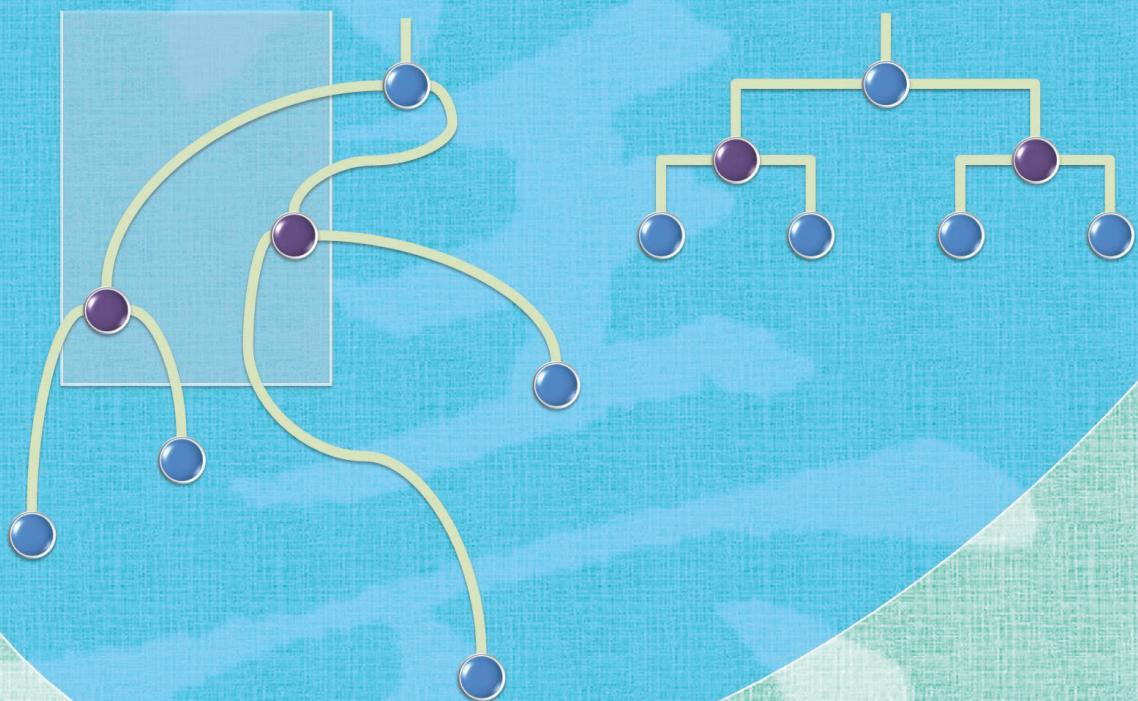
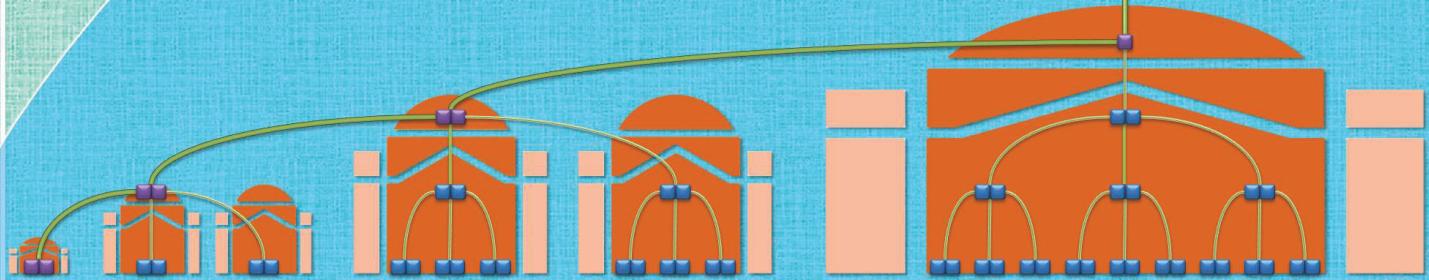


数据结构



A stylized white signature or logo, possibly representing the professor's name.

清华大学·2019秋



目 录

慕课视频 | 页码 | 章节

1. 绪论		
		A. 计算
01A[1~3]	1	1. 工具
01A[4~6]	4	2. 算法
		B. 计算模型
01B[1~4]	10	1. 统一尺度
01B[5~6]	14	2. 图灵机
	18	3. RAM
		C. 渐进复杂度
01C[1~2]	23	1. 大 O 记号
01C[3~4]	27	2. 多项式
01C[5~6]	32	3. 指数
01C7	36	4. 复杂度层次
		D. 复杂度分析
01D[1~2]	41	1. 级数
01D[3~4]	46	2. 迭代
01D[6~7]	50	3. 封底估算
		E. 迭代与递归
01E[1~5]	53	1. 减而治之
01E[6~7]	59	2. 分而治之
01E[8~9]	65	3. 总和最大区段
		XA. 动态规划
01XC[1~5]	72	1. 记忆化
01XC[6~A]	79	2. 最长公共子序列

		XB. 局限	
	87	1. 缓存	
	93	2. 字宽	
	99	3. 随机数	
		XC. 下界	
	101	1. 代数判定树	
CG-01F	108	2. 归约	
2. 向量			
	02A	A. 抽象数据类型	
	111	1. 接口与实现	
	114	2. 从数组到向量	
	119	3. 模板类	
		B. 可扩充向量	
	02B[1~4]	123	1. 算法
	02B5	127	2. 分摊
		C. 无序向量	
	02C[1~5]	132	1. 基本操作
	02C6	137	2. 查找
	02C7	141	3. 去重
	02C8	143	4. 遍历
		D. 有序向量	
	02D1	146	1. 唯一化
	02D2	151	2. 二分查找 (A)
	02D3	159	3. Fib 查找

02D4	165	4. 二分查找 (B)	03XD	268	I. 逆序对
02D4	170	5. 二分查找 (C)		273	XA. 游标实现
02D5	174	6. 插值查找		279	XB. Java 序列
02E	179	E. 起泡排序		286	XC. Python 列表
		F. 归并排序			
02F[1~2]	187	1. 分而治之			
02F[3~5]	190	2. 二路归并			
02F6	194	3. 复杂度	04A	291	A. 栈 ADT 及实现
		XA. 位图			B. 调用栈
	197	1. 结构		295	1. 原理与算法
	201	2. 应用		300	2. 实例
	208	3. 快速初始化		304	3. 消除递归
				307	4. 尾递归
			04C1	311	C. 进制转换
			04C2	316	D. 括号匹配
	215	A. 循位置访问	04C3	324	E. 栈混淆
03A	220	B. 接口与实现	04C4		F. 中缀表达式求值
		C. 无序列表		332	1. 问题与构思
03B3+03B4	227	1. 插入与删除		338	2. 算法
03B3+03B4	232	2. 构造与析构		348	3. 实例
03B2+03B5	235	3. 查找与去重	04C5		G. 逆波兰表达式
	238	4. 遍历		353	1. 定义与求值
		D. 有序列表		357	2. 转换
03C[1~2]	240	1. 唯一化		361	3. PostScript
03C3	242	2. 查找	04D	364	H. 队列 ADT 及实现
03D	245	E. 选择排序		368	I. 队列应用
	253	F. 循环节		372	XA. Steap + Queap
03E	258	G. 插入排序		377	XB. 双栈当队
	264	H. 归并排序		382	XC. 直方图内最大矩形

5. 二叉树			6. 图		
05A	389	A. 树	06A	505	A. 概述
05B	397	B. 树的表示	06B1		B. 邻接矩阵
05C	403	C. 有根有序树 = 二叉树		510	1. 构思
05D	409	D. 二叉树的实现		514	2. 实现
05E1		E. 先序遍历		518	3. 简单接口
	418	1. 迭代算法 A		522	4. 复杂接口
	425	2. 观察		527	5. 性能分析
	428	3. 迭代算法 B		530	C. 邻接表
05E2		F. 中序遍历			D. 广度优先搜索
	432	1. 观察	06C[1~4]	537	1. 算法
	437	2. 迭代算法	06C5	542	2. 实例
	441	3. 分析	06C6	545	3. 推广
	444	4. 后继与前驱	06C[7~8]	549	4. 性质及应用
		G. 后序遍历			E. 深度优先搜索
	447	1. 观察	06D[1~3]	555	1. 算法
	452	2. 迭代算法	06D4	559	2. 实例 (无向图)
	455	3. 实例	06D6	564	3. 推广
	458	4. 分析	06D5	567	4. 实例 (有向图)
	461	5. 表达式树	06D7	575	5. 性质
05E4		H. 层次遍历			F. 拓扑排序
	464	1. 算法及分析		580	1. 零入度算法
	469	2. 完全二叉树		587	2. 零出度算法
05E5	474	I. 重构			
		J. Huffman 树			
	480	1. 算法			7. 图应用
	489	2. 正确性		592	A. 优先级搜索
	496	3. 实现			B. Prim 算法
	502	4. 改进		597	1. 最小支撑树

	602	2. 极短跨边	07C	c. 平衡
	606	3. 实例	694	1. 期望树高
	610	4. 正确性	699	2. 理想平衡与适度平衡
	613	5. 实现	702	3. 等价变换
		C. Dijkstra 算法		D. AVL 树
	616	1. 最短路径	07D1	706 1. 适度平衡
	620	2. 最短路径树	07D1	710 2. 重平衡
	625	3. 算法	07D2	714 3. 插入
	630	4. 实例	07D3	718 4. 删除
	635	5. 实现	07D4	722 5. (3+4)-重构
		D. 双连通分量		
	638	1. 判定准则		
	643	2. 算法		9. 搜索树应用
	648	3. 实例		A. 范围查询
		E. Kruskal 算法	CG-07A[1-4]	727 1. 一维范围查询
	654	1. 算法	CG-07A5	732 2. 平面范围查询
	658	2. 并查集		B. kd-树
	666	F. Floyd-Warshall 算法	CG-07B[1-5]	738 1. 1d-树
			CG-07C[1-2]	743 2. 2d-树
			CG-07C[3-4]	748 3. 构造
			CG-07C5	751 4. 正则子集
			CG-07D[1-3]	754 5. 查询
			CG-07E[1-3]	759 6. 复杂度
			CG-07[FGH]	C. 多层搜索树
			CG-07I	774 XA. 范围树
			CG-08[B-D]	783 XB. 区间树
			CG-08J	796 XC. 线段树
		8. 二叉搜索树		
07A		A. 概述		
	671	1. 循关键码访问		
	674	2. 中序		
	678	3. 接口		
		B. 算法及实现		
07B1	681	1. 查找		
07B2	685	2. 插入		
07B3	688	3. 删除		

10. 高级搜索树						
		A. 伸展树		09D1	949	2. 封闭散列
08A1	810	1. 逐层伸展		09D1	953	3. 懒惰删除
08A2	815	2. 双层伸展		09D2	956	4. 平方试探
08A3	821	3. 算法实现		09D2	960	5. 双向平方试探
08A4	829	4. 分摊分析		09D2	965	6. 再散列
		B. B-树		09D2	967	7. 重散列
08B1	835	1. 大数据	09E[1~2]	969	8. 桶排序	1. 算法
08B2	840	2. 结构		972		2. 最大缝隙
08B3	849	3. 查找				E. 基数排序
08B4	856	4. 插入		976		1. 算法
08B5	865	5. 删除		981		3. 整数排序
		C. 红黑树	09E3	984	F. 计数排序	
08XA1	876	1. 持久性				XA. 跳转表
08XA2	881	2. 结构		993		1. 结构
08XA3	887	3. 插入		1000		2. 查找
08XA4	899	4. 删除		1005		3. 插入与删除

11. 词典				12. 优先级队列		
09B		A. 散列				A. 概述
	915	1. 循对象访问		10A1	1012	1. 需求与动机
	923	2. 原理		10A2	1015	2. 基本实现
	928	3. 冲突				B. 完全二叉堆
09C		B. 散列函数		10B1	1022	1. 结构
	934	1. 基本		10B2	1026	2. 插入
	940	2. 更多		10B3	1035	3. 删除
		C. 排解冲突		10B4	1044	4. 批量建堆
09D1	945	1. 开放散列		10C	1052	C. 堆排序

	D. 锦标赛排序	11E3	1155	3. 性能
1059	1. 锦标赛树			F. KR 算法
1065	2. 败者树	11F1	1157	1. 串即是数
1067	XA. 多叉堆	11F2	1162	2. 散列
	XB. 左式堆		1167	G. 键树
10XA1	1074	1. 结构		
10XA2	1081	2. 合并		
10XA3	1090	3. 插入 + 删除		
	XC. 优先级搜索树			

13. 串				
11A	1101	A. ADT	12A1	1173
11B	1105	B. 模式匹配	12A1	1177
		C. KMP 算法	12A2	1182
11C1	1110	1. 记忆法	12A2	1185
11C2	1114	2. 查询表		1190
11C3	1121	3. 理解 next[] 表		1195
11C4	1123	4. 构造 next[] 表	12A4	1201
11C5	1126	5. 分摊分析		1201
11C6	1129	6. 再改进	12B1	1206
		D. BM 算法 : BC 策略		1212
11D1	1134	1. 以终为始	12B3	1218
11D2	1138	2. 坏字符	12B3	1226
11D3	1140	3. 构造 BC[] 表		1226
11D4	1142	4. 性能	12C1	1230
		E. BM 算法 : GS 策略	12C2	1239
11E1	1145	1. 好后缀		1249
11E2	1151	2. 构造 GS[] 表		1254
				1258
				1258
				5. Sedgewick 序列

θ1-A1

绪论
计算：工具

The cubic root of 2 is not constructible by ruler and compass, but the cubic root of $2 + \sqrt{5}$, which looks more complicated, is. Things like this make it fun to be a mathematician.

邓伟辉
deng@tsinghua.edu.cn

1

绳索计算机

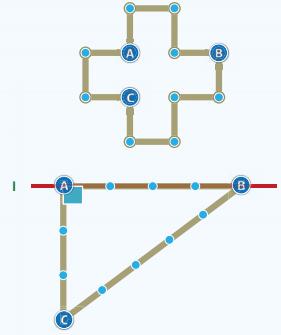
◆ 输入：任给直线 l 及其上一点A

◆ 输出：经过A做 l 的一条垂线

◆ 算法（传说，2000 B.C. 古埃及人）

- 取12段等长的绳索，首尾联接成环
- 从A点起，将4段绳索沿 l 抻直并固定于B
- 沿另一方向找到第3段绳索的终点C
- 移动点C，将剩余的3+5段绳索抻直

◆ 这里的计算机是什么？



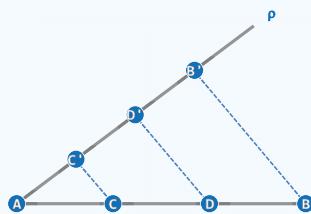
Data Structures & Algorithms, Tsinghua University

尺规计算机

3

◆ 任给平面上线段AB（输入），将其三等分（输出）

- 从A发出一条与AB不重合的射线 ρ
- 在 ρ 上取 $|AC'| = |C'D'| = |D'B'|$
- 联接 $B'C'$
- 经 D' 做 $B'C'$ 的平行线，交 AB 于D
- 经 C' 做 $B'C'$ 的平行线，交 AB 于C



◆ 这里的计算机是什么？

◆ 它能够解决什么问题？不能解决什么问题？

◆ 子程序：过直线外一点，做平行线

Data Structures & Algorithms, Tsinghua University

算法

5

◆ 计算 = 信息处理 = 借助某种工具，遵照一定规则，以明确而机械的形式进行

◆ 计算模型 = 计算机 = 信息处理工具

◆ 所谓算法，即特定计算模型下，旨在解决特定问题的指令序列

输入 待处理的信息（问题）

输出 经处理的信息（答案）

正确性 确实可以解决指定的问题

确定性 任一算法都可以描述为一个由基本操作组成的序列

可行性 每一基本操作都可实现，并在常数时间内完成

有穷性 对于任何输入，经有穷次基本操作，都可以得到输出

Data Structures & Algorithms, Tsinghua University

有穷性：程序 ~ 算法

7

```
int hailstone( int n ) {
    int length = 1;
    while ( 1 < n ) { n % 2 ? n = 3*n + 1 : n /= 2; length++; }
    return length;
} //对于任意的n，总有|Hailstone(n)| < ∞ ?
```

Data Structures & Algorithms, Tsinghua University

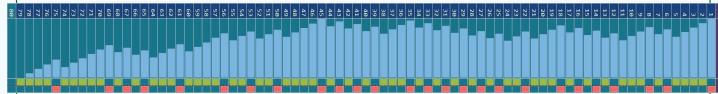
有穷性：Hailstone序列

$$Hailstone(n) = \begin{cases} \{1\} & (n \leq 1) \\ \{n\} \cup Hailstone(n/2) & (n \text{ is even}) \\ \{n\} \cup Hailstone(3n + 1) & (n \text{ is odd}) \end{cases}$$

[42], 21, 64, 32, ..., 1

[7], 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, ..., 1

[27], 82, 41, 124, 62, 31, 94, 47, 142, 71, 214, 107, ...



Data Structures & Algorithms, Tsinghua University

好算法 = ?

9

◆ 正确：符合语法，能够编译、链接

能够正确处理简单的输入

能够正确处理大规模的输入

能够正确处理一般的输入

能够正确处理退化的输入

能够正确处理任意合法的输入

◆ 健壮：能辨别不合法的输入并做适当处理，而不致非正常退出

◆ 可读：结构化 + 准确命名 + 注释 + ...

◆ 效率：速度尽可能快；存储空间尽可能少

Algorithms + Data Structures = Programs

//N. Wirth, 1976

(Algorithms + Data Structures) × Efficiency = Computation

Data Structures & Algorithms, Tsinghua University

绪论

计算模型：统一尺度

θ1-B1

邓伟辉
deng@tsinghua.edu.cn

To measure is to know. If you can not measure it, you can not improve it.
- Lord Kelvin

2

4

绪论

计算：算法

θ1-A2

邓伟辉
deng@tsinghua.edu.cn

Computer science should be called computing science, for the same reason why surgery is not called knife science.

- E. Dijkstra

6

有穷性：理想硬币 ~ 几何分布

$$Hailstone(n) = \begin{cases} \{1\} & (n \leq 1) \\ \{n\} \cup Hailstone(n/2) & (n \text{ is even}) \\ \{n\} \cup Hailstone(3n + 1) & (n \text{ is odd}) \end{cases}$$

3	3
3	3
2	2
2	2
1	1

Data Structures & Algorithms, Tsinghua University

8

◆ 假设：rand()为理想的（尽管不可能）随机整数发生器

◆ 于是：rand()为奇、偶的概率均为50%

◆ 考查算法程序：

```
void dice( ) { while ( rand() & 1 ); }
```

其中的循环，将迭代多少步？

◆ 数学期望的步数为： $E = \sum_{k=1}^{\infty} k \cdot 2^{-k} = 2$

尽管在理论上可能任意多次

10

绪论

计算模型：统一尺度

θ1-B1

邓伟辉
deng@tsinghua.edu.cn

算法分析

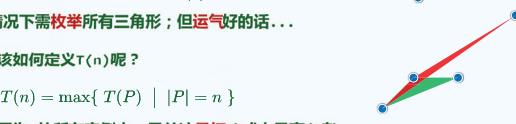
- ❖ 两个主要方面...
- ❖ 意义不大，毕竟...
- ❖ 正确：算法功能与问题要求一致？ 可能出现的问题实例太多
- ❖ 数学证明？可不那么简单...
- ❖ 如何归纳简化、概括？
- ❖ 成本：运行时间 + 所需存储空间 观察：问题实例的规模，往往是
- ❖ 如何度？如何比较？ 决定计算成本的最主要因素
- ❖ 将计算成本描述为函数，比如... 通常：规模接近，计算成本也接近
- ❖ 规模扩大，计算成本亦上升
- $T_A(P)$ = 算法A求解问题实例P的计算成本

Data Structures & Algorithms, Tsinghua University

特定算法 + 不同实例

- ❖ 令： $T_A(n)$ = 用算法A求解某一问题规模为n的实例，所需的计算成本
讨论特定算法A（及其对应的问题）时，可简记作 $T(n)$
- ❖ 然而：这一定义仍有问题...
- ❖ 观察：同一问题等规模的不同实例，计算成本毕竟不尽相同；有些场合，甚至会有实质差别...
- ❖ 例如：任给平面上n个点，在它们定义的 $\binom{n}{3}$ 个三角形中，是否某一个的面积不超过5.0？
- ❖ 蛮力：最坏情况下需枚举所有三角形；但运气好的话...
- ❖ 既然如此，又该如何定义 $T(n)$ 呢？
- ❖ 稳妥起见，取 $T(n) = \max\{T(P) \mid |P| = n\}$
- ❖ 亦即，在规模同为n的所有实例中，只关注最坏（成本最高）者

Data Structures & Algorithms, Tsinghua University



特定问题 + 不同算法

- ❖ 同一问题通常有多种算法，如何评判其优劣？
- ❖ 实验统计是最直接的方法，但足以准确反映算法的真正效率？不足够！
 - 不同的算法，可能更适应于不同规模的输入
 - 不同的算法，可能更适应于不同类型的数据
 - 同一算法，可能由不同程序员、用不同程序语言、经不同编译器生成
 - 同一算法，可能实现并运行于不同的体系结构、操作系统...
- ❖ 为给出客观的评判，需要抽象出一个理想的平台或模型
 - 不再依赖于上述种种具体的因素
 - 从而直接而准确地描述、测量并评价算法

Data Structures & Algorithms, Tsinghua University

结论

计算模型：图灵机



邓信辉

deng@tsinghua.edu.cn

Sometimes it is the people no one can imagine anything of who do the things no one can imagine.

- A. Turing

构成部件

- ❖ Tape 依次均匀地划分为单元格，各存有某一字符，初始均为'#'
- ❖ Alphabet 字符的种类有限
- ❖ Head
 - 总是对准某一单元格，并可读取或改写其中的字符
 - 每经过一个节拍可转向左侧或右侧的邻格
- ❖ State TM总是处于有限种状态中的某一种
- ❖ Transition Function 可按照规则转向另一种状态

Data Structures & Algorithms, Tsinghua University

实例：Increase

- ❖ 功能：将二进制非负整数加一
- ❖ 原理：
 - 全'1'的后缀，翻转为全'0'
 - 原最低位'0'或'#'翻转为'1'
- ❖ $(<, 1; 0, L, <) // 左行, 1->0$
- ❖ $(<, 0; 1, R, >) // 掉头, 0->1$
- ❖ $(<, #; 1, R, >) // 可否省略?$
- ❖ $(>, 0; 0, R, >) // 右行$
- ❖ $(>, #; #, L, h) // 复位$
- ❖ 规范 ~ 接口

Data Structures & Algorithms, Tsinghua University

Random Access Machine : 组成 + 语言

- ❖ 寄存器顺序编号，总数没有限制 // 但愿如此
- ❖ 可通过编号直接访问任意寄存器 // call-by-rank
- ❖ 每一基本操作仅需常数时间 // 循环及子程序本身非基本操作
- $R[i] \leftarrow c$ $R[i] \leftarrow R[R[j]]$ $R[i] \leftarrow R[j] + R[k]$
- $R[i] \leftarrow R[j]$ $R[R[i]] \leftarrow R[j]$ $R[i] \leftarrow R[j] - R[k]$
- IF $R[i] = 0$ GOTO # IF $R[i] > 0$ GOTO # GOTO #
- STOP

Data Structures & Algorithms, Tsinghua University

结论

计算模型：RAM



邓信辉

deng@tsinghua.edu.cn

There is an infinite set A that is not too big.

- J. von Neumann

Random Access Machine : 效率

- ❖ 与TM模型一样，RAM模型也是一般计算工具的简化与抽象
- ❖ 使我们可以独立于具体的平台，对算法的效率做出可信的比较与评判

在这些模型中

- 算法的运行时间 \propto 算法需要执行的基本操作次数
- $T(n)$ = 算法为求解规模为n的问题，所需执行的基本操作次数

- ❖ 思考：在TM、RAM等模型中衡量算法效率，为何通常只需考查运行时间？空间呢？

Data Structures & Algorithms, Tsinghua University

实例 : Floor Division

◆ $\forall c \geq 0$ and $d > 0$, define

$$\lfloor c/d \rfloor = \max\{x \mid d \cdot x \leq c\} \\ = \max\{x \mid d \cdot x < 1 + c\}$$

◆ 例如 : $\lfloor 5/2 \rfloor = 2$ $\lfloor 2015/56 \rfloor = 35$

$$\lfloor 6/3 \rfloor = 2$$
 $\lfloor 2016/36 \rfloor = 56$

$$\lfloor 12/5 \rfloor = 2$$

◆ 思路 : 反复地从 $R[0] = 1 + c$ 中 , 减去 $R[1] = d$

统计在下溢之前 , 所做减法的次数 x

Step	IR	R[0]	R[1]	R[2]	R[3]
0	0	12	5	0	0
1	R[3] \leftarrow 1	12	5	0	1
2	R[0] \leftarrow R[0] + R[3]	13	5	0	1
3	R[2] \leftarrow R[2] + R[3]	8	5	0	1
4	R[0] \leftarrow R[0] - R[1]	4	5	0	1
5	R[2] \leftarrow R[2] + R[3]	2	5	0	1
6	IF R[0] > 0 GOTO 2	2	5	0	1
7	R[0] \leftarrow R[2] - R[3]	3	5	0	1
8	STOP	2	5	0	1

Data Structures & Algorithms, Tsinghua University

21



Any time you are stuck on a problem, introduce more notation.
- Chris Skinner

Mathematics is more in need of good notations than of new theorems.
- A. Turing

邓伟辉
deng@tsinghua.edu.cn

Big-O notation

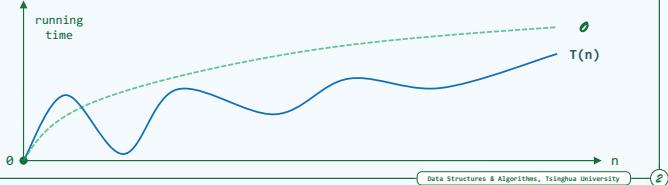
◆ Paul Bachmann, 1894: $T(n) = \mathcal{O}(f(n))$ iff $\exists c > 0$ s.t. $T(n) < c \cdot f(n)$ $\forall n \gg 2$

$$Ex: \sqrt{5n \cdot [3n \cdot (n+2) + 4] + 6} < \sqrt{5n \cdot [6n^2 + 4] + 6} < \sqrt{35n^3 + 6} < 6 \cdot n^{1.5} = \mathcal{O}(n^{1.5})$$

◆ 与 $T(n)$ 相比, $f(n)$ 在形式上更为简洁, 但依然反映前者增长趋势

$$\mathcal{O}(f(n)) = \mathcal{O}(c \cdot f(n))$$

$$\mathcal{O}(n^a + n^b) = \mathcal{O}(n^a), a \geq b > 0$$



Data Structures & Algorithms, Tsinghua University

25

结论

渐进复杂度 : 大O记号

Computational problems can be feasibly computed on some computational device only if they can be computed in polynomial time.

- A. Cobham & J. Edmonds

邓伟辉
deng@tsinghua.edu.cn

$\mathcal{O}(1)$: constant

◆ 这类算法的效率最高

// 总不能奢望不劳而获吧

◆ 什么样的代码段对应于常数执行时间 ?

// 应具体分析...

◆ 一定不含循环 ?

```
for ( i = 0; i < n; i += n/2019 + 1 ); // 2019, 常数
for ( i = 1; i < n; i = 1 << i ); // log*n, 几乎常数
```

◆ 一定不含分支转向 ?

```
if ( (n + m) * (n + m) < 4 * n * m ) goto UNREACHABLE; // 不考虑溢出
```

◆ 一定不能有(递归)调用 ?

```
if ( 2 == (n * n) % 5 ) Olop(n); // O(1)-time Operation
```

Data Structures & Algorithms, Tsinghua University

29

Step	IR	R[0]	R[1]	R[2]	R[3]
0	0	12	5	0	0
1	R[3] \leftarrow 1	12	5	0	1
2	R[0] \leftarrow R[0] + R[3]	13	5	0	1
3	R[2] \leftarrow R[2] + R[3]	8	5	0	1
4	R[0] \leftarrow R[0] - R[1]	4	5	0	1
5	R[2] \leftarrow R[2] + R[3]	2	5	0	1
6	IF R[0] > 0 GOTO 2	2	5	0	1
7	R[0] \leftarrow R[2] - R[3]	3	5	0	1
8	STOP	2	5	0	1

Data Structures & Algorithms, Tsinghua University

22

实例 : Floor Division

◆ RAM 算法

```
0 R[3]  $\leftarrow$  1 // increment
1 R[0]  $\leftarrow$  R[0] + R[3] // c++
2 R[0]  $\leftarrow$  R[0] - R[1] // c -= d
3 R[2]  $\leftarrow$  R[2] + R[3] // x++
4 IF R[0] > 0 GOTO 2 // if c > 0 goto 2
5 R[0]  $\leftarrow$  R[2] - R[3] // else x-- and
6 STOP // return R[0] = x =  $\lfloor c/d \rfloor$ 
```

Step	IR	R[0]	R[1]	R[2]	R[3]
0	0	12	5	0	0
1	1	12	5	0	1
2	2	13	5	0	1
3	3	8	5	0	1
4	4	4	5	0	1
5	5	2	5	0	1
6	6	3	5	0	1
7	7	4	5	0	1
8	8	2	5	0	1
9	9	3	5	0	1
10	10	4	5	0	1
11	11	5	5	0	1
12	12	6	2	5	1

Data Structures & Algorithms, Tsinghua University

渐进分析

◆ 回到原先的问题 :

随着问题规模的增长, 计算成本如何增长?

当输入规模 $n \gg 2$ 后, 算法

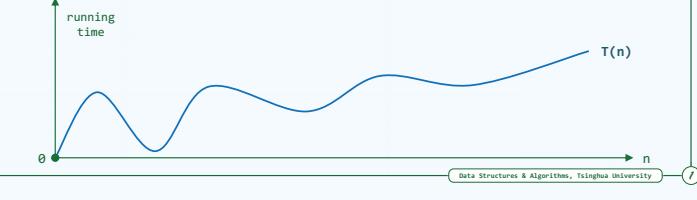
需执行的基本操作次数 $T(n) = ?$

◆ 这里更关心 :

问题规模足够大之后, 计算成本的 **增长趋势**

如欲更为精确地估计, 还可考查

需占用的存储单元数 $S(n) = ?$



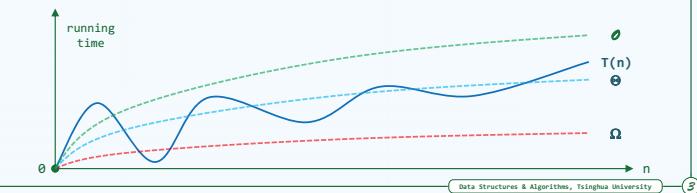
Data Structures & Algorithms, Tsinghua University

24

其它记号

$$T(n) = \Omega(f(n)) \text{ iff } \exists c > 0 \text{ s.t. } T(n) > c \cdot f(n) \quad \forall n \gg 2$$

$$T(n) = \Theta(f(n)) \text{ iff } \exists c_1 > c_2 > 0 \text{ s.t. } c_1 \cdot f(n) > T(n) > c_2 \cdot f(n) \quad \forall n \gg 2$$



Data Structures & Algorithms, Tsinghua University

26

$\mathcal{O}(1)$: constant

◆ 常数

// 含RAM各基本操作

- $2 = 2019 = 2019 \times 2019 = \mathcal{O}(1)$, 甚至

- $2019^{2019} = \mathcal{O}(1)$

◆ 从渐进的角度来看, 再大的常数, 也要小于递增的变数

// 尽管实际并非如此...

◆ [General twin prime conjecture, de Polignac 1849]

For every natural number k , there are infinitely many prime pairs p and q such that $p - q = 2k$

◆ [Yitang Zhang, April 2013] $k \leq 35,000,000$

◆ [Terence Tao, May 2013] $k \leq 6,500,000$

◆ [Polymath Project, April 2014] $k \leq 123$

Data Structures & Algorithms, Tsinghua University

28

$\mathcal{O}(1)$: constant

◆ 这类算法的效率最高

// 含RAM各基本操作

◆ 什么样的代码段对应于常数执行时间 ?

// 应具体分析...

◆ 一定不含循环 ?

```
for ( i = 0; i < n; i += n/2019 + 1 ); // 2019, 常数
for ( i = 1; i < n; i = 1 << i ); // log*n, 几乎常数
```

◆ 一定不含分支转向 ?

```
if ( (n + m) * (n + m) < 4 * n * m ) goto UNREACHABLE; // 不考虑溢出
```

◆ 一定不能有(递归)调用 ?

```
if ( 2 == (n * n) % 5 ) Olop(n); // O(1)-time Operation
```

Data Structures & Algorithms, Tsinghua University

30

$\mathcal{O}(\log n)$: poly-log

◆ 对数 $\mathcal{O}(\log n)$: $\ln n$ $\lg n$ $\log_{10} n$ $\log_{2017} n$ // 为何不注明底数 ?

◆ 常底数无所谓 : $\forall a, b > 1$, $\log_a n = [\log_a b] \cdot \log_b n = \Theta(\log_b n)$

◆ 常数次幂无所谓 : $\forall c > 0$, $\log n^c = c \cdot \log n = \Theta(\log n)$

◆ 对数多项式 : $123 \cdot \log^{321} n + \log^{205} (7 \cdot n^2 - 15 \cdot n + 31) = \Theta(\log^{321} n)$

◆ 这类算法非常有效, 复杂度无限接近于常数 : $\forall c > 0$, $\log n = \mathcal{O}(n^c)$

Data Structures & Algorithms, Tsinghua University

30

$\mathcal{O}(n^c)$: polynomial

多项式： $a_k \cdot n^k + a_{k-1} \cdot n^{k-1} + \dots + a_2 \cdot n^2 + a_1 \cdot n + a_0 = \mathcal{O}(n^k)$, $a_k > 0$

$$100 \cdot n + 203 = \mathcal{O}(n) \quad \sqrt{23 \cdot n - 472} \times \sqrt{101 \cdot n + 203} = \mathcal{O}(n)$$

$$(100 \cdot n - 532) \cdot (20 \cdot n^2 - 445 \cdot n + 2019) = \mathcal{O}(n^3)$$

$$(2019 \cdot n^2 - 129)/(1991 \cdot n - 37) = \mathcal{O}(n)$$

$$\sqrt[3]{2 \cdot n^3 - \sqrt[3]{3 \cdot n^4 - \sqrt[3]{4 \cdot n^5 + \sqrt[3]{5 \cdot n^6 + \sqrt[3]{6 \cdot n^7 + \sqrt[3]{7 \cdot n^8 + \sqrt[3]{8 \cdot n^9 + n^{2019}/\sqrt{n^6 - 5 \cdot n^3 + 1970}}}}}}}} = \mathcal{O}(n^7)$$

线性 (linear function) : 所有 $\mathcal{O}(n)$ 类函数

从 $\mathcal{O}(n)$ 到 $\mathcal{O}(n^2)$: 本课程编程习题主要覆盖的范围

这类算法的效率通常认为已可令人满意，然而...这个标准是否太低了？ //P难度！

Data Structures & Algorithms, Tsinghua University

结论

渐进复杂度：指数

$\theta_1 - C_3$

邓俊辉
deng@tsinghua.edu.cn

慌得那拿盘的小怪，战兢跑去报道：“难，难，难！难，难，难！”
老妖道：“怎么有许多难？”

“你是什么东西？”太太说。四虎子也愣住了，他自己不知道他是什么东西——这本是世上最难答的一个问题。

 $\mathcal{O}(2^n)$: exponential

指数： $T(n) = \mathcal{O}(a^n)$, $a > 1$

$$\forall c > 1, n^c = \mathcal{O}(2^n) \quad \because e^n = 1 + n + n^2/2! + n^3/3! + n^4/4! + \dots$$

$$n^{1000\dots 01} = \mathcal{O}(1.000\dots 01^n) = \mathcal{O}(2^n)$$

$$1.000\dots 01^n = \Omega(n^{1000\dots 01})$$

这类算法的计算成本增长极快，通常被认为不可忍受

从 $\mathcal{O}(n^c)$ 到 $\mathcal{O}(2^n)$ ，是从有效算法到无效算法的分水岭

很多问题的 $\mathcal{O}(2^n)$ 算法往往显而易见

然而，设计出 $\mathcal{O}(n^c)$ 算法却极其不易；甚至，有时注定地只能是徒劳无功

更糟糕的是，这类问题要远比我们想象的多得多...

incomputable
intractable
tractable
 $\mathcal{O}(n^2)$

Data Structures & Algorithms, Tsinghua University

2-Subset

$\forall S = \{a_1, a_2, \dots, a_n\} \subset \mathbb{Z}$ and $\sum_{k=1}^n a_k = 2m, \exists T \subset S$ s.t. $\sum_{a \in T} a = m$?

选举人团投票制：50个州加1个特区，共538票；获270张选举人票，即可当选

若共有两位候选人，会否恰好各得269票？

直觉上，并不难：逐一枚举S的每一子集，并统计其中元素的总和

California	Indiana	Connecticut	Idaho
Texas	Missouri	Iowa	Maine
New York	Tennessee	Oklahoma	New Hampshire
Illinois	Michigan	Arkansas	Delaware
Pennsylvania	Arizona	Kansas	D. C.
Ohio	Maryland	Louisiana	Montana
Oregon	Minnesota	Mississippi	Rhode Island
Washington	Wisconsin	West Virginia	Vermont
Georgia	Alabama	North Dakota	Nevada
New Jersey	Colorado	New Mexico	South Dakota
North Carolina	Louisiana	Utah	Vermont
Michigan	Mississippi	Virginia	Wyoming
Massachusetts	South Carolina	Hawaii	U.S.



Data Structures & Algorithms, Tsinghua University

2-Subset

定理： $|2^S| = 2^{|S|} = 2^n$

亦即：直觉算法需要迭代 2^n 轮，并（在最坏情况下）至少需要花费这么多的时间

故严格地讲，这仍只是程序，而不是算法

还是直觉：应该有更好的办法吧？

定理：2-Subset is NP-complete

— 什么意思？

意即：就目前的计算模型而言，不存在可在多项式时间内回答此问题的算法

— 就此意义而言，上述的直觉算法已属最优

结论

渐进复杂度：层级划分

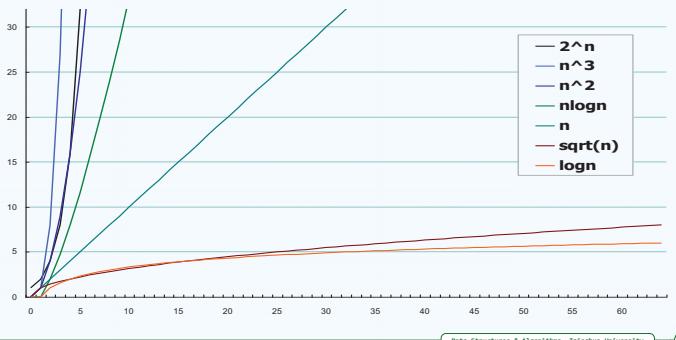
$\theta_1 - C_4$

邓俊辉
deng@tsinghua.edu.cn

好读书，不求甚解；每有会意，便欣然忘食

主啊，我向你承认，我依旧不明了时间是什么

增长速度



增长速度

层次级别

课后

常数	$\mathcal{O}(1)$	再好不过，但难得如此幸运	对数据结构的基本操作
	$\mathcal{O}(\log n)$	在这个宇宙中，几乎就是常数	逆Ackermann函数
对数	$\mathcal{O}(\log n)$	与常数无限接近，且不难遇到	有序向量的二分查找 堆、词典的查询、插入与删除
线性	$\mathcal{O}(n)$	努力目标，经常遇到	树、图的遍历
	$\mathcal{O}(n \log n)$	几乎几乎几乎接近线性	某些MST算法
	$\mathcal{O}(n \log \log n)$	几乎接近线性	某些三角剖分算法
平方	$\mathcal{O}(n^2)$	最常出现，但不见得最优	排序、EU、Huffman编码
立方	$\mathcal{O}(n^3)$	所有输入对象两两组合	Dijkstra算法
多项式	$\mathcal{O}(n^c)$	不常见	矩阵乘法
指数	$\mathcal{O}(2^n)$	P问题 == 存在多项式算法的问题	很多问题的平凡算法，再尽可能优化
	...	很多问题，并不存在算法	绝大多数问题，并不存在算法

Data Structures & Algorithms, Tsinghua University

证明、证否

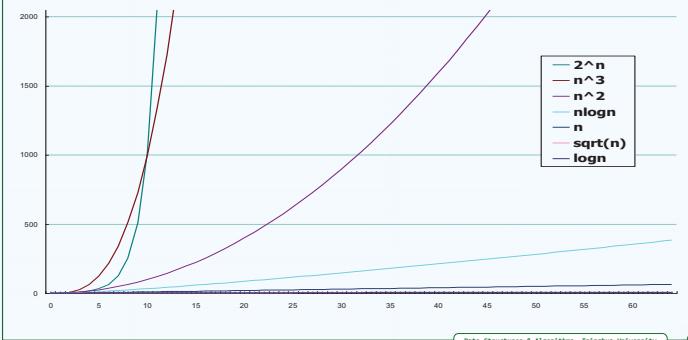
k-Subset :

- Fibonacci数 $\text{fib}(n) = \mathcal{O}(2^n)$
- $12n + 5 = \mathcal{O}(n \log n)$
- $\log^2(n^{1024} - 2 * n^6 + 101) = \mathcal{O}(?)$
- $\log^4 n = \mathcal{O}(n^c), \forall c > 0, d > 1$
- $\log^{1.001} n = \mathcal{O}(\log(n^{1001}))$
- $(n^2 + 1) / (2n + 3) = \mathcal{O}(n)$
- $n^{2013} = \mathcal{O}(n!)$
- $n! = \mathcal{O}(n^{2019})$
- $2^n = \mathcal{O}(n!)$

任给整数集S，判定S
可否划分为k个不交子集，使其和均为(ΣS)/k

证明或证否：

(k+1)-Subset的难度，不低于k-Subset



课后

自学: small-o notation

Data Structures & Algorithms, Tsinghua University

θ1-D1

绪论
复杂度分析：级数

谁对时间，谁就会突然老去。

邓俊辉
deng@tsinghua.edu.cn

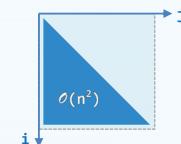
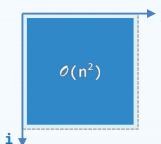
- 算术级数：与末项平方同阶 $T(n) = 1 + 2 + \dots + n = \binom{n+1}{2} = \frac{n(n+1)}{2} = \mathcal{O}(n^2)$
- 幂方级数：比幂次高出一阶 $\sum_{k=0}^n k^d \approx \int_0^n x^d dx = \frac{x^{d+1}}{d+1} \Big|_0^n = \frac{n^{d+1}}{d+1} = \mathcal{O}(n^{d+1})$
 $T_2(n) = \sum_{k=1}^n k^2 = 1^2 + 2^2 + 3^2 + \dots + n^2 = n(n+1)(2n+1)/6 = \mathcal{O}(n^3)$
 $T_3(n) = \sum_{k=1}^n k^3 = 1^3 + 2^3 + 3^3 + \dots + n^3 = n^2(n+1)^2/4 = \mathcal{O}(n^4)$
 $T_4(n) = \sum_{k=1}^n k^4 = 1^4 + 2^4 + 3^4 + \dots + n^4 = n(n+1)(2n+1)(3n^2+3n-1)/30 = \mathcal{O}(n^5)$
- 几何级数：与末项同阶 $T_a(n) = \sum_{k=0}^n a^k = a^0 + a^1 + a^2 + a^3 + \dots + a^n = \frac{a^{n+1}-1}{a-1} = \mathcal{O}(a^n), \quad 1 < a$
 $T_2(n) = \sum_{k=0}^n 2^k = 1 + 2 + 4 + 8 + \dots + 2^n = 2^{n+1} - 1 = \mathcal{O}(2^{n+1}) = \mathcal{O}(2^n)$

Data Structures & Algorithms, Tsinghua University

- 调和级数： $h(n) = \sum_{k=1}^n \frac{1}{k} = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n} = \ln n + \gamma + \mathcal{O}\left(\frac{1}{2n}\right) = \Theta(\log n)$
- 对数级数： $\sum_{k=1}^n \log k = \log 1 + \log 2 + \log 3 + \log 4 + \dots + \log n = \log(n!) = \Theta(n \cdot \log n)$
- 对数 + 线性 + 指数： $\sum_{k=1}^n k \cdot \log k \approx \int_1^n x \ln x dx = \frac{x^2 \cdot (2 \cdot \ln x - 1)}{4} \Big|_1^n = \mathcal{O}(n^2 \log n)$
 $\sum_{k=1}^n k \cdot 2^k = \sum_{k=1}^n k \cdot 2^{k+1} - \sum_{k=1}^n k \cdot 2^k = \sum_{k=1}^{n+1} (k-1) \cdot 2^k - \sum_{k=1}^n k \cdot 2^k$
 $= n \cdot 2^{n+1} - \sum_{k=1}^n 2^k = n \cdot 2^{n+1} - (2^{n+1} - 2) = (n-1) \cdot 2^{n+1} + 2 = \mathcal{O}(n \cdot 2^n)$
- 如有兴趣，不妨读读：[Concrete Mathematics](#) //ex-2.35, Goldbach Theorem

Data Structures & Algorithms, Tsinghua University

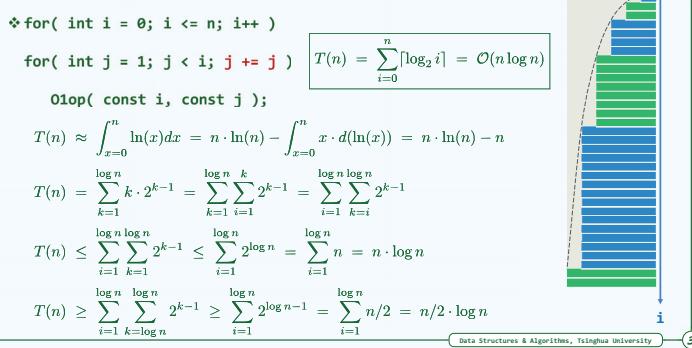
- **迭代 + 算术级数**
- **for(int i = 0; i < n; i++)** **for(int i = 0; i < n; i++)**
for(int j = 0; j < n; j++) **for(int j = 0; j < i; j++)**
O1op(const i, const j); **O1op(const i, const j);**
 $\sum_{i=0}^{n-1} n = n \times n = \mathcal{O}(n^2)$ $\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} = \mathcal{O}(n^2)$



Data Structures & Algorithms, Tsinghua University

- **迭代 + 复杂级数**
- **for(int i = 0; i <= n; i++)**
for(int j = 1; j < i; j += 2) $T(n) = \sum_{i=0}^n [\log_2 i] = \mathcal{O}(n \log n)$
O1op(const i, const j);
 $T(n) \approx \int_{x=0}^n \ln(x) dx = n \cdot \ln(n) - \int_{x=0}^n x \cdot d(\ln(x)) = n \cdot \ln(n) - n$
 $T(n) = \sum_{k=1}^{\log n} k \cdot 2^{k-1} = \sum_{k=1}^{\log n} \sum_{i=1}^k 2^{k-1} = \sum_{i=1}^{\log n} \sum_{k=i}^{\log n} 2^{k-1}$
 $T(n) \leq \sum_{i=1}^{\log n} \sum_{k=1}^{\log n} 2^{k-1} \leq \sum_{i=1}^{\log n} 2^{\log n} = \sum_{i=1}^{\log n} n = n \cdot \log n$
 $T(n) \geq \sum_{i=1}^{\log n} \sum_{k=\log n}^{\log n} 2^{k-1} \geq \sum_{i=1}^{\log n} 2^{\log n-1} = \sum_{i=1}^{\log n} n/2 = n/2 \cdot \log n$

Data Structures & Algorithms, Tsinghua University



- 两个主要任务 = 正确性 (不变性 × 单调性) + 复杂度
- 为确定后者，真地需要将算法描述为RAM的基本指令，再统计累计的执行次数？不必！
- C++等高级语言的基本指令，均等效于常数条RAM的基本指令；在渐进意义下，二者大体相当
 - 分支转向：`goto` //算法的灵魂；出于结构化考虑，被隐藏了
 - 迭代循环：`for()`、`while()`、`...` //本质上就是“`if + goto`”
 - 调用 + 递归（自我调用） //本质上也是`goto`
- 复杂度分析的主要方法
 - 迭代：级数求和
 - 递归：递归跟踪 + 递推方程
 - 猜测 + 验证

Data Structures & Algorithms, Tsinghua University

$$\begin{aligned} \sum_{k=2}^n \frac{1}{(k-1) \cdot k} &= \frac{1}{1 \cdot 2} + \frac{1}{2 \cdot 3} + \frac{1}{3 \cdot 4} + \dots + \frac{1}{(n-1) \cdot n} = 1 - \frac{1}{n} = \mathcal{O}(1) \\ \sum_{k=1}^n \frac{1}{k^2} &= 1 + \frac{1}{2^2} + \frac{1}{3^2} + \dots + \frac{1}{n^2} < 1 + \frac{1}{2^2} + \frac{1}{3^2} + \dots = \frac{\pi^2}{6} = \mathcal{O}(1) \\ \sum_{k \text{ is a perfect power}} \frac{1}{k-1} &= \frac{1}{3} + \frac{1}{7} + \frac{1}{8} + \frac{1}{15} + \frac{1}{24} + \frac{1}{26} + \frac{1}{31} + \frac{1}{35} + \dots = 1 = \mathcal{O}(1) \end{aligned}$$

$$\text{• 几何分布: } (1-\lambda) \cdot [1 + 2\lambda + 3\lambda^2 + 4\lambda^3 + \dots] = 1/(1-\lambda) = \mathcal{O}(1), \quad 0 < \lambda < 1$$

• 有必要讨论这类级数吗？

难道，基本操作次数、存储单元数可能是分数？

是的，某种意义上！

Data Structures & Algorithms, Tsinghua University

复杂度分析：迭代

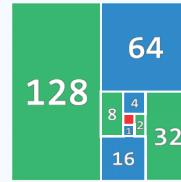
θ1-D2

邓俊辉
deng@tsinghua.edu.cn

Go To Statement Considered Harmful.

- E. Dijkstra, 1968

- **for(int i = 1; i < n; i <= 1)** **for(int i = 0; i < n; i++)**
for(int j = 0; j < i; j++) **for(int j = 0; j < i; j++)**
O1op(const i, const j); **O1op(const i, const j);**



$$\begin{aligned} 1 + 2 + 4 + \dots + 2^{\lfloor \log_2(n-1) \rfloor} &= 2^{\lfloor \log_2 n \rfloor} - 1 = \mathcal{O}(n) \\ &= 2^{\lfloor \log_2 2019 \rfloor} - 1 \end{aligned}$$

Data Structures & Algorithms, Tsinghua University

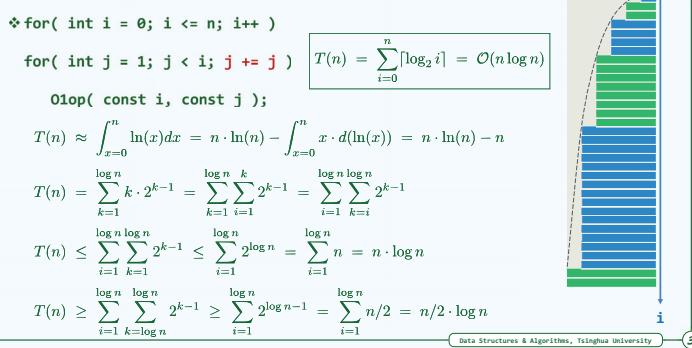
复杂度分析：封底估算

θ1-D3

邓俊辉
deng@tsinghua.edu.cn

He calculated just as men breathe, as eagles sustain themselves in the air.

我仿佛是横越三世来见你的，而你却不在



- 地球(赤道)周长 $\approx 787 \times 360 / 7.2 = 787 \times 50 = 39,350 \text{ km}$
- 1天 $= 24\text{hr} \times 60\text{min} \times 60\text{sec} = 25 \times 4000 = 10^5 \text{ sec}$
- 1生 $\approx 1\text{世纪} = 100\text{yr} \times 365 = 3 \times 10^4 \text{ day} = 3 \times 10^9 \text{ sec}$
- “为祖国健康工作五十年” $\approx 1.6 \times 10^9 \text{ sec}$
- “三生三世” $\approx 300 \text{ yr} = 10^{10} = (1 \text{ googol})^{(1/10)} \text{ sec}$
- 宇宙大爆炸至今 $= 4 \times 10^{17} > 10^8 \times \text{一生}$



Data Structures & Algorithms, Tsinghua University

$n = 1.4^+ \times 10^9$	$\mathcal{O}(10^9)$	Bubblesort $(10^9)^2$ 10^{18}	Mergesort $10^9 \times \log(10^9)$ 30×10^9	算法
普通PC 1GHz 10^9 flops		10^9 sec 30 yr	30 sec	
天河1A 千万亿次 $= 1P$ 10^{15} flops		10^3 sec 20 min	0.03 ms	

Data Structures & Algorithms, Tsinghua University

绪论

迭代与递归：减而治之

$\theta_1 - E_1$

邓俊辉
deng@tsinghua.edu.cn

虽我之死，有子存焉；子又生孙，孙又生子；子又有子，子又有孙；子子孙孙无穷匮也，而山不加增，何苦而不平？

Decrease-and-conquer



为求解一个大规模的问题，可以

- 将其划分为两个子问题：其一**平凡**，另一规模**缩减**
- 分别求解子问题
- 由子问题的解，得到原问题的解

//单调性

Data Structures & Algorithms, Tsinghua University

问题：计算任意n个整数之和

无论A[]内容如何，都有：

```
int SumI( int A[], int n ) {
    int sum = 0; // O(1)
    for ( int i = 0; i < n; i++ ) // O(n)
        sum += A[i]; // O(1)
    return sum; // O(1)
}
```

$T(n) = 1 + n \cdot 1 + 1 = n + 2 = \mathcal{O}(n) = \Omega(n) = \Theta(n)$

空间呢？

Data Structures & Algorithms, Tsinghua University

Linear Recursion: Trace

sum(int A[], int n)

```
{ return n < 1 ? 0 : sum(A, n - 1) + A[n - 1]; }
```

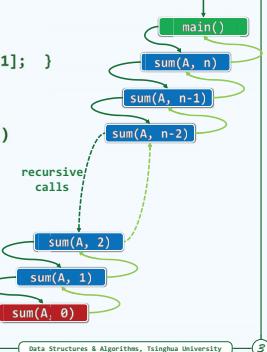
递归跟踪

- 绘出计算过程中出现过的所有**递归实例**（及其调用关系）
- 它们各自所需时间之总和，即为整体运行时间
(调用操作的成本，可计到被创建子实例的账上)

本例中，共计n+1个递归实例，各自只需 $O(1)$ 时间

故总体运行时间为： $T(n) = O(1) \times (n + 1) = O(n)$

空间复杂度呢？



Linear Recursion: Recurrence

对于大规模的问题、复杂的递归算法，递归跟踪不再适用

此时可采用另一抽象的方法...

从递推的角度看，为求解规模为n的问题sum(A, n)，需 $//T(n)$

- 递归求解规模为n-1的问题sum(A, n - 1)，再 $//T(n-1)$
- 累加上 $A[n - 1]$ $//O(1)$

递推方程： $T(n) = T(n - 1) + O(1)$ //recurrence

$T(0) = O(1)$ //base: sum(A, 0)

求解： $T(n) = T(n - 2) + O(2) = T(n - 3) + O(3) = \dots = T(0) + O(n) = O(n)$

Data Structures & Algorithms, Tsinghua University

Reverse

void reverse(int * A, int lo, int hi);

//将任一数组子区间A[lo,hi]前后颠倒

减而治之：

$reverse(lo, hi) = [hi] + reverse(lo + 1, hi - 1) + [lo]$

if (lo < hi) { //递归版

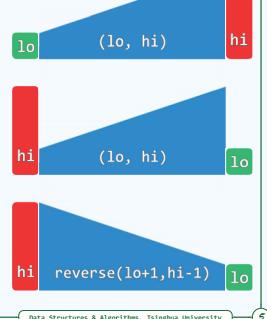
swap(A[lo], A[hi]);

reverse(A, lo + 1, hi - 1);

} //线性递归(尾递归)， $O(n)$

while (lo < hi) //迭代版

swap(A[lo++], A[hi--]); //亦是 $O(n)$



$\theta_1 - E_2$

邓俊辉
deng@tsinghua.edu.cn

为求解一个大规模的问题，可以...

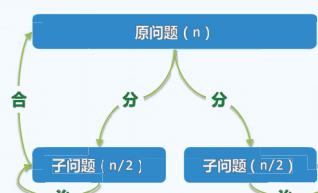
将其**划分为**若干子问题

(通常两个，且规模**大体相当**)

分别求解子问题

由子问题的解

合并得到原问题的解

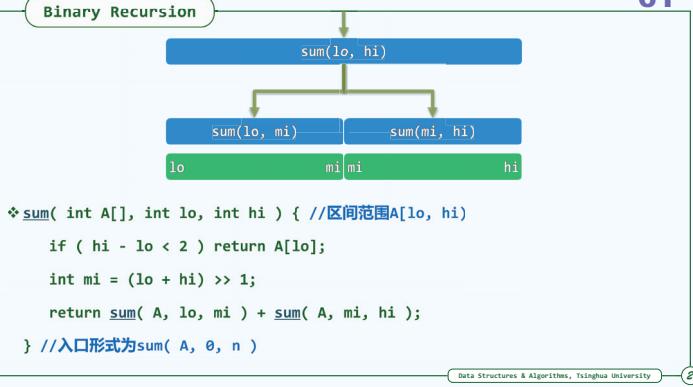


绪论

迭代与递归：分而治之

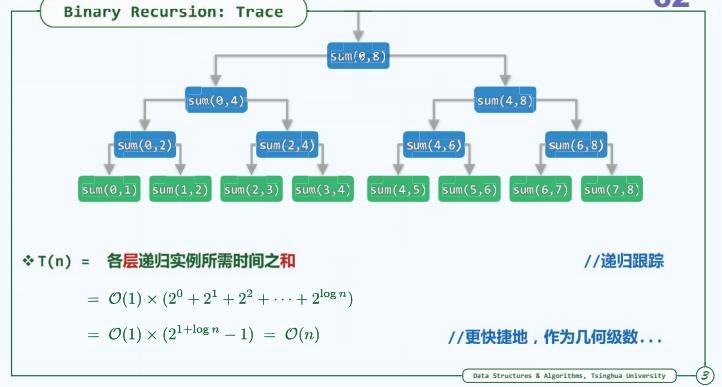
凡治众如治寡，分数是也

61



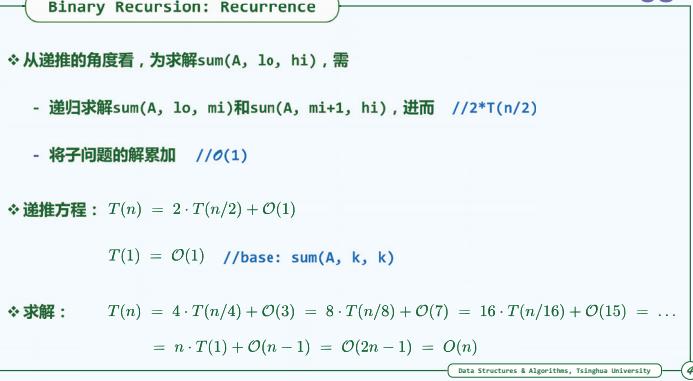
Data Structures & Algorithms, Tsinghua University

62



Data Structures & Algorithms, Tsinghua University

63



Data Structures & Algorithms, Tsinghua University

64

Master Theorem

- 分治策略对应的递推式通常（尽管不总是）形如： $T(n) = a \cdot T(n/b) + \mathcal{O}(f(n))$
(原问题被分为 a 个规模均为 n/b 的子任务；任务的划分、解的合并耗时 $f(n)$)
- 若 $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ ，则 $T(n) = \Theta(n^{\log_b a})$
 - kd-search: $T(n) = 2 \cdot T(n/4) + \mathcal{O}(1) = \mathcal{O}(\sqrt{n})$
 - 若 $f(n) = \Theta(n^{\log_b a} \cdot \log^k n)$ ，则 $T(n) = \Theta(n^{\log_b a} \cdot \log^{k+1} n)$
 - binary search: $T(n) = 1 \cdot T(n/2) + \mathcal{O}(1) = \mathcal{O}(\log n)$
 - mergesort: $T(n) = 2 \cdot T(n/2) + \mathcal{O}(n) = \mathcal{O}(n \cdot \log n)$
 - STL mergesort: $T(n) = 2 \cdot T(n/2) + \mathcal{O}(n \cdot \log n) = \mathcal{O}(n \cdot \log^2 n)$
 - 若 $f(n) = \Omega(n^{\log_b a + \epsilon})$ ，则 $T(n) = \Theta(f(n))$
 - quickselect (average case): $T(n) = 1 \cdot T(n/2) + \mathcal{O}(n) = \mathcal{O}(n)$

Data Structures & Algorithms, Tsinghua University

65

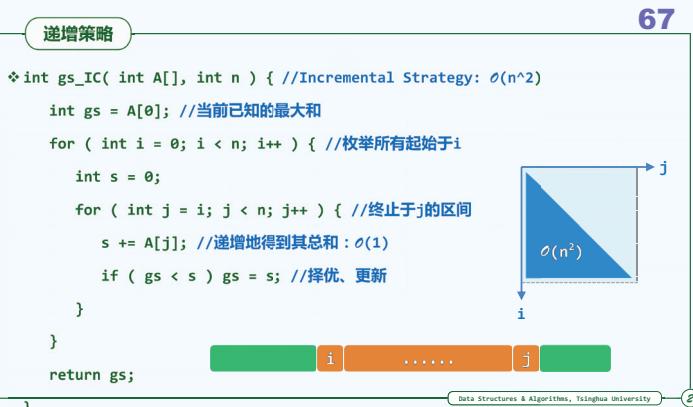
结论

迭代与递归：总和最大区段

$\theta_1 - E_3$ 邓伟辉
deng@tsinghua.edu.cn

...

67



Data Structures & Algorithms, Tsinghua University

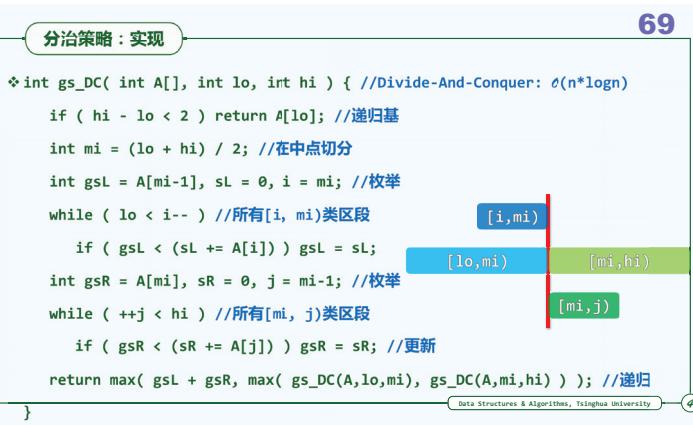
68

分治策略：构思

- 不妨将计算的范围，一般化为： $\mathcal{A}[lo, hi] = \mathcal{A}[lo, mi] \cup \mathcal{A}[mi, hi] = \mathcal{P} \cup \mathcal{S}$
- 于是借助递归，可求得 \mathcal{P} 、 \mathcal{S} 内部的 gs
- 而剩余的（也是实质的）任务无非是：
- 求得跨越前、后缀的 gs
 - 准确地说，需要考察那些“覆盖” $\mathcal{A}[mi, mi]$ 的区段： $\mathcal{A}[i, j] = \mathcal{A}[i, mi] + \mathcal{A}[mi, j]$
- 实际上，必然有： $S[i, mi] = \max\{ S[k, mi] \mid lo \leq k < mi \}$
- $$S[mi, j] = \max\{ S[mi, k] \mid mi \leq k < hi \}$$
- 更好的消息是，二者均可独立计算，且累计耗时不过 $\mathcal{O}(n)$
- 于是总体复杂度也优化为 $\mathcal{O}(n \cdot \log n)$

Data Structures & Algorithms, Tsinghua University

69



Data Structures & Algorithms, Tsinghua University

70

减治策略：构思

- 考查最短的总和非正的后缀 $A[k, hi]$ ，以及总和最大的区段 $GS(lo, hi) = \mathcal{A}[i, j]$
后者要么是前者的（真）后缀，要么与前者无交
- [反证]
- 假若二者确有非空的公共部分：
- $$S(k, hi) \leq 0$$
- $$GS(lo, hi) = A[i, j]$$
- 由 $GS[lo, hi]$ 的最大性（及最短性），必有
- $$S(k, j) > 0$$
- $$S(j, hi) < 0$$
- 基于以上事实，完全可以采用“减而治之”的策略
通过一趟线性扫描在线性时间内找出 GS ...

Data Structures & Algorithms, Tsinghua University

减治策略：实现

```
* int gs_LS( int A[], int n ) { //Linear Scan:  $\mathcal{O}(n)$ 
    int gs = A[0], s = 0, i = n, j = n;
    while ( 0 < i-- ) { //对于当前区间[i,j)
        s += A[i]; //递增地得到其总和:  $\mathcal{O}(1)$ 
        if ( gs < s ) gs = s; //择优、更新
        if ( s <= 0 ) { s = 0; j = i; } //剪除负和后缀
    }
    return gs;
}
```



Data Structures & Algorithms, Tsinghua University

绪论

动态规划：记忆法

θ_1-XA_1

邓俊辉

deng@tsinghua.edu.cn

圣人不记得，所以常记得；今人忘事，以其记事

丧失了记忆又不自知，那才是人生最快乐的时光

fib(): 递归

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

0	1	1	2	3	5	8	13	21	34	55	89
---	---	---	---	---	---	---	----	----	----	----	----

...
* int fib(n) { return (2 > n) ? n : fib(n - 1) + fib(n - 2); } //为何这么慢?
* 复杂度: $T(0) = T(1) = 1$; $T(n) = T(n-1) + T(n-2) + 1$, $\forall n > 1$
- 令 $S(n) = [T(n) + 1]/2$
- 则 $S(0) = 1 = fib(1)$, $S(1) = 1 = fib(2)$
- 故 $S(n) = S(n-1) + S(n-2) = fib(n+1)$
 $T(n) = 2 \cdot S(n) - 1 = 2 \cdot fib(n+1) - 1 = \mathcal{O}(fib(n+1)) = \mathcal{O}(\phi^n)$
- 其中 $\phi = (1 + \sqrt{5})/2 \approx 1.618$

Data Structures & Algorithms, Tsinghua University

封底估算

$$\phi^{36} \approx 2^{25} \quad \phi^{43} \approx 2^{30} \approx 10^9 flo = 1 sec$$

$$\phi^5 \approx 10 \quad \phi^{67} \approx 10^{14} flo = 10^5 sec \approx 1 day$$

$$\phi^{92} \approx 10^{19} flo = 10^{10} sec \approx 10^5 day \approx 3 century$$

Data Structures & Algorithms, Tsinghua University

递归

* 递归版fib()低效的根源在于，各递归实例均被大量地重复调用

* 先后出现的递归实例，共计 $\mathcal{O}(\phi^n)$ 个而去除重复之后，总共不过 $\mathcal{O}(n)$ 种

Data Structures & Algorithms, Tsinghua University

Memoization

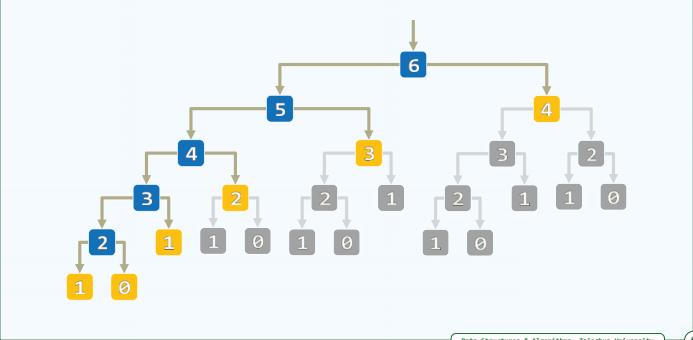
```
* T M[ N ]; #initialized with UNDEFINED
```

```
def f(n)
    if ( n < 1 ) return trivial( n );
    # recurse only when necessary &
    # always write down the result
    if ( M[n] == UNDEFINED )
        M[n] = f(n-X) + f(n-Y)*f(n-Z);
    return M[n];

```

Data Structures & Algorithms, Tsinghua University

Memoization: fib()



Data Structures & Algorithms, Tsinghua University

动态规划

* Dynamic programming

颠倒计算方向：由自顶而下递归，改为自底而上迭代

* f = 1; g = 0; //fib(-1), fib(0)

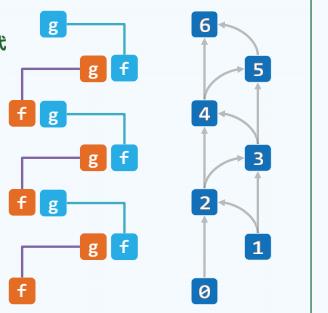
while (0 < n--) {

g = g + f;

f = g - f;

}

return g;

* $T(n) = \mathcal{O}(n)$ ，而且仅需 $\mathcal{O}(1)$ 空间！

Data Structures & Algorithms, Tsinghua University

问题

* 子序列 (Subsequence)：由序列中若干字符，按原相对次序构成

t	s	i	n	g	h	u	a
S	I	N	A				

* 最长公共子序列 (Longest Common Subsequence)：两个序列公共子序列中的最长者

e	u	c	a	o	n	a	l
D	A	T	N	A			
a	d	v	a	n	t	g	e

* 可能有多个

可能有歧义

Data Structures & Algorithms, Tsinghua University

绪论

动态规划：最长公共子序列

θ_1-XA_2

邓俊辉
deng@tsinghua.edu.cn

Make it work, make it right, make it fast

- Kent Beck

◆ 对于序列A[0, n)和B[0, m), LCS(n , m)无非三种情况

① 若n = -1或m = -1, 则取作空序列 ("") //递归基

② 若A[n] = 'X' = B[m], 则取作: LCS(n-1, m-1) + 'X' //减而治之

decrease

didactic
advant

didacticA
advantA

A[0, n-1] X

B[0, m-1] X

Data Structures & Algorithms, Tsinghua University

divide

A[0, n)

B[0, m-1)

A[0, n-1]

B[0, m]

//分而治之

2) A[n] ≠ B[m], 则在 LCS(n, m-1) 与 LCS(n-1, m) 中取更长者

didacticC
advant

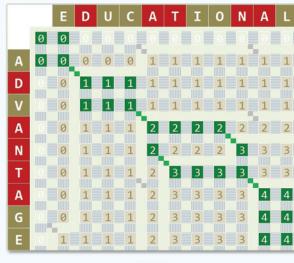
didacti
advanT

didacticC
advant

Data Structures & Algorithms, Tsinghua University

理解

◆ LCS的每一个解，对应于(0,0)与(n,m)之间的一条单调通路；反之亦然



Data Structures & Algorithms, Tsinghua University

◆ 与fib()类似，这里也有大量重复的递归实例（子问题）

各子问题，分别对应于A和B的某个前缀组合

因此实际上，总共不过 $\mathcal{O}(n \cdot m)$ 种

◆ 采用动态规划的策略

只需 $\mathcal{O}(n \cdot m)$ 时间即可计算出所有子问题

◆ 为此，只需

- 将所有子问题（假想地）列成一张表
- 颠倒计算方向

从LCS(0,0)出发，依次计算出所有项——直至LCS(n,m)

	d	i	d	a	c	t	i	c	a	l
e	0	0	0	0	0	0	0	0	0	0
a	0	0	0	0	1	1	1	1	1	1
d	0	1	1	1	1	1	1	1	1	1
v	0	1	1	1	1	1	1	1	1	1
a	0	1	1	1	2	2	2	2	2	2
n	0	1	1	1	2	2	2	2	2	2
t	0	1	1	1	2	2	3	3	3	3
a	0	1	1	1	2	2	3	3	3	4
g	0	1	1	1	2	2	3	3	3	4
e	0	1	1	1	2	2	3	3	3	4

Data Structures & Algorithms, Tsinghua University

课后

◆ 温习：程序设计基础（第3版）之第11章（动态规划）

◆ 自学：Introduction to Algorithms, §15.1, §15.3, §15.4

◆ 本节所介绍的迭代式LCS算法，似乎需要记录每个子问题的局部解，从而导致空间复杂度激增。实际上，这既不现实，亦没有必要。

试改进该算法，使得每个子问题只需常数空间，即可保证最终得到LCS的组成（而非仅仅长度）

◆ 考查序列 A = "immaculate" 和 B = "computer"

- 它们的LCS是什么？
- 这里的解是否唯一？是否有歧义性？
- 按照本节所给的算法，找出的是其中哪一个解？

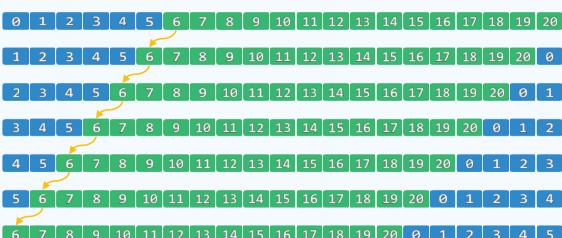
◆ 实现LCS算法的递归版和迭代版，并通过实测比较其运行时间

◆ 采用Memoization策略，实现fib与LCS算法

蛮力版

◆ void shift0(int * A, int n, int k) //反复以1为间距循环左移

{ while (k--) shift(A, n, 0, 1); } //共迭代k次, $\mathcal{O}(n \cdot k)$



Data Structures & Algorithms, Tsinghua University

迭代版

◆ int shift(int * A, int n, int s, int k) { // $\mathcal{O}(n / GCD(n, k))$

int b = A[s]; int i = s, j = (s + k) % n; int mov = 0; //mov记录移动次数

while (s != j) //从A[s]出发，以为间隔，依次左移k位

{ A[i] = A[j]; i = j; j = (j + k) % n; mov++; }

A[i] = b; return mov + 1; //最后，起始元素转入对应位置

}



◆ [0, n]由关于k的 $g = GCD(n, k)$ 个同余类组成

shift(s, k)能够且只能使其中之一就位

//各含 n/g 个元素

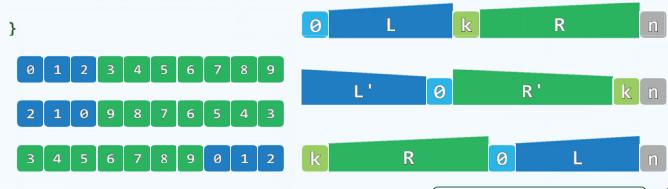
//即s所属的同余类

◆ 其它的同余类呢...

```
void shift1(int* A, int n, int k) { //经多轮迭代，实现数组循环左移k位，累计O(n+g)
    for (int s = 0, mov = 0; mov < n; s++) //O(g) = O(GCD(n, g))
        mov += shift(A, n, s, k);
}
```



```
void shift2( int * A, int n, int k ) { //借助倒置算法，将数组循环左移k位，O(3n)
    reverse( A, k ); //O(3k/2)
    reverse( A + k, n - k ); //O(3(n-k)/2)
    reverse( A, n ); //O(3n/2)
}
```



结论

局限：字宽



$$a^{9 \cdot 10^4} + 8 \cdot 10^3 + 7 \cdot 10^2 + 6 \cdot 10^1 + 5 \cdot 10^0$$

$$(a^{10^4})^9 \cdot (a^{10^3})^8 \cdot (a^{10^2})^7 \cdot (a^{10^1})^6 \cdot (a^{10^0})^5$$

Diagram showing the binary representation of the exponent 98765: 10000 1000 100 10 1. Arrows indicate powers of 10: a^{10^4} , a^{10^3} , a^{10^2} , a^{10^1} , a^{10^0} .

power_a(n) = aⁿ

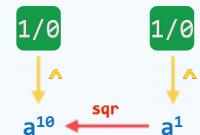
- ❖ 平凡实现：
- ```
pow = 1; //O(1)
```
- ```
while ( 0 < n ) //O(n)
```
- ```
{ pow *= a; n--; } //O(1+n)
```
- ❖  $T(n) = 1 + 2n = \mathcal{O}(n)$
- ❖ 线性？伪线性！
- ❖ 所谓输入规模，准确地应定义为用以描述输入所需的空间规模
- ❖ 对于此类数值计算即是n的二进制位数，亦即n的打印宽度
- ❖  $r = \lceil \log_2(n+1) \rceil = \mathcal{O}(\log n)$
- ❖  $T(r) = \mathcal{O}(2^r)$  //指数复杂度！

$$a^{1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0}$$

$$(a^2)^1 \cdot (a^2)^0 \cdot (a^2)^1 \cdot (a^2)^1 \cdot (a^2)^0$$

Diagram showing the binary representation of the exponent 10110<sub>b</sub>: 1 0 1 1 0. Arrows indicate powers of 2:  $a^{2^4}$ ,  $a^{2^3}$ ,  $a^{2^2}$ ,  $a^{2^1}$ ,  $a^{2^0}$ .

```
*int power(int a, int n) {
 int pow = 1, p = a; //O(1 + 1)
 while (0 < n) { //O(log n)
 if (n & 1) //O(1)
 pow *= p; //O(1)
 n >>= 1; //O(1)
 p *= p; //O(1)
 }
 return pow; //O(1)
}
```

❖ 输入规模 =  $r = \lceil \log_2(n+1) \rceil$ ❖ 运行时间 =  $T(r) = 1 + 1 + 4r + 1 = \mathcal{O}(r)$ 

❖ 如此，“实现”了从指数到线性的改进

悖论？

- ❖ 根据以上算法，“可以”在  $\mathcal{O}(\log n)$  时间内计算出  $power(n) = a^n$
- ❖ 然而， $a^n$  的二进制展开宽度是  $\Theta(n)$   
这意味着，即便是直接打印  $a^n$ ，也至少需要  $\Omega(n)$  时间.....哪里错了？
- ❖ 类似的悖论对  $fib(n)$  也存在...
- ❖ 令： $A = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} fib(0) & fib(1) \\ fib(1) & fib(2) \end{bmatrix}$ ，则： $A^n = \begin{bmatrix} fib(n-1) & fib(n) \\ fib(n) & fib(n+1) \end{bmatrix}$
- ❖ 因此参照上述  $power()$  算法，似乎也“可以”在  $\mathcal{O}(\log n)$  时间内计算出  $fib(n)$
- ❖ RAM模型：常数代价准则 (uniform cost criterion)  
对数代价准则 (logarithmic cost criterion)

结论

局限：随机数

As I have said so many times, God doesn't play dice with the world.  
- A. Einstein

那妇人道：“不好，不好！我这里有一方手帕，你顶在头上，遮了脸，撞个天婚，教我女儿从你跟前走过，你伸开手扯倒那个就把那个配了你罢。”

邓伟辉  
deng@tsinghua.edu.cn

就地随机置乱

- ❖ 给任何一个数组  $A[0, n]$ ，理想地将其中元素的次序随机打乱
- ❖ // [R. Fisher & F. Yates, 1938], [R. Durstenfeld, 1964], [D. E. Knuth, 1969]
- ```
void shuffle( int A[], int n )
{
    while ( 1 < n ) swap( A[ rand() % n ], A[ --n ] );
}
```
- ❖ 策略：自后向前，依次将各元素与随机选取的某一前驱（含自身）交换
- ❖ 的确可以等概率地生成所有 $n!$ 种排列？



此时你的思想进入我的思想
带有同样的行动和同样的面貌
使得我把二者构成同一个决定

When it is not necessary to make a decision, it is
necessary not to make a decision.

下界：代数判定树

邓俊辉
deng@tsinghua.edu.cn

时空性能、稳定性

多种角度估算的时间、空间复杂度

- 最好 / best-case
- 最坏 / worst-case
- 平均 / average-case
- 分摊 / amortized

其中，对最坏情况的估计最保守、最稳妥

因此，首先应考虑最坏情况最优的算法

//worst-case optimal

- 排序所需的时间，主要取决于
 - 关键码比较次数 / # {key comparison}
 - 元素交换次数 / # {data swap}
- 就地 (in-place)：

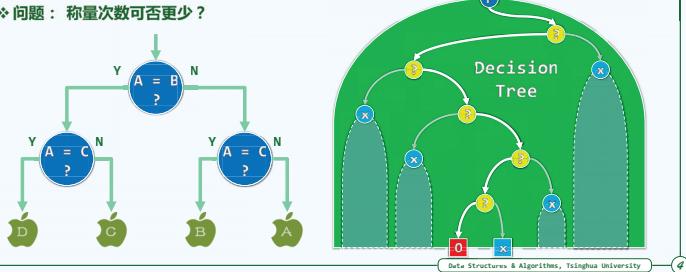
除输入数据本身外，只需 $\mathcal{O}(1)$ 附加空间
- 稳定 (stability)：

关键码相同的元素，在排序后相对位置保持

Data Structures & Algorithms, Tsinghua University

判定树

- 与CBA算法对应：每一可能的输出，都对应于至少一个叶节点；运行过程，对应于起始于根的路径
- 例如：经2/4次称量，必可从4/16只苹果中找出唯一的重量不同者
- 问题：称量次数可否更少？



下界： $\Omega(n \log n)$

比较树是三叉树 (ternary tree)

内部节点至多三个分支 (+|θ|-)

每一叶节点，各对应于

- 起自根节点的一条通路
- 某一可能的运行过程
- 运行所得的输出

叶节点深度 ~ 比较次数 ~ 计算成本

树高 ~ 最坏情况时的计算成本

树高的下界 ~ 所有CBA的时间复杂度下界

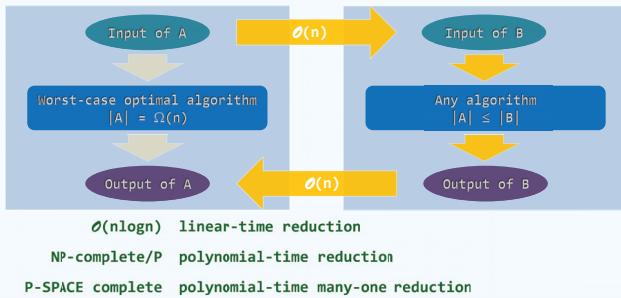
- 对于排序算法所对应的ADT，必有 $N \geq n!$
 - ADT的每一输出（叶子），对应于某一置换
 - 依此置换，可将输入序列转换为有序序列
 - 算法的输出，须覆盖所有可能的输入
- 包含 n 个叶节点的排序算法ADT，高度不低于 $\log_3 N \geq \log_3 n!$

$$= \log_3 e \cdot [n \ln n - n + \mathcal{O}(\ln n)]$$

$$= \Omega(n \cdot \log n)$$
- 这一结论，还可进一步推广到
- 理想平均情况、随机情况（概率>25%）...

Data Structures & Algorithms, Tsinghua University

除了（代数）判定树，归约（reduction）是确定下界的有力工具



Data Structures & Algorithms, Tsinghua University

难度与下界

- 由前述实例可见，同一问题的不同算法，复杂度可能相差悬殊
- 在可解的前提下，可否谈论问题的**难度**？如何比较不同问题的难度？
- 问题P若存在算法，则所有算法中**最低**的复杂度称为P的**难度**
- 为什么要确定问题的难度？给定问题P，如何确定其**难度**？
- 两个方面着手：设计复杂度更低的算法 + 证明更高的问题**难度**下界
- 一旦算法的复杂度达到**难度**下界，则说明
 - 就大O记号的意义而言，算法已经最优
 - 例如，排序问题下界为 $\Omega(n \log n)$ ，而且是累的...

Data Structures & Algorithms, Tsinghua University

最坏情况最优 + 基于比较

- 排序算法，最快能够有多快？
 - 语境1：就最坏情况最优而言
 - 语境2：就某一大类主流算法而言...
- 基于比较的算法 (comparison-based algorithm)

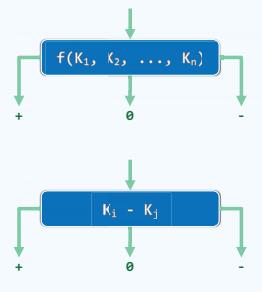
算法执行的进程，取决于一系列的数值（这里即关键码）比对结果

 - 比如，`max()` 和 `bubbleSort()`
- 任何CBA在最坏情况下，都需 $\Omega(n \log n)$ 时间才能完成排序

Data Structures & Algorithms, Tsinghua University

代数判定树

- Algebraic Decision Tree
 - 针对“比较-判定”式算法的计算模型
 - 给定输入的规模，将所有可能的输入所对应的一系列判断表示出来
- 代数判定：
 - 使用某常次数代数多项式
 - 将任意一组关键码做为变量，对多项式求值
 - 根据结果的符号，确定算法推进方向
- Comparison Tree：最简单的ADT，二元一次多项式，形如： $K_1 - K_2$



Data Structures & Algorithms, Tsinghua University



结论

下界：归约

言有易，言无难

不怕不识货，就怕货比货

deng@tsinghua.edu.cn

实例

- 【Red-Blue Matching】平面上任给n个红色点和n个蓝色点，如何用互不相交的线段配对联接

Sorting \leq_n Red-Blue Matching
- 【Element Uniqueness】任意n个实数中，是否包含雷同？ //EU的下界为 $\Omega(n \log n)$

EU \leq_n Closest Pair
- 【Integer Element Uniqueness】任意n个整数中，是否包含雷同？ //下界亦是 $\Omega(n \log n)$

IEU \leq_n Segment Intersection Detection
- 【Set Disjointness】任意一对集合A和B，是否存在公共元素？ //下界亦是 $\Omega(n \log n)$

SD \leq_n Diameter

Data Structures & Algorithms, Tsinghua University



2. 向量

抽象数据类型
接口与实现

Abstract Data Type vs. Data Structure

❖ 抽象数据类型 = 数据模型 + 定义在该模型上的一组操作

抽象定义 外部的逻辑特性

一种定义 不考虑时间复杂度

数据结构 = 基于某种特定语言，实现ADT的一整套算法

具体实现 内部的表示与实现

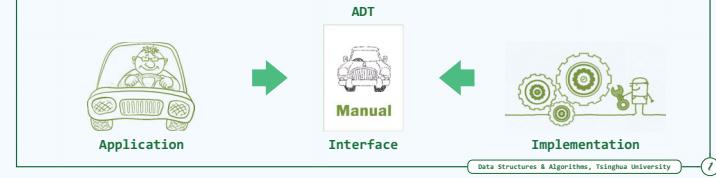
多种实现 与复杂度密切相关

操作&语义

不涉及数据的存储方式

完整的算法

要考虑数据的具体存储机制



Data Structures & Algorithms, Tsinghua University

Application = Interface × Implementation

❖ 在数据结构的具体实现与实际应用之间

ADT就分工与接口制定了统一的规范

- 实现：高效兑现数据结构的ADT接口操作

//做冰箱、造汽车

- 应用：便捷地通过操作接口使用数据结构

//用冰箱、开汽车

❖ 按照ADT规范

- 高层算法设计者与底层数据结构实现者

可高效地分工协作

- 不同的算法与数据结构可以便捷组合

- 每种操作接口只需统一地实现一次

代码篇幅缩短，软件复用度提高



Data Structures & Algorithms, Tsinghua University

循环访问

❖ C/C++语言中，数组元素与编号一一对应：A[0], A[1], A[2], ..., A[n-1]

❖ 反之，元素各由编号唯一指代，并可直接访问 //故亦称线性数组 (linear array)

A[i]的物理地址 = A + i×s, s为单个元素占用的空间量



❖ 向量是数组的抽象与泛化，由一组元素按线性次序封装而成

- 各元素与[0, n)内的秩 (rank) ——对应

typedef int Rank; //循环访问 (call-by-rank)

- 操作、管理维护更加简化、统一与安全

- 元素类型可灵活选取，便于定制复杂数据结构 //Vector< PFCTree* > pfcForest;

ADT操作实例

操作	输出	向量组成 (自左向右)	操作	输出	向量组成 (自左向右)
初始化			disordered()	3 [4 3 7 4 9 6]	
insert(0, 9)	[9]		find(9)	4 [4 3 7 4 9 6]	
insert(0, 4)	[4 9]		find(5)	-1 [4 3 7 4 9 6]	
insert(1, 5)	[4 5 9]		sort()	[3 4 4 6 7 9]	
put(1, 2)	[4 2 9]		disordered()	0 [3 4 4 6 7 9]	
get(2)	9 [4 2 9]		search(1)	-1 [3 4 4 6 7 9]	
insert(3, 6)	[4 2 9 6]		search(4)	2 [3 4 4 6 7 9]	
insert(1, 7)	[4 7 2 9 6]		search(8)	4 [3 4 4 6 7 9]	
remove(2)	2 [4 7 9 6]		search(9)	5 [3 4 4 6 7 9]	
insert(1, 3)	[4 3 7 9 6]		search(10)	5 [3 4 4 6 7 9]	
insert(3, 4)	[4 3 7 4 9 6]		uniquify()	[3 4 6 7 9]	
size()	6 [4 3 7 4 9 6]		search(9)	4 [3 4 6 7 9]	

Data Structures & Algorithms, Tsinghua University

2. 向量

抽象数据类型
模板类官职须由生处有，文章不管用时无
堪笑翰林陶学士，年年依样画葫芦

Abstract Data Type vs. Data Structure

❖ 抽象数据类型 = 数据模型 + 定义在该模型上的一组操作

抽象定义 外部的逻辑特性

一种定义 不考虑时间复杂度

数据结构 = 基于某种特定语言，实现ADT的一整套算法

具体实现 内部的表示与实现

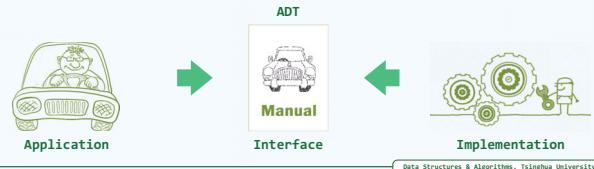
多种实现 与复杂度密切相关

操作&语义

不涉及数据的存储方式

完整的算法

要考虑数据的具体存储机制



Data Structures & Algorithms, Tsinghua University

2. 向量

抽象数据类型
从数组到向量邓稼辉
deng@tsinghua.edu.cn

阿圆很快，把手一点说：“到了，就是这里。妈妈，你只管找头，311，就是爸爸的号。”

贵贱长少，秩秩焉，莫不从祖公而敬之，是天下之大节也。

向量ADT接口

操作	功能	适用对象
size()	报告向量当前的规模 (元素总数)	向量
get(r)	获取秩为r的元素	向量
put(r, e)	用e替换秩为r元素的数值	向量
insert(r, e)	e作为秩为r元素插入，原后继依次后移	向量
remove(r)	删除秩为r的元素，返回该元素原值	向量
disordered()	判断所有元素是否已按非降序排列	向量
sort()	调整各元素的位置，使之按非降序排列	向量
find(e)	查找目标元素e	向量
search(e)	查找e，返回不大于且秩最大的元素	有序向量
deduplicate(), uniquify()	剔除重复元素	向量/有序向量
traverse()	遍历向量并统一处理所有元素	向量

Data Structures & Algorithms, Tsinghua University

STL Vector

❖ #include <iostream>

#include <vector>

using namespace std;

❖ vector<int> v; // an empty vector of integers

vector<int> s(32, 63); // { 63, 63, 63, ..., 63 }, sum = 2016

s.insert(s.begin + 2, 2017); // { 63, 63, 2017, 63, ..., 63 }

s.erase(s.end - 30, s.end); // { 63, 63, 2017 }

for (i = 0; i < s.size(); i++)

cout << s[i] << endl;

Data Structures & Algorithms, Tsinghua University

template <typename T> class Vector { //向量模板类

private: Rank _size; int _capacity; T* _elem; //规模、容量、数据区

protected:

/* ... 内部函数 */

public:

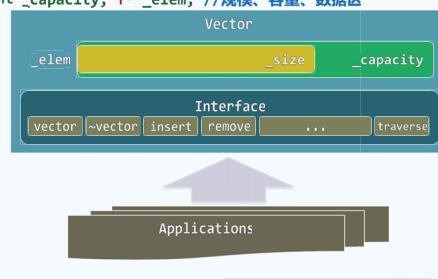
/* ... 构造函数 */

/* ... 析构函数 */

/* ... 只读接口 */

/* ... 可写接口 */

/* ... 遍历接口 */



Data Structures & Algorithms, Tsinghua University

```

template <typename T> class Vector { //向量模板类
private: Rank _size; int _capacity; T* _elem; //规模、容量、数据区
protected:
/* ... 内部函数 */
public:
/* ... 构造函数 */
/* ... 析构函数 */
/* ... 只读接口 */
/* ... 可写接口 */
/* ... 遍历接口 */
};
```

```
*#define DEFAULT_CAPACITY 3 //默认初始容量(实际应用中可设置为更大)
*Vector( int c = DEFAULT_CAPACITY )
    {_elem = new T[ _capacity = c ]; _size = 0; } //默认
*Vector( T const * A, Rank lo, Rank hi ) //数组区间复制
    { copyFrom( A, lo, hi ); }
Vector( Vector<T> const& V, Rank lo, Rank hi ) //向量区间复制
    { copyFrom( V._elem, lo, hi ); }
Vector( Vector<T> const& V ) //向量整体复制
    { copyFrom( V._elem, 0, V._size ); }
~Vector() { delete [] _elem; } //释放内部空间
```

Data Structures & Algorithms, Tsinghua University



一个人办一县事，要有一省的眼光；
办一省事，要有一国之眼光；
办一国事，要有世界的眼光。

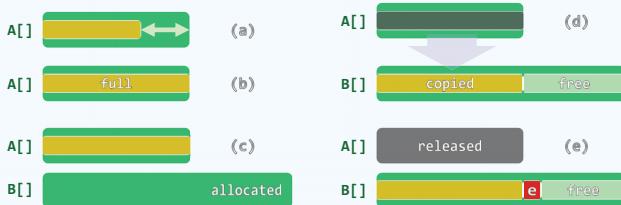
其实“我”不需扩大，宇宙只是一个“我”，只有在
我们精神往下陷落时，宇宙与我才分开

向量 可扩充向量：算法

邓伟辉

deng@tsinghua.edu.cn

- 蝉的哲学：身体经过一段时间的生长，会蜕去原先的外壳，代之以更大的新外壳
- 向量：在即将上溢时，适当扩大内部数组的容量



Data Structures & Algorithms, Tsinghua University



...在他的心理上，他总以为北平是天底下最可靠的大城，不管有什么灾难，到三个月必定消难满，而后诸事大吉。北平的灾难恰似一个人免不了有些头疼脑热，过几天自然会好了的。

向量 可扩充向量：分摊

邓伟辉

deng@tsinghua.edu.cn

- `T* oldElem = _elem; _elem = new T[_capacity <= 1];` //容量加倍
- 最坏情况：在初始容量1的满向量中，连续插入 $n = 2^m \gg 2$ 个元素...

于是，在第1、2、4、8、16、...次插入时都需扩容

每次扩容过程中复制原向量的时间成本依次为

$1, 2, 4, 8, \dots, 2^m = n$ //几何级数

总体耗时 = $\mathcal{O}(n)$ ，每次扩容的分摊成本为 $\mathcal{O}(1)$



Data Structures & Algorithms, Tsinghua University

_elem	0	hi - lo	s	2*(hi - lo)
-------	---	---------	---	-------------

copy

A[]	lo	hi
-----	----	----

```
* template <typename T> //T为基本类型，或已重载赋值操作符='
void Vector<T>::copyFrom( T const * A, Rank lo, Rank hi ) {
    _elem = new T[ _capacity = 2 * (hi - lo) ]; //分配空间
    _size = 0; //规模清零
    while ( lo < hi ) //A[lo, hi)内的元素逐一
        _elem[ _size++ ] = A[ lo++ ]; //复制至_elem[0, hi - lo)
} //O(hi - lo) = O(n)
```

Data Structures & Algorithms, Tsinghua University

- 开辟内部数组_elem[]并使用一段地址连续的物理空间

_capacity	总容量	_elem	物理空间	_size	_capacity
-----------	-----	-------	------	-------	-----------

- 若采用静态空间管理策略，容量_capacity固定，则有明显的不足...

- 上溢/overflow: _elem[]不足以存放所有元素，尽管此时系统往往仍有足够的空间
 - 下溢/underflow: _elem[]中的元素寥寥无几
 - 装填因子/load factor: $\lambda = \frac{\text{_size}}{\text{_capacity}} \ll 50\%$

- 更糟糕的是，一般的应用环境中难以准确预测空间的需求量

- 可否使得向量可随实际需求动态调整容量，并同时保证高效率？

Data Structures & Algorithms, Tsinghua University

- template <typename T> void Vector<T>::expand() { //向量空间不足时扩容

```
if ( _size < _capacity ) return; //尚未满员时，不必扩容
_capacity = max( _capacity, DEFAULT_CAPACITY ); //不低于最小容量
T* oldElem = _elem; _elem = new T[ _capacity <= 1 ]; //容量加倍
for ( int i = 0; i < _size; i++ ) //复制原向量内容
    _elem[i] = oldElem[i]; //T为基本类型，或已重载赋值操作符='
delete [] oldElem; //释放原空间
} //得益于向量的封装，尽管扩容之后数据区的物理地址有所改变，却不敢出现野指针
```

Data Structures & Algorithms, Tsinghua University

- `T* oldElem = _elem; _elem = new T[_capacity += INCREMENT];` //追加固定增量

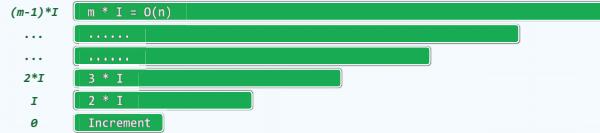
最坏情况：在初始容量0的空向量中，连续插入 $n = m \cdot I \gg 2$ 个元素...

于是，在第1、 $I+1$ 、 $2I+1$ 、 $3I+1$ 、...次插入时，都需扩容

即便不计申请空间操作，各次扩容过程中复制原向量的时间成本依次为

$0, I, 2I, \dots, (m-1)I$ //算术级数

总体耗时 = $I \cdot (m-1) \cdot m/2 = \mathcal{O}(n^2)$ ，每次扩容的分摊成本为 $\mathcal{O}(n)$



Data Structures & Algorithms, Tsinghua University

- `T* oldElem = _elem; _elem = new T[_capacity += INCREMENT];` //追加固定增量

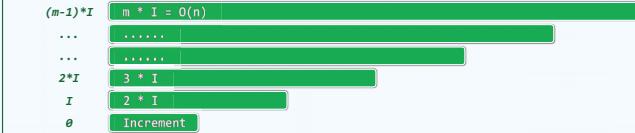
最坏情况：在初始容量0的空向量中，连续插入 $n = m \cdot I \gg 2$ 个元素...

于是，在第1、 $I+1$ 、 $2I+1$ 、 $3I+1$ 、...次插入时，都需扩容

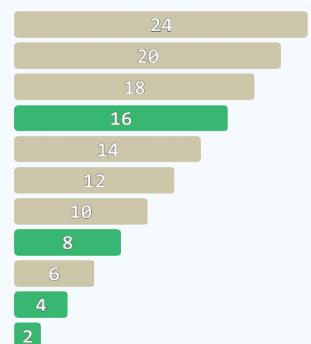
即便不计申请空间操作，各次扩容过程中复制原向量的时间成本依次为

$0, I, 2I, \dots, (m-1)I$ //算术级数

总体耗时 = $I \cdot (m-1) \cdot m/2 = \mathcal{O}(n^2)$ ，每次扩容的分摊成本为 $\mathcal{O}(n)$



Data Structures & Algorithms, Tsinghua University



Data Structures & Algorithms, Tsinghua University

◆ 平均复杂度 (average complexity)

根据数据结构各种操作出现概率的分布，将对应的成本加权平均

- 各种可能的操作，作为独立事件分别考查
- 割裂了操作之间的相关性和连贯性
- 往往不能准确地评判数据结构和算法的真实性能

◆ 分摊复杂度 (amortized complexity)

对数据结构连续地实施足够多次操作，所需总体成本分摊至单次操作

- 从实际可行的角度，对一系列操作做整体的考量
- 更加忠实地刻画了可能出现的操作序列
- 更为精准地评判数据结构和算法的真实性能

◆ 后面将看到更多、更复杂的例子

元素访问

◆ `V.get(r)` 和 `V.put(r, e)` 不够便捷、直观，可否沿用数组的访问方式 `V[r]` ?

可以！比如，通过重载下标操作符“`[]`”

◆ `template <typename T> //可作为左值: V[r] = (T) (2*x + 3)`

```
T & Vector<T>::operator[]( Rank r ) { return _elem[ r ]; }
```

◆ `template <typename T> //仅限于右值: T x = V[r] + U[s] * W[t]`

```
const T & Vector<T>::operator[]( Rank r ) const { return _elem[ r ]; }
```

◆ 这里采用了简易的方式处理意外和错误（比如，入口参数约定： $0 \leq r < _size$ ）

实际应用中，应采用更为严格的方式

区间删除

◆ `template <typename T> //删除区间[lo, hi], 0 <= lo <= hi <= size`

```
int Vector<T>::remove( Rank lo, Rank hi ) { //O(n-hi)
    if ( lo == hi ) return 0; //出于效率考虑，单独处理退化情况
    while ( hi < _size ) _elem[ lo ++ ] = _elem[ hi ++ ]; //从hi,_size顺次前移
    _size = lo; shrink(); //更新规模，若有必要则缩容
    return hi - lo; //返回被删除元素的数目
}
```



向量

无序向量：查找

θ2-C2

邓俊辉
deng@tsinghua.edu.cn

他便站将起来，背着手踱来踱去，侧眼把那些人逐个个瞧过去，内中一个果然衣领上挂着一寸来长短彩线头。

有序向量：比较器

◆ `template <typename K, typename V> struct Entry { //词条模板类`

```
K key; V value; //关键码、数值
Entry ( K k = K(), V v = V() ) : key ( k ), value ( v ) {}; //默认构造函数
Entry ( Entry<K, V> const& e ) : key ( e.key ), value ( e.value ) {}; //克隆
bool operator== ( Entry<K, V> const& e ) { return key == e.key; } //等于
bool operator!= ( Entry<K, V> const& e ) { return key != e.key; } //不等
bool operator< ( Entry<K, V> const& e ) { return key < e.key; } //小于
bool operator> ( Entry<K, V> const& e ) { return key > e.key; } //大于
}; //得益于比较器和判等器，从此往后，不必严格区分词条及其对应的关键码
```

向量

无序向量：基本操作

θ2-C1

邓俊辉
deng@tsinghua.edu.cn

它污浊，它美丽，它衰老，它活泼，它杂乱，它安闲，它可爱，它是伟大的夏初的北平。

有序向量：判等器

◆ `template <typename K, typename V> struct Entry { //词条模板类`

```
K key; V value; //关键码、数值
Entry ( K k = K(), V v = V() ) : key ( k ), value ( v ) {}; //默认构造函数
Entry ( Entry<K, V> const& e ) : key ( e.key ), value ( e.value ) {}; //克隆
bool operator== ( Entry<K, V> const& e ) { return key == e.key; } //等于
bool operator!= ( Entry<K, V> const& e ) { return key != e.key; } //不等
bool operator< ( Entry<K, V> const& e ) { return key < e.key; } //小于
bool operator> ( Entry<K, V> const& e ) { return key > e.key; } //大于
}; //得益于比较器和判等器，从此往后，不必严格区分词条及其对应的关键码
```

顺序查找

◆ `template <typename T> Rank Vector<T>:: //O(hi - lo) = O(n)`

```
find( T const & e, Rank lo, Rank hi ) const { //0 <= lo < hi <= _size
    while ( (lo < hi--) && (e != _elem[hi]) ); //逆向查找
    return hi; //hi < lo意味着失败；否则hi即命中者的秩（多个命中时，返回最大的秩）
} //Excel::match(e, range, type)
```



◆ 输入敏感 (input-sensitive)：最好O(1)，最差O(n)



你去问问你琏二婶子，正月里请吃年酒的日子拟了没有。若拟定了，叫书房里明白开了单子来，咱们再请时，就不能重犯了。旧年不留意重了几家，不说咱们不留神，倒像两宅商议定了送虚情怕费事一样。



让他们每个人轮流到你的宝座下，同样诚恳地坦白他们的内心，然后再看有没有一个人敢向你说：“我比这个人好。”

对向量中的每一元素，统一实施visit()操作 //如何指定visit()？如何将其传递到向量内部？

template <typename T> //函数指针，只读或局部性修改

```
void Vector<T>::traverse( void ( * visit )( T & ) )  
{ for ( int i = 0; i < _size; i++ ) visit( _elem[i] ); }
```

template <typename T> template <typename VST> //函数对象，全局性修改更便捷

```
void Vector<T>::traverse( VST & visit )  
{ for ( int i = 0; i < _size; i++ ) visit( _elem[i] ); }
```

比如，为统一地将向量中所有元素分别加一，只需

- 实现一个可使单个T类型元素加一的类（结构）

```
template <typename T> //假设T可直接递增或已重载操作符“++”  
struct Increase //函数对象：通过重载操作符“()”实现  
{ virtual void operator()( T & e ) { e++; } }; //加一
```

- 将其作为参数传递给遍历算法

```
template <typename T> void increase( Vector<T> & V )  
{ V.traverse( Increase<T>() ); } //即可以之作为基本操作，遍历向量
```

作为练习，可模仿此例，实现统一的减一、加倍、求和等遍历功能

Data Structures & Algorithms, Tsinghua University

还记得起泡排序的原理？有序/无序序列中，任何/总有一对相邻元素顺序/逆序

因此，相邻逆序对的数目，可在一定程度上度量向量的紊乱程度

```
template <typename T> int Vector<T>::disordered() const {  
    int n = 0; //相邻逆序对的计数器  
    for ( int i = 1; i < _size; i++ ) //逐一检查各对相邻元素  
        n += ( _elem[i-1] > _elem[i] ); //逆序则计数  
    return n; //向量有序，当且仅当n = 0  
}
```

无序向量经预处理转换为有序向量之后，相关算法多可优化

Data Structures & Algorithms, Tsinghua University

```
template <typename T> int Vector<T>::uniquify() {  
    Rank i = 0, j = 0;  
    while ( ++j < _size )  
        if ( _elem[ i ] != _elem[ j ] ) _elem[ ++i ] = _elem[ j ];  
    _size = ++i; shrink();  
    return j - i;  
}
```



Data Structures & Algorithms, Tsinghua University

template <typename T> int Vector<T>::deduplicate() {

int oldSize = _size;

(b)

Rank i = 1;

while (i < _size)

find(_elem[i], 0, i) < 0 ? (a)

[0, i)

i++

: remove(i);

(c)

return oldSize - _size;

}

Data Structures & Algorithms, Tsinghua University

template <typename T> int Vector<T>::uniquify() {

int oldSize = _size; int i = 1;

while (i < _size)

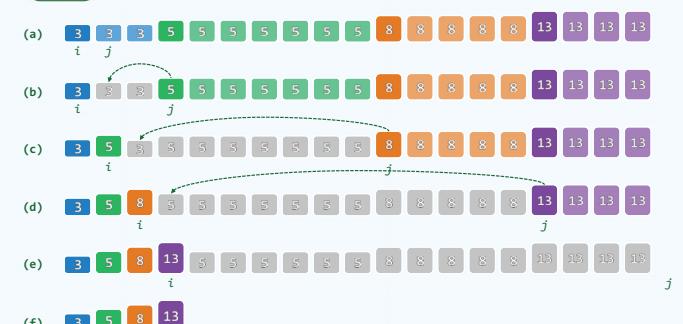
_elem[i-1] == _elem[i] ? remove(i) : i++;

return oldSize - _size;

}



Data Structures & Algorithms, Tsinghua University



Data Structures & Algorithms, Tsinghua University

θ2-D2

向量
有序向量：二分查找（版本A）

自从爷爷去后，这山被二郎菩萨点上火，烧杀了大半。我们蹲在井里，躲在洞内，藏于铁板桥下，得了性命。及至火灭烟消，出来时，又没花果养膳，难以存活，别处又去了一半。我们这一半，挺苦的住在山中，这两年，又被些打猎的抢了一半去也。

邓伟辉
deng@tsinghua.edu.cn

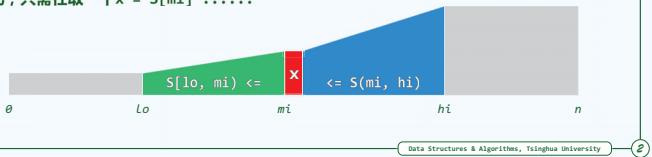
轴点

❖ 有序向量中，处处皆为轴点 //说人话！

❖ 以任一元素 $s[mi]$ 为界

- 都可将待查找区间 $[lo, hi]$ 分为三部分，且
- $s[lo, mi] \leq s[mi] \leq s(mi, hi)$

❖ 因此，只需任取一个 $x = s[mi]$



Data Structures & Algorithms, Tsinghua University

实现

❖ template <typename T> //在有序向量区间[lo, hi]内查找元素

```
static Rank binSearch( T *s, T const &e, Rank lo, Rank hi ) {
    while ( lo < hi ) { //每步迭代可能要做两次比较判断，有三个分支
        Rank mi = ( lo + hi ) >> 1; //以中点为轴点（区间宽度的折半，等效于其数值表示的轴
        if      ( e < s[mi] ) hi = mi; //深入前半段[lo, mi)继续查找
        else if ( s[mi] < e ) lo = mi + 1; //深入后半段(mi, hi)
        else                return mi; //在mi处命中
    }
    return -1; //查找失败
}
```



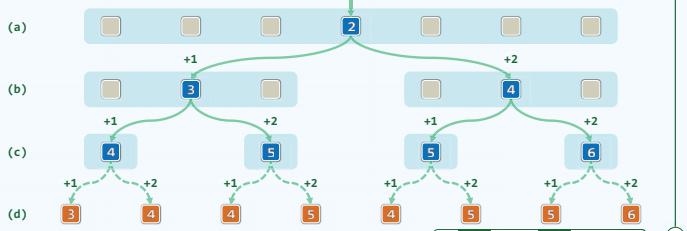
Data Structures & Algorithms, Tsinghua University

查找长度

❖ 如何更精细地评估查找算法的性能？考查关键码的比较次数 (search length)

❖ 通常，需分别针对成功与失败查找，从最好、最坏、平均等角度评估

❖ 比如，成功、失败时的平均查找长度均大致为 $\mathcal{O}(1.50 \cdot \log n)$ //详见教材、习题解析



Data Structures & Algorithms, Tsinghua University

θ2-D3

向量
有序向量：Fibonacci查找

常伟思微微一笑说：这个比例很奇怪，是吗？

他又想来想去，又想不出好地方，于是终于决心，假定这“幸福的家庭”所在的地方叫做A。

统一接口

❖ template <typename T> //查找算法统一接口， $0 \leq lo < hi \leq _size$

```
Rank Vector<T>::search( T const &e, Rank lo, Rank hi ) const {
```

return (rand() % 2) ? //按各50%的概率随机选用

binSearch(_elem, e, lo, hi) //二分查找算法，或者

: fibSearch(_elem, e, lo, hi); //Fibonacci查找算法



Data Structures & Algorithms, Tsinghua University

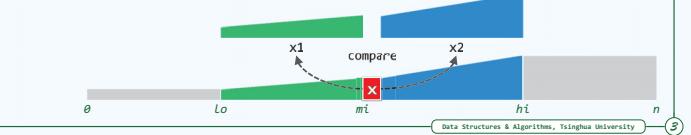
减而治之

❖ 只需将目标元素 e 与 x 做一比较，即可分三种情况进一步处理：

- $e < x$ ：则 e 若存在必属于 **左侧子区间**，故可（减除 $s[mi, hi]$ 并）递归深入 $s[lo, mi]$
- $x < e$ ：则 e 若存在必属于 **右侧子区间**，亦可（减除 $s[lo, mi]$ 并）递归深入 $s[mi, hi]$
- $e = x$ ：已在此处命中，可随即返回 //若多个，返回何者？

❖ 若轴点 mi 取作中点，则每经过至多两次比较

或者能够命中，或者将问题规模缩减一半



Data Structures & Algorithms, Tsinghua University

实例 + 复杂度

❖ $s.search(8, 0, 7)$:

经 $2 + 1 + 2 = 5$ 次比较，在 $s[4]$ 命中

	0	1	2	3	4	5	6	7
(1)	2	4	5	7	8	9	12	
(2)			8	9	12			
(3)				8				

$s.search(3, 0, 7)$:

经 $1 + 1 + 2 = 4$ 次比较，在 $s[1]$ 失败

	0	1	2	3	4	5	6	7
(1)	2	4	5	7	8	9	12	
(2)		2	4	5				
(3)			2					
(4)				1				

❖ 线性递归： $T(n) = T(n/2) + O(1) = \mathcal{O}(\log n)$ ，大大优于顺序查找

递归跟踪：轴点总能找到中点，递归深度 $\mathcal{O}(\log n)$ ；各递归实例仅耗时 $O(1)$

Data Structures & Algorithms, Tsinghua University

查找长度

❖ $n = 7$ 时，各元素对应的成功查找长度为

{ 4, 3, 5, 2, 5, 4, 6 }

在等概率情况下，平均成功查找长度

$$= 29 / 7 = 4.14$$

	0	1	2	3	4	5	6	7
(a)	2	4	5	7	8	9	12	
(b)		3		4		5		
(c)			4	5	6			
(d)				3	4	5	6	

❖ 共8种失败情况，查找长度分别为

{ 3, 4, 4, 5, 4, 5, 5, 6 }

在等概率情况下，平均失败查找长度

$$= 36 / 8 = 4.50$$

	0	1	2	3	4	5	6	7
(a)	2	4	5	7	8	9	12	
(b)		3		4		5		
(c)			4	5	6			
(d)				3	4	5	6	

Data Structures & Algorithms, Tsinghua University

向量

有序向量：Fibonacci查找

常伟思微微一笑说：这个比例很奇怪，是吗？

他又想来想去，又想不出好地方，于是终于决心，假定这“幸福的家庭”所在的地方叫做A。

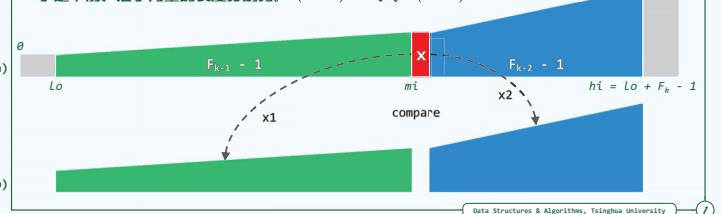
思路及原理

❖ 版本A：转向左、右分支前的关键码比较次数不等，而递归深度却相同

❖ 通过递归深度的不均衡对转向成本的不均衡做补偿，平均查找长度应能进一步缩短！

❖ 比如，若有 $n = fib(k) - 1$ ，则可取 $mi = fib(k-1) - 1$

于是，前、后子向量的长度分别为 $fib(k-1) - 1$ 、 $fib(k-2) - 1$



Data Structures & Algorithms, Tsinghua University

```

实现

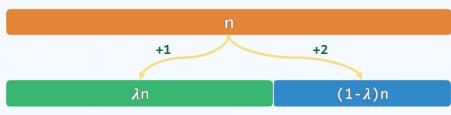
◆ template <typename T> //0 <= lo <= hi <= _size
static Rank fibSearch( T * S, T const & e, Rank lo, Rank hi ) {
    for ( Fib fib(hi - lo); lo < hi; ) { //Fib数列制表备查
        while ( hi - lo < fib.get() ) fib.prev(); //自后向前顺序查找轴点(分摊O(1))
        Rank mi = lo + fib.get() - 1; //确定形如fib(k) - 1的轴点
        if ( e < S[mi] ) hi = mi; //深入前半段[lo, mi)继续查找
        else if ( S[mi] < e ) lo = mi + 1; //深入后半段(mi, hi)
        else return mi; //在mi处命中
    }
    return -1; //查找失败
}

```

Data Structures & Algorithms, Tsinghua University

通用策略

- ◆ 在任何区间 $[0, n)$ 内，总是选取 $\lfloor \lambda \cdot n \rfloor$ 作为轴点， $0 \leq \lambda < 1$
- 比如：二分查找对应于 $\lambda = 0.5$ ，Fibonacci查找对应于 $\lambda = \phi = 0.6180339\dots$
- ◆ 这类查找算法的渐进复杂度为 $\alpha(\lambda) \cdot \log_2 n = \mathcal{O}(\log n)$
- ◆ (在常系数意义上的) 性能最优，即意味着 $\alpha(\lambda)$ 达到最小



Data Structures & Algorithms, Tsinghua University

θ2-D4

有序向量：二分查找（版本B）

和微风勾到一起的光，象冰凉的刀刃似的，把宽静的大街切成两半，一半儿黑，一半儿亮。那黑的一半，使人感到阴森，亮的一半使人感到凄凉。

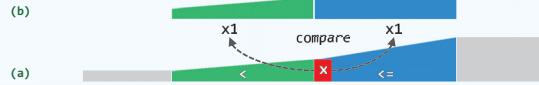
邓俊辉
deng@tsinghua.edu.cn

实现

```

◆ template <typename T> static Rank binSearch( T * S, T const & e, Rank lo, Rank hi ) {
    while ( 1 < hi - lo ) { //有效查找区间的宽度缩短至1时，算法才终止
        Rank mi = (lo + hi) >> 1; //以中点为轴点，经比较后确定深入
        e < S[mi] ? hi = mi : lo = mi; // [lo, mi) 或 [mi, hi)
    } //出口时hi = lo + 1，查找区间仅含一个元素A[lo]
    return e == S[lo] ? lo : -1; //返回命中元素的秩，或者(失败时)非法的秩
} //相对于版本A，最好(坏)情况下更坏(好)；各种情况下的SL更加接近，整体性能更趋稳定

```



Data Structures & Algorithms, Tsinghua University

语义约定

- ◆ 约定：`search()`总是返回
 - m = 不大于 e 的最后一个元素
 - (其后继 M = 大于 e 的第一个元素)

◆ 改进版本B：

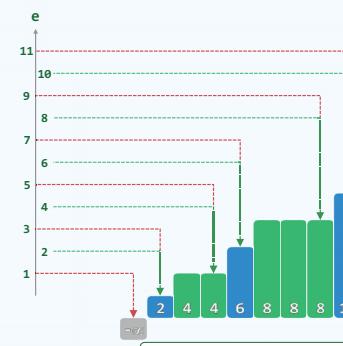
```

return e == S[lo] ? lo : -1;
return e < S[lo] ? lo-1 : lo;

```

◆ 课后：相应地改进`fibSearch()`

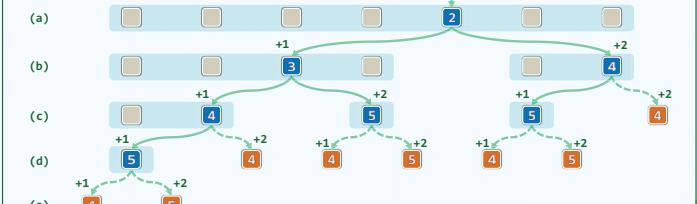
◆ 有没有更为简明的实现方式？



◆ Fibonacci查找的ASL，(在常系数的意义上)优于二分查找 //详见教材、习题解析

◆ 仍以 $n = \text{fib}(6) - 1 = 7$ 为例，在等概率情况下

- 平均成功查找长度 = $(5 + 4 + 3 + 5 + 2 + 5 + 4) / 7 = 28/7 = 4.00$
- 平均失败查找长度 = $(4 + 5 + 4 + 4 + 5 + 4 + 5 + 4) / 8 = 35 / 8 = 4.38$



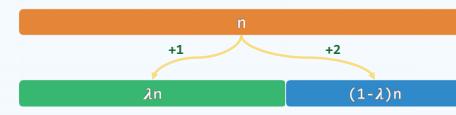
Data Structures & Algorithms, Tsinghua University

 $\phi = 0.6180339\dots$

◆ 递推式： $\alpha(\lambda) \cdot \log_2 n = \lambda \cdot [1 + \alpha(\lambda) \cdot \log_2(\lambda n)] + (1 - \lambda) \cdot [2 + \alpha(\lambda) \cdot \log_2((1 - \lambda)n)]$

◆ 整理后： $\frac{-\ln 2}{\alpha(\lambda)} = \frac{\lambda \cdot \ln \lambda + (1 - \lambda) \cdot \ln(1 - \lambda)}{2 - \lambda}$

◆ 当 $\lambda = \phi = (\sqrt{5} - 1)/2$ 时， $\alpha(\lambda) = 1.440420\dots$ 达到最小



Data Structures & Algorithms, Tsinghua University

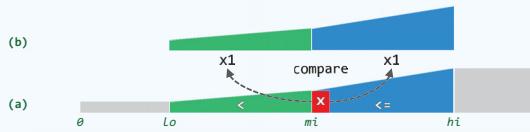
◆ 二分查找中左、右分支转向代价不平衡的问题，也可直接解决

◆ 比如，每次迭代仅做1次关键码比较；如此，所有分支只有2个方向，而不再是3个

◆ 同样地，轴点 mi 取作中点，则查找每深入一层，问题规模也缩减一半

- $e < x$ ：则 e 若存在必属于左侧子区间 $S[lo, mi]$ ，故可递归深入
- $x \leq e$ ：则 e 若存在必属于右侧子区间 $S[mi, hi]$ ，亦可递归深入

直到 $hi - lo = 1$ ，才判断是否命中



Data Structures & Algorithms, Tsinghua University

语义约定

◆ 各种特殊情况，如何统一地处置？比如

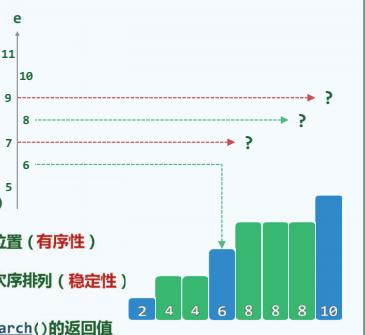
- 目标元素不存在；或反过来
- 目标元素同时存在多个

◆ 有序向量自身，如何便捷地维护？

比如：`V.insert(1 + V.search(e), e)`

- 即便失败，也应给出新元素适当的插入位置(有序性)
- 若有重复元素，每一组也需要按其插入的次序排列(稳定性)

◆ 为此，需要更为精细、明确、简捷地定义`search()`的返回值



Data Structures & Algorithms, Tsinghua University

语义约定

◆ 约定：`search()`总是返回

- m = 不大于 e 的最后一个元素
- (其后继 M = 大于 e 的第一个元素)

◆ 改进版本B：

```

return e == S[lo] ? lo : -1;
return e < S[lo] ? lo-1 : lo;

```

◆ 课后：相应地改进`fibSearch()`

◆ 有没有更为简明的实现方式？



θ2-D5

有序向量：二分查找（版本C）

向量

邓俊辉
deng@tsinghua.edu.cn

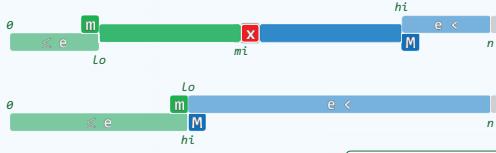
```
template <typename T> static Rank binSearch( T * S, T const & e, Rank lo, Rank hi ) {
    while ( lo < hi ) { //不变性 : A[lo, hi) ⊆ e < A[hi, n]
        Rank mi = (lo + hi) >> 1; //以中点为轴点, 经比较后确定深入
        e < S[mi] ? hi = mi : lo = mi + 1; // [lo, mi] 或 (mi, hi)
    } //出口时, 必有S[lo = hi] = M
    return lo - 1; //故, S[lo-1] = M
} //留意与版本B的差异...
待查找区间宽度缩短至 $\frac{1}{2}$ 而非 $\frac{1}{4}$ 时, 算法才结束 //lo == hi
转入右侧向量时, 左边界取作mi+1而非mi //A[mi]会被遗漏?
无论成功与否, 返回的秩严格符合接口的语义约定... //如何证明其正确性?
```

♦ 在算法执行过程中的任意时刻

- A[lo-1] 总是 (截至当前已确认的) 不大于e的最大者 (m)
- A[hi] 总是 (截至当前已确认的) 大于e的最小者 (M)

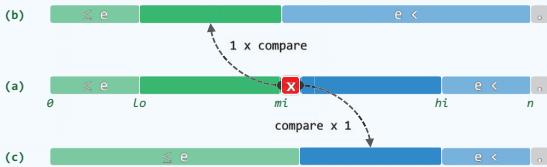
♦ 当算法终止时

- A[lo-1] = A[hi-1] 即是 (全局) 不大于e的最大者



♦ 初始时, lo = 0且hi = n, A[0, lo) = A[hi, n) = ∅, 自然成立

♦ 数学归纳: 假设不变性一直保持至(a), 以下无非两种情况...



♦ 假设: 已知有序向量中各元素随机分布的规律

比如: 独立且均匀的随机分布

♦ 于是: [lo, hi)内各元素应大致呈线性趋势增长

$$\frac{mi - lo}{hi - lo} \approx \frac{e - A[lo]}{A[hi] - A[lo]}$$

♦ 因此: 通过猜测轴点mi, 可以极大地提高收敛速度

$$mi \approx lo + (hi - lo) \cdot \frac{e - A[lo]}{A[hi] - A[lo]}$$

♦ 以英文词典为例: binary大致位于2/26处

search大致位于19/26处

[16]	0	A	1	[1, 53]
1	B	74	[131, 184]	
2	C	158	[184, 196]	
3	D	292	[196, 288]	
4	E	368	[288, 259]	
5	F	409	[259, 311]	
6	G	473	[311, 363]	
7	H	516	[363, 414]	
8	I	562	[414, 466]	
9	J	607	[466, 518]	
10	K	617	[518, 569]	
11	L	656	[569, 621]	
12	M	681	[621, 673]	
13	N	748	[673, 724]	
14	O	771	[724, 776]	
15	P	806	[776, 827]	
16	Q	915	[827, 929]	
17	R	922	[879, 931]	
18	S	1002	[931, 982]	
19	T	1176	[982, 1034]	
20	U	1253	[1034, 1086]	
21	V	1273	[1086, 1137]	
22	W	1289	[1137, 1189]	
23	X	1337	[1189, 1241]	
24	Y	1338	[1241, 1292]	
25	Z	1341	[1292, 1344]	
26		1344		

♦ 查找目标: e = 50

♦ lo = 0, hi = 18

5	10	12	14	19	23	29	36	39	41	44	51	54	59	72	79	82	86	92
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

插值: mi = 0 + (18 - 0)*(50 - 5)/(92 - 5) = 9

比较: A[9] = 41 < e

♦ lo = 10, hi = 18

5	10	12	14	19	23	29	36	39	41	44	51	54	59	72	79	82	86	92
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

插值: mi = 10 + (18 - 10)*(50 - 44)/(92 - 44) = 11

比较: A[11] = 51 > e

♦ lo = hi = 10

5	10	12	14	19	23	29	36	39	41	44	51	54	59	72	79	82	86	92
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

插值: mi = 10

比较: A[10] = 49 < e, 故返回: NOT_FOUND

♦ 从 $\mathcal{O}(log n)$ 到 $\mathcal{O}(log log n)$, 优势并不明显

(除非查找表极长, 或比较操作成本极高)

比如, $n = 2^{(2^5)} = 2^{32} = 4G$ 时

♦ 实际可行的方法

- 首先通过插值查找

迅速将查找范围缩小到一定的尺度

- 然后再进行二分查找

进一步缩小范围

- 最后 (当数据项只有200~300时)

使用顺序查找

♦ 须引入乘法、除法运算

♦ 易受小扰动的干扰和“蒙骗”

♦ 最坏: $hi - lo = \mathcal{O}(n)$

//具体实例?

♦ 平均: 每经一次比较, 待查找区间宽度由 n 缩至 \sqrt{n} // [Yao76, PIA78], 习题解析 [2-24]

$$n \rightarrow \sqrt{n} \rightarrow \sqrt{\sqrt{n}} \rightarrow \sqrt{\sqrt{\sqrt{n}}} \rightarrow \dots \rightarrow 2$$

$\log n$

$$n \rightarrow n^{1/2^1} \rightarrow n^{1/2^2} \rightarrow n^{1/2^3} \rightarrow \dots \rightarrow 2$$

$\mathcal{O}(\log \log n)$

$\log(n/2) = \log n - 1$

♦ 每经一次比较

待查找区间宽度的数值 n 开方, 有效字长 $\log n$ 减半

- 差值查找 = 在字长意义上的折半查找
- 二分查找 = 在字长意义上的顺序查找

$\log(n^{1/(2^k)}) = 0.5 \cdot \log n$

θ2-E

♦ template <typename T> void Vector<T>::sort(Rank lo, Rank hi) {

switch (rand() % 6) {

case 1: bubbleSort(lo, hi); break; //起泡排序

case 2: selectionSort(lo, hi); break; //选择排序 (习题)

case 3: mergeSort(lo, hi); break; //归并排序

case 4: heapSort(lo, hi); break; //堆排序 (第12章)

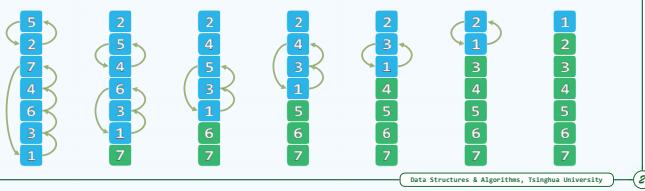
case 5: quickSort(lo, hi); break; //快速排序 (第14章)

default: shellSort(lo, hi); break; //希尔排序 (第14章)

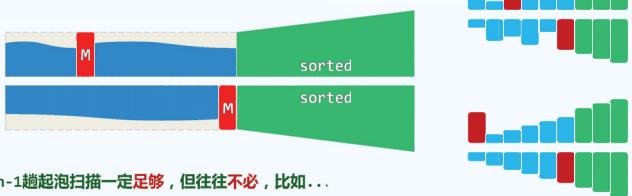
} //随机选择算法以充分测试。实用时可视具体问题的特点灵活确定或扩充

} //向量区间[lo, hi)排序

- 问题：给定n个可比较的元素，将它们按（非降）序排列
- 观察：有序/无序序列中，任何/总有一对相邻元素顺序/逆序
- 扫描交换：依次比较每一对相邻元素；如有必要，交换之
- 若整趟扫描都没有进行交换，则排序完成；否则，再做一趟扫描交换



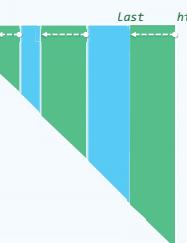
- 不变性：经k趟扫描交换后，最大的k个元素必然就位
- 单调性：经k趟扫描交换后，问题规模缩减至n-k
- 正确性：经至多n趟扫描后，算法必然终止，且能给出正确解答



n-1趟起泡扫描一定足够，但往往不必，比如...

[hi]就位后，[lo, hi]可能已经有序 (sorted)——此时，应该可以...

```
template <typename T> void Vector<T>::bubbleSort( Rank lo, Rank hi ) {
    for( Rank last = --hi; lo < hi; hi = last )
        for( Rank i = last = lo; i < hi; i++ )
            if( _elem[i] > _elem[i + 1] ) {
                swap( _elem[i], _elem[i+1] );
                last = i; //逆序对只可能残留在[lo, last)
            }
    }
    if A[lo, last] <= A[last, hi]
    A[lo, last] < A(last, hi)
}
```



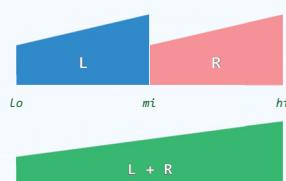
曾随逝水，岂必委芳尘
万缕千丝终不改，任他随聚随分

归并排序：分而治之

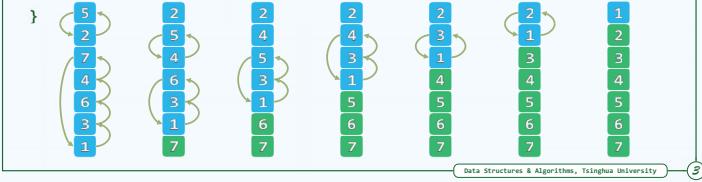
$\theta^2 - F_1$

邓俊辉
deng@tsinghua.edu.cn

```
template <typename T> void Vector<T>::mergeSort( Rank lo, Rank hi ) {
    if( hi - lo < 2 ) return; //单元素区间自然有序，否则...
    int mi = (lo + hi) >> 1; //以中点为界
    mergeSort( lo, mi ); //对前半段排序
    mergeSort( mi, hi ); //对后半段排序
    merge( lo, mi, hi ); //归并
}
```



```
template <typename T> Vector<T>::bubbleSort( Rank lo, Rank hi ) {
    while( lo < --hi ) //逐趟起泡扫描 (输入保证: 0 <= lo < hi <= size)
        for( Rank i = lo; i < hi; i++ ) //若相邻元素
            if( _elem[i] > _elem[i + 1] ) //逆序
                swap( _elem[i], _elem[i + 1] ); //则交换
}
```



```
template <typename T> void Vector<T>::bubbleSort( Rank lo, Rank hi ) {
    for( bool sorted = false; sorted = !sorted; )
        for( Rank i = lo; i < hi - 1; i++ )
            if( _elem[i] > _elem[i + 1] ) {
                swap( _elem[i], _elem[i+1] );
                sorted = false; //仍未完全有序
            }
    } //else ... 提前终止
}
```

有改进，但仍有继续改进的余地，比如...

[hi]就位后，尽管[lo, hi]未必有序，但某后缀[last, hi]可能有序——此时，应该可以...

时间效率：最好 $\Theta(n)$ ，最坏 $\Theta(n^2)$

输入含重复元素时，算法的稳定性 (stability) 是更为细致的要求

重复元素在输入、输出序列中的相对次序，是否保持不变？

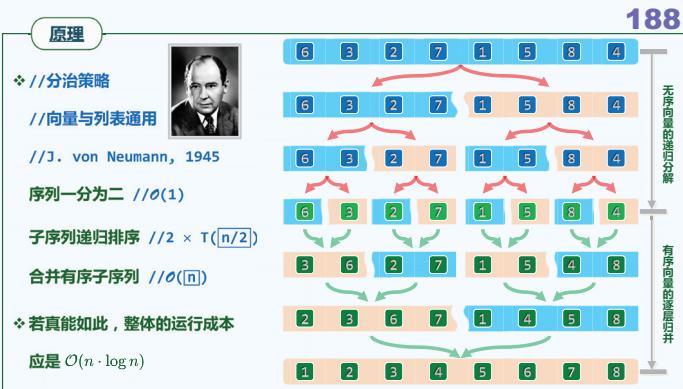
- 输入： 6, 7a, 3, 2, 7b, 1, 5, 8, 7c, 4
- 输出： 1, 2, 3, 4, 5, 6, 7a, 7b, 7c, 8 //stable
- 1, 2, 3, 4, 5, 6, 7a, 7c, 7b, 8 //unstable

以上起泡排序算法是稳定的吗？是的！

在起泡排序中，元素a和b的相对位置发生变化，只有一种可能：

- 经分别与其它元素的交换，二者相互接近直至相邻
- 在接下来一轮扫描交换中，二者因逆序而交换位置

在if一句的判断条件中，若把">"换成" \geq "，将有何变化？



若真能如此，整体的运行成本

应是 $\mathcal{O}(n \cdot \log n)$

归并排序：二路归并

$\theta^2 - F_2$

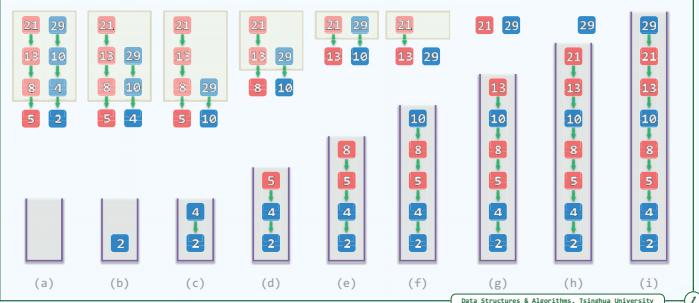
邓俊辉
deng@tsinghua.edu.cn

天下大势，分久必合，合久必分

二路归并

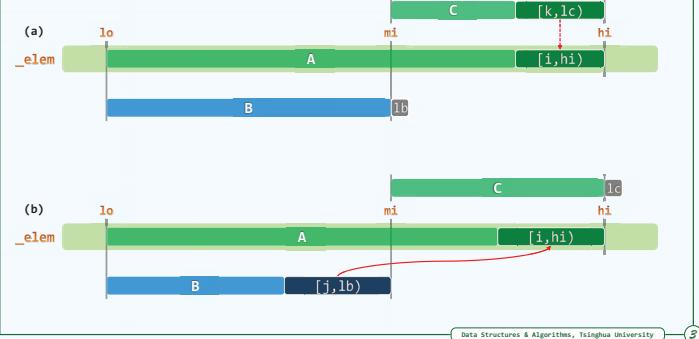
191

❖ 2-way merge : 有序序列，合二为一： $S[lo, hi] = S[lo, mi] + S[mi, hi]$



正确性

193



❖ 算法的运行时间主要消耗于for循环，共有两个控制变量

- 初始 : $j = 0, k = 0$
- 最终 : $j = lb, k = lc$
- 亦即 : $j + k = lb + lc = hi - lo = n$

❖ 观察：每经过一次迭代，j和k中至少有一个会加一 ($j+k$ 也必至少加一)

❖ 故知：`merge()` 总体迭代不过 $\mathcal{O}(n)$ 次，累计只需线性时间！

❖ 这一结论与排序算法的 $\Omega(n \log n)$ 下界并不矛盾——毕竟这里的B和C均已各自有序

❖ 注意：待归并子序列不必等长，允许： $lb \neq lc, mi \neq (lo + hi)/2$

❖ 实际上，这一算法及结论也适用于另一类序列——列表（下一章）

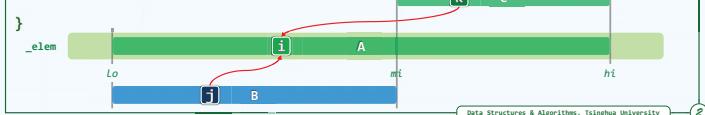
Data Structures & Algorithms, Tsinghua University

实现

192

```
template <typename T> void Vector<T>::merge( Rank lo, Rank mi, Rank hi ) {
```

```
    T* A = _elem + lo; int lb = mi - lo; T* B = new T[lb]; //A[0, hi - lo] = _elem[lo, mi - lo]
    for ( Rank i = 0; i < lb; i++ ) B[i] = A[i]; //复制前子向量B[0, lb] = _elem[lo, mi - lo]
    int lc = hi - mi; T* C = _elem + mi; //后子向量C[0, lc] = _elem[mi, hi]
    for ( Rank i = 0, j = 0, k = 0; j < lb; ) //归并：反复从B和C首元素中取出更小者
        A[i++] = ( lc <= k || B[j] <= C[k] ) ? B[j++] : C[k++];
    delete [] B; //释放临时空间
}
```



194

向量

归并排序：复杂度

$\theta 2-F3$

邓伟辉
deng@tsinghua.edu.cn

I think there is a world market for about five computers.

- T. J. Watson, 1943

195

综合评价

优点

- 实现最坏情况下最优 $\mathcal{O}(n \log n)$ 性能的第一个排序算法
- 不需随机读写，完全顺序访问——尤其适用于列表之类的序列、磁带之类的设备
- 只要实现恰当，可保证稳定——出现雷同元素时，左侧子向量优先
- 可扩展性极佳，十分适宜于外部排序——海量网页搜索结果的归并
- 易于并行化

缺点

- 非就地，需要对等规模的辅助空间——可否更加节省？
- 即便输入完全（或接近）有序，仍需 $\mathcal{O}(n \log n)$ 时间——如何改进？

196

198

整数集

$k \in S$? `bool test(int k);`

$S \cup \{k\}$ `void set(int k);`

$S \setminus \{k\}$ `void clear(int k);`



Data Structures & Algorithms, Tsinghua University

199

实现

```
*class Bitmap {
```

```
private:
```

```
    int N;
```

```
    char * M;
```

```
public:
```

```
    Bitmap( int n = 8 ) { M = new char[ N = (n+7)/8 ]; memset( M, 0, N ); }
```

```
    ~Bitmap() { delete [] M; M = NULL; }
```

```
    void set( int k ); void clear( int k ); bool test( int k );
```

}

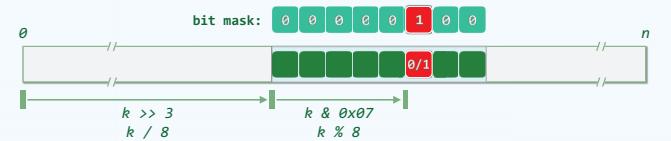
Data Structures & Algorithms, Tsinghua University

200

```
*bool test( int k ) { return M[ k >> 3 ] & ( 0x80 >> ( k & 0x07 ) ); }
```

```
*void set( int k ) { expand( k ); M[ k >> 3 ] |= ( 0x80 >> ( k & 0x07 ) ); }
```

```
*void clear( int k ) { expand( k ); M[ k >> 3 ] &= ~( 0x80 >> ( k & 0x07 ) ); }
```



Data Structures & Algorithms, Tsinghua University

θ2-XA2

向量
位图：典型应用

邓俊辉
deng@tsinghua.edu.cn

Those too big to pass through are our friends.

小集合 + 大数据

```
* Bitmap B( m ); //O(m)
for (int i = 0; i < n; i++)
    B.set( A[i] ); //O(n)
for (int k = 0; k < m; k++)
    if (B.test( k ) )
        /* ... */ //O(m)
```

◆ 拓展：搜索引擎的使用规律亦是如此
词表规模不大，但重复度极高
——如何从中剔除重复的索引词？

◆ 总体运行时间 = $O(n+m) = O(n)$
◆ 空间 = $O(m)$
- 就上例而言，降至：
 $m/8 = 2^{21} = 2\text{MB} \ll 40\text{GB}$
- 即便 $m = 2^{32}$ ，也不过：
 $2^{29} = 0.5\text{GB}$
◆ 关键在于
如何将查询词表转换为某一集合
——留作习题

Data Structures & Algorithms, Tsinghua University

筛法：实现

```
void Eratosthenes( int n, char * file ) {
    Bitmap B( n );
    B.set( 0 ); B.set( 1 );
    for ( int i = 2; i < n; i++ )
        if ( ! B.test( i ) )
            for ( int j = 2*i; j < n; j += i )
                B.set( j );
    B.dump( file );
}
```



Data Structures & Algorithms, Tsinghua University

◆ 不计内循环，外循环自身每次仅一次加法、两次判断，累计 $O(n)$
◆ 内循环每趟迭代 $O(n/i)$ 步，由素数定理至多 $n/\ln n$ 趟，累计耗时不过

$$\frac{n}{2} + \frac{n}{3} + \frac{n}{5} + \frac{n}{7} + \dots + \frac{n}{n/\ln n} < n/2 + n/3 + n/4 + n/5 + n/6 + \dots + n/(n/\ln n) = O(n \cdot (\ln(n/\ln n) - 1)) = O(n \cdot \ln n - n \cdot \ln(\ln(n))) = O(n \cdot \log n)$$

◆ 循环起点 “ $i+i$ ” 可改作 “ $i*i$ ” //为什么？
◆ 如此，内循环的长度将由 $O(n/i)$ 降至 $O(\max(1, n/i - i))$ //从渐进的角度看，是否实质的改进？
◆ 基于以上，如何实现 primeNLT(int low)？



Data Structures & Algorithms, Tsinghua University

$O(n) \sim O(1)$

◆ Bitmap 的构造函数中，通过 `memset(M, 0, N)` 统一清零
这一步只需 $O(1)$ 时间？不，实际上仍等效于诸位清零， $O(N) = O(n)$ ！
◆ 尽管这并不会影响上例的渐进复杂度，但并非所有问题都是如此
◆ 有时，对于大规模的散列表，初始化的效率直接影响到实际性能
例如：第 11 章中 `b2c[]` 表的构造算法，需要 $O(|\Sigma|+m) = O(s+m)$ 时间
若能省去 `b2c[]` 表各项的初始化，则可严格地保证是 $O(m)$
◆ 有时，甚至会影响到算法的整体渐进复杂度
例如，从 $n=10^8$ 个 32 位整数中找出重复者，可构造剔除算法... //但这里无需回收
因此，若能省去 Bitmap 的初始化，则只需 $O(n)$ 时间

Data Structures & Algorithms, Tsinghua University

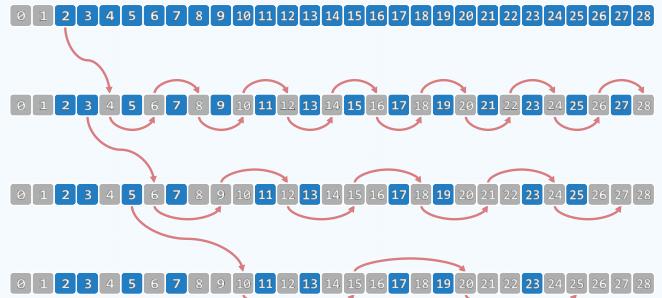
小集合 + 大数据

老问题

- int A[n] 的元素均取自 [0, m)
- 如何剔除其中的重复者？
◆ 比如
 $2^{24} = m \ll n = 10^{10}$
亦即，10,000,000,000 个 24 位无符号整数
◆ 如果采用内部排序算法
先排序，再扫描
 $\sim O(n \log n + n)$ —— 毫无压力
◆ 新特点：数据量虽大，但重复度极高
- 想想我们电脑里的 mp3、mp4
- 还有，朋友圈 ...
◆ 那么
 $m \ll n$ 的条件，又应如何加以利用？

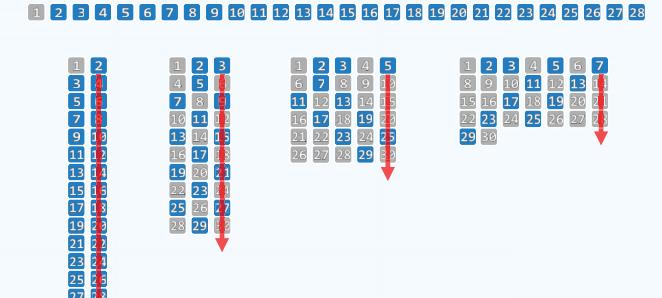
Data Structures & Algorithms, Tsinghua University

筛法：思路



Data Structures & Algorithms, Tsinghua University

筛法：效果



Data Structures & Algorithms, Tsinghua University

效率与改进

◆ 不计内循环，外循环自身每次仅一次加法、两次判断，累计 $O(n)$
◆ 内循环每趟迭代 $O(n/i)$ 步，由素数定理至多 $n/\ln n$ 趟，累计耗时不过

$$\frac{n}{2} + \frac{n}{3} + \frac{n}{5} + \frac{n}{7} + \dots + \frac{n}{n/\ln n} < n/2 + n/3 + n/4 + n/5 + n/6 + \dots + n/(n/\ln n) = O(n \cdot (\ln(n/\ln n) - 1)) = O(n \cdot \ln n - n \cdot \ln(\ln(n))) = O(n \cdot \log n)$$

◆ 循环起点 “ $i+i$ ” 可改作 “ $i*i$ ” //为什么？
◆ 如此，内循环的长度将由 $O(n/i)$ 降至 $O(\max(1, n/i - i))$ //从渐进的角度看，是否实质的改进？
◆ 基于以上，如何实现 primeNLT(int low)？



Data Structures & Algorithms, Tsinghua University

向量

位图：快速初始化

从那天看见他我心里就放不下呀
因此上我偷偷地就爱上他呀
但愿这个年轻的人哪也把我爱呀
过了门，他劳动，他生产，又织布，纺棉花
我们学文化，他帮助我，我帮助他
争一对模范夫妻立业成家呀

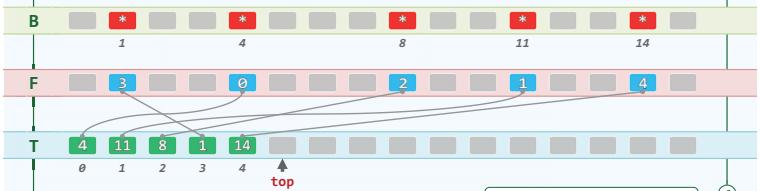
$O(n) \sim O(1)$

◆ Bitmap 的构造函数中，通过 `memset(M, 0, N)` 统一清零
这一步只需 $O(1)$ 时间？不，实际上仍等效于诸位清零， $O(N) = O(n)$ ！
◆ 尽管这并不会影响上例的渐进复杂度，但并非所有问题都是如此
◆ 有时，对于大规模的散列表，初始化的效率直接影响到实际性能
例如：第 11 章中 `b2c[]` 表的构造算法，需要 $O(|\Sigma|+m) = O(s+m)$ 时间
若能省去 `b2c[]` 表各项的初始化，则可严格地保证是 $O(m)$
◆ 有时，甚至会影响到算法的整体渐进复杂度
例如，从 $n=10^8$ 个 32 位整数中找出重复者，可构造剔除算法... //但这里无需回收
因此，若能省去 Bitmap 的初始化，则只需 $O(n)$ 时间

Data Structures & Algorithms, Tsinghua University

结构：校验环

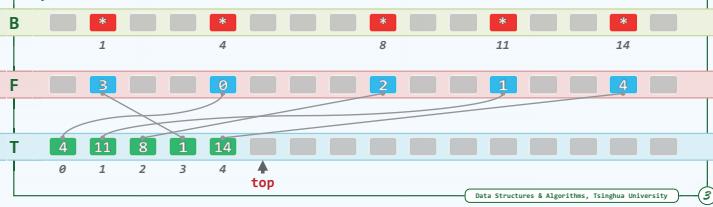
◆ // [J. Hopcroft, 1974] 将 $B[m]$ 拆分为一对等长的 Rank 型向量
// 有效位须满足: $T[F[k]] = k, F[T[k]] = k$ // From
Rank F[m]; //From
Rank T[m]; Rank top = 0; // To 及其栈顶指示



判断

211

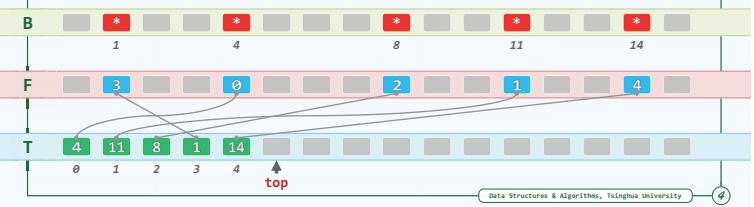
```
* bool Bitmap::test( Rank k ) {
    return ( 0 <= F[ k ] ) && ( F[ k ] < top )
        && ( k == T[ F[ k ] ] );
}
```



复位

212

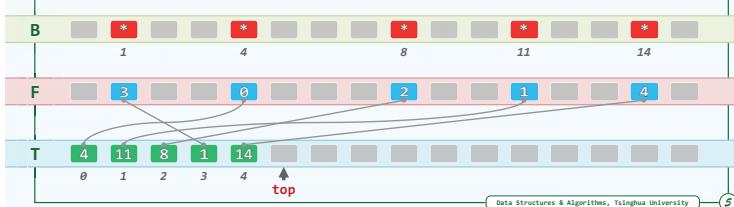
```
* void Bitmap::reset() { top = 0; }
```



插入

213

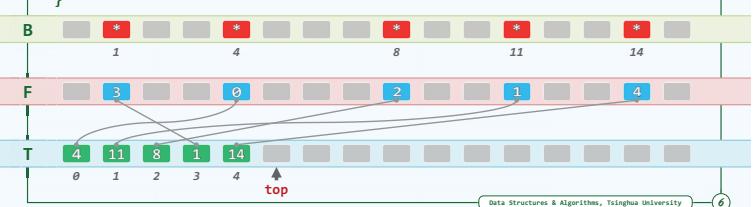
```
* void Bitmap::set( Rank k ) {
    if ( ! test( k ) ) { T[ top ] = k; F[ k ] = top++; }
}
```



删除

214

```
* void Bitmap::clear( Rank k ) {
    if ( test( k ) && ( --top ) )
        { F[ T[ top ] ] = F[ k ]; T[ F[ k ] ] = T[ top ]; }
}
```



215

列表

循位置访问

θ3-A

邓俊辉
deng@tsinghua.edu.cnDon't lose the link.
- Robin Milner

从静态到动态

根据是否修改数据结构，所有操作大致分为两类方式

- 静态：仅读取，数据结构的内容及组成一般不变：get、search
- 动态：需写入，数据结构的局部或整体将改变：insert、remove

与操作方式相对应地，数据元素的存储与组织方式也分为两种

- 静态：数据空间整体创建或销毁
数据元素的物理存储次序与其逻辑次序严格一致；可支持高效的静态操作
比如向量，元素的物理地址与其逻辑次序线性对应
- 动态：为各数据元素动态地分配和回收的物理空间
相邻元素记录彼此的物理地址，在逻辑上形成一个整体；可支持高效的动态操作

216

从向量到列表

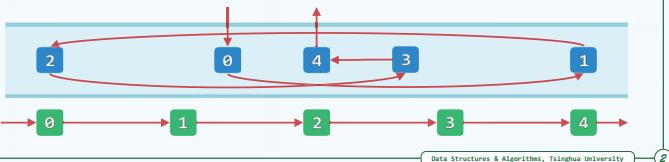
217

◆ 列表（list）是采用动态储存策略的典型结构

- 其中的元素称作节点（node），通过指针或引用彼此联接
- 在逻辑上构成一个线性序列： $L = \{ a_0, a_1, \dots, a_{n-1} \}$

◆ 相邻节点彼此互称前驱（predecessor）或后继（successor）

- 没有前驱/后继的节点称作首（first/front）/末（last/rear）节点



从秩到位置

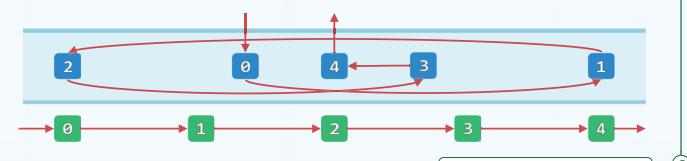
218

◆ 向量支持循秩访问（call-by-rank）：根据元素的秩，可在 $O(1)$ 时间内直接确定其物理地址

A[] 0 1 2 3 n-1 n

◆ 这种高效的方式，可否被列表沿用？

◆ 比如，从头/尾端出发，沿后继/前驱引用... //List::operator[](Rank r)，下节详解



从秩到位置

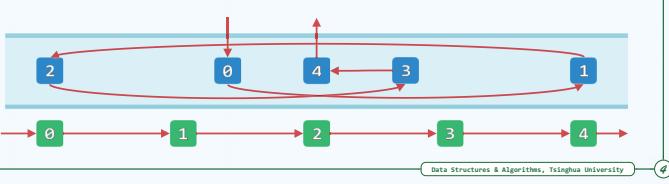
219

◆ 然而，此时的循秩访问成本过高，已不合时宜

◆ 因此，应改用循位置访问（call-by-position）的方式

亦即，转而利用节点之间的相互引用，找到特定的节点

◆ 比喻：找到我的朋友A的亲戚B的同事C的战友D的...的同学Z



220

列表

接口与实现

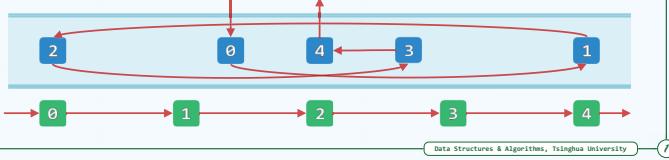
θ3-B

邓俊辉
deng@tsinghua.edu.cn百只骆驼绕山走，九十八只在山后
尾驼羞尾不见头，头驼露头出山沟

列表节点 : ADT接口

◆ 作为列表的基本元素，列表节点首先需要独立地“封装”实现

操作接口	功能
pred()	当前节点前驱节点的位置
succ()	当前节点后继节点的位置
data()	当前节点所存数据对象
insertAsPred(e)	插入前驱节点，存入被引用对象e，返回新节点位置
insertAsSucc(e)	插入后继节点，存入被引用对象e，返回新节点位置



Data Structures & Algorithms, Tsinghua University

列表节点 : 模板类

```
#define Posi(T) ListNode<T>* //列表节点位置 ( ISO C++.0x, template alias )
template <typename T> class ListNode { //列表节点模板类 (以双向链表形式实现)
    T data; //数据
    Posi(T) pred; //前驱
    Posi(T) succ; //后继
    ListNode() {} //针对header和trailer的构造
    ListNode(T e, Posi(T) p = NULL, Posi(T) s = NULL)
        : data(e), pred(p), succ(s) {} //默认构造器
    Posi(T) insertAsPred(T const& e); //前插入
    Posi(T) insertAsSucc(T const& e); //后插入
};
```



Data Structures & Algorithms, Tsinghua University

列表 : ADT接口

操作接口	功能	适用对象
size()	报告列表当前的规模 (节点总数)	列表
first(), last()	返回首、末节点的位置	列表
insertAsFirst(e), insertAsLast(e)	将e当作首、末节点插入	列表
insertA(p, e), insertB(p, e)	将e当作节点p的直接后继、前驱插入	列表
remove(p)	删除位置p处的节点，返回其引用	列表
disordered()	判断所有节点是否已按非降序排列	列表
sort()	调整各节点的位置，使之按非降序排列	列表
find(e)	查找目标元素e，失败时返回NULL	列表
search(e)	查找e，返回不大于e且秩最大的节点	有序列表
deduplicate(), unify()	剔除重复节点	列表/有序列表
traverse()	遍历列表	列表

Data Structures & Algorithms, Tsinghua University

列表 : 模板类

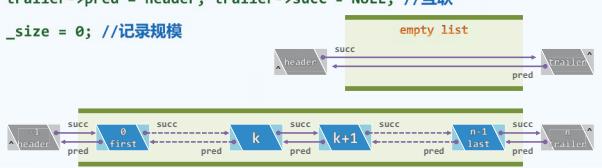
```
#include "listNode.h" //引入列表节点类
template <typename T> class List { //列表模板类
private: int _size; //规模
    Posi(T) header; Posi(T) trailer; //哨兵
protected: /* ... 内部函数 */
public: /* ... 构造函数、析构函数、只读接口、可写接口、遍历接口 */
};
```



◆ 等效地，头、首、末、尾节点的秩可分别理解为-1、0、n-1、n

构造

```
template <typename T> void List<T>::init() { //初始化，创建列表对象时统一调用
    header = new ListNode<T>; //创建头哨兵节点
    trailer = new ListNode<T>; //创建尾哨兵节点
    header->succ = trailer; header->pred = NULL; //互联
    trailer->pred = header; trailer->succ = NULL; //互联
    _size = 0; //记录规模
}
```



Data Structures & Algorithms, Tsinghua University

循环访问

◆ 通过重载下标操作符，可模仿向量的循环访问方式

```
template <typename T> //assert: 0 <= r < size
List<T>::operator[]( Rank r ) const { //O(r)效率低下，可偶尔为之，却不宜常用
    Posi(T) p = first(); //从首节点出发
    while ( 0 < r-- ) p = p->succ; //顺数第r个节点即是
    return p->data; //目标节点
} //秩 == 前驱的总数
```

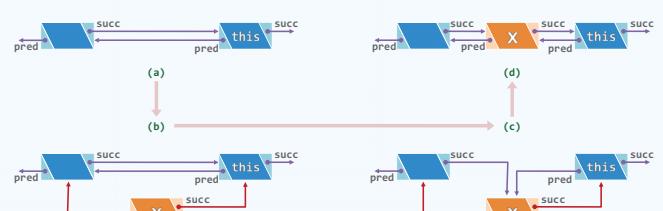


◆ 时间复杂度为O(r)

均匀分布时，期望复杂度为 $(1 + 2 + 3 + \dots + n)/n = O(n)$

插入 : 思路 + 过程

```
template <typename T> Posi(T) List<T>::insertB( Posi(T) p, T const & e )
{ _size++; return p->insertAsPred( e ); } //e当作p的前驱插入 ( Before )
```

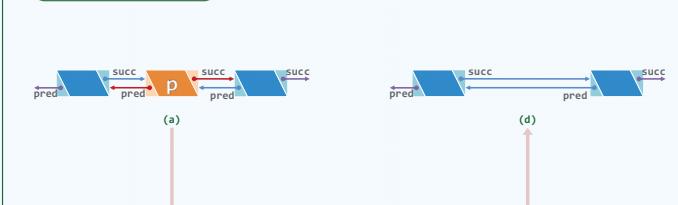


插入 : 实现

```
template <typename T> //前插入算法 ( 后插入算法完全对称 )
Posi(T) ListNode<T>::insertAsPred( T const & e ) { //O(1)
    Posi(T) x = new ListNode<T>( e, pred, this ); //创建 ( 耗时100倍 )
    pred->succ = x; pred = x; //次序不可颠倒
    return x; //建立链接，返回新节点的位置
} //得益于哨兵，即便this为首节点亦不必特殊处理——此时等效于insertAsFirst(e)
```



删除 : 思路 + 过程



Data Structures & Algorithms, Tsinghua University

Data Structures & Algorithms, Tsinghua University

删除：实现

```
* template <typename T> //删除合法位置p处节点，返回其数值
T List<T>::remove( Posi(T) p ) { //O(1)
    T e = p->data; //备份待删除节点数值（设类型T可直接赋值）
    p->pred->succ = p->succ;
    p->succ->pred = p->pred;
    delete p; _size--; return e; //返回备份数值
}
```



```
* template <typename T> void List<T>::copyNodes( Posi(T) p, int n ) { //O(n)
    init(); //创建头、尾哨兵节点并做初始化
    while (n--) { //将起自p的n项依次作为末节点
        insertAsLast( p->data ); //插入
        p = p->succ;
    }
    List<T>::List( List<T> const & L ) { copyNodes( L.first(), L._size ); }
}
```

Data Structures & Algorithms, Tsinghua University

列表

无序列表：查找与去重

θ3-C3

邓俊辉
deng@tsinghua.edu.cn

顧長康啜甘蔗，先食尾。問所以，云：“漸至佳境。”

有些事，你一辈子总也忘不掉。凡是让你揪心的事，在你身上，都会发生两次。或两次以上。

去重

```
* template <typename T> int List<T>::deduplicate() {
    int oldSize = _size;
    ListNodePosi(T) p = first(); ListNodePosi(T) q = NULL;
    for ( Rank r = 0; p != trailer; p = p->succ, q = find( p->data, r, p ) )
        q ? remove( q ): r++; //r为无重前缀的长度
    return oldSize - _size; //即被删除元素总数
} //正确性及效率分析的方法与结论，与Vector::deduplicate()相同
```

Data Structures & Algorithms, Tsinghua University

```
* template <typename T>
void List<T>::traverse( void ( * visit )( T & ) ) //函数指针
{ Posi(T) p = header; while ( ( p = p->succ ) != trailer ) visit( p->data ); }

* template <typename T> template <typename VST>
void List<T>::traverse( VST & visit ) //函数对象
{ Posi(T) p = header; while ( ( p = p->succ ) != trailer ) visit( p->data ); }
```

②

列表

无序列表：构造与析构

θ3-C2

邓俊辉
deng@tsinghua.edu.cn

A scientist discovers that which exists, an engineer creates that which never was.

“宇宙里有生有死……爱情里也有死有生。”

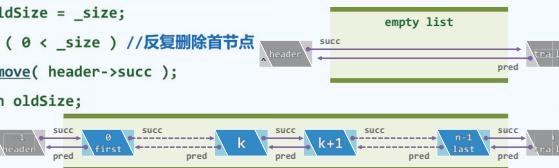
“这是什么意思？”剑云低声说，没有人回答他。

* template <typename T> List<T>::~List() //列表析构

{ clear(); delete header; delete trailer; } //清空列表，释放头、尾哨兵节点

* template <typename T> int List<T>::clear() { //清空列表, O(n)

```
int oldSize = _size;
while ( 0 < _size ) //反复删除首节点
    remove( header->succ );
return oldSize;
```



* 若remove(header->succ)改作remove(trailer->pred)呢？

Data Structures & Algorithms, Tsinghua University

查找

* template <typename T> //0 ≤ n ≤ rank(p) < _size

Posi(T) List<T>::find(T const & e, int n, Posi(T) p) const { //O(n)

while (0 < n--) //自后向前逐个对比

if (e == (p = p->pred) ->data) //假定类型T已重载 “==”

return p; //在p的n个前驱中，等于e的最靠后者

return NULL; //失败



* Posi(T) find(T const & e) const { return find(e, _size, trailer); }

Data Structures & Algorithms, Tsinghua University

列表

无序列表：遍历

θ3-C4

邓俊辉
deng@tsinghua.edu.cn

八戒道：“不要扯，等我一家家吃将来。”

列表

有序列表：唯一化

θ3-D1

邓俊辉
deng@tsinghua.edu.cn

昨夜
我梦见
你和我
说同一个字

Most people are other people. Their thoughts are someone else's opinions, their lives a mimicry, their passions a quotation.

```
* template <typename T> int List<T>::uniquify() { //剔除重复元素
    if (_size < 2) return 0; //平凡列表自然无重复
    int oldSize = _size; //记录原规模
    ListNodePosi(T) p = first(); ListNodePosi(T) q; //各区段起点及其直接后继
    while ( trailer != ( q = p->succ ) ) //反复考查紧邻的节点对(p,q)
        if ( p->data != q->data ) p = q; //若互异，则转向下一对
        else remove(q); //否则(雷同)，删除后者
    return oldSize - _size; //规模变化量，即被删除元素总数
} //只需遍历整个列表一趟，O(n)
```

Data Structures & Algorithms, Tsinghua University

在有序列表内节点p的n个(真)前驱中，找到不大于e的最靠后者

```
* template <typename T> // assert: 0 <= n <= rank(p) < _size
Posi(T) List<T>::search( T const & e, int n, Posi(T) p ) const {
    do { p = p->pred; n--; } //从右向左
    while ( ( -1 < n ) && ( e < p->data ) ); //逐个比较，直至命中或越界
    return p;
}
```

失败时，返回区间左边界的前驱(可能是header)

最好O(1)，最坏O(n)；等概率时平均O(n)，正比于区间宽度

Data Structures & Algorithms, Tsinghua University

* template <typename T>

```
Posi(T) List<T>::search( T const & e, int n, Posi(T) p ) const ;
```

语义与向量相似，便于插入排序等后续操作：

```
insertA( search( e, r, p ), e )
```

为何未能借助有序性提高查找效率？实现不当，还是根本不可能？

按照循位置访问的方式，物理存储地址与其逻辑次序无关

依据秩的随机访问无法高效实现，而只能依据元素间的引用顺序访问

邓俊辉
deng@tsinghua.edu.cn

列表

选择排序

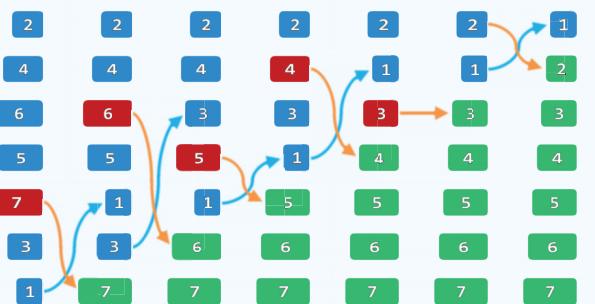
θ3-E

天下只有两种人。譬如一串葡萄到手，一种人挑最好的先吃，另一种人把最好的留在最后吃。

当下又选了几样果菜与凤姐送去，凤姐儿也送了几样来。

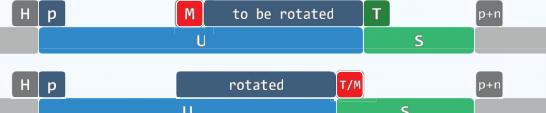
邓俊辉
deng@tsinghua.edu.cn

交换法

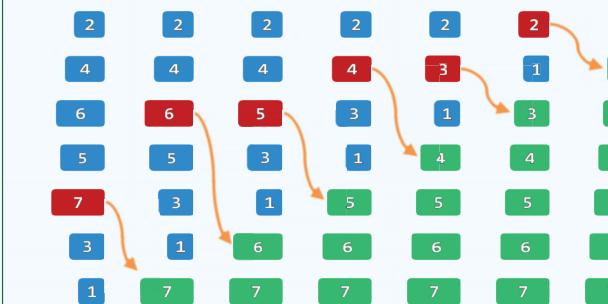


Data Structures & Algorithms, Tsinghua University

```
* 对列表中起始于位置p的连续n个元素做选择排序，valid(p) && rank(p) + n <= size
template <typename T> void List<T>::selectionSort( Posi(T) p, int n ) {
    Posi(T) head = p->pred; Posi(T) tail = p; //待排序区间(head, tail)
    for ( int i = 0; i < n; i++ ) tail = tail->succ; //head/tail可能是头/尾哨兵
    while ( 1 < n ) { //反复从(非平凡)待排序区间内找出最大者，并移至有序区间前端
        insertB( tail, remove( selectMax( head->succ, n ) ) ); //改进...
        tail = tail->pred; n--; //待排序区间、有序区间的范围，均同步更新
    }
}
```



平移法



Data Structures & Algorithms, Tsinghua University

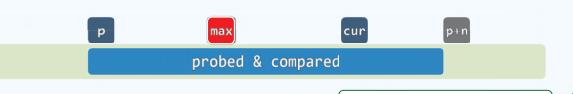
selectionSort()

```
* 对列表中起始于位置p的连续n个元素做选择排序，valid(p) && rank(p) + n <= size
template <typename T> void List<T>::selectionSort( Posi(T) p, int n ) {
    Posi(T) head = p->pred; Posi(T) tail = p; //待排序区间(head, tail)
    for ( int i = 0; i < n; i++ ) tail = tail->succ; //head/tail可能是头/尾哨兵
    while ( 1 < n ) { //反复从(非平凡)待排序区间内找出最大者，并移至有序区间前端
        insertB( tail, remove( selectMax( head->succ, n ) ) ); //改进...
        tail = tail->pred; n--; //待排序区间、有序区间的范围，均同步更新
    }
}
```

Data Structures & Algorithms, Tsinghua University

selectMax()

```
* template <typename T> //从起始于位置p的n个元素中选出最大者，1 < n
Posi(T) List<T>::selectMax( Posi(T) p, int n ) { //O(n)
    Posi(T) max = p; //最大者暂定为p
    for ( Posi(T) cur = p; 1 < n; n-- ) //后续节点逐一与max比较
        if ( ! lt( ( cur = cur->succ )->data, max->data ) ) //data ≥ max
            max = cur; //则更新最大元素位置记录
    return max; //返回最大节点位置
}
```



列表

有序列表：查找

θ3-D2

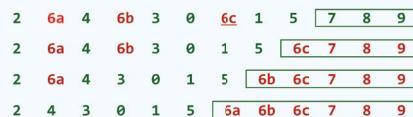
种种念起，无不出于找。离了现前，向外去找，根本让我们直觉麻木的这个找。

于是，他们急忙把自己的布袋卸在地上，各人打开自己的布袋。管家就搜查，从最大的开始，查到最小的。那怀亮在便雅悯的布袋里搜出来了

邓俊辉
deng@tsinghua.edu.cn

Data Structures & Algorithms, Tsinghua University

- ◆ 有多个重复元素同时命中时，往往需要按照某种附加的约定，返回其中特定的某一个
- ◆ 比如，通常都约定“靠后者优先返回”
- ◆ 为此，必须采用比较器`lt()`或`ge()`，即等效于后者优先



◆ 如此即可保证，重复元素在列表中的相对次序，与其插入次序一致



Data Structures & Algorithms, Tsinghua University 6



.....莫非你吃了我吩咐你不可吃的那树上的果子吗？”

“你所赐给我、与我同居的女人，她把那树上的果子给我，我就吃了。”

“你作的是什么事呢？”

“那蛇引诱我，我就吃了。”

列表

循环节

邓俊辉
deng@tsinghua.edu.cn

实例

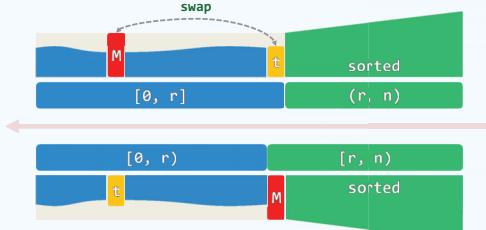
rank:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A[]:	J	N	P	M	A	I	G	O	D	C	H	B	K	L	F	E
S[]:	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
r[]:	9	13	15	12	0	8	6	14	3	2	7	1	10	11	5	4
	J . P . A . . . C E	. N B . L M . I . O D . H . K . F G												

Data Structures & Algorithms, Tsinghua University 2

255

无效的交换

◆ M已经就位，无需交换——这种情况会出现几次？



◆ 有c个循环节，就出现c-1次——最大值为n，期望 $\Theta(n \log n)$

Data Structures & Algorithms, Tsinghua University 3

257

性能分析

◆ 共迭代n次，在第k次迭代中

- `selectMax()` 为 $\Theta(n - k)$ //算术级数
- `swap()` 为 $\Theta(1)$ //或 `remove()` + `insertB()`
- 故总体复杂度应为 $\Theta(n^2)$

◆ 尽管如此，元素的移动操作远远少于起泡排序 //实际更为费时
也就是说， $\Theta(n^2)$ 主要来自于元素的比较操作 //成本相对更低

◆ 可否...每轮只做 $\Theta(n)$ 次比较，即找出当前的最大元素？

◆ 可以！...利用高级数据结构，`selectMax()` 可改进至 $\Theta(\log n)$ //稍后分解
当然，如此立即可以得到 $\Theta(n \log n)$ 的排序算法 //保持兴趣

Data Structures & Algorithms, Tsinghua University 7

254

Cycle

◆ 任何一个序列 $A[0, n]$ ，都可以分解为若干个循环节 //设元素之间可定义次序

◆ 任何一个序列 $A[0, n]$ ，都对应于一个有序序列 $S[0, n]$ //经排序之后

◆ 元素 $A[k]$ 在 S 中对应的秩，记作 $r(A[k]) = r(k) \in [0, n]$

◆ 元素 $A[k]$ 所属的循环节是：

$$A[k], A[r(k)], A[r(r(k))], A[r(r(r(k)))], \dots, A[r(\dots(r(r(k))\dots))] = A[k]$$

◆ 每个循环节，长度均不超过n

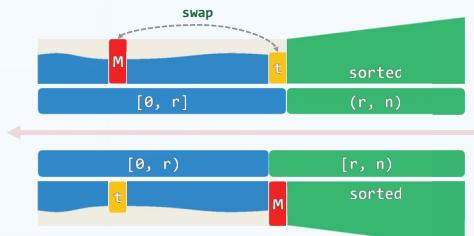
◆ 循环节之间，互不相交

Data Structures & Algorithms, Tsinghua University 7

255

单调性

◆ 采用交换法，每迭代一步，M都会脱离原属的循环节，自成一个循环节



◆ M所属循环节，长度恰好减少一个单位；其余循环节，保持不变

Data Structures & Algorithms, Tsinghua University 8

256



一语未了，只见宝玉笑嘻嘻的摘了一枝红梅进来，众丫鬟忙已接过，插入瓶内。

减而治之

259

◆ 不变性

序列总能视作两部分：

$$S[0, r] + U[r, n]$$

◆ 初始化

$$|S| = r = 0$$

◆ 反复地，针对 $e = A[r]$

在 S 中查找适当位置，以插入 e

◆ 二分查找？顺序查找？

实例

迭代轮次	前缀有序子序列	当前元素	后缀无序子序列
-1	^	^	5 2 7 4 6 3 1
0	^	5	2 7 4 6 3 1
1	(5)	2	7 4 6 3 1
2	(2) 5	7	4 6 3 1
3	2 5 (7)	4	6 3 1
4	2 (4) 5 7	6	3 1
5	2 4 5 (6) 7	3	1
6	2 (3) 4 5 6 7	1	^
7	(1) 2 3 4 5 6 7	^	^

Data Structures & Algorithms, Tsinghua University 9

260

```
//对列表中起始于位置p的连续n个元素做插入排序，valid(p) && rank(p) + n <= size
template <typename T> void List<T>::insertionSort( Posi(T) p, int n ) {
    for ( int r = 0; r < n; r++ ) { //逐一引入各节点，由Sr得到Sr+1
        insertA( search( p->data, r, p ), p->data ); //查找 + 插入
        p = p->succ; remove( p->pred ); //转向下一节点
    } //n次迭代，每次O(1)
} //仅使用O(1)辅助空间，属于就地算法

◆ 紧邻于search()接口返回的位置之后插入当前节点，总是保持有序

◆ 验证各种情况下的正确性，体会哨兵节点的作用：
    Sr中含有/不含与p相等的元素；Sr中的元素均严格小于/大于p
```

Data Structures & Algorithms, Tsinghua University

平均性能：后向分析

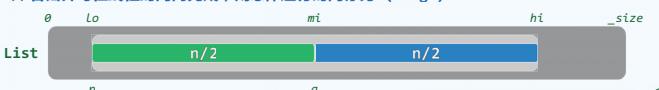
- ◆ 假定：各元素的取值系均匀独立分布
- 于是：平均要做多少次元素比较？
- ◆ 考查：e=[r]刚插入完成的那一时刻...
- 试问：此时的有序前缀[0,r]中，谁是e？
- ◆ 观察：其中的r+1个元素均有可能，且概率均为1/(r+1)
- ◆ 因此，在刚完成的这次迭代中，为引入S[r]所花费时间的数学期望为
$$[r + (r - 1) + \dots + 3 + 2 + 1 + 0] / (r + 1) + 1 = r/2 + 1$$
- ◆ 于是，总体时间为 $\sum_{r=0}^{n-1} r/2 + 1 = O(n^2)$
- ◆ 再问：在n次迭代中，平均有多少次无需交换呢？

//习题[3-10]

Data Structures & Algorithms, Tsinghua University

主算法

```
template <typename T> //valid(p) && rank(p) + n <= size
void List<T>::mergeSort( Posi(T) & p, int n ) { //对起始于位置p的n个元素排序
    if ( n < 2 ) return; //待排序范围足够小时直接返回，否则...
    Posi(T) q = p; int m = n >> 1; //以中点为界
    for ( int i = 0; i < m; i++ ) q = q->succ; //均分列表：O(m) = O(n)
    mergeSort( p, m ); mergeSort( q, n - m ); //子序列分别排序
    merge( p, m, *this, q, n - m ); //归并
} //若归并可在线性时间内完成，则总体运行时间亦为O(nlogn)
```

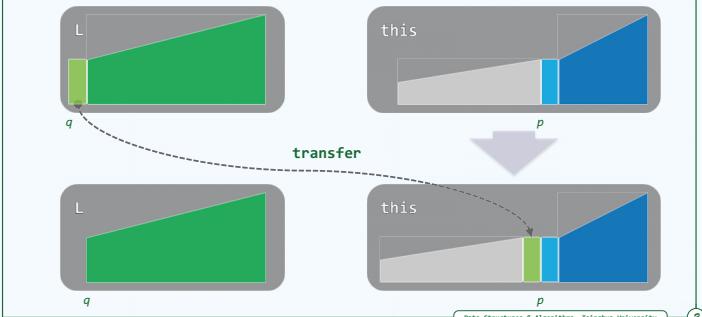


Data Structures & Algorithms, Tsinghua University

二路归并：实现

```
template <typename T> //自p起的n个元素，与L中自q起的m个元素归并（归并排序时L==this）
void List<T>::merge( Posi(T) & p, int n, List<T> & L, Posi(T) q, int m ) {
    ListNodePosi(T) pp = p->pred; //借助前驱（可能是header），以便返回前...
    while ( 0 < m ) //在q尚未移出区间之前
        if ( ( 0 < n ) && ( p->data <= q->data ) ) //若p仍在区间内且v(p) <= v(q)
            { if ( q == ( p = p->succ ) ) break; n--; } //则将p直接后移
        else //若p已超出右界或v(q) < v(p)，则将q插至p之前
            { insertB( p, L.remove( ( q = q->succ )->pred ) ); m--; }
        p = pp->succ; //确定归并后区间的（新）起点
} //运行时间O(n + m)，线性正比于节点总数
```

二路归并：算法



Inversion

- ◆ 考查序列A[0, n]，设元素之间可比较大大小
- (i, j) is called an inversion if $0 \leq i < j < n$ and $A[i] > A[j]$
- ◆ 为便于统计，可将逆序对统一记到**后者的**账上
- $I(j) = \{ 0 \leq i < j \mid A[i] > A[j] \text{ and hence } (i, j) \text{ is an inversion} \}$
- ◆ 例：A[] = { 5, 3, 1, 4, 2 } 中，共有 $0 + 1 + 2 + 1 + 3 = 7$ 个逆序对
- A[] = { 1, 2, 3, 4, 5 } 中，共有 $0 + 0 + 0 + 0 + 0 = 0$ 个逆序对
- A[] = { 5, 4, 3, 2, 1 } 中，共有 $0 + 1 + 2 + 3 + 4 = 10$ 个逆序对
- ◆ 显然，逆序对总数 $I = \sum_j I(j) \leq \binom{n}{2} = O(n^2)$

Data Structures & Algorithms, Tsinghua University

- ◆ 属于就地算法 (in-place)
- ◆ 属于在线算法 (online)
- ◆ 具有输入敏感性 (input sensitivity)
 - 后面将会看到：Shellsort之类算法的高效性，完全依赖于insertionsort的这一特性
- ◆ 最好情况：完全（或几乎）有序
 - 每次迭代，只需1次比较，0次交换：累计O(n)时间！
- ◆ 最坏情况：完全（或几乎）逆序
 - 第k次迭代，需O(k)次比较，1次交换：累计O(n²)时间！
- ◆ “优化”的可能：在有序前缀中的查找定位，为何采用了顺序查找，而不是二分查找？

Data Structures & Algorithms, Tsinghua University

列表

归并排序



邓俊辉

deng@tsinghua.edu.cn

日两美其必合兮，孰信修而慕之？思九州岛之博大兮，岂惟是具有女？

列表

逆序对

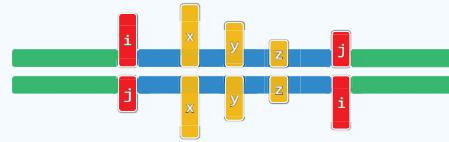


邓俊辉

deng@tsinghua.edu.cn

有象斯有对，对必反其为；有反斯有雠，雠必和而解

BubbleSort



- ◆ 在序列中交换一对逆序元素，逆序对总数必然减少



- ◆ 在序列中交换一对紧邻的逆序元素，逆序对总数恰好减一

- ◆ 因此对于BubbleSort算法而言，交换操作的次数恰等于输入序列所含逆序对的总数

Data Structures & Algorithms, Tsinghua University

◆ 针对任一元素 $e = A[r]$ 的那一步迭代

恰好需要做 $I(r)$ 次比较

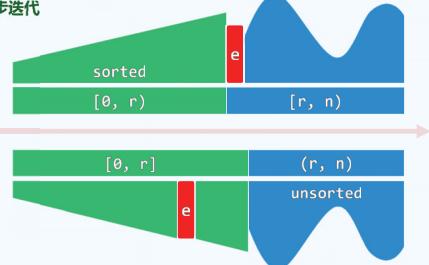
◆ 若共含 I 个逆序对，则

- 关键码比较次数为 $O(I)$

- 运行时间为 $O(n + I)$

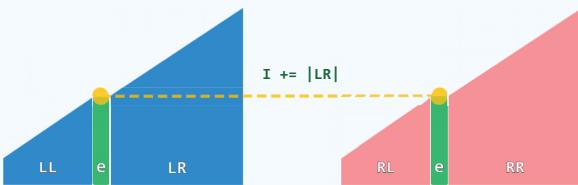
//习题[3-11]

//输入敏感性



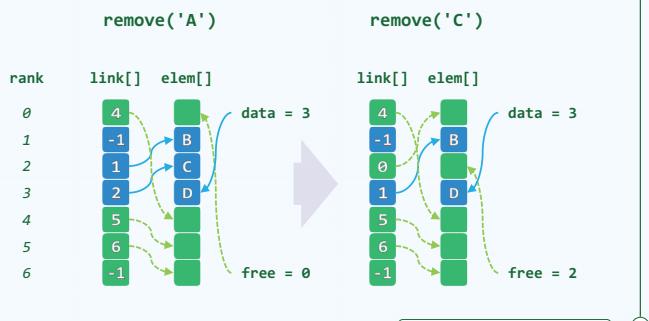
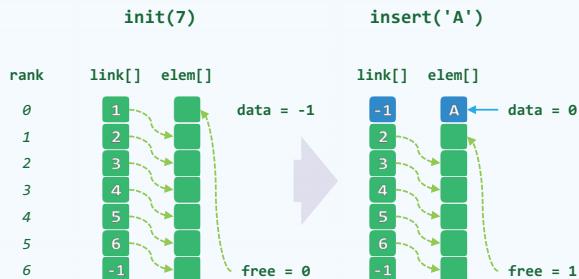
◆ 任意给定一个序列，如何统计其中逆序对的总数？

◆ 蛮力算法需要 $\Omega(n^2)$ 时间；借助归并排序，仅需 $\mathcal{O}(n \log n)$ 时间...



列表

游标实现



列表

Java序列



◆ 某些特定语言或环境中

- 或者不(直接)支持指针

- 或者不支持动态空间分配

此时，如何实现列表结构呢？

◆ 利用线性数组，以游标方式模拟列表

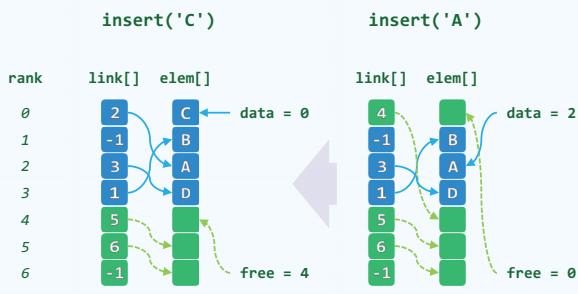
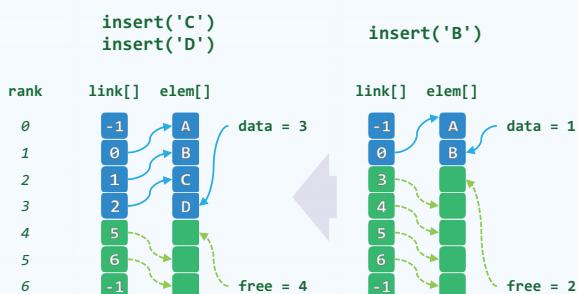
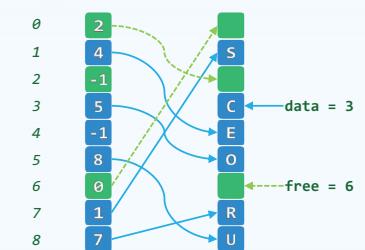
- elem[] : 对外可见的数据项

- link[] : 数据项之间的引用

◆ 维护逻辑上互补的列表data和free

◆ 在插入或删除元素时，应如何调整？

rank link[] elem[]



◆ Java支持ADT的一种机制：在同一接口规范下，允许不同的实现

◆ interface Geometry { //几何物体

```
final double PI = 3.1415926; //常量定义，类定义可直接使用
```

```
double area(); //无参数的接口方法
```

```
boolean inside( Point p ); //带参数的接口方法
```

}

◆ interface不能直接实例化为对象

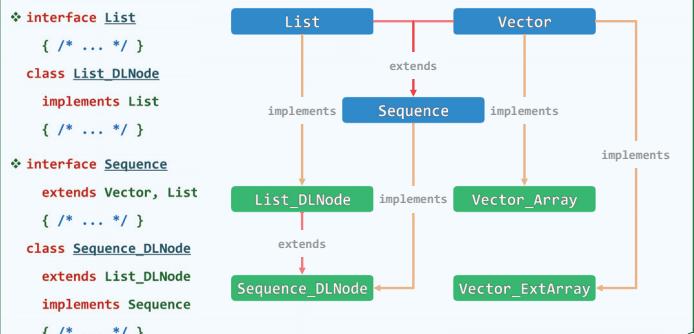
符合interface定义的任何类，都需要具体地实现其中的接口方法

```
class Disk implements Geometry { //符合Geometry接口的Disk类
    Point c; double r;
    public Disk( Point center, double radius ) //构造方法
        {c = center; r = radius;}
    public double perimeter() { return 2 * PI * r; } //类方法
    public double area() { return PI * r * r; } //接口方法的实现
    public boolean inside( Point p ) { //接口方法的实现
        double dx = p.x - c.x, dy = p.y - c.y;
        return dx*dx + dy*dy < r*r;
    }
}
```

```
public interface Vector {
    public int getSize();
    public boolean isEmpty();
    public Object getAtRank( int r ) throws ExceptionBoundaryViolation;
    public Object replaceAtRank( int r, Object obj )
        throws ExceptionBoundaryViolation;
    public Object insertAtRank( int r, Object obj )
        throws ExceptionBoundaryViolation;
    public Object removeAtRank( int r ) throws ExceptionBoundaryViolation;
}
```

```
public class Vector_Array implements Vector {
    private final int N = 1024; //数组容量固定
    private Object[] A; private int n = 0;
    public Vector_Array() { A = new Object[N]; n = 0; }
    public int getSize() { return n; }
    public boolean isEmpty() { return 0 == n; }
    public Object insertAtRank( int r, Object obj ) throws ExceptionBoundaryViolation {
        if ( 0 > r || r > n ) throw new ExceptionBoundaryViolation( "out of range" );
        if ( n >= N ) throw new ExceptionBoundaryViolation( "overflow" );
        for ( int i = n; i > r; i-- ) A[i] = A[i - 1];
        A[r] = obj; n++; return obj;
    }
    /* ..... */
}
```

```
public class Vector_ExtArray implements Vector {
    private int N = 8; //数组的初始容量, 可不断增加
    /* ..... */
    public Object insertAtRank( int r, Object obj ) throws ExceptionBoundaryViolation {
        if ( 0 > r || r > n ) throw new ExceptionBoundaryViolation( "out of range" );
        if ( N <= n ) { //空间溢出的处理
            N *= 2; Object B[] = new Object[ N ]; //容量加倍
            for ( int i = 0; i < n; i++ ) B[i] = A[i]; A = B; //用B[]替换A[]
        }
        for ( int i = n; i > r; i-- ) A[i] = A[i - 1]; //后续元素顺次后移
        A[r] = obj; n++; return obj;
    }
    /* ..... */
}
```



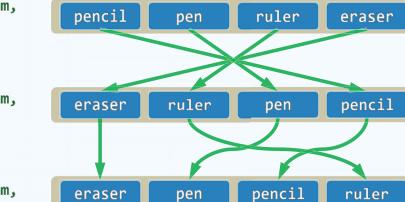
♦ 在Python中，List属于内置的标准数据类型

```
box = [ 'pencil', 'pen', 'ruler', 'eraser' ]; print box
# ['pencil', 'pen', 'ruler', 'eraser']

for item in box: print item,
# pencil pen ruler eraser

box.reverse()
for item in box: print item,
# eraser ruler pen pencil

box.sort()
for item in box: print item,
# erase pen pencil ruler
```



Python列表
邓俊辉
deng@tsinghua.edu.cn

```
for i in range(0, len(box)): # [0, n)
```

```
print box[i],
```

```
# eraser pen pencil ruler
```

```
for i in range(len(box)-1, -1, -1): # [n-1, -1)
```

```
print box[i],
```

```
# ruler pencil pen eraser
```

```
for i in range(-1, -len(box)-1, -1): # [-1, -n-1)
```

```
print box[i],
```

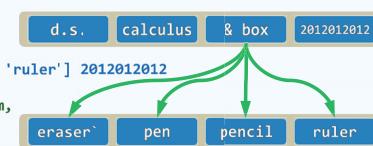
```
# ruler pencil pen eraser
```

♦ bag = ['data structures', 'calculus', box, 2012012012]
print bag
['data structures', 'calculus',
['eraser', 'pen', 'pencil', 'ruler'], 2012012012]

for item in bag: print item,
data structures calculus
[d.s. calculus & box 2012012012]
['eraser', 'pen', 'pencil', 'ruler'] 2012012012

for item in bag[2]: print item,
eraser pen pencil ruler

for item in bag[2][1:3]: print item,
pen pencil



```
def reverse_1(L): # 循秩访问?
```

```
lo, hi = 0, len(L) - 1 # 从首、末元素开始
```

```
while lo < hi: # 依次令对称元素
```

```
L[lo], L[hi] = L[hi], L[lo] # 互换, 然后
```

```
lo, hi = lo + 1, hi - 1 # 考查下一对元素
```

```
return L # 最终即得倒置后的列表
```

```
def reverse_2(L): # 循位置访问?
```

```
for i in range( len(L) ): # 对[0,n)内的每个i, 依次
```

```
L.insert(i, L.pop()) # 将未元素转移至位置i
```

```
return L # 最终即得倒置后的列表
```