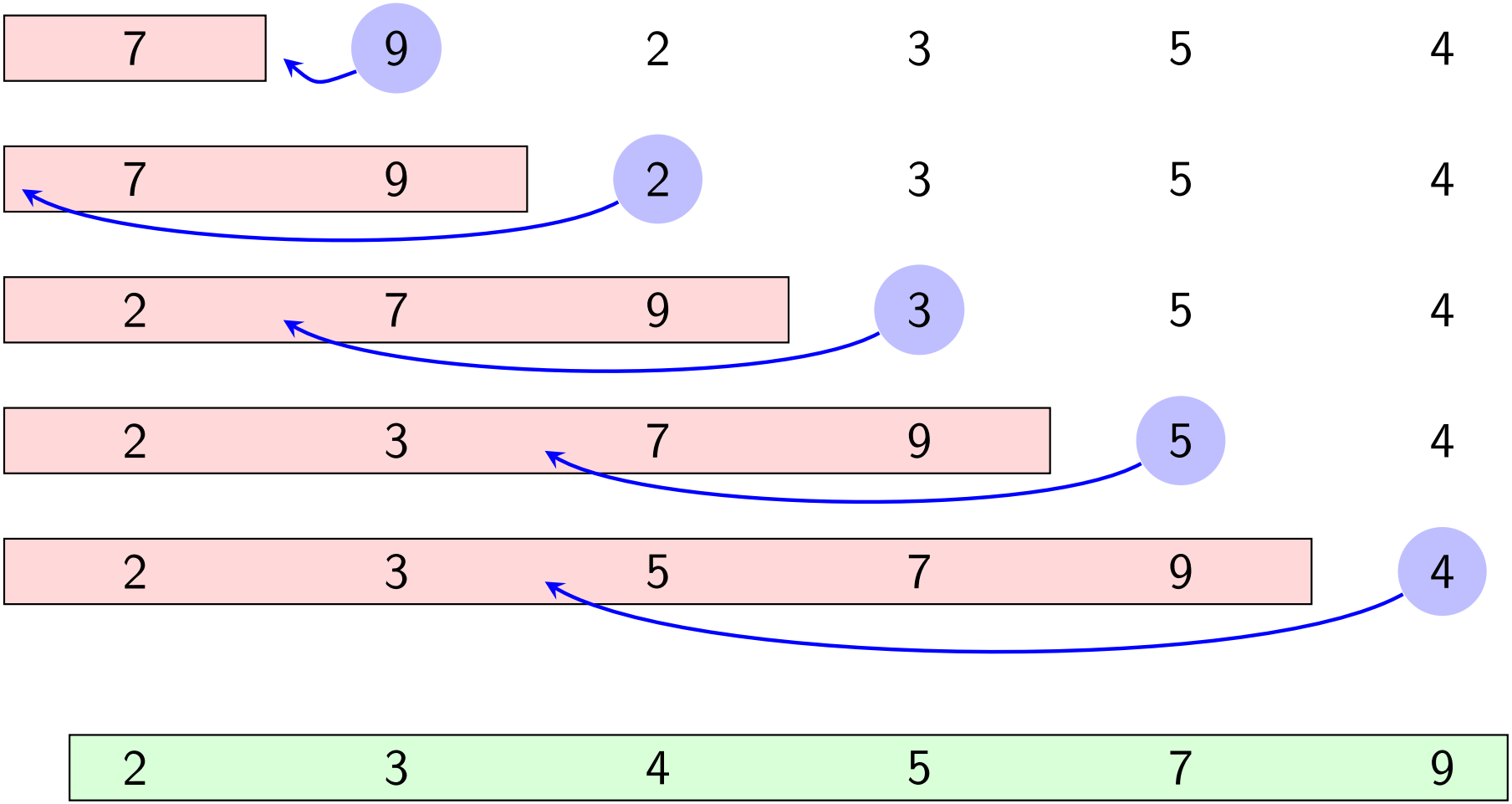
基本策略



基本策略



基本策略



算法

Algorithm `InsertionSort`$(E[], n)$

```
1 for xindex ← 1 to n − 1 do
2 |   current ← E[xindex];
3 |   xloc ← ShiftVac(E, xindex, current);
4 |   E[xLoc] ← current;
5 end
```

ShiftVac

Algorithm `ShiftVac`$(E[], xindex, x)$

```
1  vacant ← xindex;
2  xloc ← 0;
3  while vacant > 0 do
4      if E[vacant − 1] ≤ x then
5          xloc ← vacant;
6          break;
7      end
8      E[vacant] ← E[vacant − 1];
9      vacant ← vacant − 1;
10 end
11 return xloc;
```

复杂度分析

- 基本操作：元素比较
- Worst-Case：
  - $W(n) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \in \Theta(n^2)$
- Average-Case：
  - 不妨假设序列元素各不相同
  - 对于 ShiftVac，将当前第 i 个待排序元素插入前 i 个排好序的元素序列中，共有 $i+1$ 个可能的插入位置，在每个位置插入的可能性为 $1/(i+1)$，则平均情况下比较次数为

$$\frac{1}{i+1}\sum_{j=1}^{i} j + \frac{i}{i+1} = \frac{i}{2} + 1 - \frac{1}{i+1}$$

  - InsertionSort 总共进行了 $n-1$ 次插入操作，比较次数总和为

$$A(n) = \sum_{i=1}^{n-1}\left(\frac{i}{2} + 1 - \frac{1}{i+1}\right) = \frac{n(n-1)}{4} + n - 1 - \sum_{j=2}^{n}\frac{1}{j} \approx \frac{n^2}{4} \in \Theta(n^2)$$

类似算法的复杂度下界

- 类似算法：每次比较之后仅交换一对相邻元素位置的排序算法
- 设原序列 $E = (x_1, x_2, \cdots, x_n)$，定义 π 为原序列的一种排列，即对于 $1 \leq i \leq n$，$\pi(i)$ 是 x_i 在排好序的序列中的实际位置
- 排列中的反序对 $(\pi(i), \pi(j))$：$i < j$ 且 $\pi(i) > \pi(j)$
- 上述算法的最少比较次数等于输入序列中反序对的个数
- 有一种排列其反序对个数为 $n(n-1)/2$，因此上述算法最坏情况下复杂度下界为 $n(n-1)/2 \in \Omega(n^2)$
- 平均情况？

Theorem

任何使用比较为基本操作，并且每次比较后最多去除一个反序对的排序算法，输入规模为 n 时，在最坏情况下的最少比较次数为 $n(n-1)/2$，在平均情况下的最少比较次数为 $n(n-1)/4$

另一个类似算法：选择排序

Algorithm `SelectionSort`$(E[], n)$

```
1 for i ← 0 to n − 1 do
2 |   k ← i;
3 |   for j ← i to n − 1 do
4 |   |   if E[j] < E[k] then k ← j;
5 |   end
6 |   E[i] ↔ E[k];
7 end
```

- Worst-Case：$W(n) = n(n-1)/2$
- Average-Case：$A(n) = n(n-1)/2$
- 优点：不需频繁移动数组元素

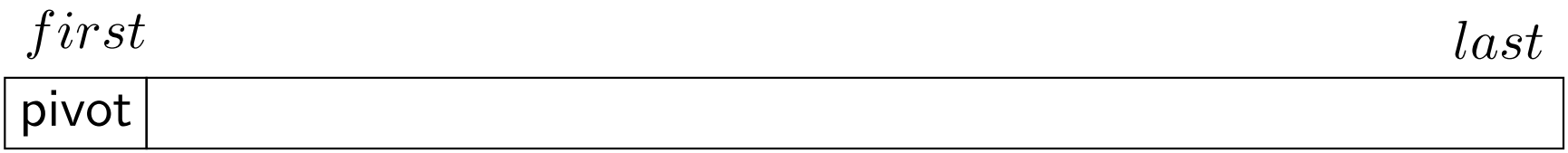
主要内容

基本策略

- 将原序列分解为两个子序列，使一个子序列里面的元素小于另一个子序列中的元素
- 对两个子序列分别排序 (递归求解)
- C. A. R. Hoare 在 1962 年提出

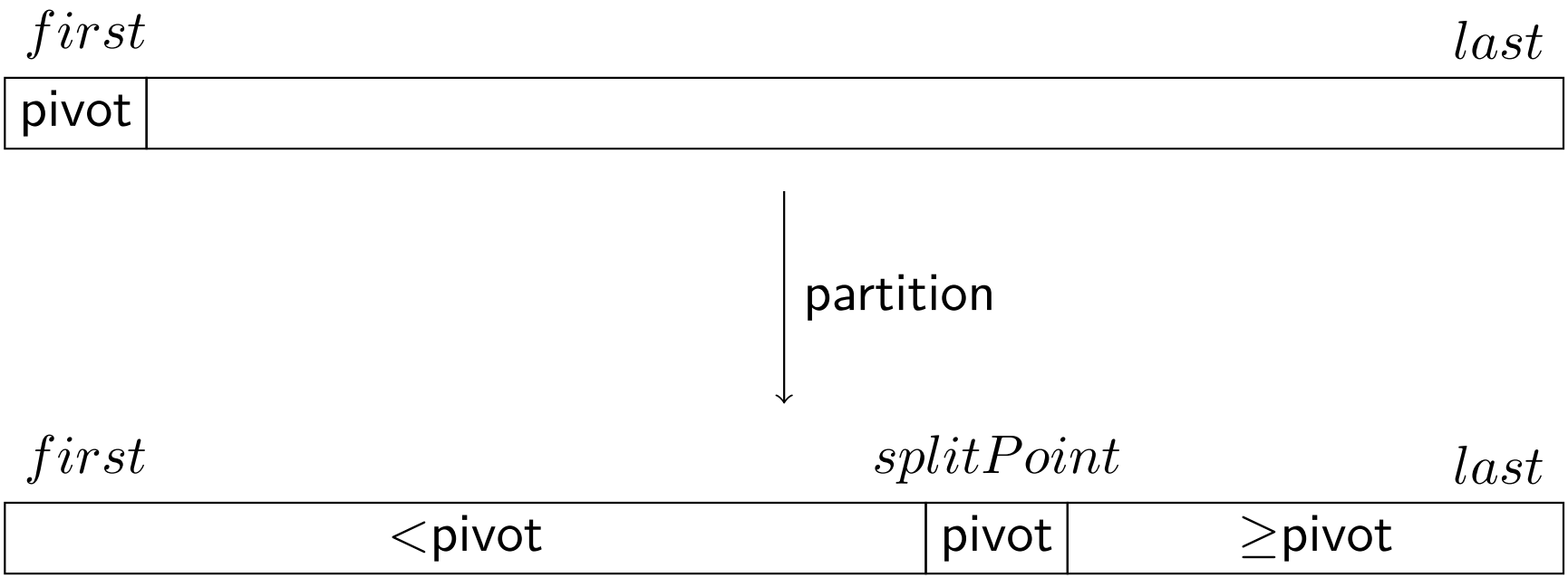
基本策略

- 将原序列分解为两个子序列，使一个子序列里面的元素小于另一个子序列中的元素
- 对两个子序列分别排序 (递归求解)
- C. A. R. Hoare 在 1962 年提出



基本策略

- 将原序列分解为两个子序列，使一个子序列里面的元素小于另一个子序列中的元素
- 对两个子序列分别排序 (递归求解)
- C. A. R. Hoare 在 1962 年提出



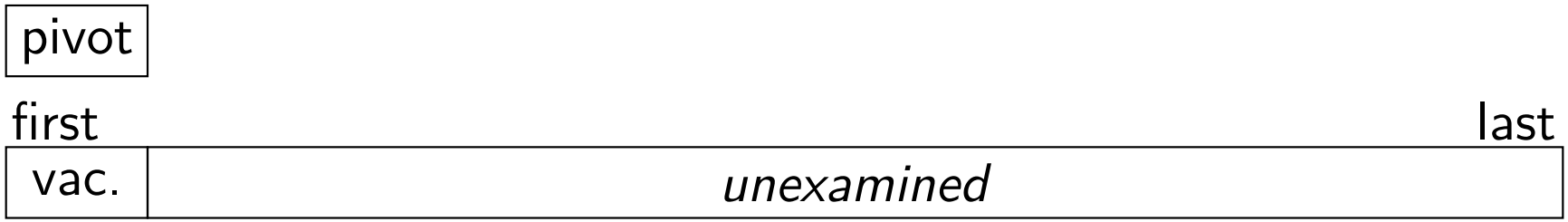
QuickSort 算法描述

Algorithm QuickSort($E[], first, last$)

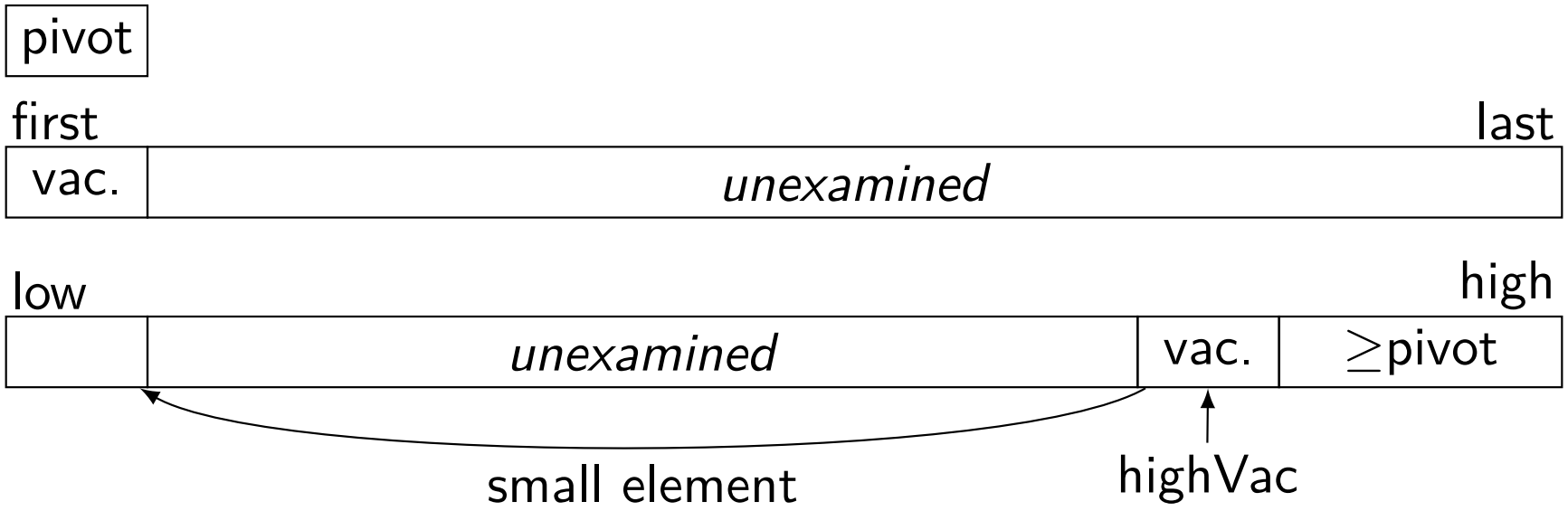
```
1 if first < last then
2 |   pivot ← E[first];
3 |   splitPoint ← Partition(E, pivot, first, last);
4 |   E[splitPoint] ← pivot;
5 |   QuickSort(E, first, splitPoint − 1);
6 |   QuickSort(E, splitPoint + 1, last);
7 end
```

In Place Partition

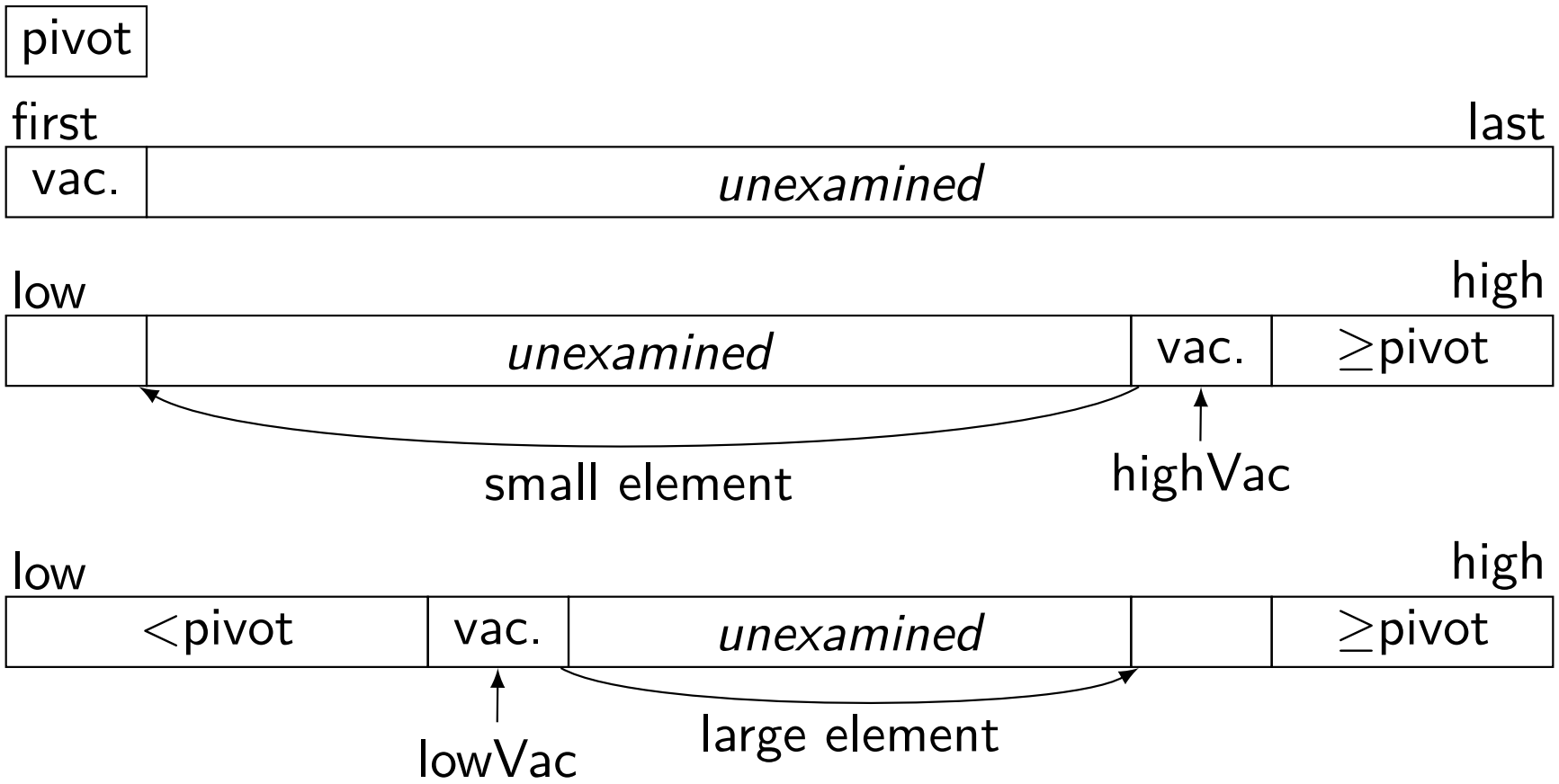
In Place Partition



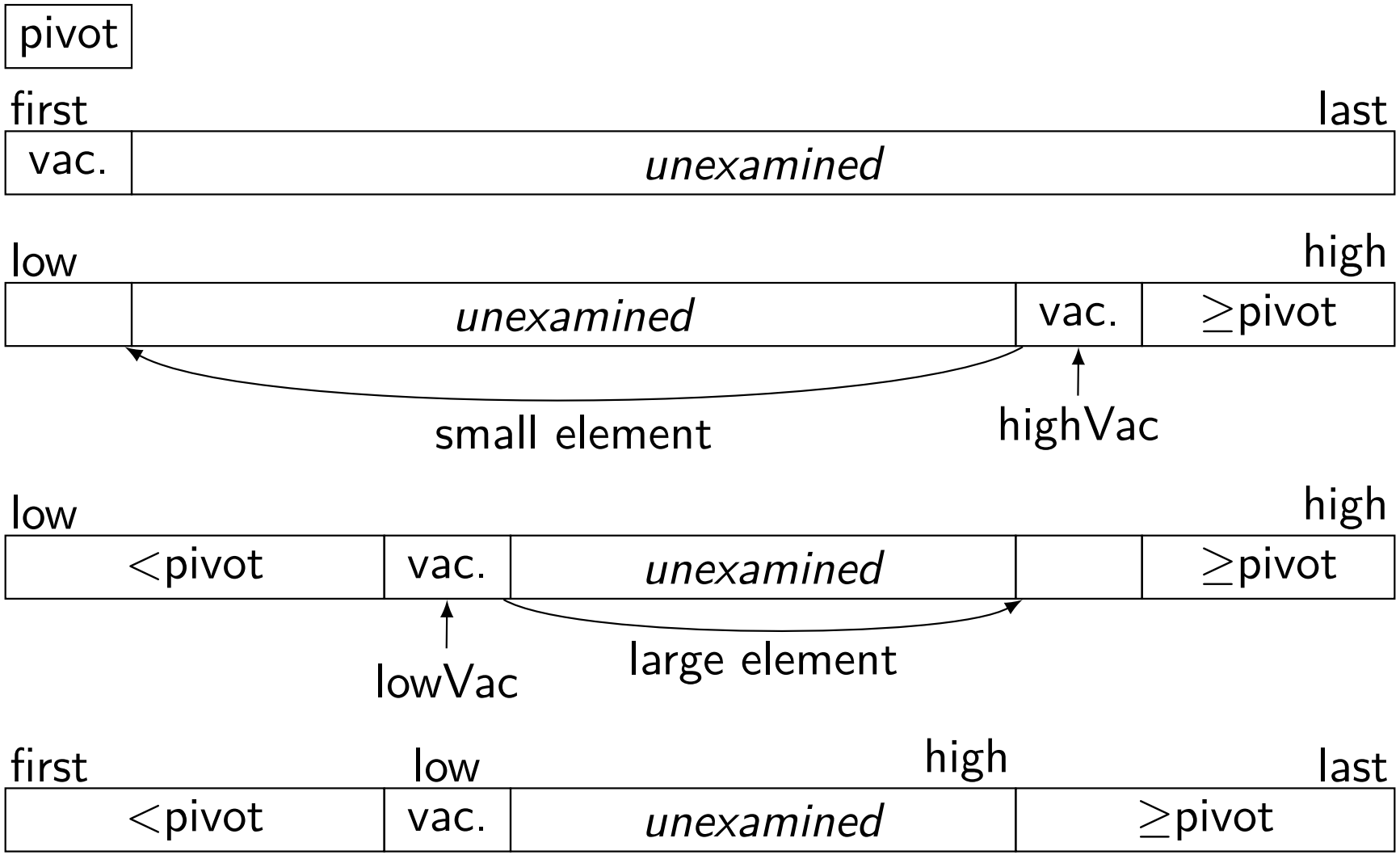
In Place Partition



In Place Partition



In Place Partition



Partition

Procedure `Partition`$(E[], pivot, first, last)$

```
1 low ← first;
2 high ← last;
3 while low < high do
4 |   highVac ← ExtendLargeRegion(E, pivot, low, high);
5 |   lowVac ←
  |   ExtendSmallRegion(E, pivot, low + 1, highVac);
6 |   low ← lowVac;
7 |   high ← highVac − 1;
8 end
9 return low;
```

ExtendLargeRegion

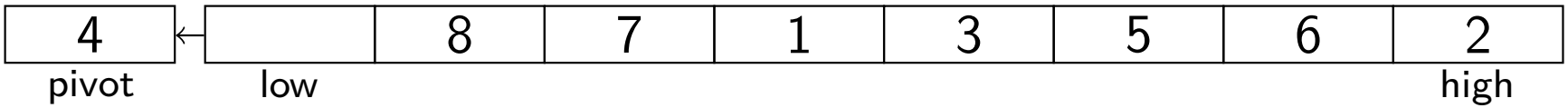
Procedure ExtendLargeRegion($E[], pivot, lowVac, high$)

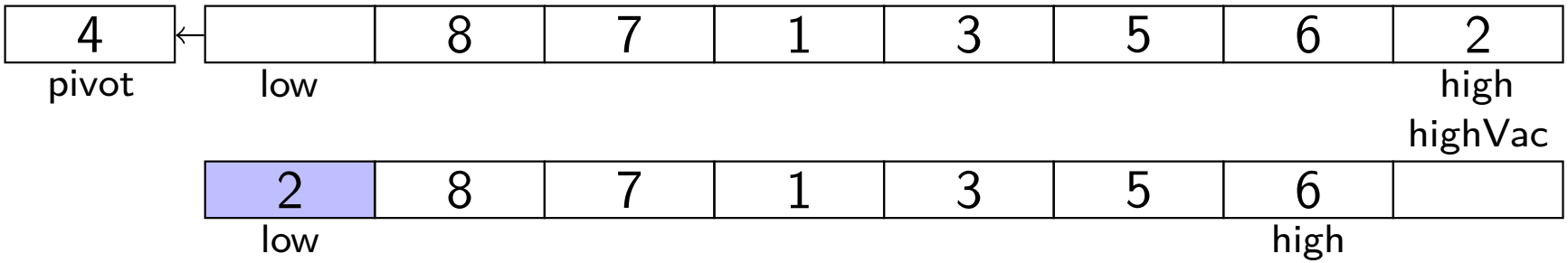
```
1  highVac ← lowVac;        // In case no element < pivot.
2  curr ← high;
3  while curr > lowVac do
4  |   if E[curr] < pivot then
5  |   |   E[lowVac] ← E[curr];
6  |   |   highVac ← curr;
7  |   |   break;
8  |   end
9  |   curr ← curr − 1;
10 end
11 return highVac;
```

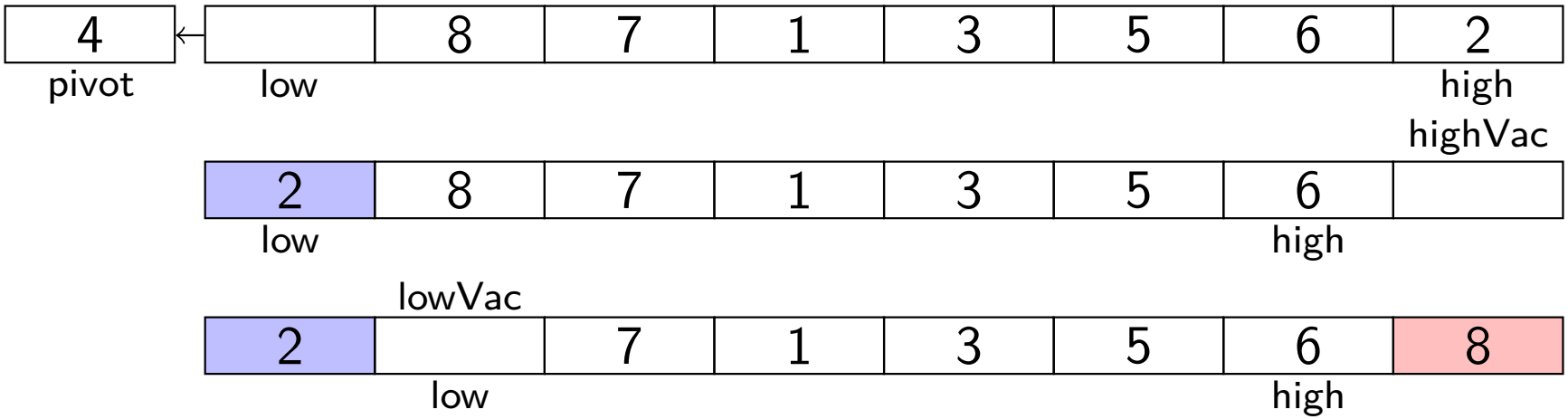
ExtendSmallRegion

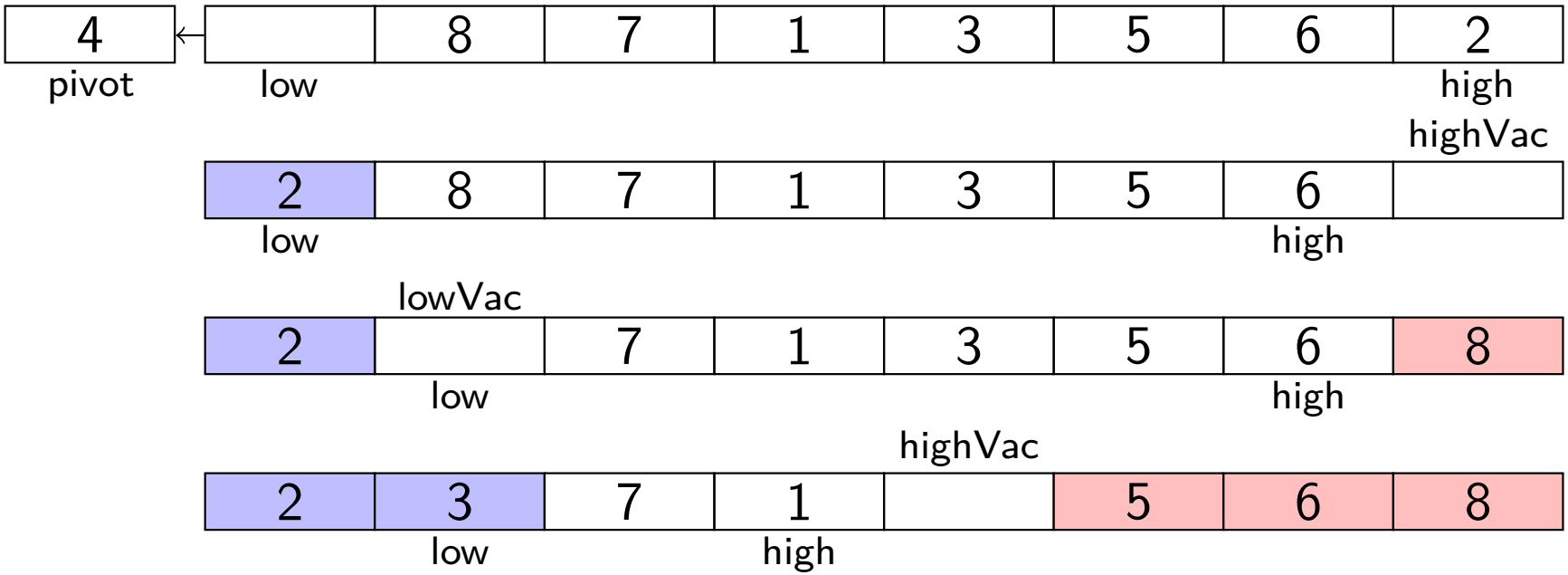
Procedure ExtendSmallRegion($E[], pivot, low, highVac$)

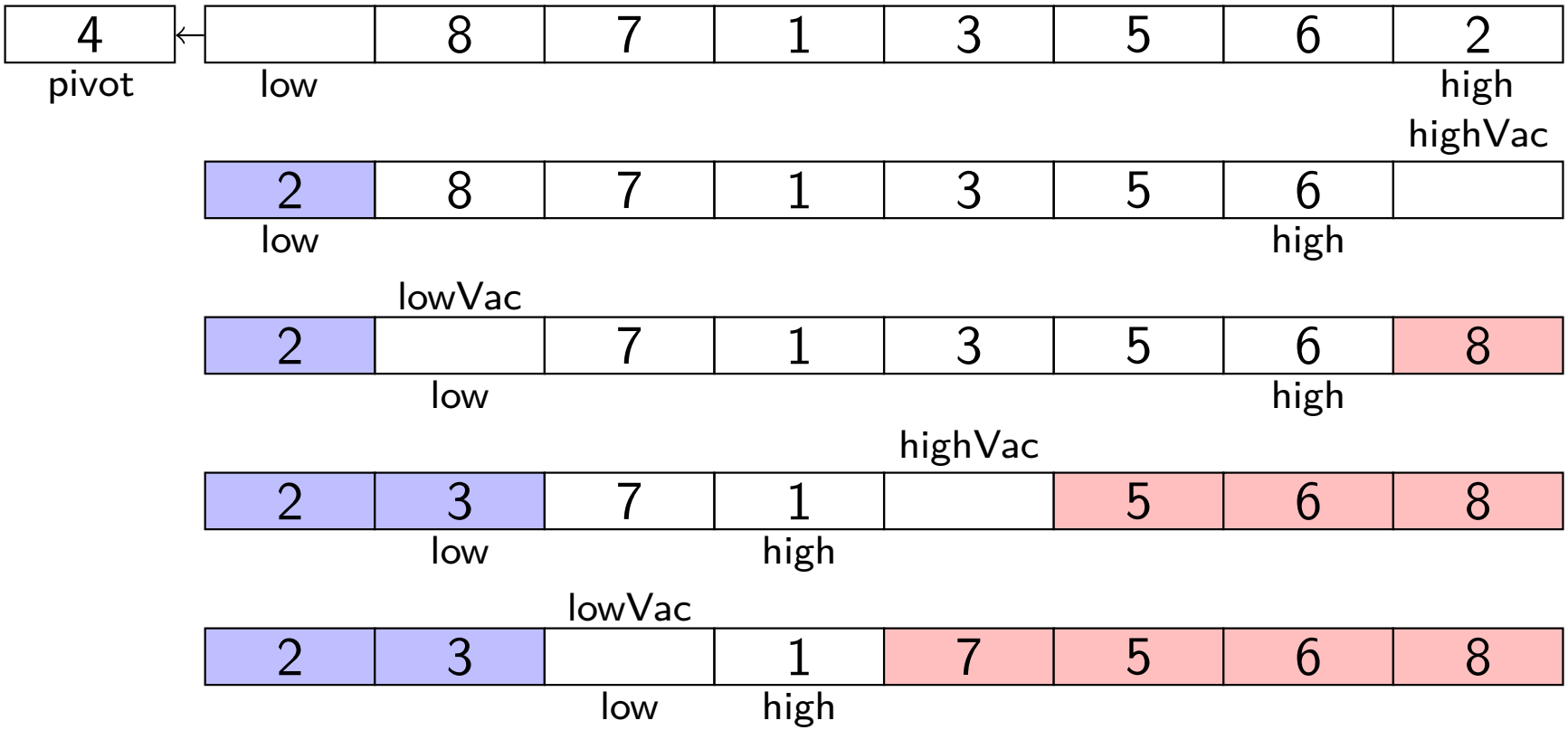
```
1  lowVac ← highVac;        // In case no element ≥ pivot.
2  curr ← low;
3  while curr < highVac do
4  |   if E[curr] ≥ pivot then
5  |   |   E[highVac] ← E[curr];
6  |   |   lowVac ← curr;
7  |   |   break;
8  |   end
9  |   curr ← curr + 1;
10 end
11 return lowVac;
```

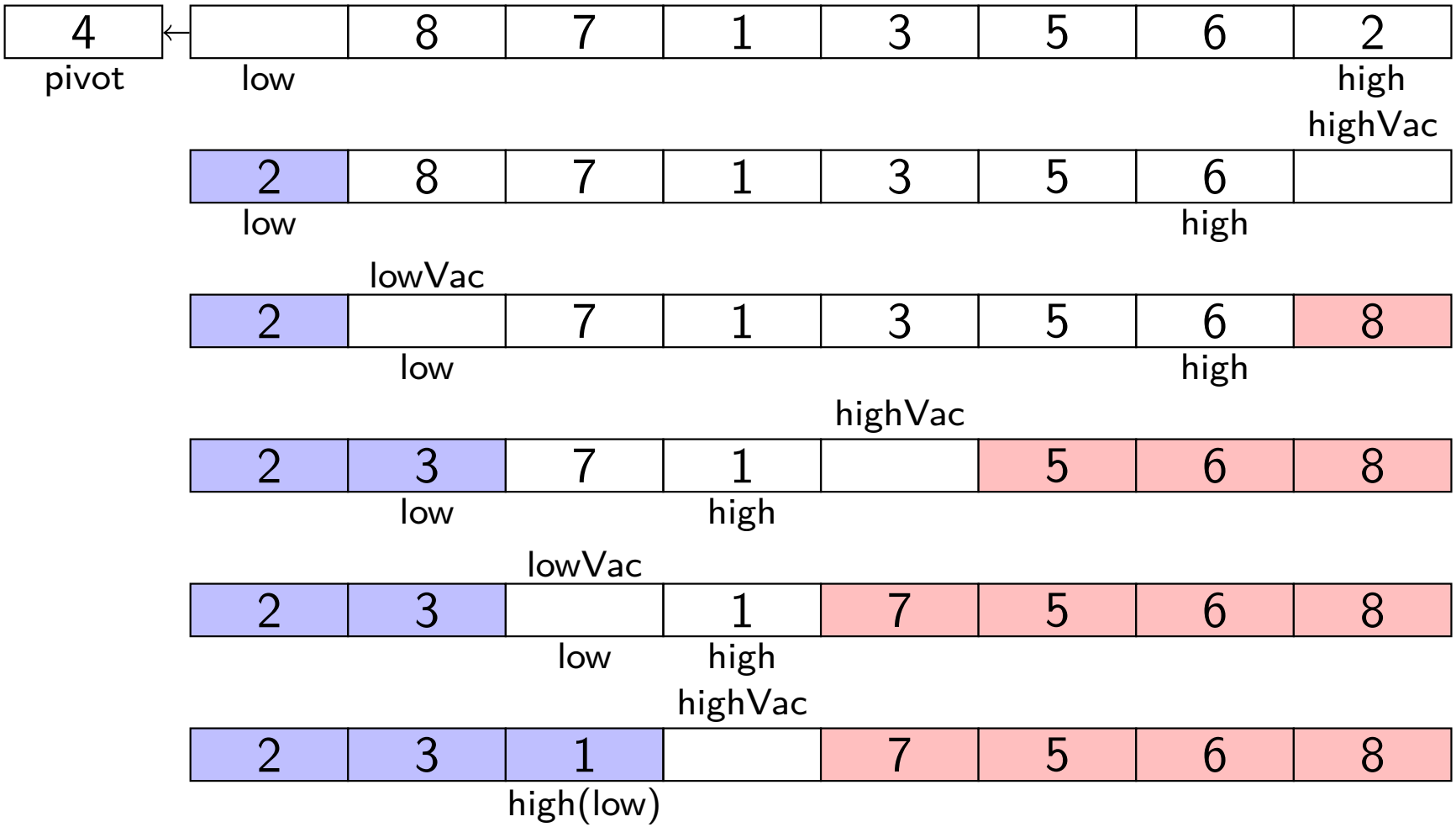

4
8
7
1
3
5
6
2
pivot
low
high

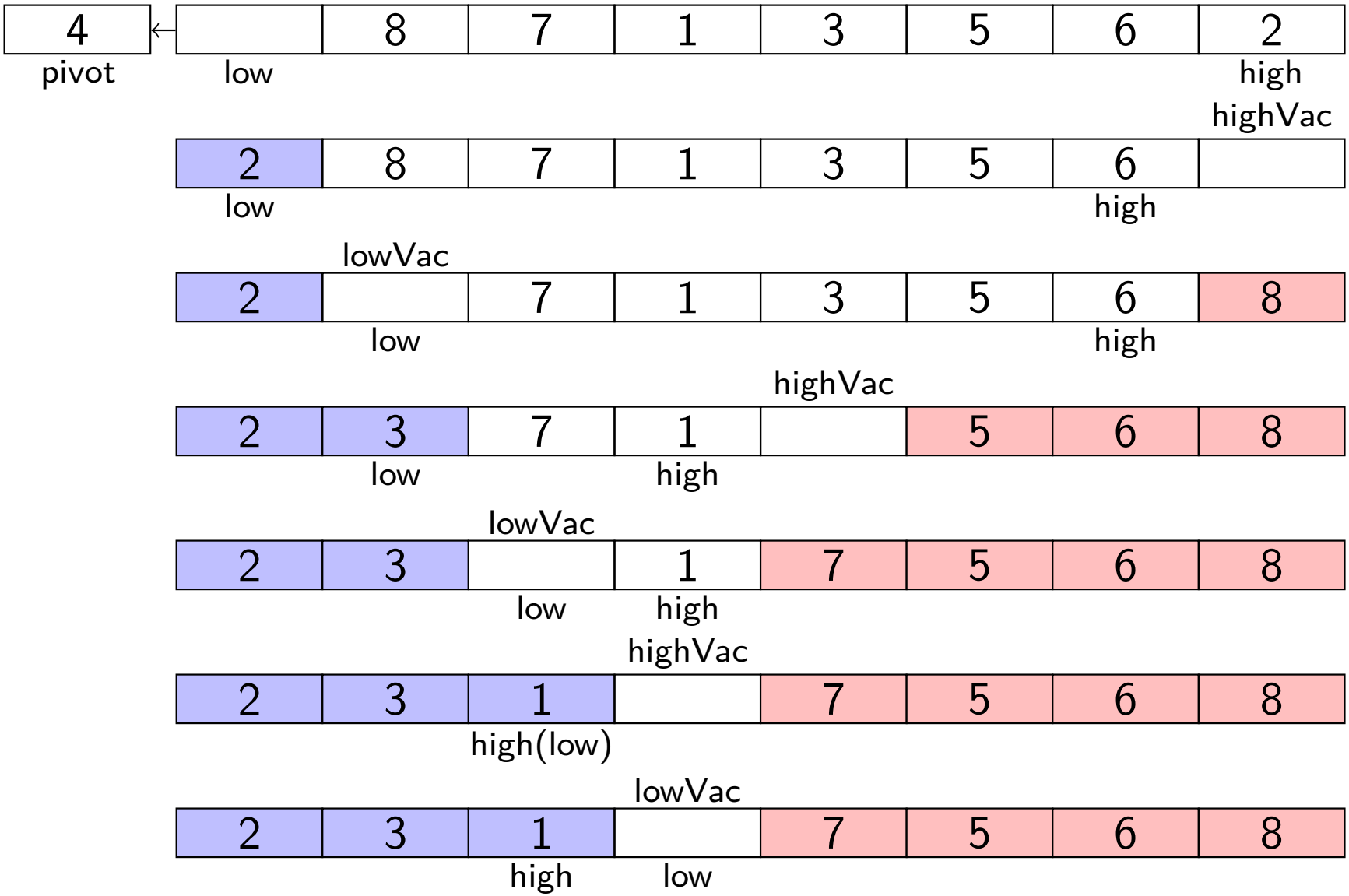

4
8
7
1
3
5
6
2
pivot
low
high
highVac
2
8
7
1
3
5
6
low
high

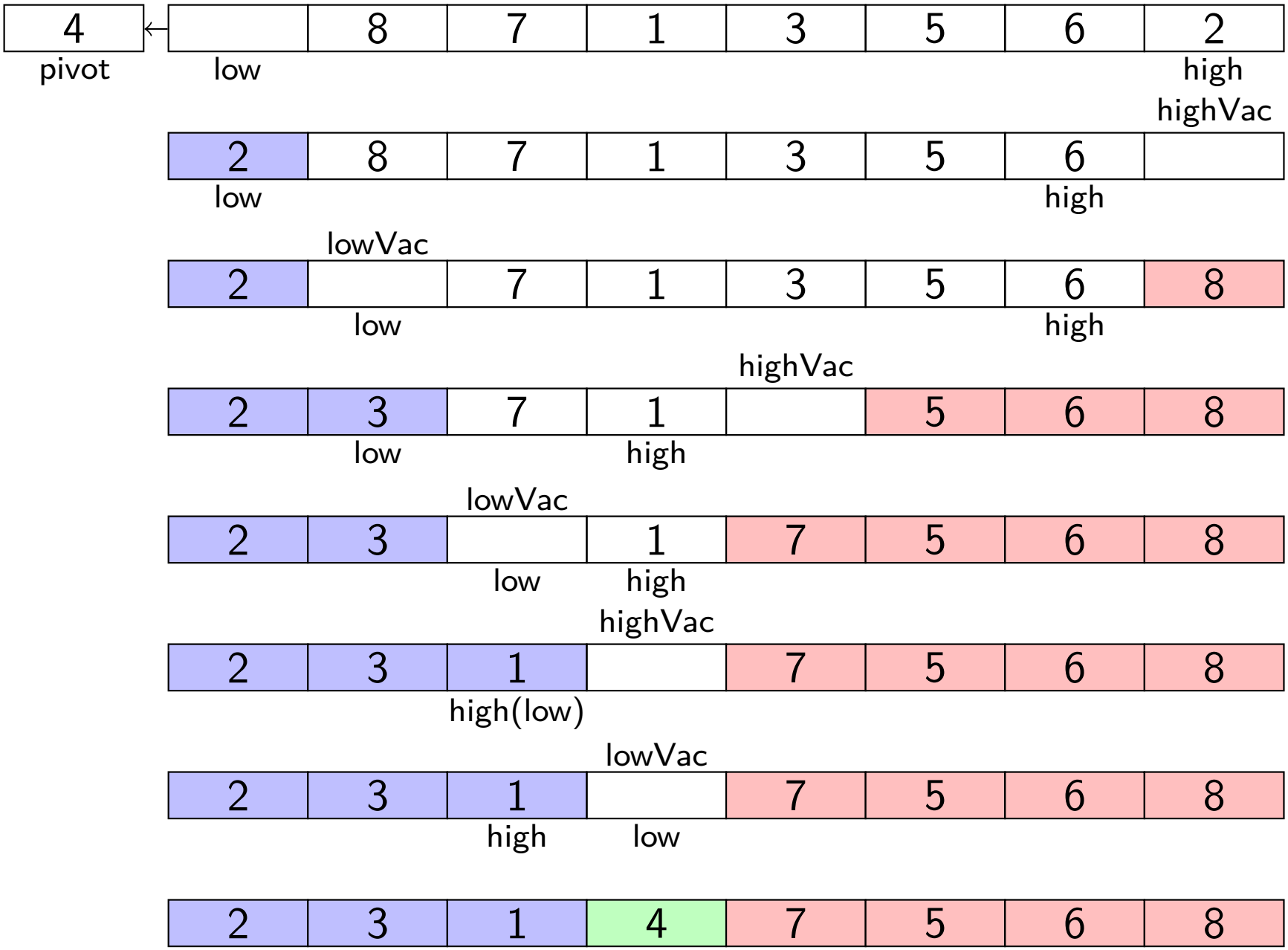

4
8
7
1
3
5
6
2
pivot
low
high
highVac
2
8
7
1
3
5
6
low
high
lowVac
2
7
1
3
5
6
8
low
high


4
8
7
1
3
5
6
2
pivot
low
high
highVac
2
8
7
1
3
5
6
low
high
lowVac
2
7
1
3
5
6
8
low
high
highVac
2
3
7
1
5
6
8
low
high


4
8
7
1
3
5
6
2
pivot
low
high
highVac
2
8
7
1
3
5
6
low
high
lowVac
2
7
1
3
5
6
8
low
high
highVac
2
3
7
1
5
6
8
low
high
lowVac
2
3
1
7
5
6
8
low
high


4
8
7
1
3
5
6
2
pivot
low
high
highVac
2
8
7
1
3
5
6
low
high
lowVac
2
7
1
3
5
6
8
low
high
highVac
2
3
7
1
5
6
8
low
high
lowVac
2
3
1
7
5
6
8
low
high
highVac
2
3
1
7
5
6
8
high(low)


4
8 7 1 3 5 6 2
pivot
low
high
highVac
2 8 7 1 3 5 6
low
high
lowVac
2 7 1 3 5 6 8
low
high
highVac
2 3 7 1 5 6 8
low
high
lowVac
2 3 1 7 5 6 8
low
high
highVac
2 3 1 7 5 6 8
high(low)
lowVac
2 3 1 7 5 6 8
high
low


4
8 7 1 3 5 6 2
pivot
low
high
highVac
2
8 7 1 3 5 6
low
high
lowVac
2
7 1 3 5 6 8
low
high
highVac
2 3 7 1 5 6 8
low
high
lowVac
2 3 1 7 5 6 8
low
high
highVac
2 3 1 7 5 6 8
high(low)
lowVac
2 3 1 7 5 6 8
high
low
2 3 1 4 7 5 6 8

最坏情况复杂度分析

- `Partition`: 对于长度为 k 的序列，共需比较 $k-1$ 次；
- 最坏情况下的 `Partition` 结果是 $splitPoint = first$ or $last$，原序列被分成一个空序列和一个比原序列少一个元素 (pivot) 的子序列；
- 因此，最坏情况下时间复杂度为：

$$W(n) = \sum_{k=2}^{n}(k-1) = \frac{n(n-1)}{2}$$

最坏情况复杂度分析

- `Partition`：对于长度为 k 的序列，共需比较 $k-1$ 次；
- 最坏情况下的 `Partition` 结果是 $splitPoint = first$ or $last$，原序列被分成一个空序列和一个比原序列少一个元素 (pivot) 的子序列；
- 因此，最坏情况下时间复杂度为：

$$W(n) = \sum_{k=2}^{n}(k-1) = \frac{n(n-1)}{2} \in \Theta(n^2)$$

最坏情况复杂度分析

- Partition: 对于长度为 k 的序列，共需比较 $k-1$ 次；
- 最坏情况下的 Partition 结果是 $splitPoint = first$ or $last$，原序列被分成一个空序列和一个比原序列少一个元素 (pivot) 的子序列；
- 因此，最坏情况下时间复杂度为：

$$W(n) = \sum_{k=2}^{n}(k-1) = \frac{n(n-1)}{2} \in \Theta(n^2)$$

- *QuickSort 徒有虚名？*

平均情况复杂度分析

- 我们已经知道：当一次比较最多仅能消除一个反序对的时候，平均情况下比较次数至少为 $n(n-1)/4$
- 对于 QuickSort 来说，一次比较后，元素在序列中可能移动相当大的距离，这意味着每次比较后可能消除多个反序对 (最多 $n-1$)
- 每次 Partition 之后得到的两个子序列各自的元素之间两两未经比较，因此可以认为子序列各种排序出现的概率保持相等
- 假设序列元素每种排列出现的概率相等，Partition 返回的 $splitPoint$ 位置出现在每个序列元素位置的概率也相等，则：

$$A(n) = (n-1) + \sum_{i=0}^{n-1} \frac{1}{n}(A(i) + A(n-1-i)) \qquad \text{for } n \geq 2$$

$$A(1) = A(0) = 0$$

- 整理一下得：$A(n) = (n-1) + \frac{2}{n}\sum_{i=1}^{n-1} A(i) \quad \text{for } n \geq 1$

求 $A(n)$

- 做一个猜想：如果 `Partition` 每次都将序列分为长度相同的两个子序列

$$A(n) \approx n + 2A(n/2)$$

求 $A(n)$

- 做一个猜想：如果 `Partition` 每次都将序列分为长度相同的两个子序列

$$A(n) \approx n + 2A(n/2) \xrightarrow{\text{主定理}} A(n) \in \Theta(n \log n)$$

求 $A(n)$

- 做一个猜想：如果 `Partition` 每次都将序列分为长度相同的两个子序列

$$A(n) \approx n + 2A(n/2) \xrightarrow{\text{主定理}} A(n) \in \Theta(n \log n)$$

Theorem

对于如下定义的 $A(n)$：

$$A(n) = (n-1) + \frac{2}{n}\sum_{i=1}^{n-1} A(i) \quad \textit{for } n \geq 1$$

存在常数 c，使得 $A(n) \leq cn \ln n$.

求 $A(n)$

- 做一个猜想：如果 `Partition` 每次都将序列分为长度相同的两个子序列

$$A(n) \approx n + 2A(n/2) \xrightarrow{\text{主定理}} A(n) \in \Theta(n \log n)$$

Theorem

对于如下定义的 $A(n)$：

$$A(n) = (n-1) + \frac{2}{n}\sum_{i=1}^{n-1} A(i) \quad \textit{for } n \geq 1$$

存在常数 c，使得 $A(n) \leq cn \ln n$.

- 一个更精确的近似：$A(n) \approx 1.386n \lg n - 2.846n$

改进

- 选择合适的 pivot：随机选择；选择 $E[first]$, $E[(first + last)/2]$, $E[last]$ 的中间值
- 减少递归调用
  - 将递归变为循环
  - 减小递归调用深度 (SmallSort)

改进

- 选择合适的 pivot：随机选择；选择 $E[first]$, $E[(first + last)/2]$, $E[last]$ 的中间值
- 减少递归调用
  - 将递归变为循环
  - 减小递归调用深度 (SmallSort)

Algorithm QuickSort($E[], first, last$)

```
1 if last − first > smallSize then
2 |   pivot ← E[first];
3 |   splitPoint ← Partition(E, pivot, first, last);
4 |   E[splitPoint] ← pivot;
5 |   QuickSort(E, first, splitPoint − 1);
6 |   QuickSort(E, splitPoint + 1, last);
7 else
8 |   SmallSort(E, first, last);
9 end
```

改进 (cont.)

- 快速排序的递归深度：$O(n) \Rightarrow O(\lg n)$

改进 (cont.)

- 快速排序的递归深度：$O(n) \Rightarrow O(\lg n)$

Algorithm `QuickSort2`$(E[], first, last)$

```
1  while first < last do
2      pivot ← E[first];
3      splitPoint ← Partition(E, pivot, first, last);
4      E[splitPoint] ← pivot;
5      if splitPoint − first < last − splitPoint then
6          QuickSort2(E, first, splitPoint − 1);
7          first ← splitPoint + 1;
8      else
9          QuickSort2(E, splitPoint + 1, last);
10         last ← splitPoint − 1;
11     end
12 end
```

主要内容

归并操作 (Merge)

- 归并操作，或归并算法，指的是将两个有序序列合并成一个有序序列的操作

- 基本原理：
  1. 申请空间，使其大小为两个已经排序序列之和，用来存放合并后的序列；
  2. 设定两个指针，最初位置分别为两个已经排序序列的起始位置；
  3. 比较两个指针所指向的元素，选择相对小的元素放入到合并空间，并移动指针到下一位置；
  4. 重复上一步骤直到某一指针达到序列尾；
  5. 将另一序列剩下的所有元素直接复制到合并序列尾

递归描述

Algorithm `Merge`$(A[], B[], C[])$

```
1  if A is empty then rest of C ← rest of B;
2  else if B is empty then rest of C ← rest of A;
3  else
4      if first of A ≤ first of B then
5          first of C ← first of A;
6          Merge(rest of A, B, rest of C);
7      else
8          first of C ← first of B;
9          Merge(A, rest of B, rest of C);
10     end
11 end
```

非递归版本

Algorithm `Merge`$(A[], k, B[], m, C[])$

```
1  n ← k + m;
2  indexA ← indexB ← indexC ← 0;
3  while indexA < k and indexB < m do
4      if A[indexA] ≤ B[indexB] then
5          C[indexC] ← A[indexA];
6          indexA ← indexA + 1;
7          indexC ← indexC + 1;
8      else
9          C[indexC] ← B[indexB];
10         indexB ← indexB + 1;
11         indexC ← indexC + 1;
12     end
13 end
14 if indexA ≥ m then  Copy B[indexB, ..., m − 1] to C[indexC, ..., n − 1] ;
15 else  Copy A[indexA, ..., k − 1] to C[indexC, ..., n − 1] ;
```

归并操作时间复杂度分析

- Worst-Case:
  - 每做完一次比较，至少有一个元素被移到合并后的序列中
  - 最后一次比较时，至少还剩 2 个元素未移到合并序列中，这时合并序列中最多有 $n-1$ 个元素，也即至多经过了 $n-1$ 次比较
  - $W(n) = n - 1 \in \Theta(n)$
- Average-Case?

归并操作时间复杂度分析

- Worst-Case:
  - 每做完一次比较，至少有一个元素被移到合并后的序列中
  - 最后一次比较时，至少还剩 2 个元素未移到合并序列中，这时合并序列中最多有 $n-1$ 个元素，也即至多经过了 $n-1$ 次比较
  - $W(n) = n - 1 \in \Theta(n)$
- Average-Case?

Theorem

任何对两个均包含 $k = m = n/2$ 个元素的有序序列，仅使用元素间的比较操作进行归并的算法，在最坏情况下至少需要 $n-1$ 次比较

归并操作时间复杂度分析

- Worst-Case:
  - 每做完一次比较，至少有一个元素被移到合并后的序列中
  - 最后一次比较时，至少还剩 2 个元素未移到合并序列中，这时合并序列中最多有 $n-1$ 个元素，也即至多经过了 $n-1$ 次比较
  - $W(n) = n - 1 \in \Theta(n)$
- Average-Case?

Theorem

任何对两个均包含 $k = m = n/2$ 个元素的有序序列，仅使用元素间的比较操作进行归并的算法，在最坏情况下至少需要 $n-1$ 次比较

Corollary

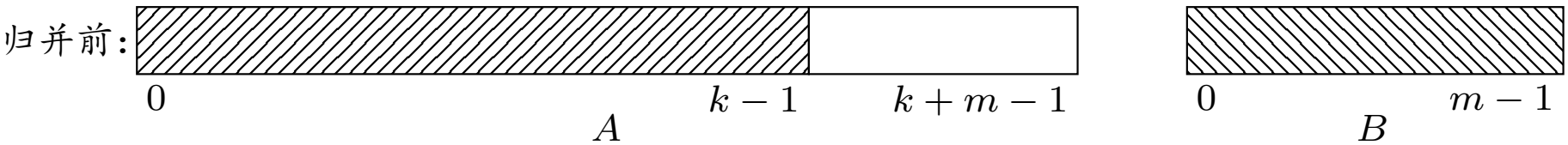
任何对两个分别包含 k 和 m 个元素的有序序列，并且 $|k-m| = 1$，仅使用元素间的比较操作进行归并的算法，在最坏情况下至少需要 $n-1$ 次比较

归并操作空间复杂度分析

- 前面的算法在存储空间占用上，除了输入数据 $k+m=n$ 个元素占用的空间外，显然还需要 n 个元素的额外空间存储结果
- 如果输入序列 A 和 B 用链表存储的话，则可以做到不需要额外存储空间，但前提是不需要保持输入数据
- 如果输入序列 A 和 B 用数组存储，且输入数据不需要保持，则额外空间占用可以有一定程度的减少

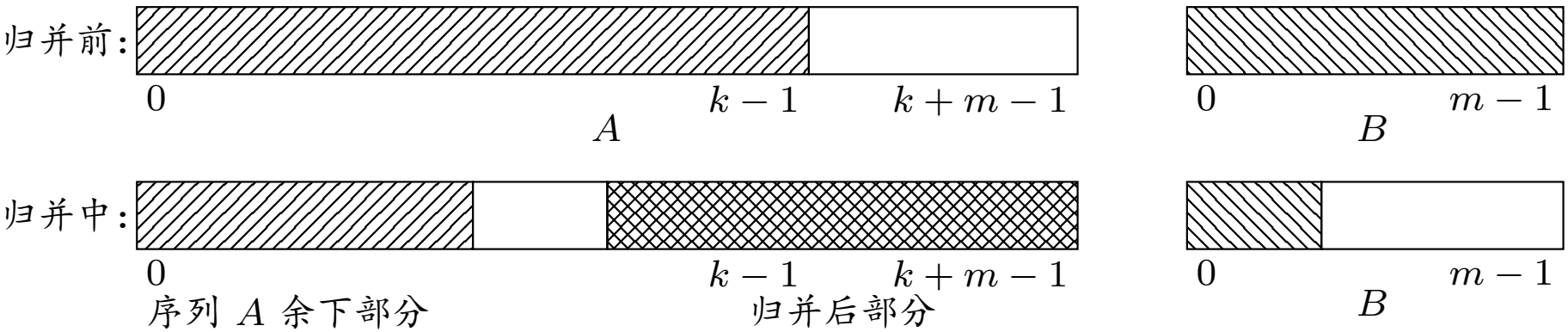
归并操作空间复杂度分析

- 前面的算法在存储空间占用上，除了输入数据 $k+m=n$ 个元素占用的空间外，显然还需要 n 个元素的额外空间存储结果
- 如果输入序列 A 和 B 用链表存储的话，则可以做到不需要额外存储空间，但前提是不需要保持输入数据
- 如果输入序列 A 和 B 用数组存储，且输入数据不需要保持，则额外空间占用可以有一定程度的减少



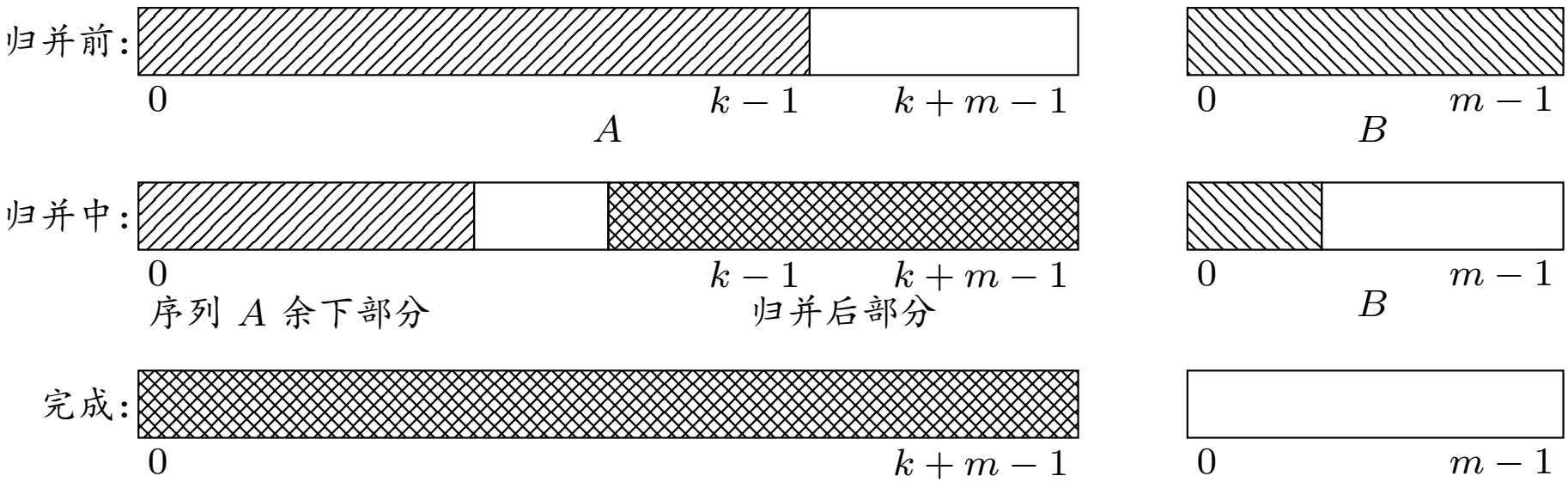
归并操作空间复杂度分析

- 前面的算法在存储空间占用上，除了输入数据 $k+m=n$ 个元素占用的空间外，显然还需要 n 个元素的额外空间存储结果
- 如果输入序列 A 和 B 用链表存储的话，则可以做到不需要额外存储空间，但前提是不需要保持输入数据
- 如果输入序列 A 和 B 用数组存储，且输入数据不需要保持，则额外空间占用可以有一定程度的减少



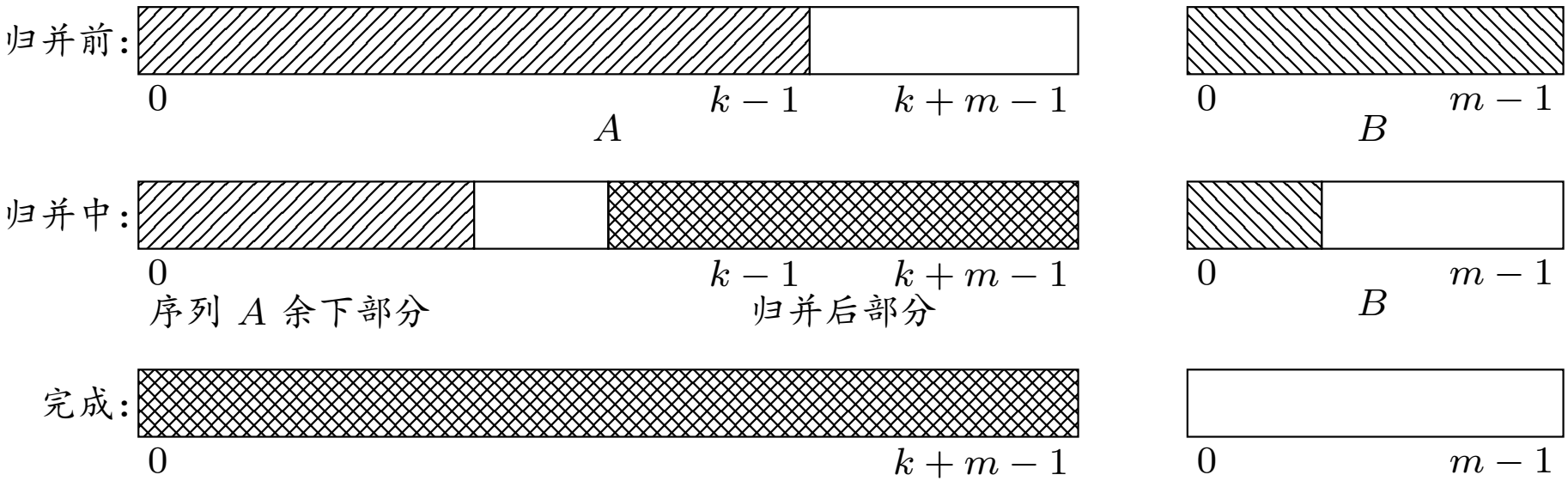
归并操作空间复杂度分析

- 前面的算法在存储空间占用上，除了输入数据 $k+m=n$ 个元素占用的空间外，显然还需要 n 个元素的额外空间存储结果
- 如果输入序列 A 和 B 用链表存储的话，则可以做到不需要额外存储空间，但前提是不需要保持输入数据
- 如果输入序列 A 和 B 用数组存储，且输入数据不需要保持，则额外空间占用可以有一定程度的减少



归并操作空间复杂度分析

- 前面的算法在存储空间占用上，除了输入数据 $k+m=n$ 个元素占用的空间外，显然还需要 n 个元素的额外空间存储结果
- 如果输入序列 A 和 B 用链表存储的话，则可以做到不需要额外存储空间，但前提是不需要保持输入数据
- 如果输入序列 A 和 B 用数组存储，且输入数据不需要保持，则额外空间占用可以有一定程度的减少



最坏情况下额外占用的存储空间：$n/2 \in \Theta(n)$ 当 $k=m=n/2$

归并排序 (Mergesort)

- 快速排序最大的问题实际上是每次序列分割不一定能将原序列分成长度相等的两个子序列
- 归并排序则每次将序列分割为恰好相等的两个子序列，分别对子序列进行递归排序，再将排好序的两个子序列通过归并操作合并成一个序列

归并排序 (Mergesort)

- 快速排序最大的问题实际上是每次序列分割不一定能将原序列分成长度相等的两个子序列
- 归并排序则每次将序列分割为恰好相等的两个子序列，分别对子序列进行递归排序，再将排好序的两个子序列通过归并操作合并成一个序列

Algorithm `Mergesort`$(E[], first, last)$

```
1 if first < last then
2 |   mid ← (first + last)/2;
3 |   Mergesort(E, first, mid);
4 |   Mergesort(E, mid + 1, last);
5 |   Merge(E, first, mid, last);
6 end
```

归并排序时间复杂度

- Worst-Case:

$$W(n) = W(\lfloor n/2 \rfloor) + W(\lceil n/2 \rceil) + n - 1$$
$$W(1) = 0$$

归并排序时间复杂度

- Worst-Case:

$$W(n) = W(\lfloor n/2 \rfloor) + W(\lceil n/2 \rceil) + n - 1$$
$$W(1) = 0$$
$$\Downarrow \text{主定理}$$
$$W(n) \in \Theta(n \log n)$$

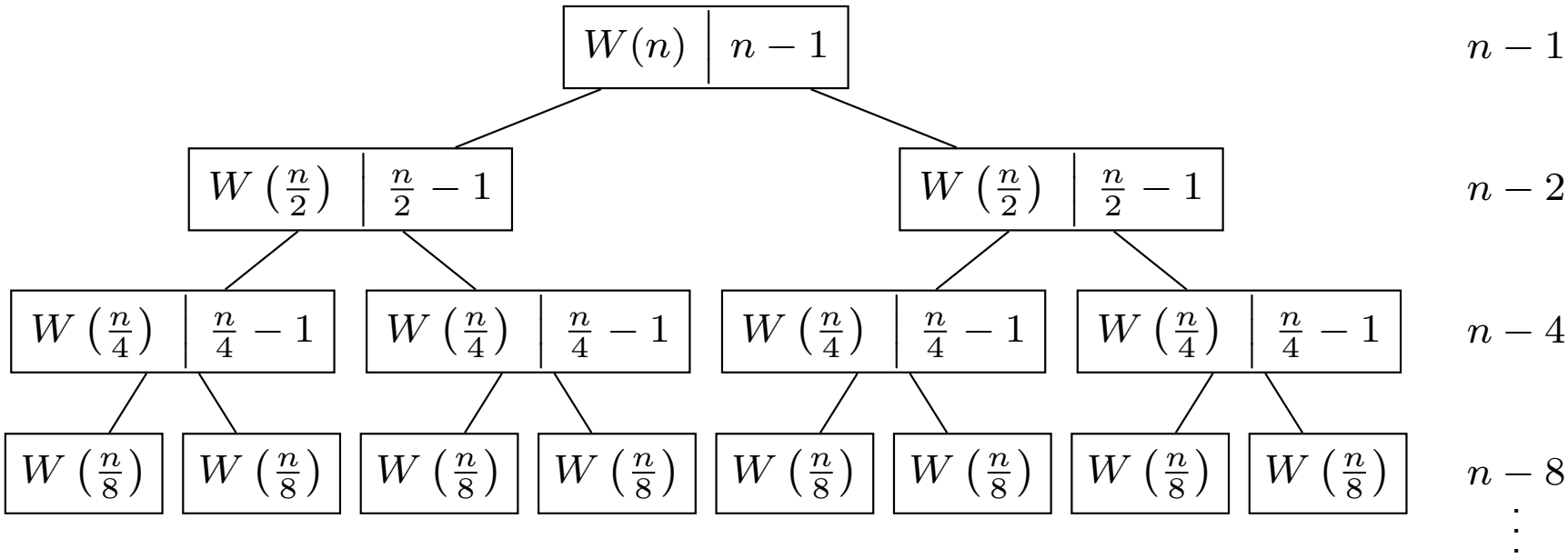
归并排序时间复杂度

- Worst-Case:

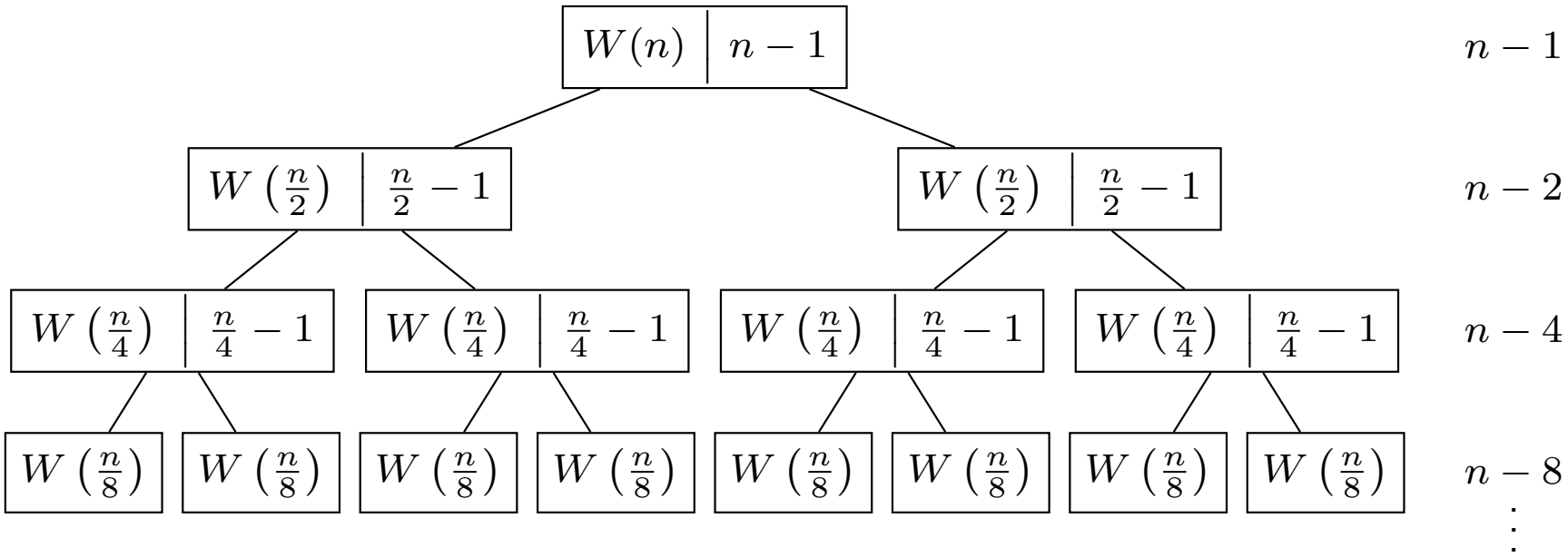
$$W(n) = W(\lfloor n/2 \rfloor) + W(\lceil n/2 \rceil) + n - 1$$
$$W(1) = 0$$
$$\Downarrow \text{主定理}$$
$$W(n) \in \Theta(n \log n)$$

- 一个更精确的结果：$\lceil n \lg n - n + 1 \rceil \leq W(n) \leq \lceil n \lg n - 0.914n \rceil$

Mergesort 递归树

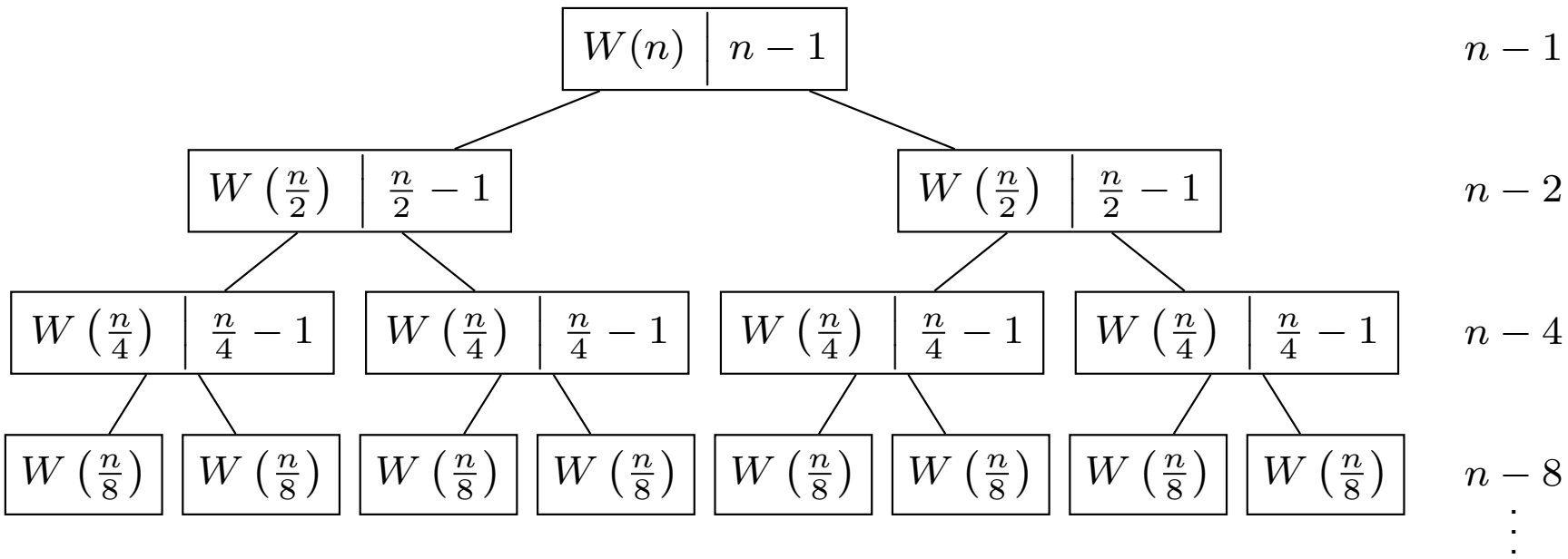


Mergesort 递归树



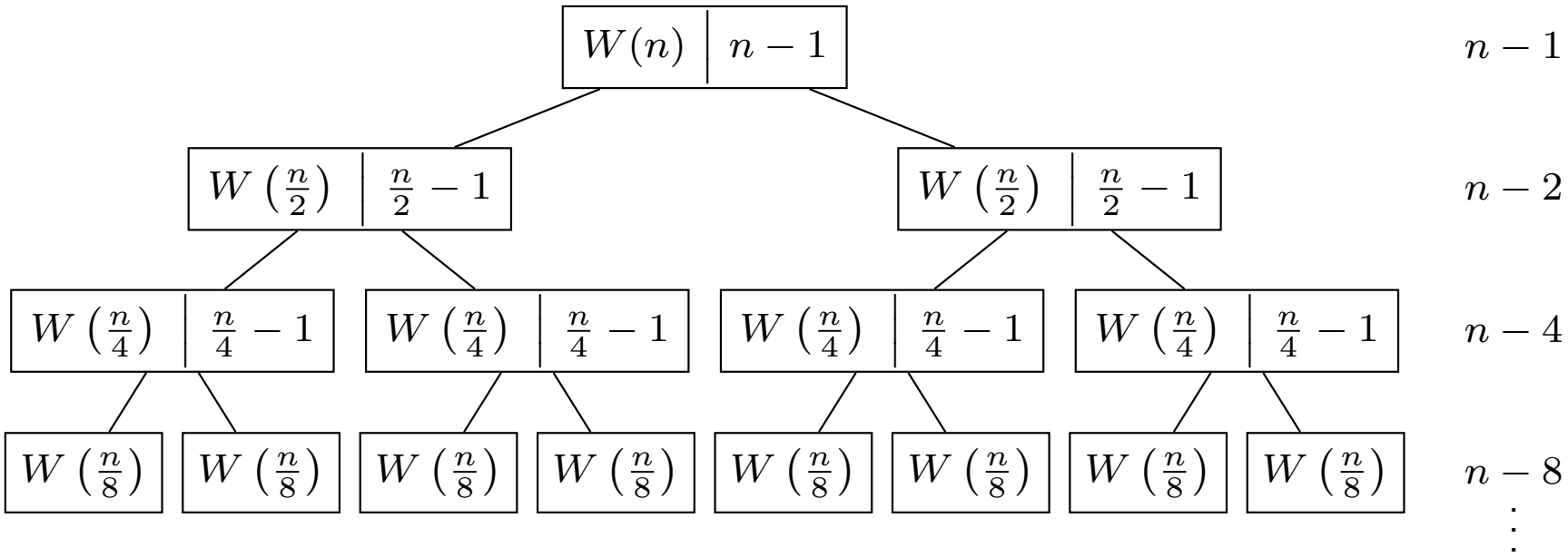
- 不包含叶结点的每层非递归开销为：$n-2^d$

Mergesort 递归树



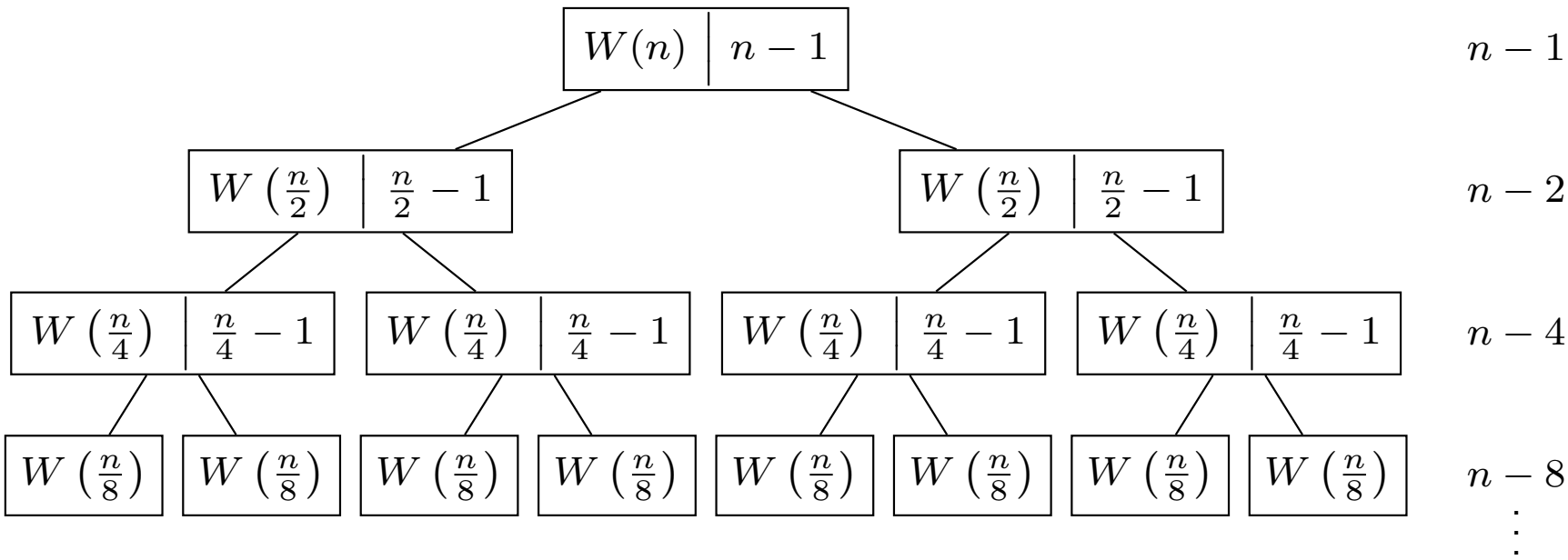
- 不包含叶结点的每层非递归开销为：$n - 2^d$
- 叶结点 $(W(1) = 0)$ 深度为 $\lceil \lg(n+1) \rceil - 1$ 或 $\lceil \lg(n+1) \rceil$

Mergesort 递归树



- 不包含叶结点的每层非递归开销为：$n - 2^d$
- 叶结点 $(W(1) = 0)$ 深度为 $\lceil \lg(n+1) \rceil - 1$ 或 $\lceil \lg(n+1) \rceil$
- 恰有 n 个叶结点

Mergesort 递归树



- 不包含叶结点的每层非递归开销为：$n - 2^d$
- 叶结点 $(W(1) = 0)$ 深度为 $\lceil \lg(n+1) \rceil - 1$ 或 $\lceil \lg(n+1) \rceil$
- 恰有 n 个叶结点
- 令递归树最大深度 (高度) 为 $D = \lceil \lg(n+1) \rceil$，在深度为 $D-1$ 层有 B 个叶结点，则在 D 层有 $n - B$ 个叶结点，在 $D-1$ 层有 $(n-B)/2$ 个非叶结点，且 $B = 2^D - n$ (*Why?*)

时间复杂度计算

$$
\begin{aligned}
W(n) &= \sum_{d=0}^{D-2}(n-2^d) + \frac{n-B}{2}W(2) \\
&= n(D-1) - 2^{D-1} + 1 + \frac{n-B}{2} \\
&= nD - 2^D + 1
\end{aligned}
$$

时间复杂度计算

$$
\begin{aligned}
W(n) &= \sum_{d=0}^{D-2}(n-2^d) + \frac{n-B}{2}W(2) \\
&= n(D-1) - 2^{D-1} + 1 + \frac{n-B}{2} \\
&= nD - 2^D + 1
\end{aligned}
$$

$$
\alpha = \frac{2^D}{n} \Rightarrow 1 \leq \alpha < 2
$$

时间复杂度计算

$$
\begin{aligned}
W(n) &= \sum_{d=0}^{D-2}(n-2^d) + \frac{n-B}{2}W(2) \\
&= n(D-1) - 2^{D-1} + 1 + \frac{n-B}{2} \\
&= nD - 2^D + 1
\end{aligned}
$$

$$
\alpha = \frac{2^D}{n} \Rightarrow 1 \leq \alpha < 2
$$

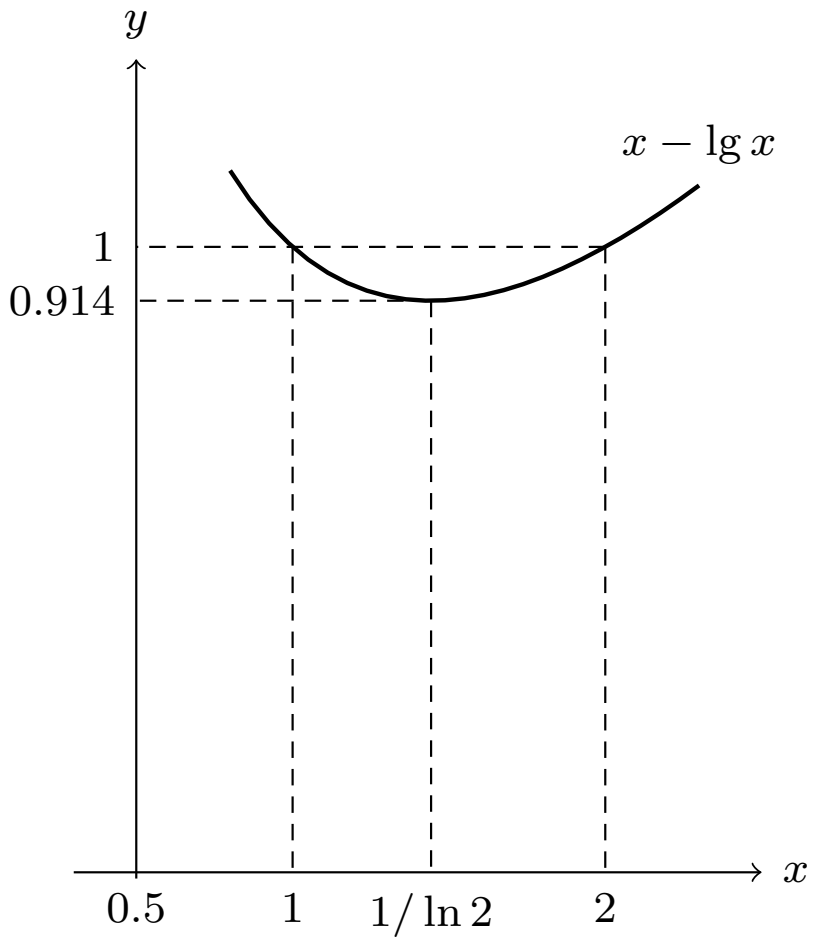
$$
W(n) = n\lg n - (\alpha - \lg\alpha)n + 1
$$

时间复杂度计算

$$W(n) = \sum_{d=0}^{D-2}(n - 2^d) + \frac{n-B}{2}W(2)$$
$$= n(D-1) - 2^{D-1} + 1 + \frac{n-B}{2}$$
$$= nD - 2^D + 1$$

$$\alpha = \frac{2^D}{n} \Rightarrow 1 \le \alpha < 2$$

$$W(n) = n\lg n - (\alpha - \lg\alpha)n + 1$$



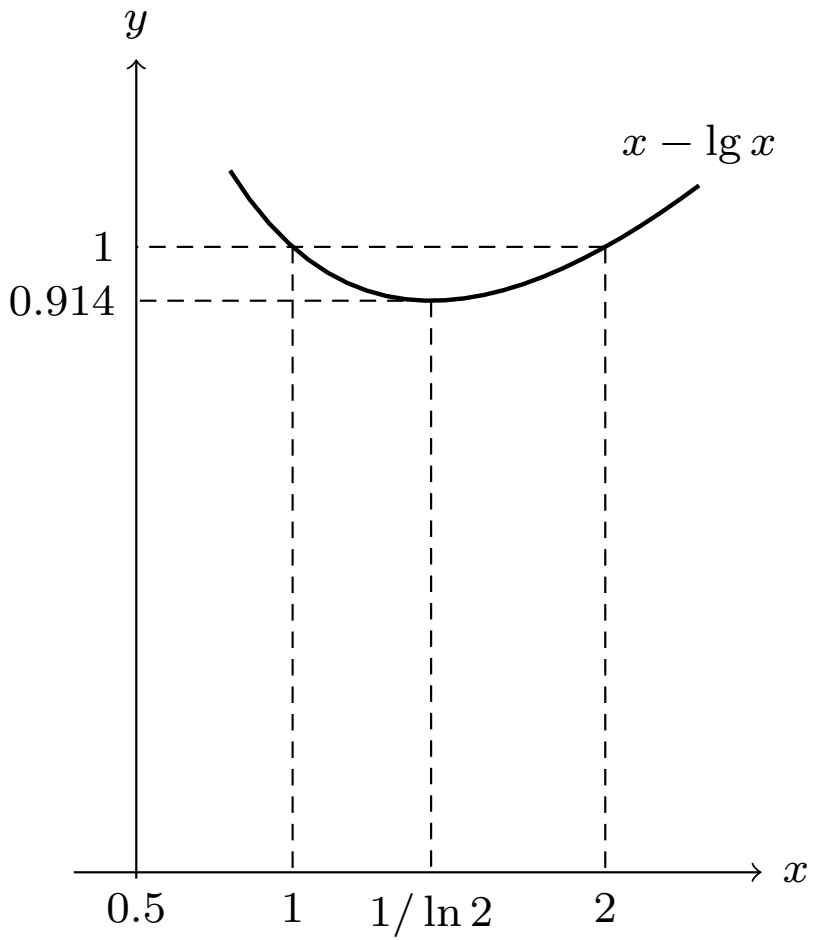
时间复杂度计算

$$W(n) = \sum_{d=0}^{D-2}(n - 2^d) + \frac{n-B}{2}W(2)$$
$$= n(D-1) - 2^{D-1} + 1 + \frac{n-B}{2}$$
$$= nD - 2^D + 1$$

$$\alpha = \frac{2^D}{n} \Rightarrow 1 \le \alpha < 2$$

$$W(n) = n\lg n - (\alpha - \lg\alpha)n + 1$$

$$0.914 \le \alpha - \lg\alpha \le 1$$



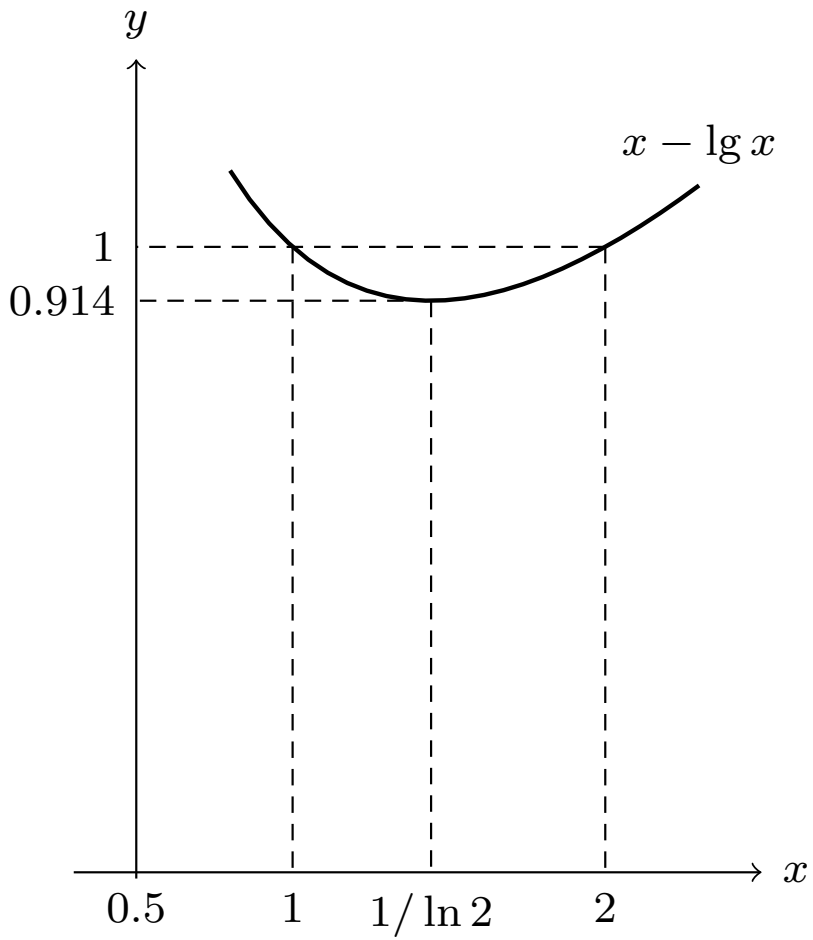
时间复杂度计算

$$W(n) = \sum_{d=0}^{D-2}(n-2^d) + \frac{n-B}{2}W(2)$$
$$= n(D-1) - 2^{D-1} + 1 + \frac{n-B}{2}$$
$$= nD - 2^D + 1$$

$$\alpha = \frac{2^D}{n} \Rightarrow 1 \le \alpha < 2$$

$$W(n) = n\lg n - (\alpha - \lg\alpha)n + 1$$

$$0.914 \le \alpha - \lg\alpha \le 1$$



$$\lceil n\lg n - n + 1\rceil \le W(n) \le \lceil n\lg n - 0.914n\rceil$$

一点讨论

- 快速排序时间复杂度：$W(n) \in \Theta(n^2),\ A(n) \approx 1.386n\lg n - 2.846n$
- 归并排序：$\lceil n\lg n - n + 1\rceil \le W(n) \le \lceil n\lg n - 0.914n\rceil$
- 是否意味着归并排序比快速排序更好？
  - 时间复杂度
  - 空间复杂度

Exercise (3)

试分析比较快速排序和归并排序在 平均情况 下的 元素移动次数。

deadline: 2015.04.11

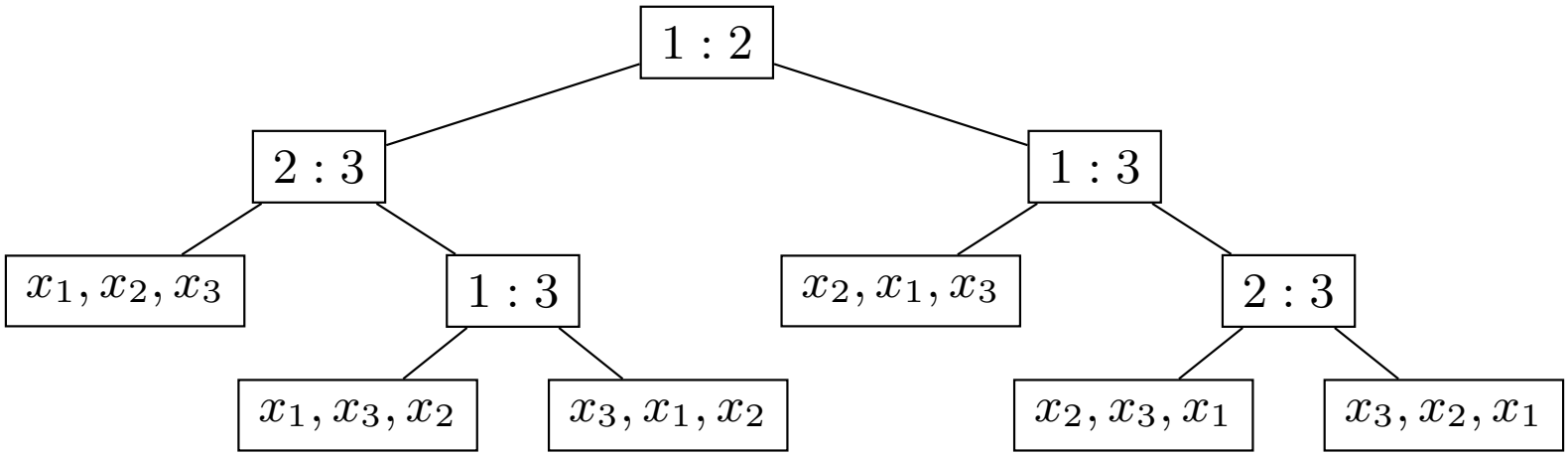
主要内容

排序算法的决策树

- 任意一种以元素间的比较为基本操作的排序算法其执行过程都对应于一棵二叉决策树：
  - 内结点标注 $(i:j)$，表示数组元素 x_i 与 x_j 的一次比较操作，其左子树对应 $x_i < x_j$ 的情况下下一步要进行的比较操作 (或输出结果操作)，右子树对应 $x_i > x_j$ 的情况下下一步要进行的比较操作 (或输出结果操作)
  - 每个外结点表示一个排序结果

排序算法的决策树

- 任意一种以元素间的比较为基本操作的排序算法其执行过程都对应于一棵二叉决策树：
  - 内结点标注 $(i:j)$，表示数组元素 x_i 与 x_j 的一次比较操作，其左子树对应 $x_i < x_j$ 的情况下下一步要进行的比较操作 (或输出结果操作)，右子树对应 $x_i > x_j$ 的情况下下一步要进行的比较操作 (或输出结果操作)
  - 每个外结点表示一个排序结果
- 例：输入规模 $n = 3$ 时的决策树：



最坏情况复杂度下界

Lemma

若高度为 h 的二叉树有 L 个叶结点，则有：$L \leq 2^h$ 及 $h \geq \lceil \lg L \rceil$

Lemma

对于给定输入规模 n，任何以元素比较为基本操作的排序算法其对应决策树的高度至少为 $\lceil \lg n! \rceil$

最坏情况复杂度下界

Lemma

若高度为 h 的二叉树有 L 个叶结点，则有：$L \leq 2^h$ 及 $h \geq \lceil \lg L \rceil$

Lemma

对于给定输入规模 n，任何以元素比较为基本操作的排序算法其对应决策树的高度至少为 $\lceil \lg n! \rceil$

Theorem

对于给定输入规模 n，任何以元素比较为基本操作的排序算法在最坏情况下至少要执行 $\lceil \lg n! \rceil \approx \lceil n \lg n - 1.443n \rceil$ 次比较操作

最坏情况复杂度下界

Lemma

若高度为 h 的二叉树有 L 个叶结点，则有：$L \leq 2^h$ 及 $h \geq \lceil \lg L \rceil$

Lemma

对于给定输入规模 n，任何以元素比较为基本操作的排序算法其对应决策树的高度至少为 $\lceil \lg n! \rceil$

Theorem

对于给定输入规模 n，任何以元素比较为基本操作的排序算法在最坏情况下至少要执行 $\lceil \lg n! \rceil \approx \lceil n \lg n - 1.443n \rceil$ 次比较操作

$$\text{注：}\ \lg n! = \sum_{j=1}^{n} \lg j \geq n \lg n - (\lg e)n$$

平均情况复杂度下界

- 根据决策树定义，很显然是 2-tree，而每一条从根结点到叶结点的路径代表了一次成功的排序过程
- 设决策树外路径长度为 epl，叶结点个数为 L，则平均比较次数为 epl/L

平均情况复杂度下界

- 根据决策树定义，很显然是 2-tree，而每一条从根结点到叶结点的路径代表了一次成功的排序过程
- 设决策树外路径长度为 epl，叶结点个数为 L，则平均比较次数为 epl/L
- $\Longrightarrow$ 找到 epl 的下界：$epl \geq L \lg L$

recall

平均情况复杂度下界

- 根据决策树定义，很显然是 2-tree，而每一条从根结点到叶结点的路径代表了一次成功的排序过程
- 设决策树外路径长度为 epl，叶结点个数为 L，则平均比较次数为 epl/L
- $\Longrightarrow$ 找到 epl 的下界：$epl \geq L \lg L$

recall

Theorem

对于给定输入规模 n，任何以元素比较为基本操作的排序算法在平均情况下至少要执行 $\lg n! \approx n \lg n - 1.443n$ 次比较操作

主要内容

回顾：二叉堆 (Binary Heap)

- 堆 (Heap)：基于树的数据结构，其满足堆特性：若 B 是 A 的子结点，则 $\text{key}(A) \geq \text{key}(B)$ (大根堆)
- 二叉堆：用完全二叉树来表达的堆结构，是实现优先队列的最有效的数据结构之一
- 二叉堆的基本操作：
  - 往堆中添加元素 (heapify-up, up-heap, bubble-up)：
    1. 将新元素插入堆的尾部
    2. 将新元素与其父结点比较，若满足堆特性，则结束
    3. 否则，将其与父结点交换位置，并重复上一步骤
  - 提取并删除堆首元素 (heapify-down, down-heap, bubble-down)：
    1. 将堆尾元素放到堆首代替已删除的堆首元素
    2. 将其与子结点比较，若满足堆特性，则结束
    3. 否则，将其与子结点中的一个交换位置，使三者满足堆特性，并重复上一步骤
- ✠ 效率：入队和出队的复杂度 $W(n) \in O(\log n)$

堆排序 (Heapsort) 算法要点

Algorithm `Heapsort`$(E[], n)$

1 从 E 构造堆 H;
2 **for** $i = n$ **down to** 1 **do**
3 $\quad curMax \leftarrow$ `GetMax`(H);
4 $\quad$ `DeleteMax`(H);
5 $\quad E[i] \leftarrow curMax$;
6 **end**

堆排序 (Heapsort) 算法要点

Algorithm `Heapsort`$(E[], n)$

1 从 E 构造堆 H;
2 **for** $i = n$ **down to** 1 **do**
3 $\quad curMax \leftarrow$ `GetMax`(H);
4 $\quad$ `DeleteMax`(H);
5 $\quad E[i] \leftarrow curMax$;
6 **end**

Algorithm `DeleteMax`(H)

1 拷贝堆 H 的底层最右边结点中的元素到 K 中;
2 删除堆 H 的底层最右边结点中的元素;
3 `FixHeap`(H, K);

堆的调整：FixHeap

Algorithm FixHeap(H, K)

```
1  if H 是叶结点 then
2  |   将 K 插入 Root(H);
3  else
4  |   取左右子树中根结点元素较大的那棵作为 largerSubHeap;
5  |   if K ≥Root(largerSubHeap) then
6  |   |   将 K 插入 Root(H);
7  |   else
8  |   |   将 Root(largerSubHeap) 插入 Root(H);
9  |   |   FixHeap(largerSubHeap, K);
10 |   end
11 end
```

堆的构造：ConstructHeap

Algorithm ConstructHeap(H)

```
1 if H 不是叶结点 then
2 |   ConstructHeap(H 的左子树);
3 |   ConstructHeap(H 的右子树);
4 |   K ← Root(H);
5 |   FixHeap(H, K);
6 end
```

- 最坏情况复杂度分析：

堆的构造：ConstructHeap

Algorithm ConstructHeap(H)

```
1 if H 不是叶结点 then
2 |   ConstructHeap(H 的左子树);
3 |   ConstructHeap(H 的右子树);
4 |   K ← Root(H);
5 |   FixHeap(H, K);
6 end
```

- 最坏情况复杂度分析：
  - FixHeap：大约需要 $2\lg n$ 次比较

堆的构造：ConstructHeap

Algorithm ConstructHeap(H)

```
1 if H 不是叶结点 then
2 |   ConstructHeap(H 的左子树);
3 |   ConstructHeap(H 的右子树);
4 |   K ← Root(H);
5 |   FixHeap(H, K);
6 end
```

- 最坏情况复杂度分析：
  - FixHeap：大约需要 $2\lg n$ 次比较
  - ConstructHeap：$W(n) = W(n-r-1) + W(r) + 2\lg n$，$r$ 为右子树结点数，且 $n > 1$

堆的构造：ConstructHeap

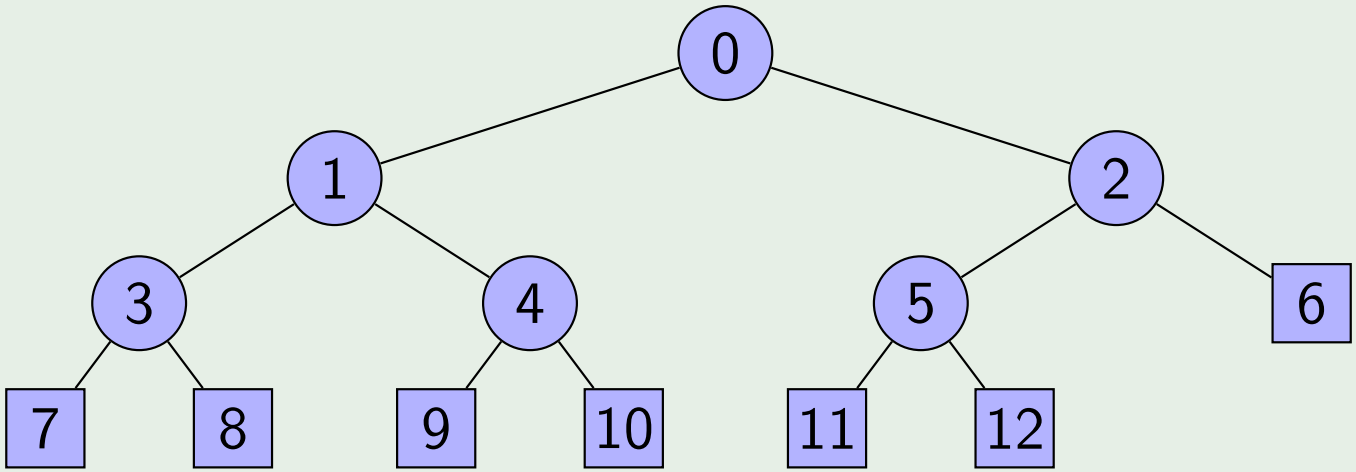
Algorithm ConstructHeap(H)

```
1 if H 不是叶结点 then
2 |   ConstructHeap(H 的左子树);
3 |   ConstructHeap(H 的右子树);
4 |   K ← Root(H);
5 |   FixHeap(H, K);
6 end
```

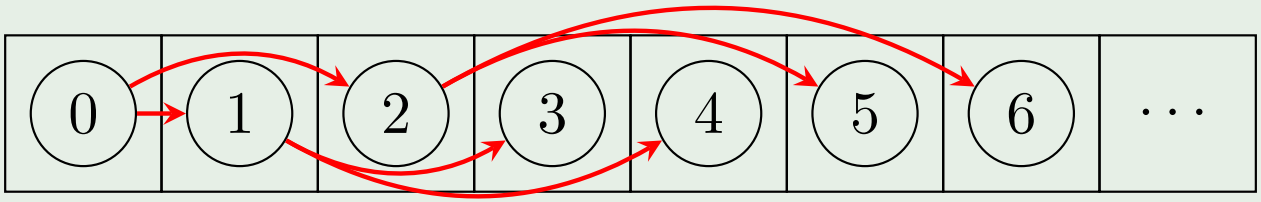
- 最坏情况复杂度分析：
  - FixHeap：大约需要 $2\lg n$ 次比较
  - ConstructHeap：$W(n) = W(n-r-1) + W(r) + 2\lg n$，$r$ 为右子树结点数，且 $n > 1$
  - $\Rightarrow W(n) \in \Theta(n)$ (*Why?*)

回顾：二叉堆 (完全二叉树) 的数组存储方式

Example (完全二叉树)



完全二叉树的顺序存储方式：



数组实现版本

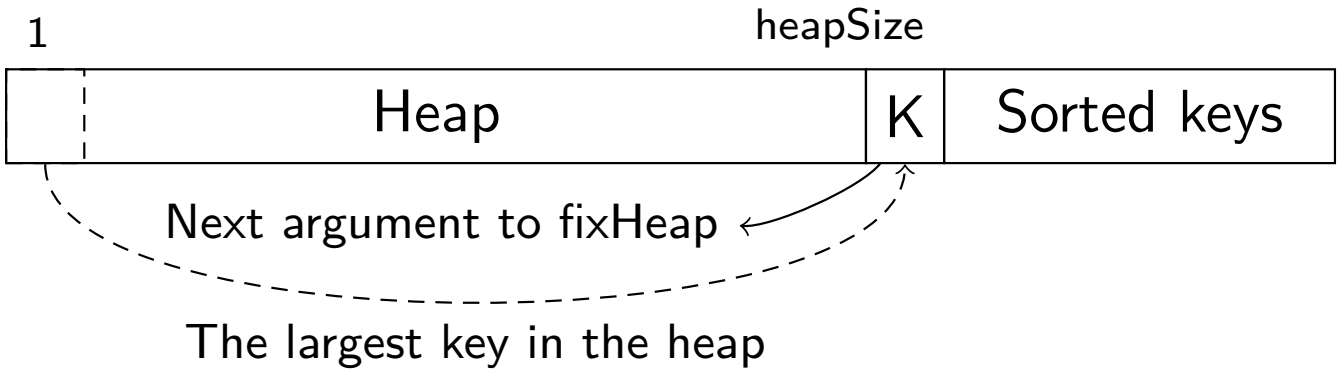
Algorithm `Heapsort`$(E[], n)$

```
1 ConstructHeap(E, n);
2 for heapSize ← n to 2 do
3 |   curMax ← E[0];
4 |   K ← E[heapSize − 1];
5 |   FixHeap(E, heapSize − 1, 0, K);
6 |   E[heapSize − 1] ← curMax;
7 end
```

数组实现版本

Algorithm `Heapsort`$(E[], n)$

```
1 ConstructHeap(E, n);
2 for heapSize ← n to 2 do
3 |   curMax ← E[0];
4 |   K ← E[heapSize − 1];
5 |   FixHeap(E, heapSize − 1, 0, K);
6 |   E[heapSize − 1] ← curMax;
7 end
```



FixHeap 的数组实现

Algorithm FixHeap$(E[], heapSize, root, K)$

```
1  left ← 2 * root + 1, right ← 2 * root + 2;
2  if left ≥ heapSize then E[root] ← K;
3  else
4      if left = heapSize − 1 then largerSubHeap ← left;
5      else if E[left] > E[right] then largerSubHeap ← left;
6      else largerSubHeap ← right;
7      if K ≥ E[largerSubHeap] then E[root] ← K;
8      else
9          E[root] ← E[largerSubHeap];
10         FixHeap(E, heapSize, largerSubHeap, K);
11     end
12 end
```

堆排序算法复杂度分析

最坏情况：

- `ConstructHeap`：$\in \Theta(n)$
- `FixHeap`：$2\lfloor \lg k \rfloor$
- 循环：$2\sum_{k=1}^{n-1} \lfloor \lg k \rfloor$

堆排序算法复杂度分析

最坏情况：

- `ConstructHeap`：$\in \Theta(n)$
- `FixHeap`：$2\lfloor \lg k \rfloor$
- 循环：$2\sum_{k=1}^{n-1}\lfloor \lg k \rfloor$

$$
\begin{aligned}
2\sum_{k=1}^{n-1}\lfloor \lg k \rfloor &\leq 2\int_{1}^{n}(\lg e)\ln x dx \\
&= 2(\lg e)(n\ln n - n) \\
&= 2(n\lg n - 1.443n)
\end{aligned}
$$

堆排序算法复杂度分析

最坏情况：

- `ConstructHeap`：$\in \Theta(n)$
- `FixHeap`：$2\lfloor \lg k \rfloor$
- 循环：$2\sum_{k=1}^{n-1} \lfloor \lg k \rfloor$

$$
\begin{aligned}
2\sum_{k=1}^{n-1} \lfloor \lg k \rfloor &\leq 2\int_1^n (\lg e) \ln x dx \\
&= 2(\lg e)(n \ln n - n) \\
&= 2(n \lg n - 1.443n)
\end{aligned}
$$

Theorem

`Heapsort` 排序算法在最坏情况下所需的比较操作次数为 $2n \lg n + O(n)$，其时间复杂度为 $\Theta(n \log n)$

堆排序算法复杂度分析

最坏情况：

- `ConstructHeap`：$\in \Theta(n)$
- `FixHeap`：$2\lfloor \lg k \rfloor$
- 循环：$2\sum_{k=1}^{n-1} \lfloor \lg k \rfloor$

$$
\begin{aligned}
2\sum_{k=1}^{n-1} \lfloor \lg k \rfloor &\leq 2\int_1^n (\lg e) \ln x dx \\
&= 2(\lg e)(n \ln n - n) \\
&= 2(n \lg n - 1.443n)
\end{aligned}
$$

Theorem

`Heapsort` 排序算法在最坏情况下所需的比较操作次数为 $2n \lg n + O(n)$，其时间复杂度为 $\Theta(n \log n)$

平均情况？

改进

Algorithm `BubbleUpHeap`$(E[], root, K, vacant)$

```
1 if vacant = root then E[vacant] = K;
2 else
3 |   parent ← (vacant − 1)/2;
4 |   if K ≤ E[parent] then E[vacant] ← K ;
5 |   else
6 |   |   E[vacant] ← E[parent];
7 |   |   BubbleUpHeap(E, root, K, parent);
8 |   end
9 end
```

改进

Algorithm `BubbleUpHeap`$(E[], root, K, vacant)$

```
1 if vacant = root then E[vacant] = K;
2 else
3     parent ← (vacant − 1)/2;
4     if K ≤ E[parent] then E[vacant] ← K ;
5     else
6         E[vacant] ← E[parent];
7         BubbleUpHeap(E, root, K, parent);
8     end
9 end
```

- 取出堆的最大元素后，首先每次将左右子结点中的较大元素放到根结点空位上而暂时不与 K 比较 (risky `FixHeap`)

改进

Algorithm `BubbleUpHeap`$(E[], root, K, vacant)$

```
1 if vacant = root then E[vacant] = K;
2 else
3     parent ← (vacant − 1)/2;
4     if K ≤ E[parent] then E[vacant] ← K ;
5     else
6         E[vacant] ← E[parent];
7         BubbleUpHeap(E, root, K, parent);
8     end
9 end
```

- 取出堆的最大元素后，首先每次将左右子结点中的较大元素放到根结点空位上而暂时不与 K 比较 (risky `FixHeap`)
- 当空位降到最底层后使用 `BubbleUpHeap` 将 K “提升” (bubble up) 到适当位置

改进

Algorithm `BubbleUpHeap`$(E[], root, K, vacant)$

```
1 if vacant = root then E[vacant] = K;
2 else
3     parent ← (vacant − 1)/2;
4     if K ≤ E[parent] then E[vacant] ← K ;
5     else
6         E[vacant] ← E[parent];
7         BubbleUpHeap(E, root, K, parent);
8     end
9 end
```

- 取出堆的最大元素后，首先每次将左右子结点中的较大元素放到根结点空位上而暂时不与 K 比较 (risky `FixHeap`)
- 当空位降到最底层后使用 `BubbleUpHeap` 将 K “提升” (bubble up) 到适当位置
- 由于我们取的 K 是堆尾元素，应该是一个“相当小”的元素，因此在平均情况下应该用比空位下降更少的比较就能上升到合适位置

改进

Algorithm `BubbleUpHeap`$(E[], root, K, vacant)$

```
1 if vacant = root then E[vacant] = K;
2 else
3     parent ← (vacant − 1)/2;
4     if K ≤ E[parent] then E[vacant] ← K ;
5     else
6         E[vacant] ← E[parent];
7         BubbleUpHeap(E, root, K, parent);
8     end
9 end
```

- 取出堆的最大元素后，首先每次将左右子结点中的较大元素放到根结点空位上而暂时不与 K 比较 (risky `FixHeap`)
- 当空位降到最底层后使用 `BubbleUpHeap` 将 K “提升” (bubble up) 到适当位置
- 由于我们取的 K 是堆尾元素，应该是一个“相当小”的元素，因此在平均情况下应该用比空位下降更少的比较就能上升到合适位置
- 注意：最坏情况仍然需要 $2\lfloor \lg k \rfloor$ 次比较

改进

Algorithm `BubbleUpHeap`$(E[], root, K, vacant)$

```
1 if vacant = root then E[vacant] = K;
2 else
3     parent ← (vacant − 1)/2;
4     if K ≤ E[parent] then E[vacant] ← K ;
5     else
6         E[vacant] ← E[parent];
7         BubbleUpHeap(E, root, K, parent);
8     end
9 end
```

- 取出堆的最大元素后，首先每次将左右子结点中的较大元素放到根结点空位上而暂时不与 K 比较 (risky `FixHeap`)
- 当空位降到最底层后使用 `BubbleUpHeap` 将 K “提升” (bubble up) 到适当位置
- 由于我们取的 K 是堆尾元素，应该是一个“相当小”的元素，因此在平均情况下应该用比空位下降更少的比较就能上升到合适位置
- 注意：最坏情况仍然需要 $2\lfloor \lg k \rfloor$ 次比较
- 能不能做得更好一些？

进一步的改进 —— Divide and Conquer

- Risky FixHeap 每次只下降当前堆 $\frac{h}{2}$ 高度 (可以称为 Promote)
- 当下降到当前空位父结点元素小于 K 时开始 BubbleUpHeap

进一步的改进 —— Divide and Conquer

- Risky FixHeap 每次只下降当前堆 $\frac{h}{2}$ 高度 (可以称为 Promote)
- 当下降到当前空位父结点元素小于 K 时开始 BubbleUpHeap

Algorithm Promote$(E[], hStop, vacant, h)$

```
1  left ← 2 * vacant + 1;
2  right ← 2 * vacant + 2;
3  if h ≤ hStop then
4  |   vacStop ← vacant;
5  else if E[left] ≤ E[right] then
6  |   E[vacant] ← E[right];
7  |   vacStop ← Promote(E, hStop, right, h − 1);
8  else
9  |   E[vacant] ← E[left];
10 |   vacStop ← Promote(E, hStop, left, h − 1);
11 end
12 return vacStop;
```

更快的 FixHeap

Algorithm FixHeapFast$(E[], n, K, vacant, h)$

```
1  if h ≤ 1 then
2  |   处理高度为 0 或 1 的情况;
3  else
4  |   hStop ← h/2;
5  |   vacStop ← Promote(E, hStop, vacant, h);
6  |   vacParent ← (vacStop − 1)/2;
7  |   if E[vacParent] ≤ K then
8  |   |   E[vacStop] = E[vacParent];
9  |   |   BubbleUpHeap(E, vacant, K, vacParent);
10 |   else
11 |   |   FixHeapFast(E, n, K, vacStop, hStop);
12 |   end
13 end
```

时间复杂度分析

- 直观地看：
  - 如果在 $\frac{h}{2}$ 处 BubbleUpHeap 就被调用，需要最多 $\frac{h}{2}$ 次比较找到 K 的合适位置，而之前 Promote 也用去了 $\frac{h}{2}$ 次元素比较
  - 若在 $\frac{h}{4}$ 处 BubbleUpHeap 被调用，则 Promote 之前共进行了 $\frac{3h}{4}$ 次比较，而 BubbleUpHeap 此时只需返回最多 $\frac{h}{4}$ 的高度
  - 以此类推，递归过程中 Promote 的比较次数加上大约一次 BubbleUpHeap 调用需要约 $h+1$ 次比较
  - FixHeapFast 本身的非递归开销大约为 $\lg h$ 次比较
  - 因此，总的元素比较次数大约为 $h+\lg h$

时间复杂度分析

- 直观地看：
  - 如果在 $\frac{h}{2}$ 处 BubbleUpHeap 就被调用，需要最多 $\frac{h}{2}$ 次比较找到 K 的合适位置，而之前 Promote 也用去了 $\frac{h}{2}$ 次元素比较
  - 若在 $\frac{h}{4}$ 处 BubbleUpHeap 被调用，则 Promote 之前共进行了 $\frac{3h}{4}$ 次比较，而 BubbleUpHeap 此时只需返回最多 $\frac{h}{4}$ 的高度
  - 以此类推，递归过程中 Promote 的比较次数加上大约一次 BubbleUpHeap 调用需要约 $h+1$ 次比较
  - FixHeapFast 本身的非递归开销大约为 $\lg h$ 次比较
  - 因此，总的元素比较次数大约为 $h+\lg h$
- 递推方程：$T(h) = \lceil h/2 \rceil + 1 + \max\left(\lceil h/2 \rceil, T(\lfloor h/2 \rfloor)\right) \quad T(1) = 2$

时间复杂度分析

- 直观地看：
  - 如果在 $\frac{h}{2}$ 处 BubbleUpHeap 就被调用，需要最多 $\frac{h}{2}$ 次比较找到 K 的合适位置，而之前 Promote 也用去了 $\frac{h}{2}$ 次元素比较
  - 若在 $\frac{h}{4}$ 处 BubbleUpHeap 被调用，则 Promote 之前共进行了 $\frac{3h}{4}$ 次比较，而 BubbleUpHeap 此时只需返回最多 $\frac{h}{4}$ 的高度
  - 以此类推，递归过程中 Promote 的比较次数加上大约一次 BubbleUpHeap 调用需要约 $h+1$ 次比较
  - FixHeapFast 本身的非递归开销大约为 $\lg h$ 次比较
  - 因此，总的元素比较次数大约为 $h+\lg h$
- 递推方程：$T(h) = \lceil h/2 \rceil + 1 + \max\left(\lceil h/2 \rceil, T(\lfloor h/2 \rfloor)\right) \quad T(1) = 2$
- 假设 $T(h) \geq h$：$T(h) = \lceil h/2 \rceil + 1 + T(\lfloor h/2 \rfloor) \quad T(1) = 2$

时间复杂度分析

- 直观地看：
  - 如果在 $\frac{h}{2}$ 处 BubbleUpHeap 就被调用，需要最多 $\frac{h}{2}$ 次比较找到 K 的合适位置，而之前 Promote 也用去了 $\frac{h}{2}$ 次元素比较
  - 若在 $\frac{h}{4}$ 处 BubbleUpHeap 被调用，则 Promote 之前共进行了 $\frac{3h}{4}$ 次比较，而 BubbleUpHeap 此时只需返回最多 $\frac{h}{4}$ 的高度
  - 以此类推，递归过程中 Promote 的比较次数加上大约一次 BubbleUpHeap 调用需要约 $h+1$ 次比较
  - FixHeapFast 本身的非递归开销大约为 $\lg h$ 次比较
  - 因此，总的元素比较次数大约为 $h+\lg h$
- 递推方程：$T(h) = \lceil h/2 \rceil + 1 + \max\left(\lceil h/2 \rceil, T(\lfloor h/2 \rfloor)\right) \quad T(1) = 2$
- 假设 $T(h) \geq h$：$T(h) = \lceil h/2 \rceil + 1 + T(\lfloor h/2 \rfloor) \quad T(1) = 2$
  $\Longrightarrow T(h) \approx h + \lceil \lg(h+1) \rceil \approx \lg(n+1) + \lg\lg(n+1)$

时间复杂度分析

- 直观地看：
  - 如果在 $\frac{h}{2}$ 处 BubbleUpHeap 就被调用，需要最多 $\frac{h}{2}$ 次比较找到 K 的合适位置，而之前 Promote 也用去了 $\frac{h}{2}$ 次元素比较
  - 若在 $\frac{h}{4}$ 处 BubbleUpHeap 被调用，则 Promote 之前共进行了 $\frac{3h}{4}$ 次比较，而 BubbleUpHeap 此时只需返回最多 $\frac{h}{4}$ 的高度
  - 以此类推，递归过程中 Promote 的比较次数加上大约一次 BubbleUpHeap 调用需要约 $h+1$ 次比较
  - FixHeapFast 本身的非递归开销大约为 $\lg h$ 次比较
  - 因此，总的元素比较次数大约为 $h+\lg h$
- 递推方程：$T(h) = \lceil h/2 \rceil + 1 + \max\left(\lceil h/2 \rceil, T(\lfloor h/2 \rfloor)\right) \quad T(1) = 2$
- 假设 $T(h) \geq h$：$T(h) = \lceil h/2 \rceil + 1 + T(\lfloor h/2 \rfloor) \quad T(1) = 2$
  $\Longrightarrow T(h) \approx h + \lceil \lg(h+1) \rceil \approx \lg(n+1) + \lg\lg(n+1)$

Theorem

使用 FixHeapFast 来调整堆结构的堆排序算法在最坏情况下其时间复杂度为 $n\lg n + \Theta(n \log\log n)$

主要内容

基本思想

- 回顾：插入排序 (InsertionSort)
  - 如果原始数据的大部分元素已经有序，那么插入排序的速度很快 (因为需要移动的元素很少)
  - 从这个事实我们可以想到，如果原始数据只有很少元素，那么排序的速度也很快 (SmallSort)
  - 插入排序的低效之处在于每次比较交换相邻元素最多只能消除一个反序对

基本思想

- 回顾：插入排序 (InsertionSort)
  - 如果原始数据的大部分元素已经有序，那么插入排序的速度很快 (因为需要移动的元素很少)
  - 从这个事实我们可以想到，如果原始数据只有很少元素，那么排序的速度也很快 (SmallSort)
  - 插入排序的低效之处在于每次比较交换相邻元素最多只能消除一个反序对
- 希尔排序 (Shell Sort) 又称缩小增量排序 (diminishing increment sort)，就是利用插入排序的特点设计的一种很优秀的排序法，算法本身不难理解，也很容易实现，而且它的速度很快，其基本原理为：
  - 指定一组递减增量序列 (最后一个增量为 1)
  - 依次取序列中的增量为矩阵行宽，将待排序序列按矩阵排列
  - 对矩阵每一列进行插入排序

基本思想

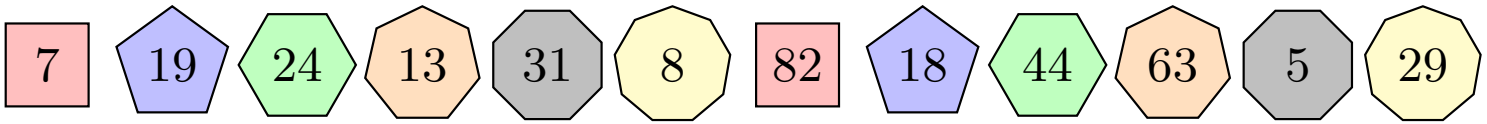
- 回顾：插入排序 (InsertionSort)
  - 如果原始数据的大部分元素已经有序，那么插入排序的速度很快 (因为需要移动的元素很少)
  - 从这个事实我们可以想到，如果原始数据只有很少元素，那么排序的速度也很快 (SmallSort)
  - 插入排序的低效之处在于每次比较交换相邻元素最多只能消除一个反序对
- 希尔排序 (Shell Sort) 又称缩小增量排序 (diminishing increment sort)，就是利用插入排序的特点设计的一种很优秀的排序法，算法本身不难理解，也很容易实现，而且它的速度很快，其基本原理为：
  - 指定一组递减增量序列 (最后一个增量为 1)
  - 依次取序列中的增量为矩阵行宽，将待排序序列按矩阵排列
  - 对矩阵每一列进行插入排序
- 算法名称取自其发明人 D. L. Shell 的名字，于 1959 年发表

示例

7	19	24	13	31	8	82	18	44	63	5	29

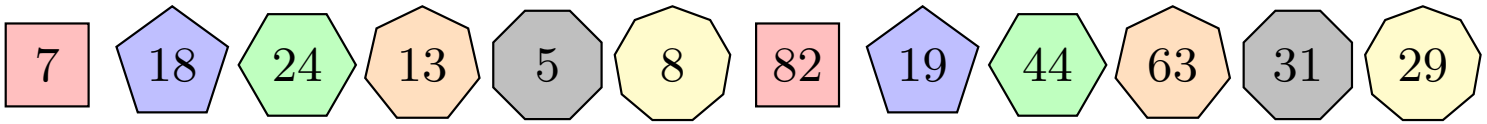
示例

$h_5 = 6$



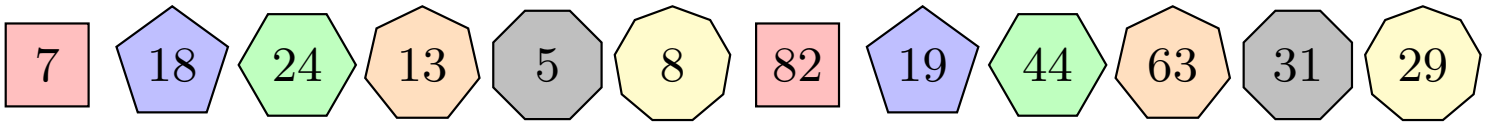
示例

$h_5 = 6$

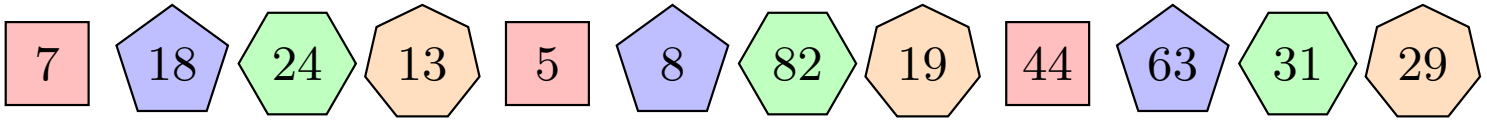


示例

$h_5 = 6$

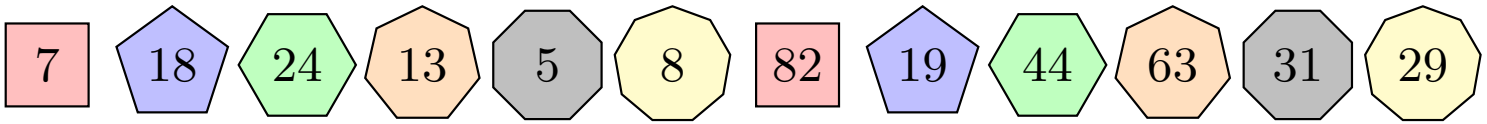


$h_4 = 4$

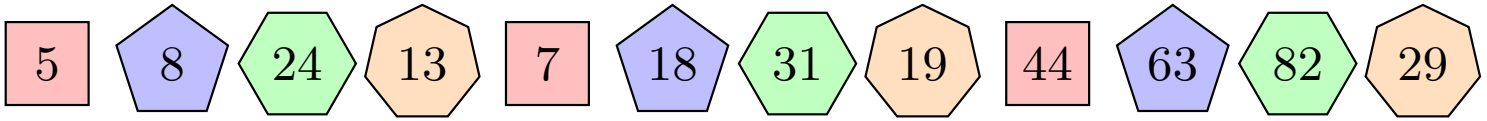


示例

$h_5 = 6$

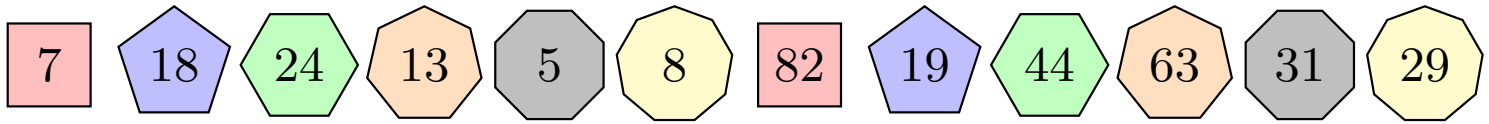


$h_4 = 4$

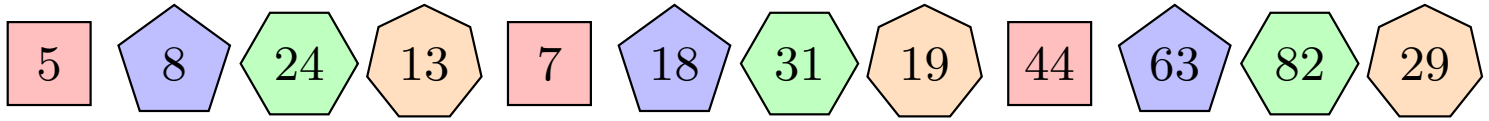


示例

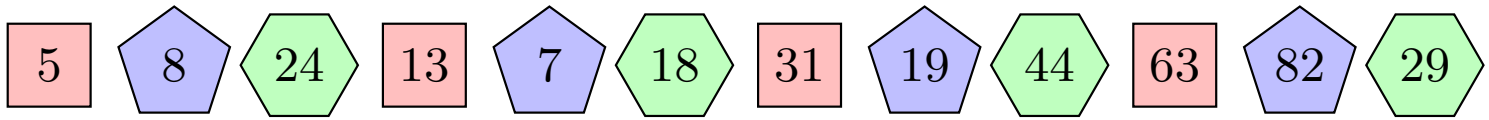
$h_5 = 6$



$h_4 = 4$

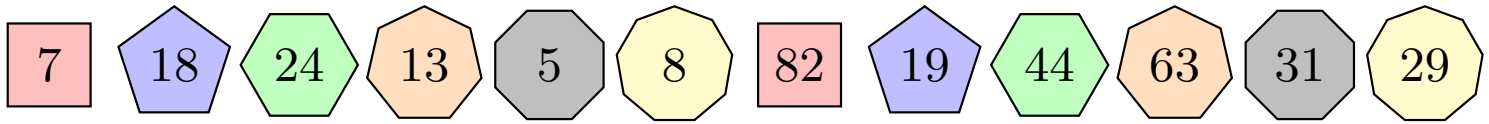


$h_3 = 3$

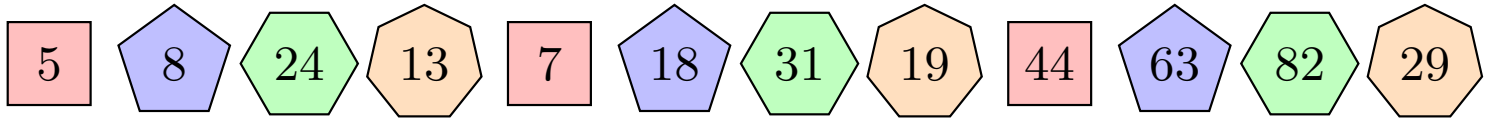


示例

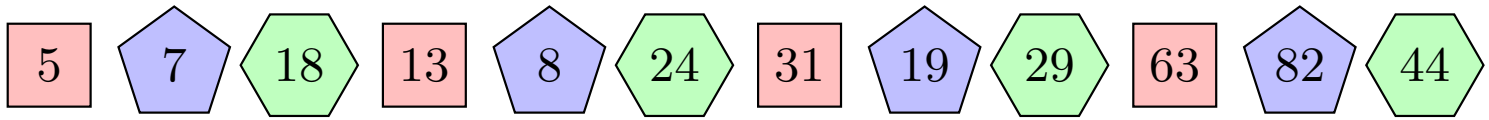
$h_5 = 6$



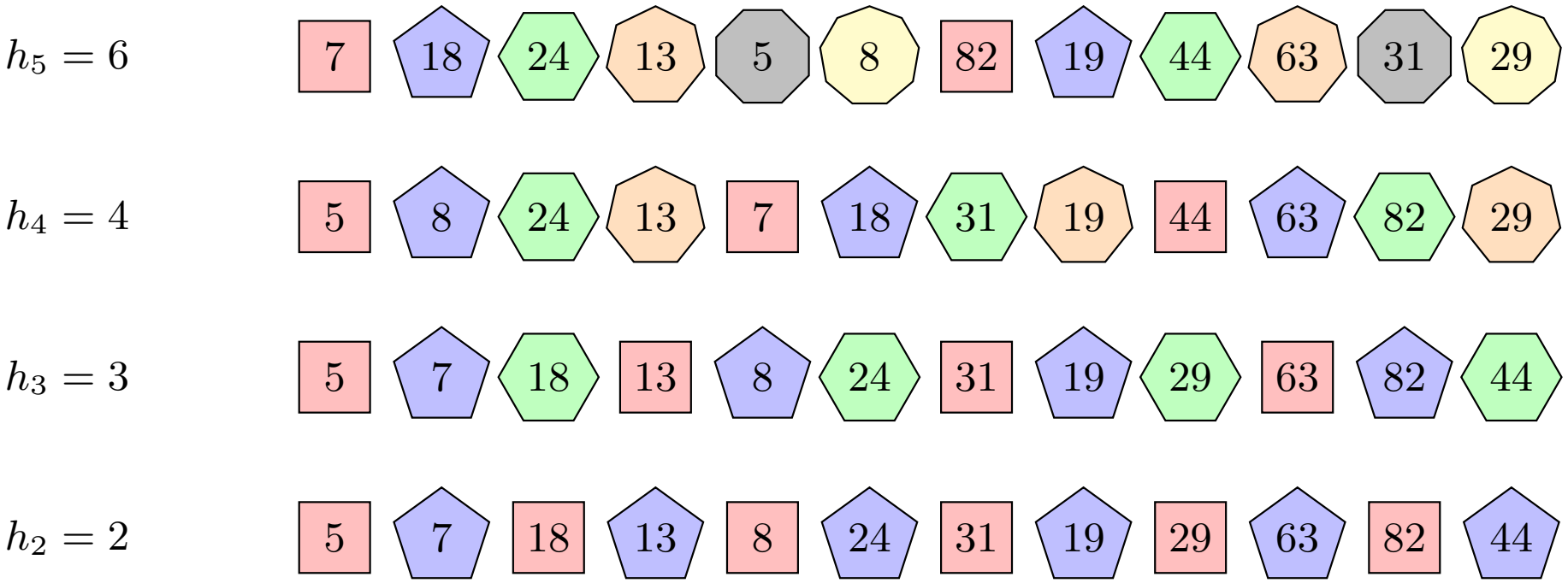
$h_4 = 4$



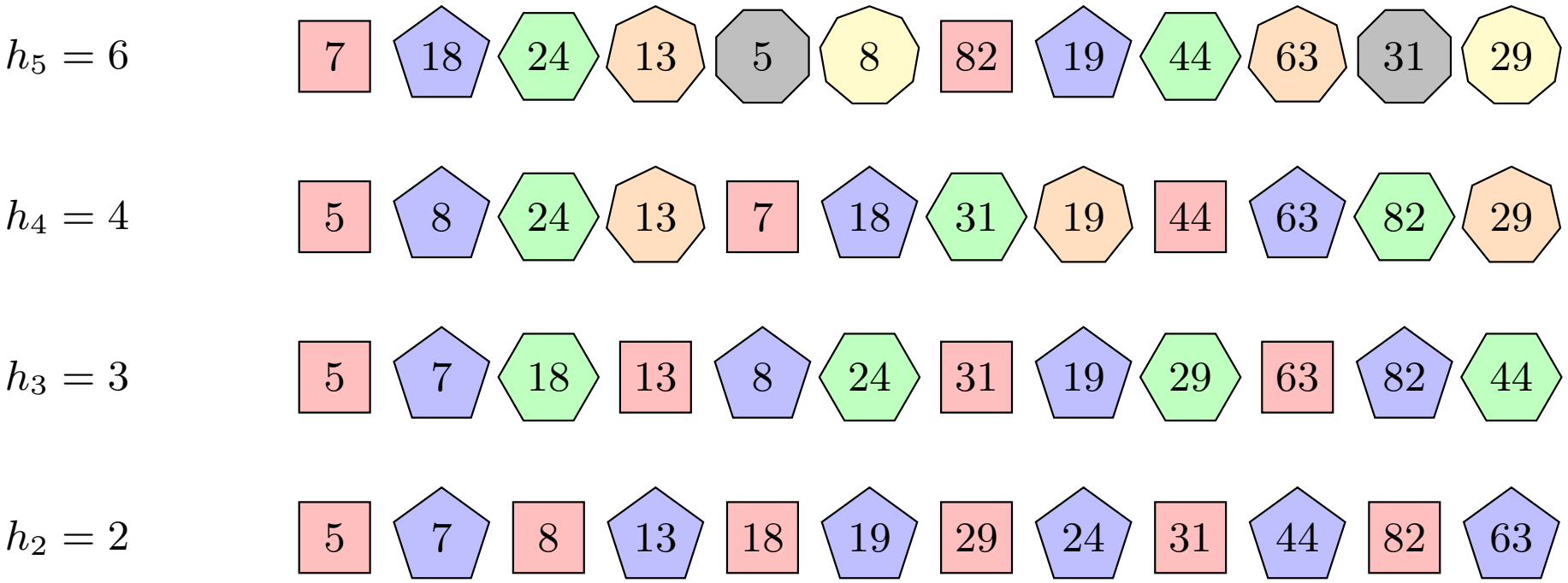
$h_3 = 3$



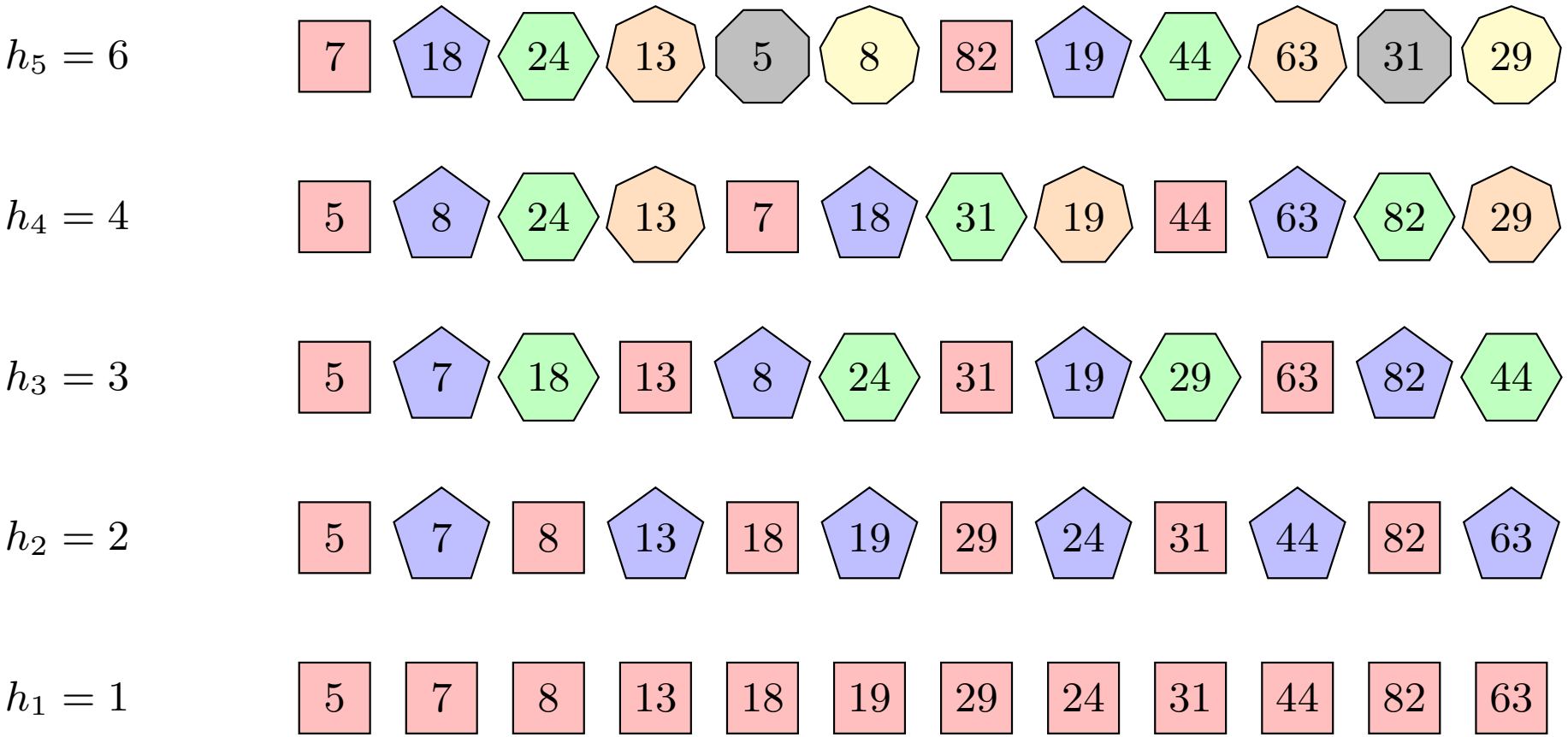
示例


$h_5 = 6$
7 18 24 13 5 8 82 19 44 63 31 29
$h_4 = 4$
5 8 24 13 7 18 31 19 44 63 82 29
$h_3 = 3$
5 7 18 13 8 24 31 19 29 63 82 44
$h_2 = 2$
5 7 18 13 8 24 31 19 29 63 82 44

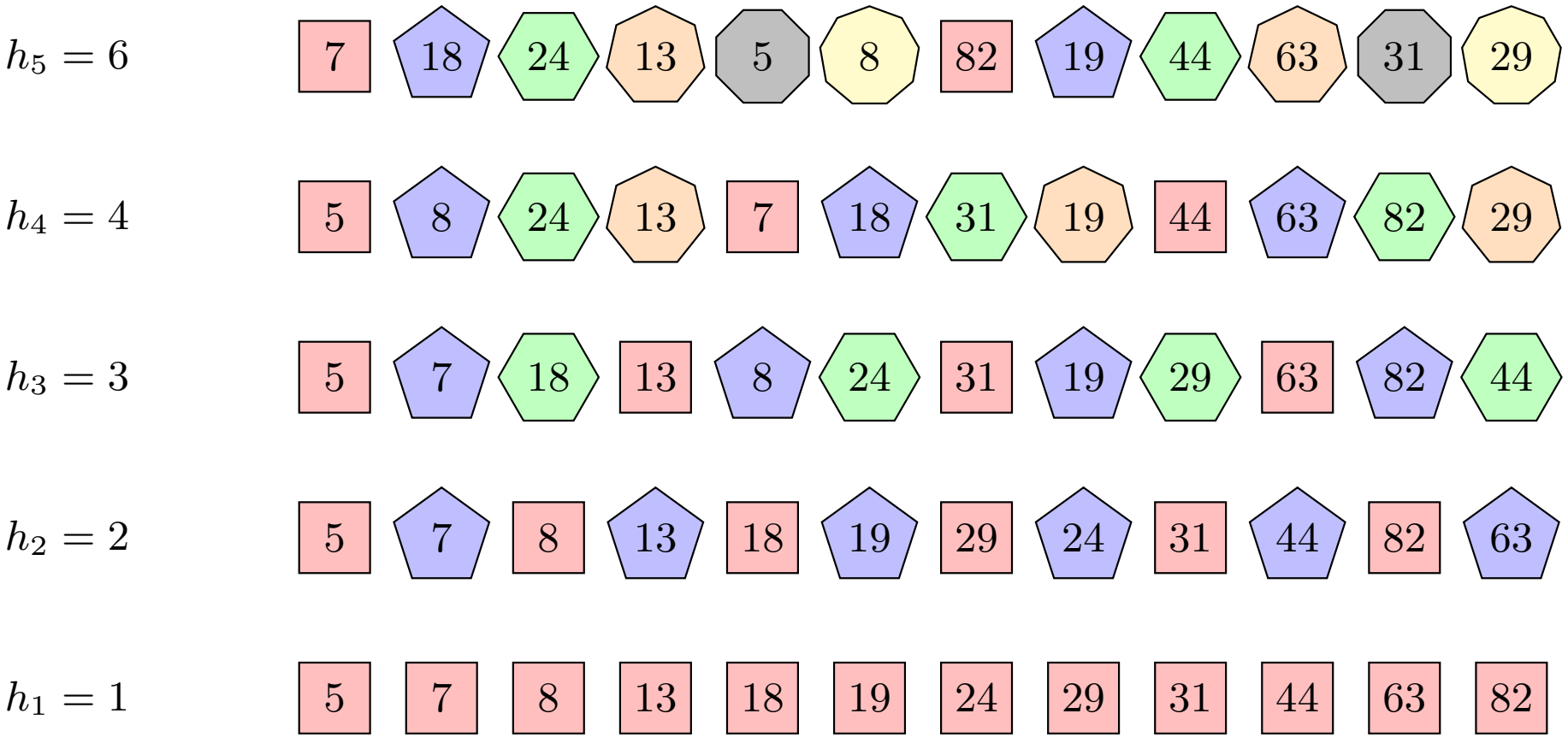
示例


$h_5 = 6$
7 18 24 13 5 8 82 19 44 63 31 29
$h_4 = 4$
5 8 24 13 7 18 31 19 44 63 82 29
$h_3 = 3$
5 7 18 13 8 24 31 19 29 63 82 44
$h_2 = 2$
5 7 8 13 18 19 29 24 31 44 82 63

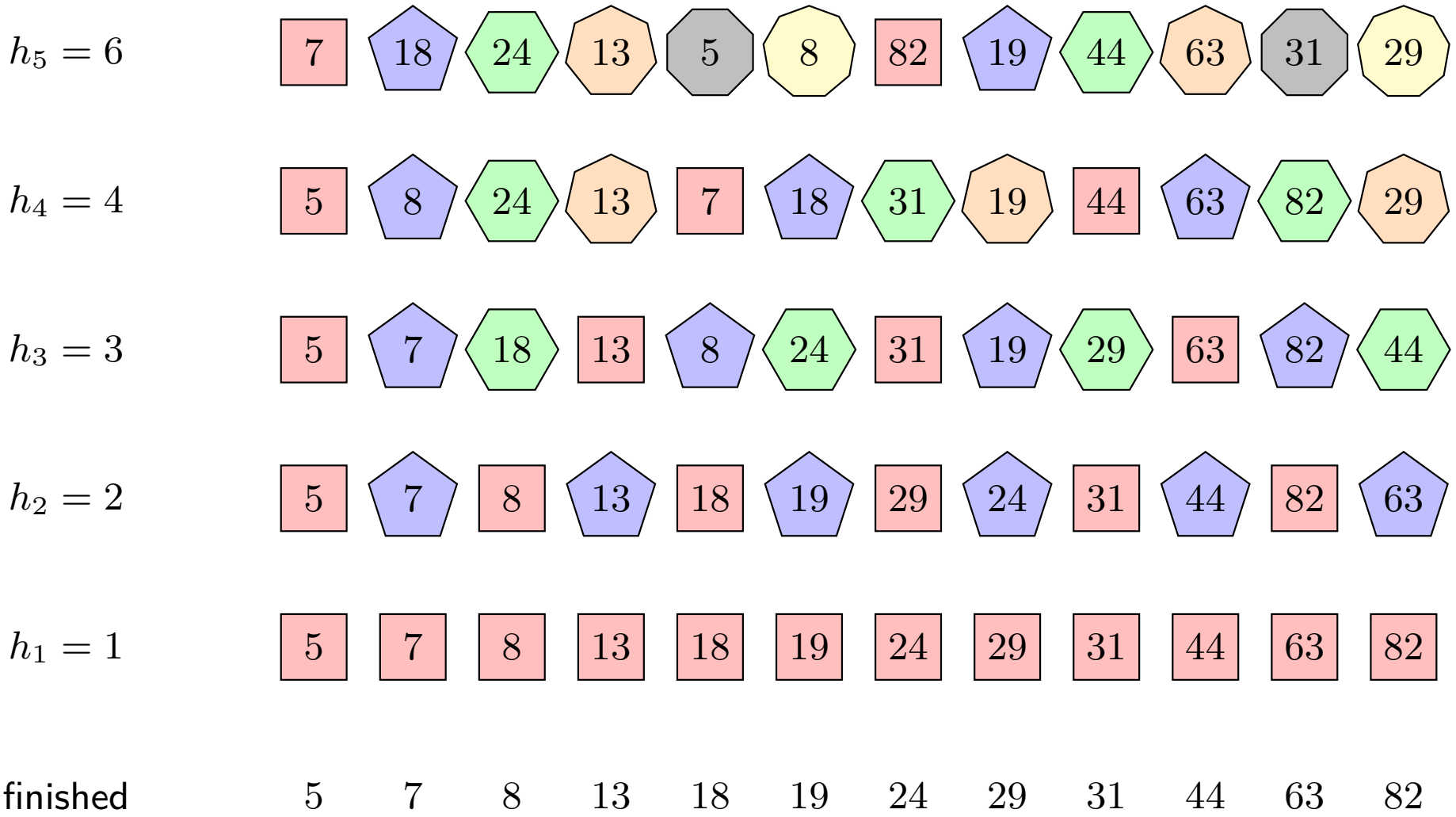
示例


$h_5 = 6$
7 18 24 13 5 8 82 19 44 63 31 29
$h_4 = 4$
5 8 24 13 7 18 31 19 44 63 82 29
$h_3 = 3$
5 7 18 13 8 24 31 19 29 63 82 44
$h_2 = 2$
5 7 8 13 18 19 29 24 31 44 82 63
$h_1 = 1$
5 7 8 13 18 19 29 24 31 44 82 63

示例


$h_5 = 6$
7 18 24 13 5 8 82 19 44 63 31 29
$h_4 = 4$
5 8 24 13 7 18 31 19 44 63 82 29
$h_3 = 3$
5 7 18 13 8 24 31 19 29 63 82 44
$h_2 = 2$
5 7 8 13 18 19 29 24 31 44 82 63
$h_1 = 1$
5 7 8 13 18 19 24 29 31 44 63 82

示例


$h_5 = 6$
7 18 24 13 5 8 82 19 44 63 31 29
$h_4 = 4$
5 8 24 13 7 18 31 19 44 63 82 29
$h_3 = 3$
5 7 18 13 8 24 31 19 29 63 82 44
$h_2 = 2$
5 7 8 13 18 19 29 24 31 44 82 63
$h_1 = 1$
5 7 8 13 18 19 24 29 31 44 63 82
finished
5 7 8 13 18 19 24 29 31 44 63 82

算法描述

Algorithm `Shellsort`$(E[], n, h[], t)$

```
1 for s ← t − 1 to 0 do
2 |   for xindex ← h[s] to n − 1 do
3 |   |   current ← E[xindex];
4 |   |   xloc ←ShiftVacH(E, h[s], xindex, current);
5 |   |   E[xloc] ← current;
6 |   end
7 end
```

ShiftVacH

Procedure ShiftVacH($E[], h, xindex, cur$)

```
1 vacant ← xindex;
2 while vacant ≥ h do
3 |   if E[vacant − h] ≤ cur then
4 |   |   break;
5 |   end
6 |   E[vacant] ← E[vacant − h];
7 |   vacant ← vacant − h;
8 end
9 return vacant;
```

复杂度分析

- Shellsort 的时间复杂度取决于增量序列的选择，其分析难度较大，仍然是一个开放的问题
- 已知复杂度上界：
  - [Pratt, 1971]：$\Theta(n(\log n)^2)$，
    增量序列：$1, 2, 3, 4, 6, 9, 8, 12, 18, 27, \cdots, 2^i 3^j$
  - [Papernov-Stasevich, 1965; Pratt, 1971]：$\Theta(n^{3/2})$，
    增量序列：$1, 3, 7, 15, \cdots, 2^k - 1$
  - [Sedgewick, 1982]：$O(n^{4/3})$，
    增量序列：$1, 8, 23, 77, 281, 1073, 4193, 16577, \cdots, 4^{j+1} + 3 \cdot 2^j + 1$
  - [Incerpi-Sedgewick, 1985]：对于任意 $\epsilon > 0$，必存在一种增量序列使得 Shellsort 的时间复杂度为 $O(n^{1+\epsilon/\log n})$，且只用 $\frac{8}{\epsilon^2} \log n$ 遍排序
- 已知复杂度下界：
  - [Poonen, 1993]：对大小为 n 的序列进行 m 遍排序的 Shellsort 至少需要 $n^{1+c/\sqrt{m}}$ 次比较 $(c > 0)$

复杂度分析 (cont.)

- 平均情况：
  - [Knuth, 1973]：两遍 $(h, 1)$ `Shellsort` 需要 $2n^2/h + \sqrt{\pi n^3 h}$ 次比较 $(h \in O(n^{1/3}) \Rightarrow A(n) \in O(n^{5/3}))$
  - [Yao, 1980]：三遍 $(h, k, 1)$ `Shellsort` 需要 $\frac{2n^2}{h} + \frac{1}{k}\left(\sqrt{\frac{\pi n^3 h}{8}} - \sqrt{\frac{\pi n^3}{8h}}\right) + \psi(h, k)n$ 次比较
- 悬而未决的问题：
  - 是否还存在更好的增量序列？
  - 最坏情况下 `Shellsort` 时间复杂度是否能达到 $O(n \log n)$？
  - 平均情况下 `Shellsort` 时间复杂度是否能达到 $O(n \log n)$？
  - 如果使用其他排序算法代替每遍排序使用的插入排序算法，时间复杂度是否还能降低？
- 参考文献：
  - R. Sedgewick: "Analysis of Shellsort and Related Algorithms." (1996)

主要内容

桶排序

- 前面介绍的排序算法基于这样的假设：基本操作只是元素之间的比较

桶排序

- 前面介绍的排序算法基于这样的假设：基本操作只是元素之间的比较
- 如果改变基本操作或增加其他前提假设，是否会对降低时间复杂度有所帮助？

桶排序

- 前面介绍的排序算法基于这样的假设：基本操作只是元素之间的比较
- 如果改变基本操作或增加其他前提假设，是否会对降低时间复杂度有所帮助？
- 桶排序 (Bucket Sorts)：

桶排序

- 前面介绍的排序算法基于这样的假设：基本操作只是元素之间的比较
- 如果改变基本操作或增加其他前提假设，是否会对降低时间复杂度有所帮助？
- 桶排序 (Bucket Sorts)：
  - 假设已知待排序元素的某些结构特性使其可以等可能的被安排在等间隔的区间内

桶排序

- 前面介绍的排序算法基于这样的假设：基本操作只是元素之间的比较
- 如果改变基本操作或增加其他前提假设，是否会对降低时间复杂度有所帮助？
- 桶排序 (Bucket Sorts)：
  - 假设已知待排序元素的某些结构特性使其可以等可能的被安排在等间隔的区间内
  - 等间隔的区间称为桶，每个桶内存放该区间的元素

桶排序

- 前面介绍的排序算法基于这样的假设：基本操作只是元素之间的比较
- 如果改变基本操作或增加其他前提假设，是否会对降低时间复杂度有所帮助？
- 桶排序 (Bucket Sorts)：
  - 假设已知待排序元素的某些结构特性使其可以等可能的被安排在等间隔的区间内
  - 等间隔的区间称为桶，每个桶内存放该区间的元素
  - 算法的基本思想是：(1) 分配待排序元素到各个桶；(2) 对每个桶内分别进行排序；(3) 将排序结果组合到一起

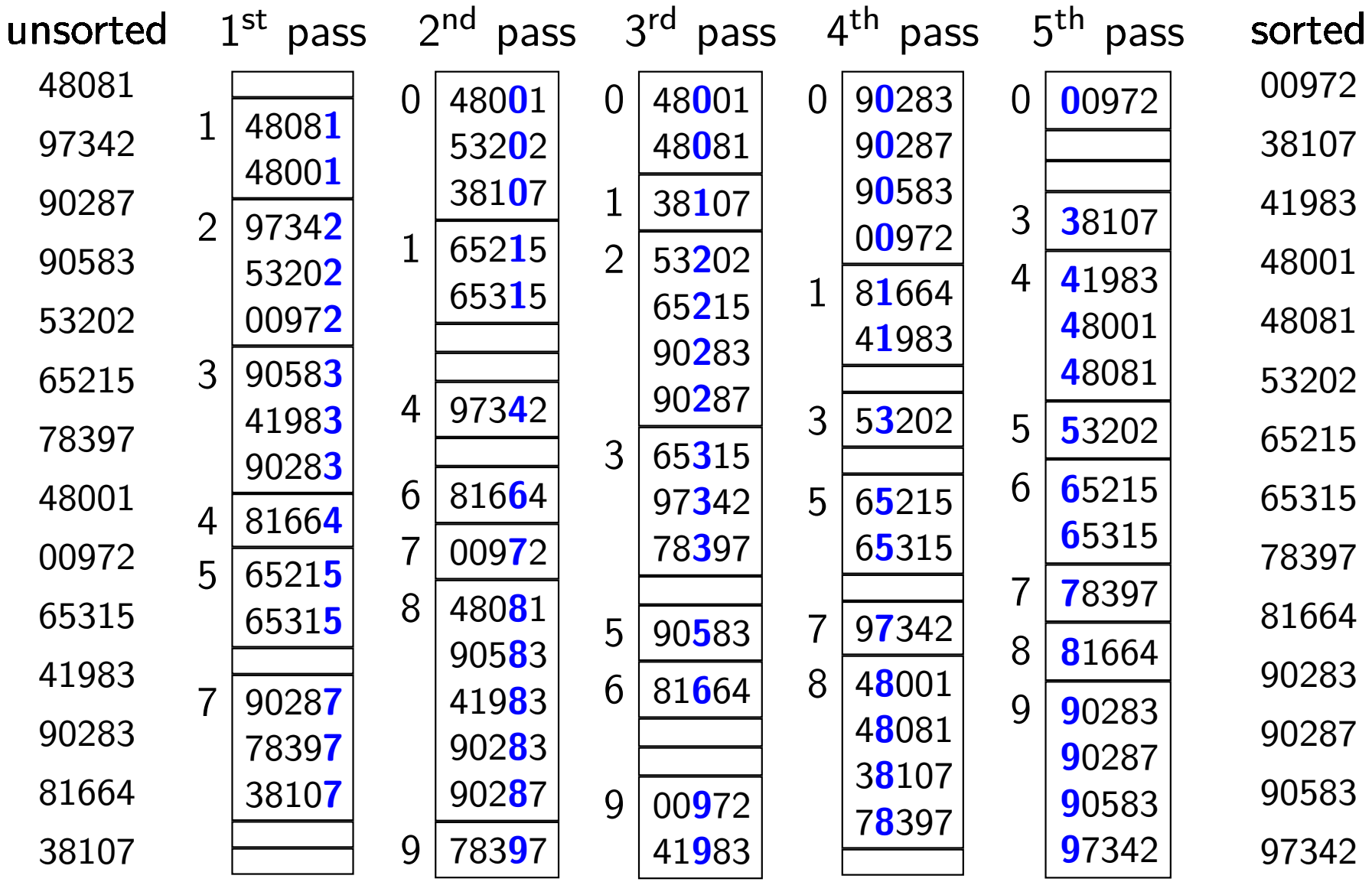
桶排序

- 前面介绍的排序算法基于这样的假设：基本操作只是元素之间的比较
- 如果改变基本操作或增加其他前提假设，是否会对降低时间复杂度有所帮助？
- 桶排序 (Bucket Sorts)：
  - 假设已知待排序元素的某些结构特性使其可以等可能的被安排在等间隔的区间内
  - 等间隔的区间称为桶，每个桶内存放该区间的元素
  - 算法的基本思想是：(1) 分配待排序元素到各个桶；(2) 对每个桶内分别进行排序；(3) 将排序结果组合到一起
  - 假定元素分配尽可能平均，每个桶内排序基于比较操作，则桶排序的时间复杂度可以达到 $O(n\log(n/k))$，其中 k 为桶的个数，若取 $k = n/c$，复杂度为 $O(n\log c) = O(n)$

桶排序

- 前面介绍的排序算法基于这样的假设：基本操作只是元素之间的比较
- 如果改变基本操作或增加其他前提假设，是否会对降低时间复杂度有所帮助？
- 桶排序 (Bucket Sorts)：
  - 假设已知待排序元素的某些结构特性使其可以等可能的被安排在等间隔的区间内
  - 等间隔的区间称为桶，每个桶内存放该区间的元素
  - 算法的基本思想是：(1) 分配待排序元素到各个桶；(2) 对每个桶内分别进行排序；(3) 将排序结果组合到一起
  - 假定元素分配尽可能平均，每个桶内排序基于比较操作，则桶排序的时间复杂度可以达到 $O(n\log(n/k))$，其中 k 为桶的个数，若取 $k = n/c$，复杂度为 $O(n\log c) = O(n)$
- 基数排序 (Radix Sort)：一种复杂度能达到 $O(n)$ 的桶排序算法

基数排序


unsorted
1st pass
2nd pass
3rd pass
4th pass
5th pass
sorted
48081
97342
90287
90583
53202
65215
78397
48001
00972
65315
41983
90283
81664
38107
1 48081 48001
2 97342 53202 00972
3 90583 41983 90283
4 81664
5 65215 65315
7 90287 78397 38107
0 48001 53202 38107
1 65215 65315
4 97342
6 81664
7 00972
8 48081 90583 41983 90283 90287
9 78397
0 48001 48081
1 38107
2 53202 65215 90283 90287
3 65315 97342 78397
5 90583
6 81664
9 00972 41983
0 90283 90287 90583 00972
1 81664 41983
3 53202
5 65215 65315
7 97342
8 48001 48081 38107 78397
0 00972
3 38107
4 41983 48001 48081
5 53202
6 65215 65315
7 78397
8 81664
9 90283 90287 90583 97342
00972
38107
41983
48001
48081
53202
65215
65315
78397
81664
90283
90287
90583
97342

算法描述

- 用链表存储待排序元素和桶元素

Algorithm RadixSort(List L, int $radix$, int $numFields$)

```
1 List[] buckets ← new List[radix];
2 List newL ← L;
3 for field ← 0 to numFields do
4 |   初始化 buckets 中元素为空链表;
5 |   Distribute(newL, buckets, radix, field);
6 |   newL ← Combine(buckets, radix);
7 end
8 return newL;
```

Distribute

Procedure Distribute(List L, List[] $buckets$, $radix$, $field$)

```
1 List remL ← L;
2 while remL ≠ null do
3 |   K ← First(remL);                          /* 取表头元素 */
4 |   b ← MaskShift(field, radix, K);           /* 取当前位数字 */
5 |   buckets[b] ← Cons(K, buckets[b]);         /* 插入到相应桶 */
6 |   remL ← Rest(remL);                        /* 取表尾 */
7 end
```

Combine

Procedure `Combine`(List[] $buckets$, int $radix$)

```
1 for b ← radix − 1 down to 0 do
2 |   remBucket ← buckets[b];
3 |   while remBucket ≠null do
4 |   |   K ← First(remBucket);
5 |   |   L ← Cons(K, L);
6 |   |   remBucket ← Rest(remBucket);
7 |   end
8 end
9 return L;
```

复杂度分析

- 时间复杂度：
  - `Distribute`: $\Theta(n)$
  - `Combine`: $\Theta(n)$
  - `RadixSort`: 总共调用 $numFields$ 次 `Distribute` 和 `Combine`，一般情况下 $numFields$ 为常数，因此 `RadixSort` 复杂度为 $\Theta(n)$
- 空间复杂度：
  - 额外空间主要用在存储元素的链表结构上，所需空间为 $\Theta(n)$

主要内容

排序算法的稳定性

- **排序算法的稳定性**：若待排序的序列中，存在多个具有相同键值的记录，经过排序，这些记录的相对次序保持不变，则称该算法是稳定的；若经排序后，记录的相对次序发生了改变，则称该算法是不稳定的。
- **稳定性的好处**：排序算法如果是稳定的，那么从一个键上排序，然后再从另一个键上排序，第一个键排序的结果可以为第二个键排序所用。
- **何时需要稳定的排序算法**：不希望改变具有相同键值的记录原有的顺序。例如：对学生按成绩排序，对于成绩相同的学生，希望保持原有的学号顺序

排序算法比较

算法	最坏情况	平均情况	空间占用	稳定性
插入排序	$n^2/2$	$n^2/4$	$\Theta(1)$	是
选择排序	$n^2/2$	$n^2/2$	$\Theta(1)$	否
快速排序	$n^2/2$	$1.386n\lg n$	$\Theta(\log n)$	否
归并排序	$n\lg n$	$n\lg n$	$\Theta(n)$	是
堆排序	$2n\lg n$	$\Theta(n\log n)$	$\Theta(1)$	否
快速堆排序	$n\lg n$	$\Theta(n\log n)$	$\Theta(1)$	否
希尔排序	$\Theta(n\log^2 n)$	?	$\Theta(1)$	否
基数排序	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	是

主要内容

主要内容

斯特拉森矩阵乘法

- 由德国数学家 Volker Strassen 于 1969 年提出，被称为 Strassen Algorithm
- A 和 B 是两个 $n \times n$ 矩阵，A 和 B 相加时间复杂度为 $\Theta(n^2)$，A 和 B 相乘时间复杂度为 $\Theta(n^3)$
- 分治法：将矩阵分成大小为 $\frac{n}{2} \times \frac{n}{2}$ 的子矩阵 (假设 $n = 2^k$)

$$AB = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}$$

- $T(n) = 8T(n/2) + c_1 n^2, \quad T(n) = c_2 \ (n \leq 2) \Rightarrow T(n) \in \Theta(n^3)$

斯特拉森矩阵乘法 (cont.)

- 在分治法基础上通过增加加法次数来减少乘法次数

$$AB = \begin{bmatrix} P+S-T+V & R+T \\ Q+S & P+R-Q+U \end{bmatrix}$$

其中：

$$\begin{aligned} P &= (A_{11}+A_{22})(B_{11}+B_{22}) \\ Q &= (A_{21}+A_{22})B_{11} \\ R &= A_{11}(B_{12}-B_{22}) \\ S &= A_{22}(B_{21}-B_{11}) \\ T &= (A_{11}+A_{12})B_{22} \\ U &= (A_{21}-A_{11})(B_{11}+B_{12}) \\ V &= (A_{12}-A_{22})(B_{21}+B_{22}) \end{aligned}$$

- $T(n) = 7T(n/2) + c_1 n^2, \quad T(n) = c_2 \ (n \le 2) \Rightarrow T(n) \in \Theta(n^{2.807})$

主要内容

问题描述

- 在分析算法复杂度时经常将乘法作为基本运算处理，即认为其时间复杂度为 $\Theta(1)$
- 但这样的分析有一个前提，就是计算机字长允许直接表示和处理参与运算的整数
- 在某些情况下需要处理很大的整数，超过了计算机字长表示范围
- 用浮点数运算限制了计算精度和有效数字位数
- 因此，对于大整数相乘，只能以软件的方式设计算法来实现
- 目标：设计一个有效的算法进行两个 n 位大整数的乘法运算

设计思路

- 采用列竖式笔算的方法，若将每 2 个 1 位数的相乘作为基本运算单位，其时间复杂度为 $O(n^2)$

设计思路

- 采用列竖式笔算的方法，若将每 2 个 1 位数的相乘作为基本运算单位，其时间复杂度为 $O(n^2)$
- 分治法思想：
  - 将 2 个 n 位 b 进制整数各分为两段表示如下：(为便于分析，不妨设 $n = 2^k$)
$$X = Ab^{n/2} + B, \quad Y = Cb^{n/2} + D$$
  - 则 $XY = ACb^n + (AD + BC)b^{n/2} + BD$
  - 递归开销为 4 次 $n/2$ 位整数的乘法，非递归开销为 3 次不超过 n 位整数的加法 $(O(n))$、2 次数组移位 $(O(n))$
  - 时间复杂度：$W(n) = 4W(n/2) + O(n),\ W(1) = 1 \Rightarrow W(n) \in \Theta(n^2)$

设计思路

- 采用列竖式笔算的方法，若将每 2 个 1 位数的相乘作为基本运算单位，其时间复杂度为 $O(n^2)$
- 分治法思想：
  - 将 2 个 n 位 b 进制整数各分为两段表示如下：(为便于分析，不妨设 $n = 2^k$)
    $$X = Ab^{n/2} + B, \quad Y = Cb^{n/2} + D$$
  - 则 $XY = ACb^n + (AD + BC)b^{n/2} + BD$
  - 递归开销为 4 次 $n/2$ 位整数的乘法，非递归开销为 3 次不超过 n 位整数的加法 $(O(n))$、2 次数组移位 $(O(n))$
  - 时间复杂度：$W(n) = 4W(n/2) + O(n),\ W(1) = 1 \Rightarrow W(n) \in \Theta(n^2)$
  - 继续改进：$XY = ACb^n + ((A - B)(D - C) + AC + BD)b^{n/2} + BD$
  - 递归开销为 3 次 $n/2$ 位整数的乘法，非递归开销为 6 次加减法和 2 次移位
  - 时间复杂度：
    $W(n) = 3W(n/2) + O(n),\ W(1) = 1 \Rightarrow W(n) \in \Theta(n^{\lg 3}) \in O(n^{1.59})$

算法描述

- 假定整数保存在大小为 $n+1(n>0)$ 的数组 A 中，$A[0]$ 为符号位，取 $+1$ 或 -1，$A[i]$ 对应第 i 位数字

Algorithm `LargeMul`$(X[], Y[], n)$

```
1  S ← Array(2 * n + 1, 0);
2  S[0] ← X[0] * Y[0];
3  if n = 1 then
4  |   S ← Mul(X[1], Y[1]);
5  else
6  |   mid ← n div 2;
7  |   A ← X[mid + 1..n]; B ← X[1..mid];
8  |   C ← Y[mid + 1..n]; D ← Y[1..mid];
9  |   m1 ← LargeMul(A, C, n − mid);
10 |   m2 ← LargeMul(Sub(A, B), Sub(D, C), Max(mid, n − mid));
11 |   S ← LargeMul(B, D, mid);
12 |   S ← Add(S, ShiftLeft(m2, n − mid));
13 |   S ← Add(S, ShiftLeft(m1, 2n − 2mid));
14 end
```

主要内容

问题描述

- 最小套圈问题：给定一个套圈游戏场中的布局，固定每个玩具的位置，试设计圆环套圈的半径尺寸，使得它每次最多只能套中一个玩具。但同时为了让游戏看起来更具有吸引力，这个套圈的半径又需要尽可能大
- 空中交通控制问题：监控空中飞行的飞机，找出其中碰撞危险最大的飞机 (给予警告或修正航线)

问题描述

- 最小套圈问题：给定一个套圈游戏场中的布局，固定每个玩具的位置，试设计圆环套圈的半径尺寸，使得它每次最多只能套中一个玩具。但同时为了让游戏看起来更具有吸引力，这个套圈的半径又需要尽可能大
- 空中交通控制问题：监控空中飞行的飞机，找出其中碰撞危险最大的飞机 (给予警告或修正航线)
- 问题的抽象 —— 最接近点对问题[2]：给定平面上 (空间) n 个点，找其中的一对点，使得在 n 个点的所有点对中，该点对的距离最小

[2] 王晓东. 计算机算法设计与分析 (第 2 版). 电子工业出版, 2004.

问题描述

- 最小套圈问题：给定一个套圈游戏场中的布局，固定每个玩具的位置，试设计圆环套圈的半径尺寸，使得它每次最多只能套中一个玩具。但同时为了让游戏看起来更具有吸引力，这个套圈的半径又需要尽可能大
- 空中交通控制问题：监控空中飞行的飞机，找出其中碰撞危险最大的飞机 (给予警告或修正航线)
- 问题的抽象 —— 最接近点对问题[2]：给定平面上 (空间) n 个点，找其中的一对点，使得在 n 个点的所有点对中，该点对的距离最小
- 解决方案：
  - 两两计算点对距离，找出其中的最小值，复杂度为 $O(n^2)$

[2] 王晓东. 计算机算法设计与分析 (第 2 版). 电子工业出版, 2004.

问题描述

- 最小套圈问题：给定一个套圈游戏场中的布局，固定每个玩具的位置，试设计圆环套圈的半径尺寸，使得它每次最多只能套中一个玩具。但同时为了让游戏看起来更具有吸引力，这个套圈的半径又需要尽可能大
- 空中交通控制问题：监控空中飞行的飞机，找出其中碰撞危险最大的飞机 (给予警告或修正航线)
- 问题的抽象 —— 最接近点对问题[2]：给定平面上 (空间) n 个点，找其中的一对点，使得在 n 个点的所有点对中，该点对的距离最小
- 解决方案：
  - 两两计算点对距离，找出其中的最小值，复杂度为 $O(n^2)$
  - 可以证明该问题的复杂度下界为 $\Omega(n \log n)$

[2] 王晓东. 计算机算法设计与分析 (第 2 版). 电子工业出版, 2004.

问题描述

- 最小套圈问题：给定一个套圈游戏场中的布局，固定每个玩具的位置，试设计圆环套圈的半径尺寸，使得它每次最多只能套中一个玩具。但同时为了让游戏看起来更具有吸引力，这个套圈的半径又需要尽可能大
- 空中交通控制问题：监控空中飞行的飞机，找出其中碰撞危险最大的飞机 (给予警告或修正航线)
- 问题的抽象 —— 最接近点对问题[2]：给定平面上 (空间) n 个点，找其中的一对点，使得在 n 个点的所有点对中，该点对的距离最小
- 解决方案：
  - 两两计算点对距离，找出其中的最小值，复杂度为 $O(n^2)$
  - 可以证明该问题的复杂度下界为 $\Omega(n \log n)$
  - 是否存在复杂度为 $\Theta(n \log n)$ 的算法？

[2] 王晓东. 计算机算法设计与分析 (第 2 版). 电子工业出版, 2004.

问题描述

- 最小套圈问题：给定一个套圈游戏场中的布局，固定每个玩具的位置，试设计圆环套圈的半径尺寸，使得它每次最多只能套中一个玩具。但同时为了让游戏看起来更具有吸引力，这个套圈的半径又需要尽可能大
- 空中交通控制问题：监控空中飞行的飞机，找出其中碰撞危险最大的飞机 (给予警告或修正航线)
- 问题的抽象 —— 最接近点对问题[2]：给定平面上 (空间) n 个点，找其中的一对点，使得在 n 个点的所有点对中，该点对的距离最小
- 解决方案：
  - 两两计算点对距离，找出其中的最小值，复杂度为 $O(n^2)$
  - 可以证明该问题的复杂度下界为 $\Omega(n \log n)$
  - 是否存在复杂度为 $\Theta(n \log n)$ 的算法？ —— 分治法？

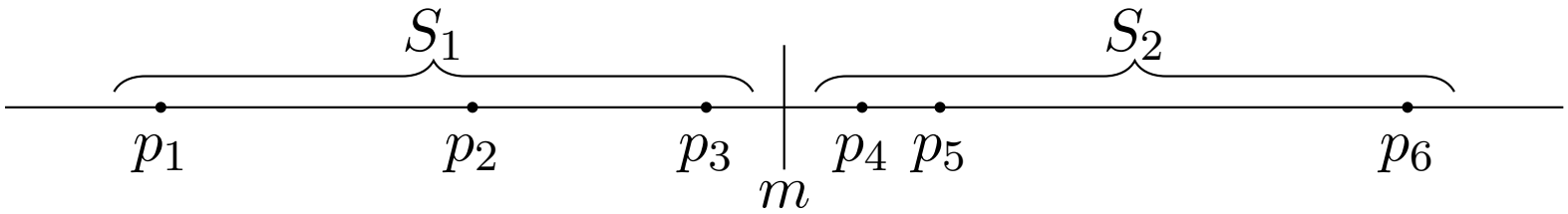
[2] 王晓东. 计算机算法设计与分析 (第 2 版). 电子工业出版, 2004.

分治法思路

- 将所给的平面上 n 个点的集合 S 分成 2 个子集 S_1 和 S_2，每个子集中约有 $n/2$ 个点
- 在每个子集中递归地求其最接近的点对，但 S_1 和 S_2 的最接近点对未必就是 S 的最接近点对
- 关键问题：如何合并子问题求解的结果，即由 S_1 和 S_2 的最接近点对，如何求得原集合 S 中的最接近点对

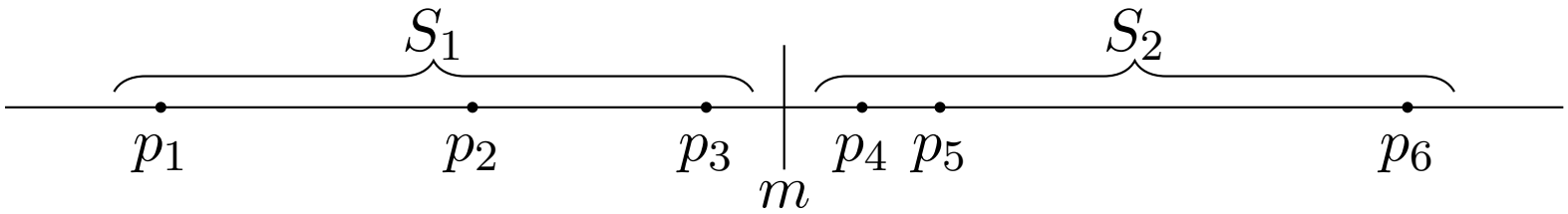
分治法思路

- 将所给的平面上 n 个点的集合 S 分成 2 个子集 S_1 和 S_2，每个子集中约有 $n/2$ 个点
- 在每个子集中递归地求其最接近的点对，但 S_1 和 S_2 的最接近点对未必就是 S 的最接近点对
- 关键问题：如何合并子问题求解的结果，即由 S_1 和 S_2 的最接近点对，如何求得原集合 S 中的最接近点对
- 一维的情况：



分治法思路

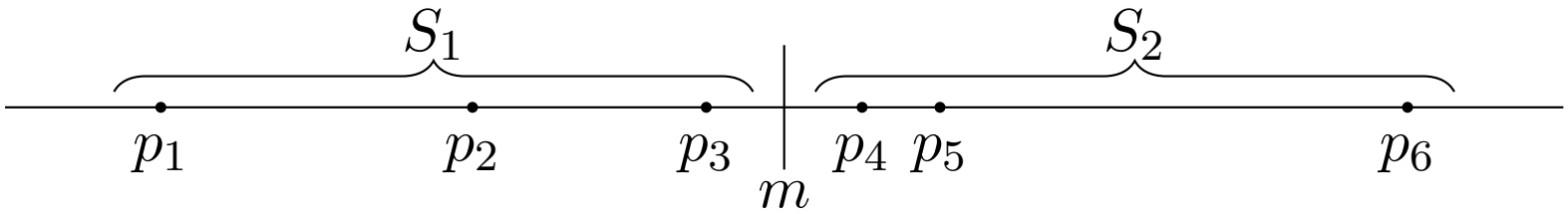
- 将所给的平面上 n 个点的集合 S 分成 2 个子集 S_1 和 S_2，每个子集中约有 $n/2$ 个点
- 在每个子集中递归地求其最接近的点对，但 S_1 和 S_2 的最接近点对未必就是 S 的最接近点对
- 关键问题：如何合并子问题求解的结果，即由 S_1 和 S_2 的最接近点对，如何求得原集合 S 中的最接近点对
- 一维的情况：



  - 假定分割点为 m，则如果出现最小点对分别落在 S_1 和 S_2 中的情况，这两点只可能是 S_1 中的最大点和 S_2 中的最小点 $(O(n))$

分治法思路

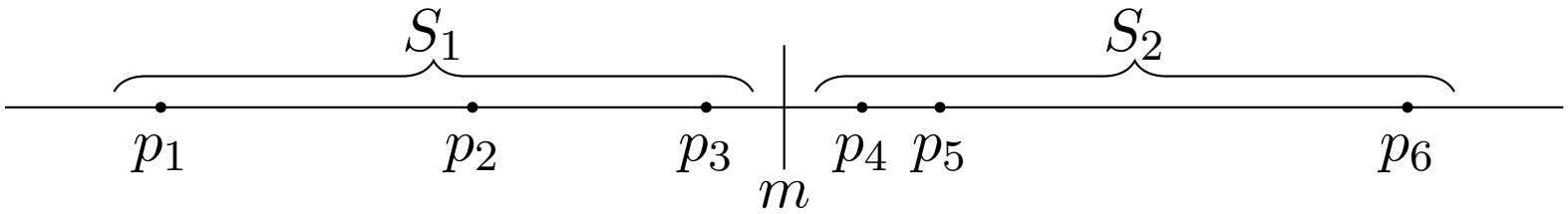
- 将所给的平面上 n 个点的集合 S 分成 2 个子集 S_1 和 S_2，每个子集中约有 $n/2$ 个点
- 在每个子集中递归地求其最接近的点对，但 S_1 和 S_2 的最接近点对未必就是 S 的最接近点对
- 关键问题：如何合并子问题求解的结果，即由 S_1 和 S_2 的最接近点对，如何求得原集合 S 中的最接近点对
- 一维的情况：



  - 假定分割点为 m，则如果出现最小点对分别落在 S_1 和 S_2 中的情况，这两点只可能是 S_1 中的最大点和 S_2 中的最小点 $(O(n))$
  - 分割点 m 的选择：中位数 $(\Theta(n))$

分治法思路

- 将所给的平面上 n 个点的集合 S 分成 2 个子集 S_1 和 S_2，每个子集中约有 $n/2$ 个点
- 在每个子集中递归地求其最接近的点对，但 S_1 和 S_2 的最接近点对未必就是 S 的最接近点对
- 关键问题：如何合并子问题求解的结果，即由 S_1 和 S_2 的最接近点对，如何求得原集合 S 中的最接近点对
- 一维的情况：



  - 假定分割点为 m，则如果出现最小点对分别落在 S_1 和 S_2 中的情况，这两点只可能是 S_1 中的最大点和 S_2 中的最小点 $(O(n))$
  - 分割点 m 的选择：中位数 $(\Theta(n))$
  - 时间复杂度：$W(n) = 2W(n/2) + O(n) \Rightarrow W(n) \in \Theta(n \log n)$

一维情况算法描述

Algorithm `MinPairI`(S)

```
1  if |S| < 2 then d ← ∞ ;
2  else
3      m ← S 中点坐标中位数;
4      S_1 ← {x ∈ S | x ≤ m};
5      S_2 ← {x ∈ S | x > m};
6      d_1 ← MinPairI(S_1);
7      d_2 ← MinPairI(S_2);
8      x_1 ← max(S_1);
9      x_2 ← min(S_2);
10     d ← min(d_1, d_2, x_2 − x_1);
11 end
12 return d;
```

推广到二维

- 点集分割：为了将平面上点集线性分割为大小大致相等的 2 个子集 S_1 和 S_2，选取一垂直线 $l : x = m$ 来作为分割直线。其中 m 为 S 中各点 x 坐标的中位数
- 递归求解：分别得到 S_1 和 S_2 中的最小距离 d_1 和 d_2
- 若设 $d = \min(d_1, d_2)$，则如果 S 中的最接近点对 (p, q) 距离小于 d，则 p 和 q 必分属于 S_1 和 S_2，如何找到距离可能小于 d 的点对？

推广到二维

- 点集分割：为了将平面上点集线性分割为大小大致相等的 2 个子集 S_1 和 S_2，选取一垂直线 $l : x = m$ 来作为分割直线。其中 m 为 S 中各点 x 坐标的中位数
- 递归求解：分别得到 S_1 和 S_2 中的最小距离 d_1 和 d_2
- 若设 $d = \min(d_1, d_2)$，则如果 S 中的最接近点对 (p, q) 距离小于 d，则 p 和 q 必分属于 S_1 和 S_2，如何找到距离可能小于 d 的点对？
  - p, q 必然位于直线 l 两侧左右各宽 d 的垂直长条内，设为 P_1、P_2

推广到二维

- 点集分割：为了将平面上点集线性分割为大小大致相等的 2 个子集 S_1 和 S_2，选取一垂直线 $l : x = m$ 来作为分割直线。其中 m 为 S 中各点 x 坐标的中位数
- 递归求解：分别得到 S_1 和 S_2 中的最小距离 d_1 和 d_2
- 若设 $d = \min(d_1, d_2)$，则如果 S 中的最接近点对 (p, q) 距离小于 d，则 p 和 q 必分属于 S_1 和 S_2，如何找到距离可能小于 d 的点对？
  - p, q 必然位于直线 l 两侧左右各宽 d 的垂直长条内，设为 P_1、P_2
  - 对于 P_1 中任一候选点，P_2 中能与其构成候选点对的点必然落在一个 $d \times 2d$ 的矩形 R 中

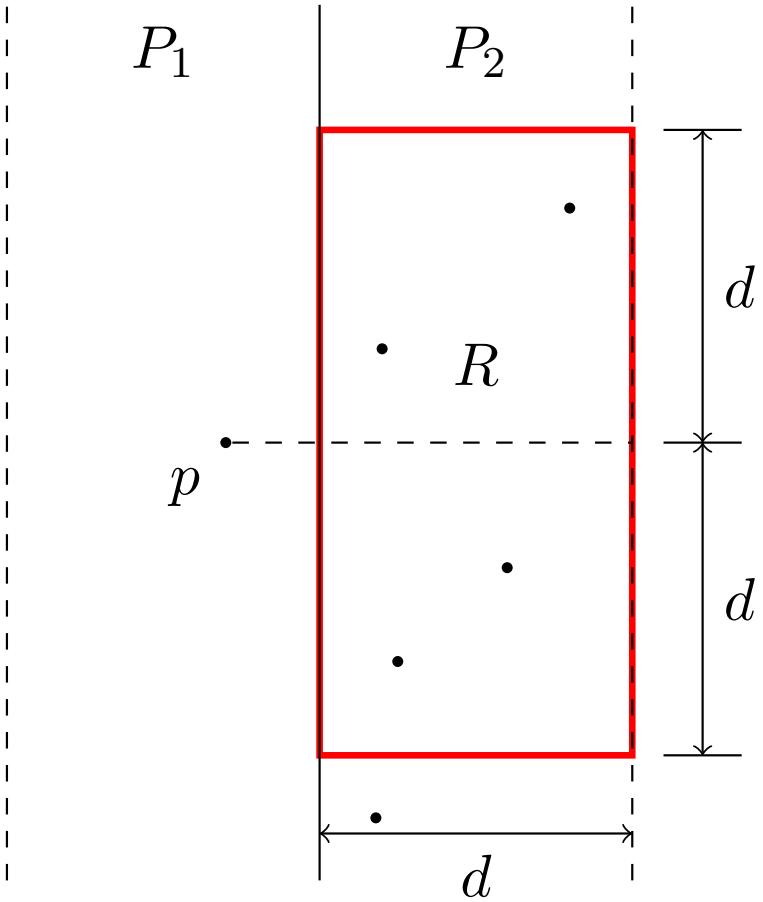
推广到二维

- 点集分割：为了将平面上点集线性分割为大小大致相等的 2 个子集 S_1 和 S_2，选取一垂直线 $l : x = m$ 来作为分割直线。其中 m 为 S 中各点 x 坐标的中位数
- 递归求解：分别得到 S_1 和 S_2 中的最小距离 d_1 和 d_2
- 若设 $d = \min(d_1, d_2)$，则如果 S 中的最接近点对 (p, q) 距离小于 d，则 p 和 q 必分属于 S_1 和 S_2，如何找到距离可能小于 d 的点对？
  - p, q 必然位于直线 l 两侧左右各宽 d 的垂直长条内，设为 P_1、P_2
  - 对于 P_1 中任一候选点，P_2 中能与其构成候选点对的点必然落在一个 $d \times 2d$ 的矩形 R 中
  - 矩形 R 中最多只有 6 个点 (鸽笼原理)

推广到二维

- 点集分割：为了将平面上点集线性分割为大小大致相等的 2 个子集 S_1 和 S_2，选取一垂直线 $l : x = m$ 来作为分割直线。其中 m 为 S 中各点 x 坐标的中位数
- 递归求解：分别得到 S_1 和 S_2 中的最小距离 d_1 和 d_2
- 若设 $d = \min(d_1, d_2)$，则如果 S 中的最接近点对 (p, q) 距离小于 d，则 p 和 q 必分属于 S_1 和 S_2，如何找到距离可能小于 d 的点对？
  - p, q 必然位于直线 l 两侧左右各宽 d 的垂直长条内，设为 P_1、P_2
  - 对于 P_1 中任一候选点，P_2 中能与其构成候选点对的点必然落在一个 $d \times 2d$ 的矩形 R 中
  - 矩形 R 中最多只有 6 个点 (鸽笼原理)
  - 如何在线性时间复杂度内找到这 6 个点？—— 对 y 坐标预排序

示意图



算法描述

Algorithm MinPairIIRec(P)

1 **if** $|S| < 2$ **then** $d \leftarrow \infty$;
2 **else**
3 $m \leftarrow P$ 中点的 x 坐标中位数;
4 $P_1[] \leftarrow \{P[i]_x \leq m\}$;
5 $P_2[] \leftarrow \{P[i]_x > m\}$;
6 $d_1 \leftarrow$ MinPairIIRec(P_1);
7 $d_2 \leftarrow$ MinPairIIRec(P_2);
8 $d \leftarrow d_m \leftarrow \min(d_1, d_2)$;
9 $P_1^*[] \leftarrow \{m - P_1[i]_x < d_m\}$;
10 $P_2^*[] \leftarrow \{P_2[i]_x - m < d_m\}$;
11 **foreach** $P_1^*[i]$ **do**
12 搜索 P_2^* 中与 $P_1^*[i]$ 距离小于 d_m 的点 (最多 6 个) 更新 d;
13 **end**
14 **end**
15 **return** d;

算法分析

- 3, 4, 5, 9, 10 显然复杂度为 $O(n)$
- 8 的开销是常数
- 6, 7 是递归部分，复杂度为 $2W(n/2)$
- 关键是 11 ～ 13，在 P 是有序的前提下才能够达到线性复杂度，而且在递归调用的任何时候都不会破坏子集中的点 y 坐标的有序性
- 在上述条件下，`MinPairIIRec` 的时间复杂度：
  $W(n) = 2W(n/2) + O(n) \Rightarrow W(n) \in \Theta(n \log n)$

算法分析

- 3, 4, 5, 9, 10 显然复杂度为 $O(n)$
- 8 的开销是常数
- 6, 7 是递归部分，复杂度为 $2W(n/2)$
- 关键是 11 ～ 13，在 P 是有序的前提下才能够达到线性复杂度，而且在递归调用的任何时候都不会破坏子集中的点 y 坐标的有序性
- 在上述条件下，`MinPairIIRec` 的时间复杂度：
  $W(n) = 2W(n/2) + O(n) \Rightarrow W(n) \in \Theta(n \log n)$
- 因此在调用 `MinPairIIRec` 之前需要对 P 中的点按 y 坐标排序，排序的时间复杂度可以达到 $\Theta(n \log n)$，所以不影响整个算法的时间复杂度

Procedure `MinPairII`(S)

---

1 $P \leftarrow S$ 中的点按 y 坐标排序;
2 `MinPairIIRec`(P);

主要内容

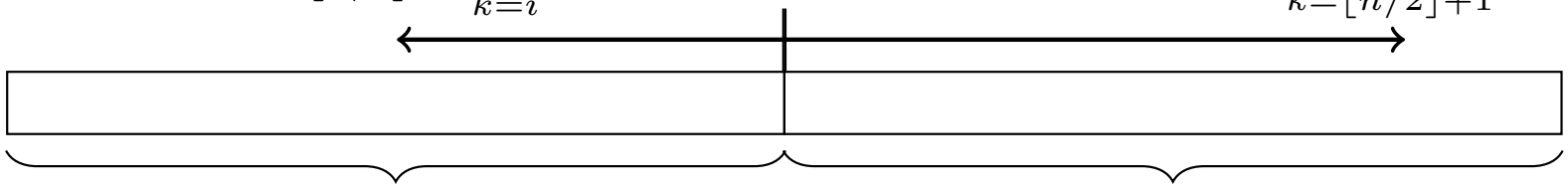
问题描述及分析

- 某公司过去 n 天的股票价格波动保存在数组 A 中，求在哪段时间 (连续哪几天) 其股价的累计增长最大
- 例如：过去 7 天内其股价波动序列为 $+3, -6, +5, +2, -3, +4, -4$，累计增长最大值出现在第 3 天到第 6 天，累计增长值为 $5 + 2 - 3 + 4 = 8$
- 实际上就是求 i 和 j $(0 \leq i \leq j \leq n-1)$，使 $\sum_{k=i}^{j} A[k]$ 最大

问题描述及分析

- 某公司过去 n 天的股票价格波动保存在数组 A 中，求在哪段时间 (连续哪几天) 其股价的累计增长最大
- 例如：过去 7 天内其股价波动序列为 $+3, -6, +5, +2, -3, +4, -4$，累计增长最大值出现在第 3 天到第 6 天，累计增长值为 $5+2-3+4=8$
- 实际上就是求 i 和 j $(0 \le i \le j \le n-1)$，使 $\sum_{k=i}^{j} A[k]$ 最大
- 采用分治法设计算法：

$$L = \max_{0 \le i \le \lfloor n/2 \rfloor} \Big(\sum_{k=i}^{\lfloor n/2 \rfloor} A[k] \Big) \qquad R = \max_{\lfloor n/2 \rfloor + 1 \le j \le n} \Big(\sum_{k=\lfloor n/2 \rfloor + 1}^{j} A[k] \Big)$$



$$m_l = \max_{0 \le i \le j \le \lfloor n/2 \rfloor} \Big(\sum_{k=i}^{j} A[k] \Big) \qquad m_r = \max_{\lfloor n/2 \rfloor + 1 \le i \le j \le n} \Big(\sum_{k=i}^{j} A[k] \Big)$$

$$\boldsymbol{m = \max(m_l, m_r, L + R)}$$

算法描述

Algorithm `LargestIncrease`$(A[], first, last)$

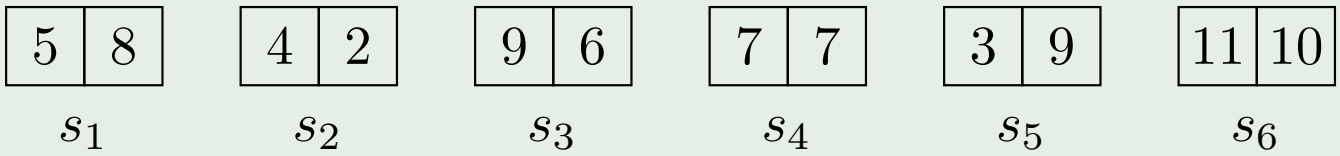
```
1  if first ≥ last then i ← first, j ← last, m ← A[last] ;
2  else
3      mid ← ⌊(first + last)/2⌋;
4      (il, jl, ml) ← LargestIncrease(A, first, mid);
5      (ir, jr, mr) ← LargestIncrease(A, mid + 1, last);
6      找到 im 使得 L = Σ_{k=im}^{mid} A[k] 最大;
7      找到 jm 使得 R = Σ_{k=mid+1}^{jm} A[k] 最大;
8      if ml ≥ mr and ml ≥ L + R then i ← il, j ← jl, m ← ml ;
9      else if mr ≥ ml and mr ≥ L + R then i ← ir, j ← jr, m ← mr ;
10     else i ← im, j ← jm, m ← L + R ;
11 end
12 return (i, j, m);
```

- 算法复杂度：$W(n) = 2W(n/2) + \Theta(n) \Rightarrow \Theta(n \log n)$

课后习题：多米诺骨牌

Exercise (4)

现有 n 块“多米诺骨牌” $s_1, s_2, \cdots, s_n$ 水平放成一排，每块骨牌 s_i 包含左右两个部分，每个部分赋予一个非负整数值，如下图所示为包含 6 块骨牌的序列。骨牌可做 180 度旋转，使得原来在左边的值变到右边，而原来在右边的值移到左边，假设不论 s_i 如何旋转，$L[i]$ 总是存储 s_i 左边的值，$R[i]$ 总是存储 s_i 右边的值，$W[i]$ 用于存储 s_i 的状态：
当 $L[i] \leq R[i]$ 时记为 0，否则记为 1，试采用分治法设计算法
求 $\sum_{i=1}^{n-1} R[i] \cdot L[i+1]$ 的最大值，以及当取得最大值时每个骨牌的状态。



deadline: 2015.04.11

本讲回顾

1. 递归和数学归纳法
   - 递归
   - 数学归纳法
   - 程序正确性证明
   - 递推方程
2. 分治法
3. 排序算法设计
   - 插入排序
   - 快速排序
   - 归并排序
   - 比较排序的复杂度下界
   - 堆排序
   - 希尔排序
   - 基数排序
   - 排序算法比较
4. 其他分治法实例
   - 矩阵相乘
   - 大整数乘法
   - 最接近点对问题
   - 股价增值问题