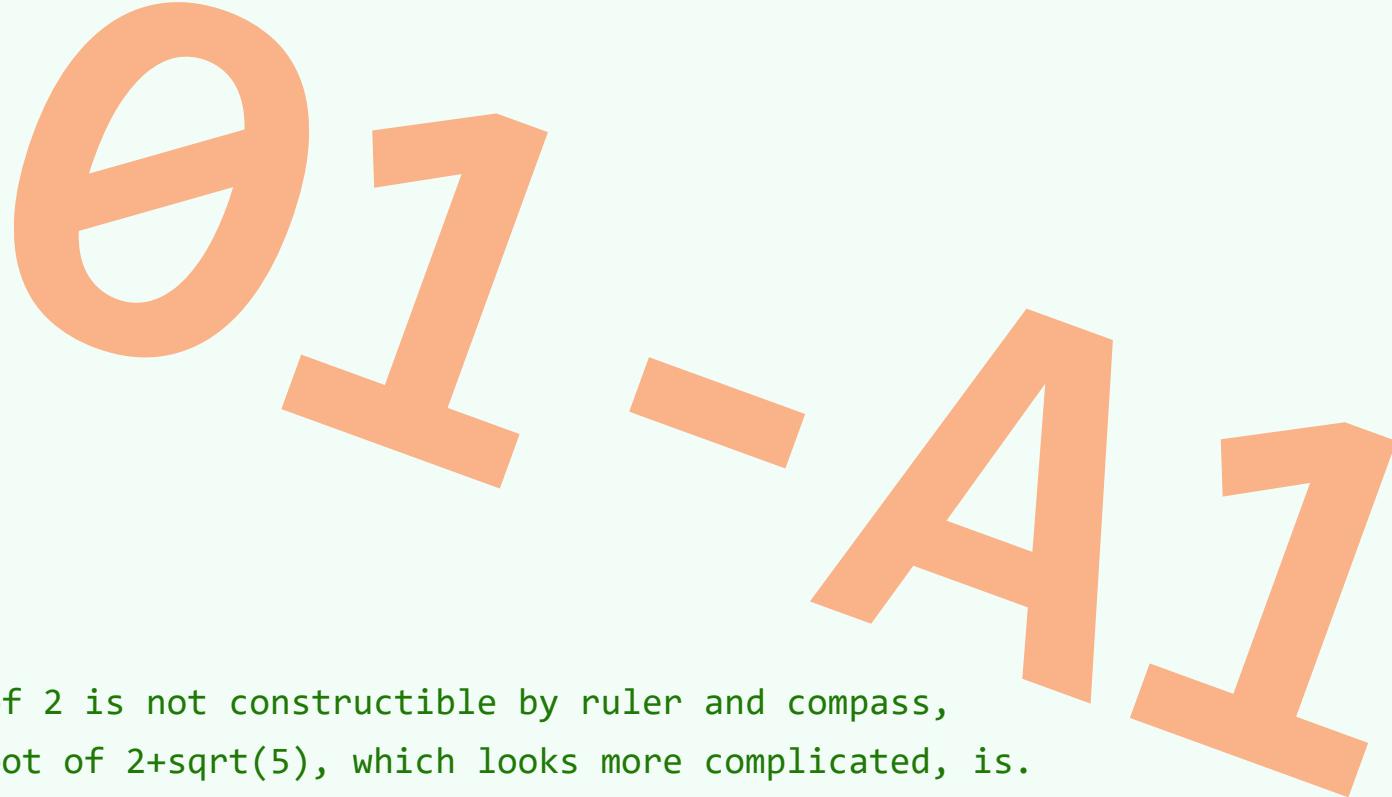


绪论

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.

邓俊辉

deng@tsinghua.edu.cn

尺规计算机

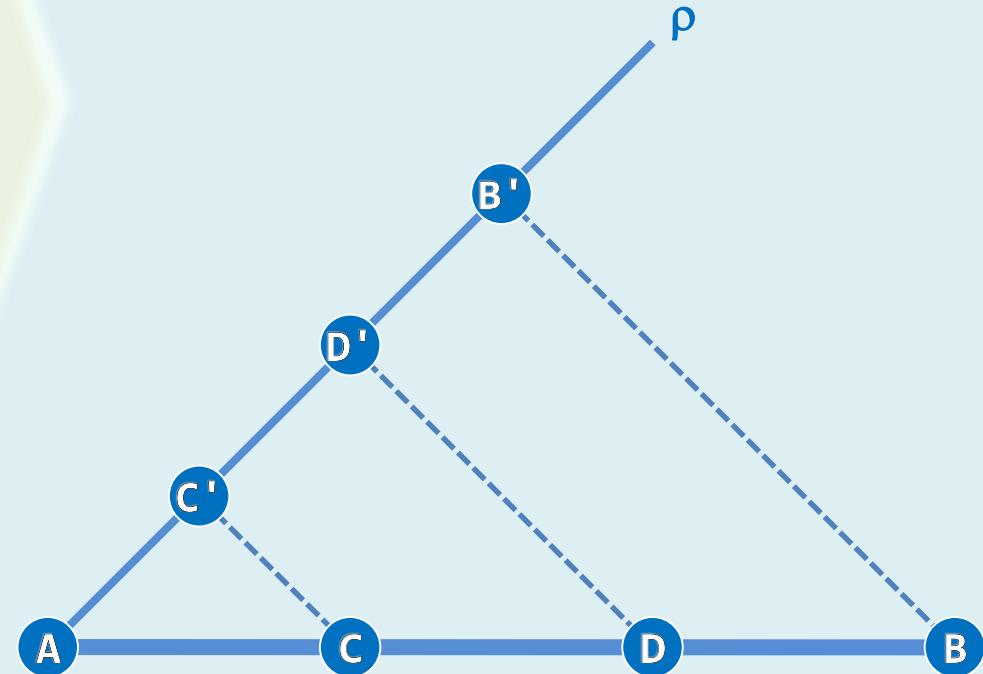
❖ 任给平面上线段AB（输入），将其三等分（输出）

- 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 $B'B$
- 经 D' 做 $B'B$ 的平行线，交 AB 于D
- 经 C' 做 $B'B$ 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



绪论

计算：算法

θ_1

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

- J. von Neumann

A₂

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

- E. Dijkstra

邓俊辉

deng@tsinghua.edu.cn

算法

- ❖ 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 的确可以解决指定的问题

确定性 可描述为一个由基本操作组成的序列 //加盐少许，加糖适量，煮至半熟...

可行性 每一基本操作都可实现，且在常数时间内完成 //把大象放进冰箱，不过三步..."

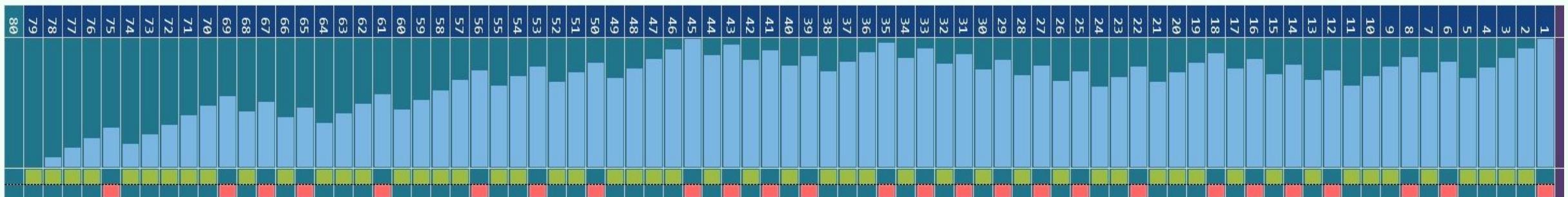
有穷性 对于任何输入，经有穷次基本操作，都可以得到输出

有穷性: Hailstone序列

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

$\boxed{42}, 21, 64, 32, \dots, 1$

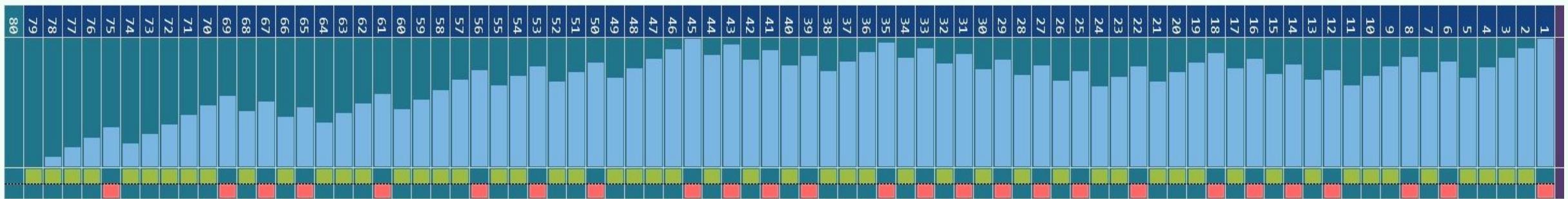
$\boxed{7}, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, \dots, 1$



$\boxed{27}, 82, 41, 124, 62, 31, 94, 47, 142, 71, 214, 107, \dots$

有穷性：程序~算法

```
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)| < ∞ ?
```



❖ Erdos: Mathematics is NOT yet ready for such problems

有穷性：理想硬币 ~ 几何分布

❖ 假设：rand()为理想的随机整数发生器

❖ 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？

❖ $Pr(s = k) = (1/2)^{k-1} \cdot (1 - 1/2) = 2^{-k}$

s的数学期望： $\mathbb{E}(s) = \sum_{k=1}^{\infty} k \cdot 2^{-k} = 1/(1 - 1/2) = 2$

尽管在理论上可能任意多次

3	3
3	4
2	4
2	4
2	4
2	4
2	5
2	5
1	5

正确的算法 ~ 好的算法

- ❖ 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**一般性的输入**
- 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

(Algorithms + Data Structures) x Efficiency = Computation

绪论

计算模型：统一尺度

θ₁

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

- Lord Kelvin

B₁

洞察一个对象的形式的性质，把它同它的质料部分完全分开，然后沉思
它，然后判断时间，即这一特殊形式的事物自然要持续的最长时间。

邓俊辉

deng@tsinghua.edu.cn

算法分析

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

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: 同等规模的不同问题实例, 计算成本也不尽相同, 甚至差异极大

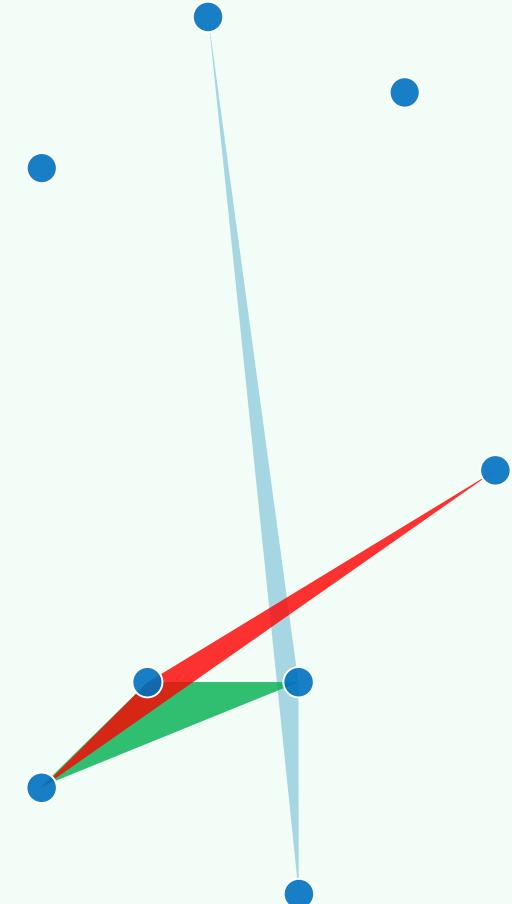
❖ 例如: 任给平面上n个点, 在它们定义的 $\binom{n}{3}$ 个三角形中,
是否某一个的面积不超过5.0?

❖ 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?

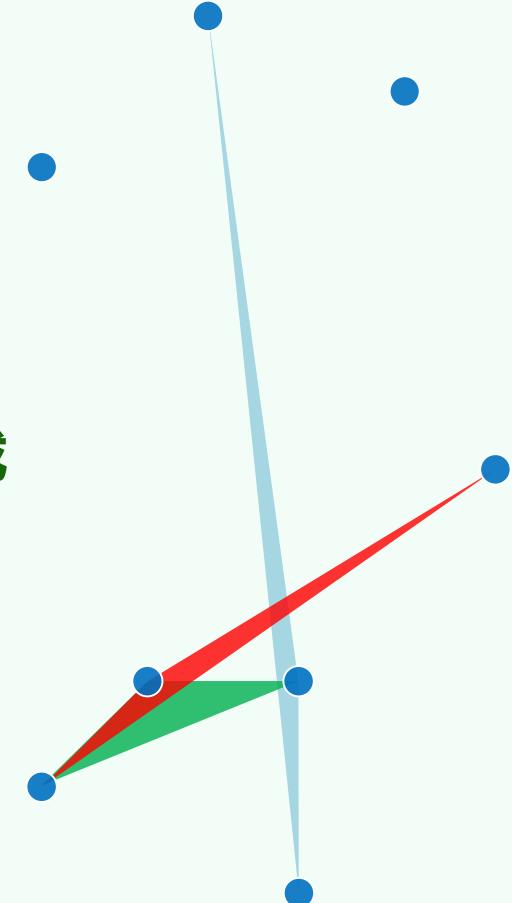
❖ 稳妥起见, 取 $T(n) = \max\{ T(P) \mid |P| = n \}$

亦即, 在规模同为n的所有实例中, 只关注最坏 (成本最高) 者



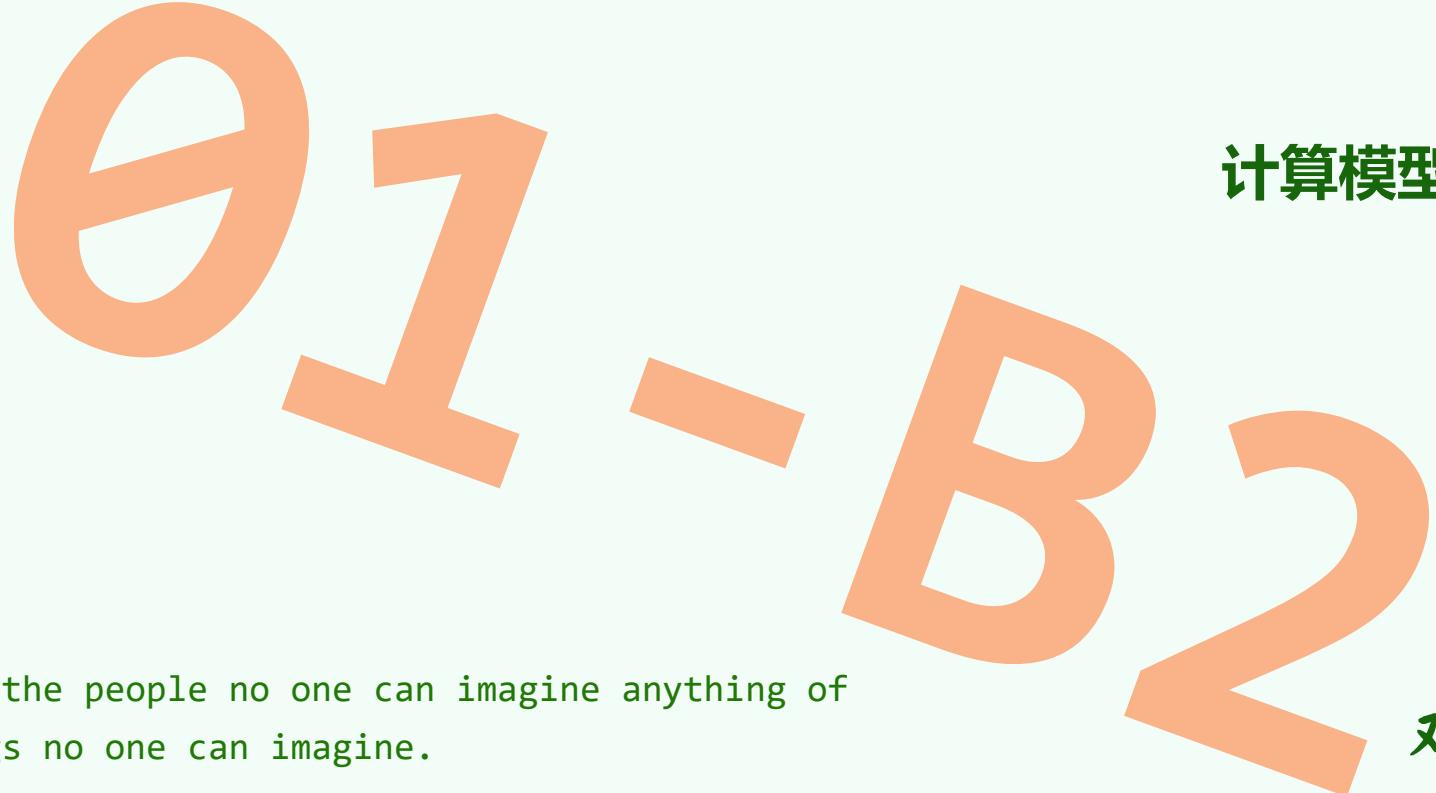
特定问题 + 不同算法

- ❖ 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输入
 - 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



绪论

计算模型：图灵机



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

- A. Turing

邓俊辉

deng@tsinghua.edu.cn

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'#'

❖ 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

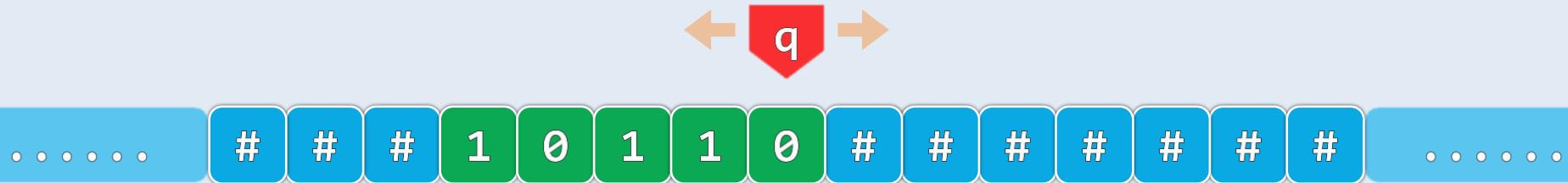
❖ Alphabet

- 字符的种类有限

❖ State

- TM总是处于有限种状态中的某一种
- 每经过一个节拍
可按照规则转向另一种状态
- 统一约定，'h' = halt

转换函数



- ❖ Transition Function: $(q, c; d, L/R, p)$
 - ❖ 特别地, 一旦转入约定的状态 ' h ' , 则停机
 - ❖ 从启动至停机, 所经历的节拍数目
即可用以度量计算的成本
 - ❖ 亦等于Head累计的移动次数 (无量纲)
 - ❖ 若当前状态为 q , 且当前字符为 c , 则
 - 将当前字符改写为 d
 - 转向左/右侧邻格
 - 转入 ' p ' 状态

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

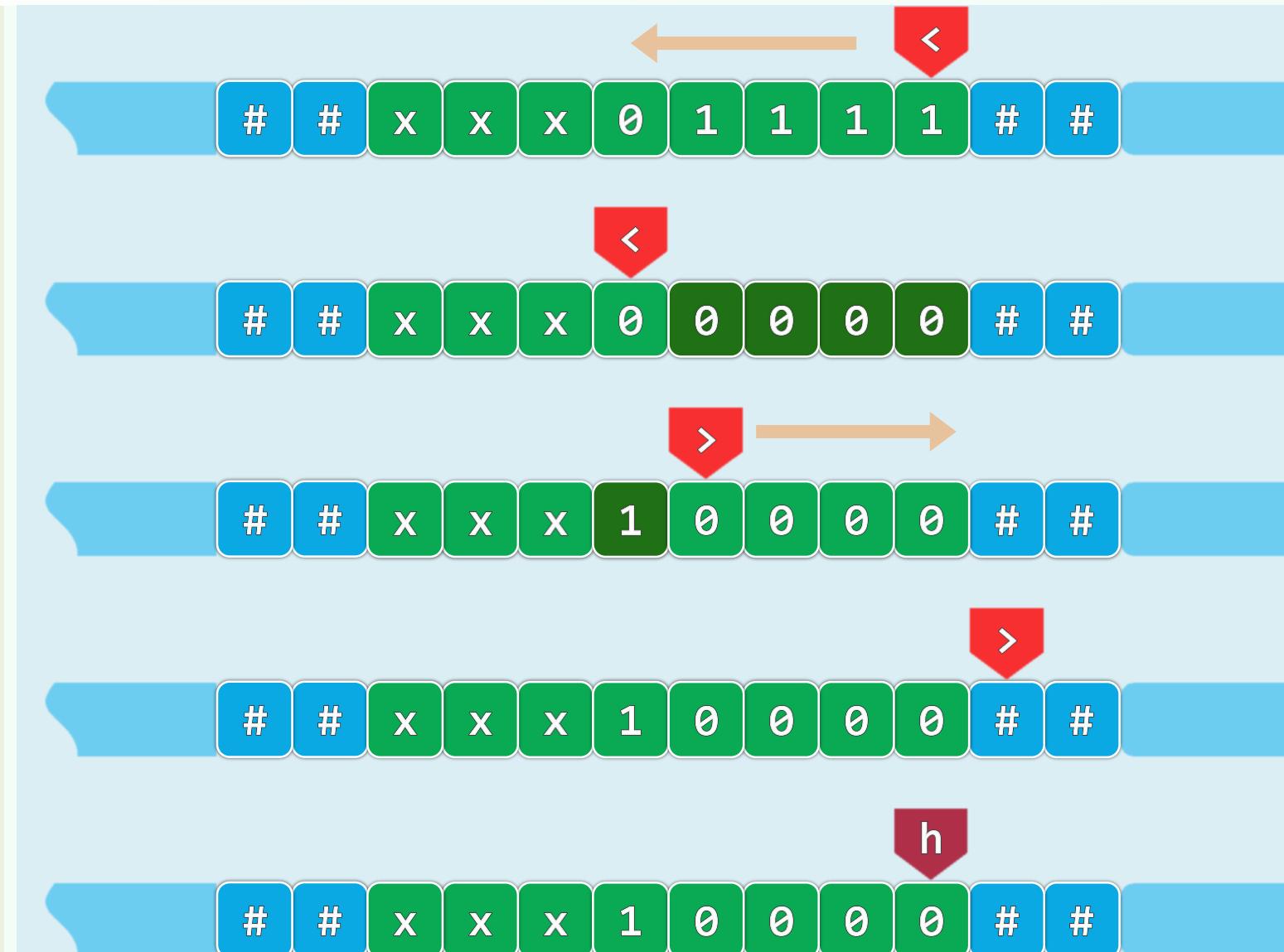
(**<**, 0; 1, R, **>**) //掉头, 0->1

(**<**, #; 1, R, **>**) //可否省略?

(**>**, 0; 0, R, **>**) //右行

(**>**, #; #, L, h/**<**) //?

❖ 规范 ~ 接口



绪论

计算模型：RAM

01 - B3

当有加减施加拳脚的地方，理性便有了容身之处；而在加减无所适从的地方，理性也就失去了容身之所。

三年之喪，二十五月而畢，哀痛未盡，思慕未忘，然而禮以是斷之者，
豈不以送死有已，復生有節也哉！

邓俊辉

deng@tsinghua.edu.cn

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

$R[i] \leftarrow c$ GOTO #
 $R[i] \leftarrow R[j]$
 $R[i] \leftarrow R[R[j]]$ IF $R[i] = 0$ GOTO #
 $R[R[i]] \leftarrow R[j]$ IF $R[i] > 0$ GOTO #
 $R[i] \leftarrow R[j] + R[k]$
 $R[i] \leftarrow R[j] - R[k]$ STOP

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？

实例: Ceiling Division: 思路

- ❖ $\forall c \geq 0$ and $d > 0$, define

$$\begin{aligned}\lceil c/d \rceil &= \min\{x \mid c \leq d \cdot x\} \\ &= \min\{x \mid c - 1 < d \cdot x\}\end{aligned}$$

- ❖ 例如: $\lceil 2/7 \rceil = 1$

$$\lceil 35/5 \rceil = 7 \quad \lceil 2021/43 \rceil = 47$$

$$\lceil 41/7 \rceil = 6 \quad \lceil 2022/43 \rceil = 48$$

- ❖ 思路: 反复地从 $R[0] = c$ 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]	25	12	0	0
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					
16					
17					
18					
19					
20					
21					
22					
23					
24					
25					
26					
27					
28					
29					
30					
31					
32					
33					
34					
35					
36					
37					
38					
39					
40					
41					
42					
43					
44					
45					
46					
47					
48					
49					
50					
51					
52					
53					
54					
55					
56					
57					
58					
59					
60					
61					
62					
63					
64					
65					
66					
67					
68					
69					
70					
71					
72					
73					
74					
75					
76					
77					
78					
79					
80					
81					
82					
83					
84					
85					
86					
87					
88					
89					
90					
91					
92					
93					
94					
95					
96					
97					
98					
99					
100					

实例: Ceiling Division: 算法

```

[0] R[3] <- 1 //constant increment

[1] GOTO 4 //in case R[0] = c == 0

[2] R[2] <- R[2] + R[3] //x++

[3] R[0] <- R[0] - R[1] //c -= d

[4] IF R[0] > 0 GOTO 2 //if c > 0 goto 2

[5] R[0] <- R[2] //else copy x to R[0] and

[6] STOP //return R[0] = x = ⌈c/d⌉
    
```

时间成本 ~ 各条指令执行次数之总和

Step	IR	R[0]	R[1]	R[2]	R[3]
0	[0]	25	12	0	0
1	[1]	^	^	^	1
2	[4]	^	^	^	^
3	[2]	^	^	^	^
4	[3]	^	^	1	^
5	[4]	13	^	^	^
6	[2]	^	^	^	^
7	[3]	^	^	2	^
8	[4]	1	^	^	^
9	[2]	^	^	^	^
10	[3]	^	^	3	^
11	[4]	-11	^	^	^
12	[5]	^	^	^	^
13	[6]	3	^	^	^

绪论

渐近复杂度：大 θ 记号

Θ_1

C_1

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

邓俊辉

deng@tsinghua.edu.cn

渐近分析

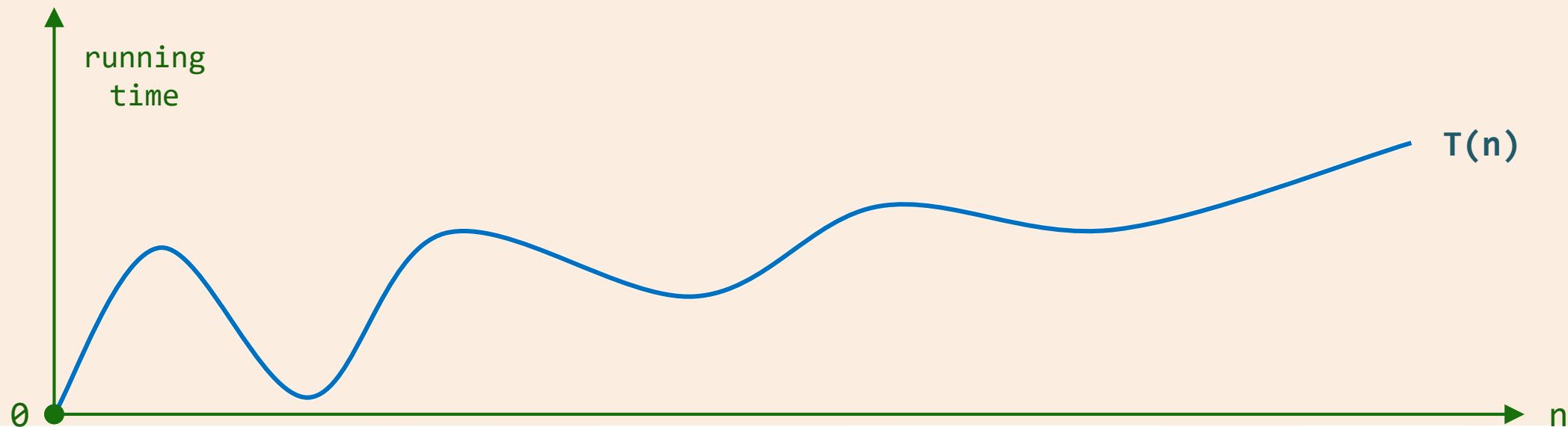
❖ 回到原先的问题：

随着问题规模的增长，计算成本如何增长？

❖ 这里更关心：

问题规模**足够大之后**，计算成本的**增长趋势**

- ❖ 当输入规模 $n \gg 2$ 后，算法需执行的基本操作次数 $T(n) = ?$
- ❖ 如欲更为精确地估计，还可考查需占用的存储单元数 $S(n) = ?$



Big-O notation

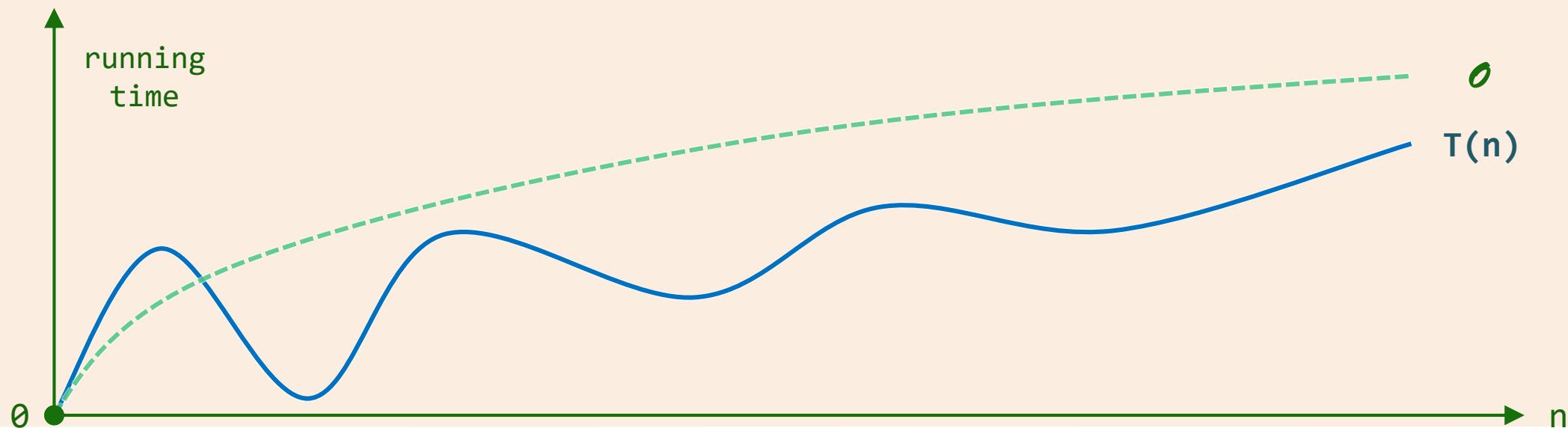
❖ Paul Bachmann, 1894: $T(n) = \mathcal{O}(f(n))$ iff $\exists c > 0$ s.t. $T(n) < c \cdot f(n) \quad \forall n \gg 2$

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

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

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

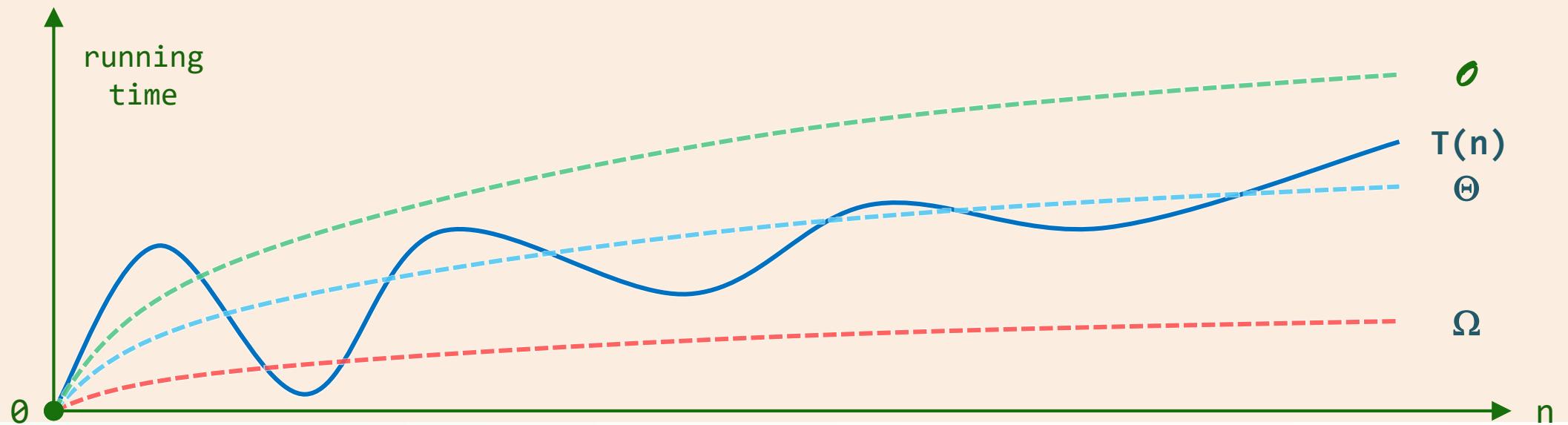
$$\mathcal{O}(n^a + n^b) = \mathcal{O}(n^a), a \geq b > 0$$



其它记号

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

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



绪论

渐近复杂度：多项式

θ₁ - C₂

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

- A. Cobham & J. Edmonds

邓俊辉
deng@tsinghua.edu.cn

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

- ❖ 常数: $2 = 2023 = 2023 \times 2023 = \mathcal{O}(1)$ //含RAM的所有基本操作, 甚至
 $2023^{2023} = \mathcal{O}(1)$
- ❖ 从渐近的角度来看, 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] $k \leq 123$

$\Theta(1)$: 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/2023 + 1 ); //2023, 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 ) o1op(n); // $\Theta(1)$ -time Operation
```

...

$\Theta(\log^c n)$: poly-log

- ❖ 对数 $\mathcal{O}(\log n)$: $\ln n$ $\lg n$ $\log_{100} n$ $\log_{2022} n$ //为何不注明底数?
- ❖ 常底数无所谓: $\forall a, b > 1, \log_a n = \boxed{\log_a b} \cdot \log_b n = \Theta(\log_b n)$
- ❖ 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: $\forall c > 0, \log n = \mathcal{O}(n^c)$

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

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

$$100 \cdot n + 2023 = \mathcal{O}(n) \quad \sqrt{23 \cdot n - 472} \times \sqrt{101 \cdot n + 2023} = \mathcal{O}(n)$$

$$(100 \cdot n - 532) \cdot (20 \cdot n^2 - 445 \cdot n + 2023) = \mathcal{O}(n^3) \quad (2023 \cdot n^2 - 129)/(1911 \cdot n - 37) = \mathcal{O}(n)$$

$$\sqrt[3]{2 \cdot n^3 - \sqrt[3]{3 \cdot n^4 - \sqrt{4 \cdot n^5 + \sqrt{5 \cdot n^6 + \sqrt{6 \cdot n^7 + \sqrt{7 \cdot n^8 + \sqrt{8 \cdot n^9 + n^{2023}}}}}}}} / \sqrt{n^{14} - 5 \cdot n^9 + 2023} = \mathcal{O}(n^7)$$

❖ 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!

绪论

渐近复杂度：指数

01



C3

邓俊辉

deng@tsinghua.edu.cn

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

“你是什么东西？”太太说。四虎子也楞住了，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。

$\Theta(2^n)$: exponential

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

$$\because e^n = 1 + n + n^2/2! + n^3/3! + n^4/4! + \dots$$

$$\therefore \forall c > 1, n^c = \mathcal{O}(2^n)$$

$$n^{1000\dots01} = \mathcal{O}(1.000\dots01^n) = \mathcal{O}(2^n)$$

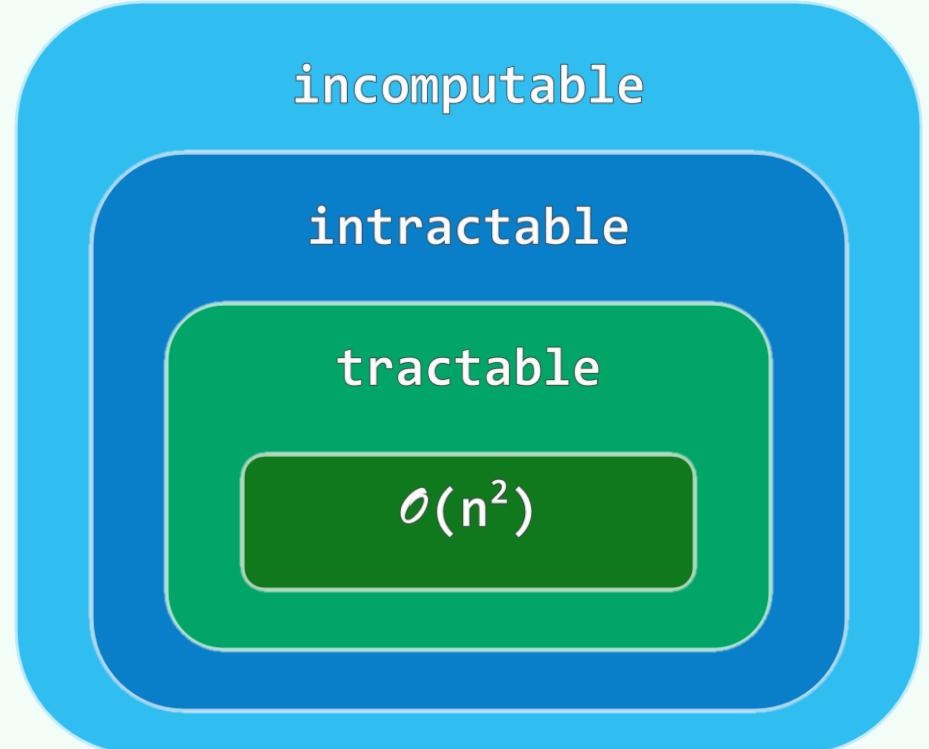
$$1.000\dots01^n = \Omega(n^{1000\dots01})$$

❖ 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

❖ $\mathcal{O}(2^n)$ 算法往往显而易见，然而设计出 $\mathcal{O}(n^c)$ 算法却极其不易，有时甚至注定是徒劳无功

❖ 更糟糕的是，这类问题要远比我们想象的多得多...



SubsetSum: 问题

$\forall S = \{a_1, a_2, \dots, a_n\} \subset \mathbb{Z}^+$ and $0 \leq M \leq \sum_{k=1}^n a_k$

$\exists T \subset S$ s.t. $\sum_{a \in T} a = M$?

❖ 选举人团投票制：

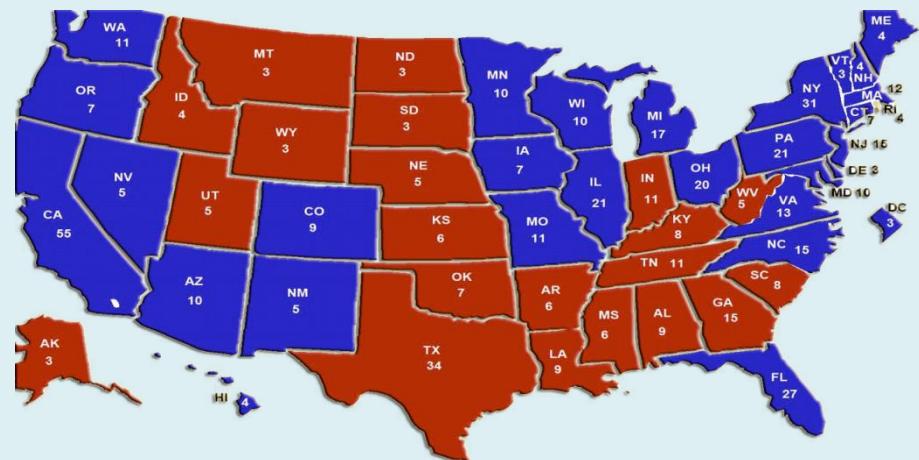
50个州加1个特区，共538票 //n, 2M

获270张选举人票，即可当选 //M+1

❖ 若仅两位候选人，会否恰好各得269票？ //M

❖ 可视作SubsetSum的特例： $\sum_{k=1}^n a_k = 2M$

55	California	11	Indiana	7	Connecticut	4	Idaho
34	Texas	11	Missouri	7	Iowa	4	Maine
31	New York	11	Tennessee	7	Oklahoma	4	New Hampshire
27	Florida	11	Washington	7	Oregon	4	Rhode Island
21	Illinois	10	Arizona	6	Arkansas	3	Alaska
21	Pennsylvania	10	Maryland	6	Kansas	3	Delaware
20	Ohio	10	Minnesota	6	Mississippi	3	D. C.
17	Michigan	10	Wisconsin	5	Nebraska	3	Montana
15	Georgia	9	Alabama	5	Nevada	3	North Dakota
15	New Jersey	9	Colorado	5	New Mexico	3	South Dakota
15	North Carolina	9	Louisiana	5	Utah	3	Vermont
13	Virginia	8	Kentucky	5	West Virginia	3	Wyoming
12	Massachusetts	8	South Carolina	4	Hawaii		$538 = \Sigma$



SubsetSum: 算法 程序

❖ 直觉上，并~~不~~难：

逐一枚举S的每一子集，统计总和并核对

❖ sSum($S = \{a_1, a_2, \dots, a_n\}$, M)

```
if ( M == 0 ) return true;
```

```
if ( n == 0 ) return false;
```

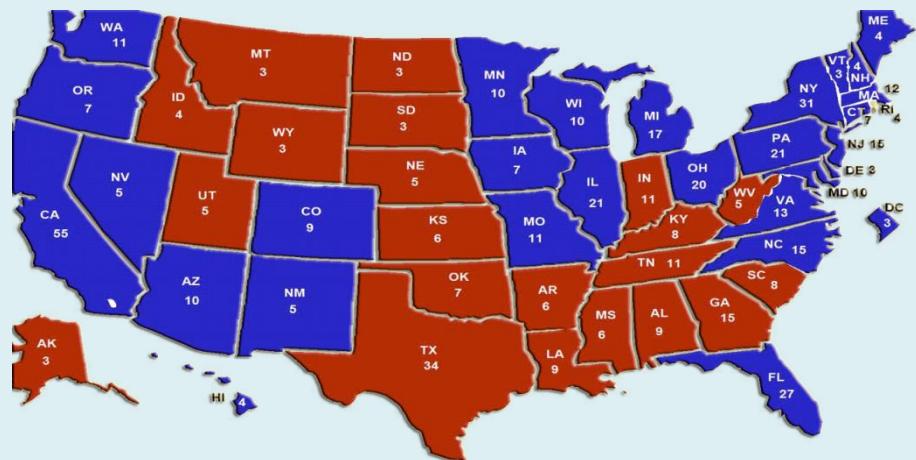
```
S = S\{a_n\}; //classification
```

```
return sSum(S, M) || sSum(S, M-a_n);
```

❖ 最坏情况下，需要检视每一个子集，然而...

$$|2^S| = 2^{|S|} = 2^n$$

55	California	11	Indiana	7	Connecticut	4	Idaho
34	Texas	11	Missouri	7	Iowa	4	Maine
31	New York	11	Tennessee	7	Oklahoma	4	New Hampshire
27	Florida	11	Washington	7	Oregon	4	Rhode Island
21	Illinois	10	Arizona	6	Arkansas	3	Alaska
21	Pennsylvania	10	Maryland	6	Kansas	3	Delaware
20	Ohio	10	Minnesota	6	Mississippi	3	D. C.
17	Michigan	10	Wisconsin	5	Nebraska	3	Montana
15	Georgia	9	Alabama	5	Nevada	3	North Dakota
15	New Jersey	9	Colorado	5	New Mexico	3	South Dakota
15	North Carolina	9	Louisiana	5	Utah	3	Vermont
13	Virginia	8	Kentucky	5	West Virginia	3	Wyoming
12	Massachusetts	8	South Carolina	4	Hawaii		$538 = \Sigma$



SubsetSum: NPC

- ◆ 故严格地讲，这类方法只能算作**程序**，而非**算法**

- ### ❖ 还是直觉：应该有更好的办法吧？

比如...转化为...背包问题...?

很遗憾，这里对于整数的取值范围未作任何假定

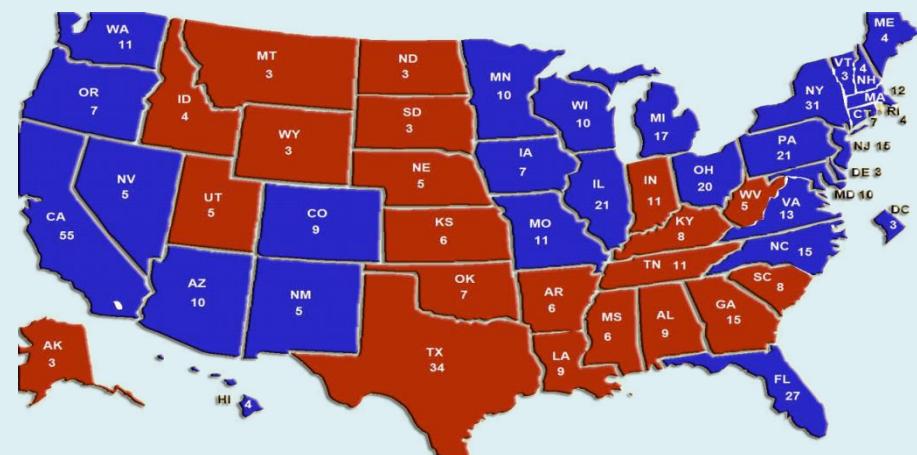
- ❖ 定理: SubsetSum is NP-complete — 什么意思?

- ❖ 意即：就目前的计算模型而言

不存在可在多项式时间内解决此问题的算法

上述的直觉算法，居然就是最优的

55	California	11	Indiana	7	Connecticut	4	Idaho
34	Texas	11	Missouri	7	Iowa	4	Maine
31	New York	11	Tennessee	7	Oklahoma	4	New Hampshire
27	Florida	11	Washington	7	Oregon	4	Rhode Island
21	Illinois	10	Arizona	6	Arkansas	3	Alaska
21	Pennsylvania	10	Maryland	6	Kansas	3	Delaware
20	Ohio	10	Minnesota	6	Mississippi	3	D. C.
17	Michigan	10	Wisconsin	5	Nebraska	3	Montana
15	Georgia	9	Alabama	5	Nevada	3	North Dakota
15	New Jersey	9	Colorado	5	New Mexico	3	South Dakota
15	North Carolina	9	Louisiana	5	Utah	3	Vermont
13	Virginia	8	Kentucky	5	West Virginia	3	Wyoming
12	Massachusetts	8	South Carolina	4	Hawaii		538 = Σ



绪论

渐近复杂度：层级划分

e1 - c4

邓俊辉

deng@tsinghua.edu.cn

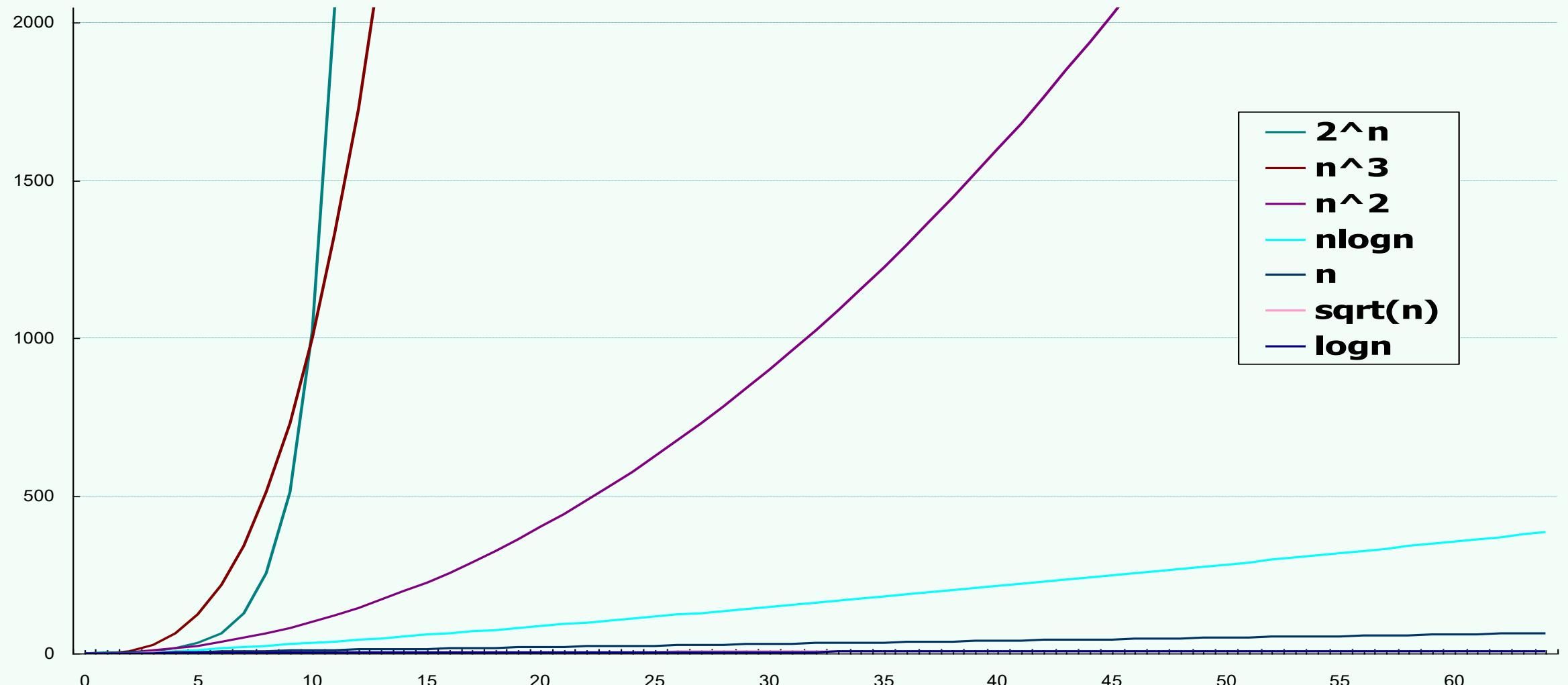
好读书，不求甚解；每有会意，便欣然忘食

主啊，我向你承认，我依旧不明了时间是什么

增长速度：先胖不算胖



增长速度：路遥知马力



层次级别

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...就是线性	某些MST算法
	$\mathcal{O}(n \log \log n)$	非常非常非常...接近线性	某些三角剖分算法
	$\mathcal{O}(n \log n)$	最常出现，但不见得最优	排序、EU、Huffman编码
平方	$\mathcal{O}(n^2)$	所有输入对象两两组合	Dijkstra算法
立方	$\mathcal{O}(n^3)$	不常见	矩阵乘法
多项式	$\mathcal{O}(n^c)$	P问题 = 存在多项式算法的问题	
指数	$\mathcal{O}(2^n)$	很多问题的平凡算法，再尽可能优化	
...		绝大多数问题，并不存在算法	

绪论

复杂度分析：级数

$\theta_1 - D_1$

邓俊辉

deng@tsinghua.edu.cn

谁校对时间，谁就会突然老去。

算法分析

- ❖ 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，再累计各条代码的执行次数？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；在渐近意义下，二者大体相当
 - 分支转向：`goto` //算法的灵魂；为结构化而被隐藏了而已
 - 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`
- ❖ 主要方法：迭代（级数求和）、递归（递归跟踪 + 递推方程）、实用（猜测 + 验证）

级数

❖ 算术级数：与末项平方同阶 $T(n) = 1 + 2 + \dots + n = \frac{(n+1)}{2} = \frac{n(n+1)}{2} = \mathcal{O}(n^2)$

❖ 幂方级数：比幂次高出一阶 $\sum_{k=0}^n k^d \approx \int_0^n x^d dx = \frac{x^{d+1}}{d+1} \Big|_0^n = \frac{n^{d+1}}{d+1} = \mathcal{O}(n^{d+1})$

$$T_2(n) = \sum_{k=1}^n k^2 = 1^2 + 2^2 + 3^2 + \dots + n^2 = \cancel{n(n+1)(2n+1)/6} = \mathcal{O}(n^3)$$

$$T_3(n) = \sum_{k=1}^n k^3 = 1^3 + 2^3 + 3^3 + \dots + n^3 = \cancel{n^2(n+1)^2/4} = \mathcal{O}(n^4)$$

$$T_4(n) = \sum_{k=1}^n k^4 = 1^4 + 2^4 + 3^4 + \dots + n^4 = \cancel{n(n+1)(2n+1)(3n^2+3n-1)/30} = \mathcal{O}(n^5)$$

❖ 几何级数：与末项同阶

$$T_a(n) = \sum_{k=0}^n a^k = a^0 + a^1 + a^2 + a^3 + \dots + a^n = \cancel{\frac{a^{n+1}-1}{a-1}} = \mathcal{O}(a^n), \quad 1 < a$$

$$T_2(n) = \sum_{k=0}^n 2^k = 1 + 2 + 4 + 8 + \dots + 2^n = \cancel{2^{n+1}-1} = \mathcal{O}(2^{n+1}) = \mathcal{O}(2^n)$$

收敛级数

❖ $\sum_{k=2}^n \frac{1}{(k-1) \cdot k} = \frac{1}{1 \cdot 2} + \frac{1}{2 \cdot 3} + \frac{1}{3 \cdot 4} + \dots + \frac{1}{(n-1) \cdot n} = 1 - \frac{1}{n} = \mathcal{O}(1)$

$$\sum_{k=1}^n \frac{1}{k^2} = 1 + \frac{1}{2^2} + \frac{1}{3^2} + \dots + \frac{1}{n^2} < 1 + \frac{1}{2^2} + \frac{1}{3^2} + \dots = \frac{\pi^2}{6} = \mathcal{O}(1)$$

$$\sum_{k \text{ is a perfect power}} \frac{1}{k-1} = \frac{1}{3} + \frac{1}{7} + \frac{1}{8} + \frac{1}{15} + \frac{1}{24} + \frac{1}{26} + \frac{1}{31} + \frac{1}{35} + \dots = 1 = \mathcal{O}(1)$$

❖ 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, 某种意义上的确是!

不收敛，但有限

❖ 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: $\sum_{k=1}^n \ln k = \ln \prod_{k=1}^n k = \ln n! \approx (n + 0.5) \cdot \ln n - n = \Theta(n \cdot \log n)$

❖ 对数 + 线性 + 指数: $\sum_{k=1}^n k \cdot \log k \approx \int_1^n x \ln x dx = \frac{x^2 \cdot (2 \cdot \ln x - 1)}{4} \Big|_1^n = \mathcal{O}(n^2 \log n)$

$$\begin{aligned} \sum_{k=1}^n k \cdot 2^k &= \sum_{k=1}^n k \cdot 2^{k+1} - \sum_{k=1}^n k \cdot 2^k = \sum_{k=1}^{n+1} (k-1) \cdot 2^k - \sum_{k=1}^n k \cdot 2^k \\ &= n \cdot 2^{n+1} - \sum_{k=1}^n 2^k = n \cdot 2^{n+1} - (2^{n+1} - 2) = (n-1) \cdot 2^{n+1} + 2 = \mathcal{O}(n \cdot 2^n) \end{aligned}$$

❖ 如有兴趣，不妨读读: Concrete Mathematics

//ex-2.35, Goldbach Theorem

绪论

复杂度分析：迭代

θ₁ - D₂

邓俊辉

deng@tsinghua.edu.cn

Go To Statement Considered Harmful.

- E. Dijkstra, 1968

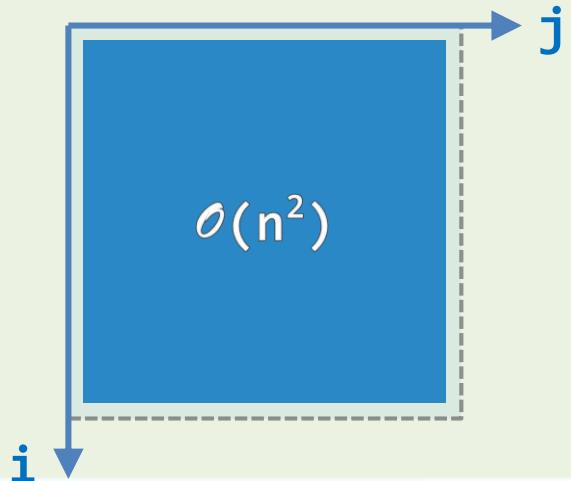
迭代 + 算术级数

```
for( int i = 0; i < n; i++ )
```

```
for( int j = 0; j < n; j++ )
```

```
    O1op(const i, const j);
```

$$\sum_{i=0}^{n-1} n = n \times n = \mathcal{O}(n^2)$$

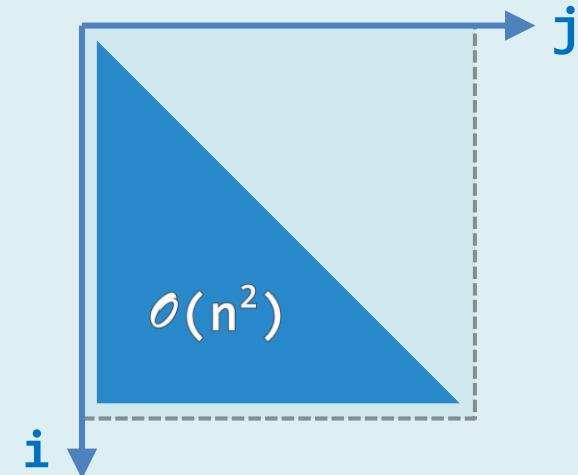


```
for( int i = 0; i < n; i++ )
```

```
for( int j = 0; j < i; j++ )
```

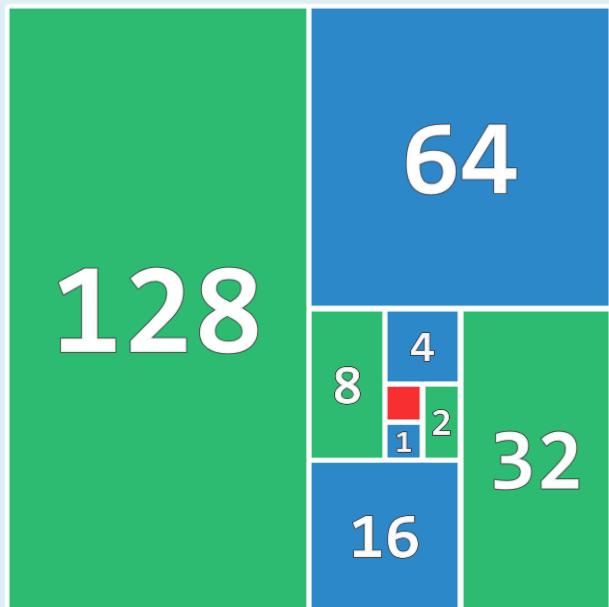
```
    O1op(const i, const j);
```

$$\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} = \mathcal{O}(n^2)$$



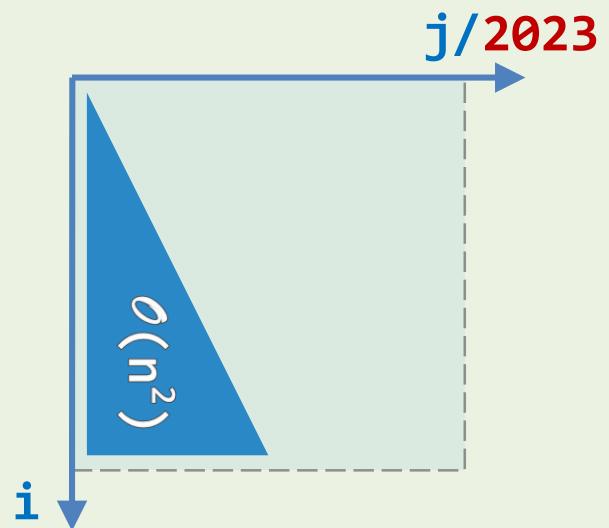
迭代 vs. 级数

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



$$1 + 2 + 4 + \dots + 2^{\lfloor \log_2(n-1) \rfloor} = 2^{\lceil \log_2 n \rceil} - 1 = \mathcal{O}(n)$$

```
for( int i = 0; i < n; i++ )  
  
for( int j = 0; j < i; j += 2023 )  
  
    O1op( const i, const j );
```



迭代 + 复杂级数

```
for( int i = 0; i <= n; i++ )
```

```
for( int j = 1; j < i; j += j )
```

```
    Olop( const i, const j );
```

$$T(n) \approx \int_{x=0}^n \ln(x) dx = n \cdot \ln(n) - \int_{x=0}^n x \cdot d(\ln(x)) = n \cdot \ln(n) - n$$

$$T(n) = \sum_{k=1}^{\log n} k \cdot 2^{k-1} = \sum_{k=1}^{\log n} \sum_{i=1}^k 2^{k-1} = \sum_{i=1}^{\log n} \sum_{k=i}^{\log n} 2^{k-1}$$

$$T(n) \leq \sum_{i=1}^{\log n} \sum_{k=1}^{\log n} 2^{k-1} \leq \sum_{i=1}^{\log n} 2^{\log n} = \sum_{i=1}^{\log n} n = n \cdot \log n$$

$$T(n) \geq \sum_{i=1}^{\log n} \sum_{k=\log n}^{\log n} 2^{k-1} \geq \sum_{i=1}^{\log n} 2^{\log n-1} = \sum_{i=1}^{\log n} n/2 = n/2 \cdot \log n$$



绪论

复杂度分析：封底估算

Theta-1-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

Back-Of-The-Envelope Calculation

❖ 地球（赤道）周长 $\approx 787 \times 360/7.2$

$$= 787 \times 50 = 39,350 \text{ km}$$

❖ 1天 $= 24\text{hr} \times 60\text{min} \times 60\text{sec}$

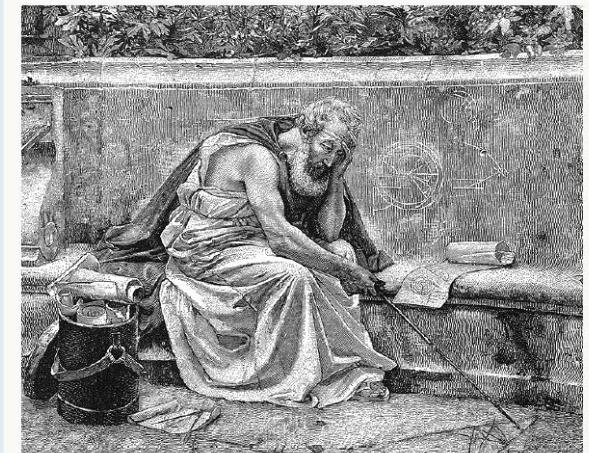
$$\approx 25 \times 4000 = 10^5 \text{ sec}$$

❖ 1生 $\approx 1\text{世纪} = 100\text{yr} \times 365 = 3 \times 10^4 \text{ day} = 3 \times 10^9 \text{ sec}$

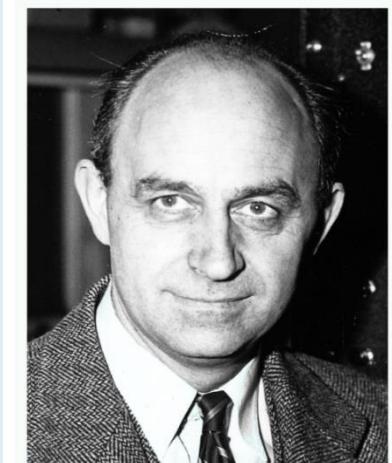
❖ “为祖国健康工作五十年” $\approx 1.6 \times 10^9 \text{ sec}$

❖ “三生三世” $\approx 300 \text{ yr} = 10^{10} = (1 \text{ googel})^{(1/10)} \text{ sec}$

❖ 宇宙大爆炸至今 $= 4 \times 10^{17} > 10^8 \times \text{一生}$

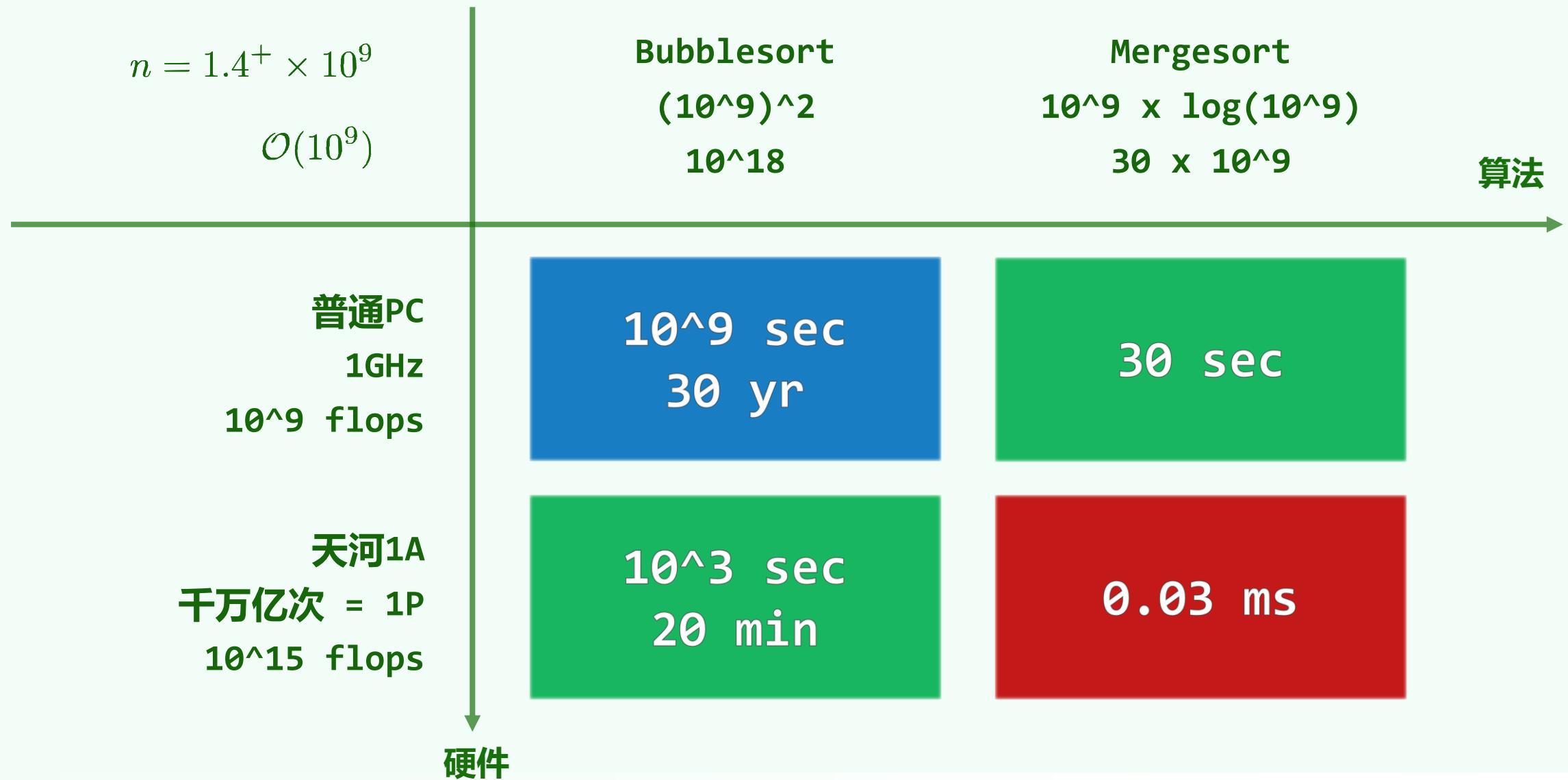


Eratosthenes
(276 ~ 194 B.C.)



Enrico Fermi
(1901 - 1954)

实例：全国人口普查数据的排序



绪论

迭代与递归：减而治之

让最后一个数结束这些争吵.....

否则我就利用你留给我的权限

像一根一根扯马尾鬃一般

——删去个位数直至啥也看不见

您也会被我的推理逗着玩

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？

E1

邓俊辉

deng@tsinghua.edu.cn

Sum: 计算任意n个整数之和

❖ 思路：逐一取出每个元素，累加之

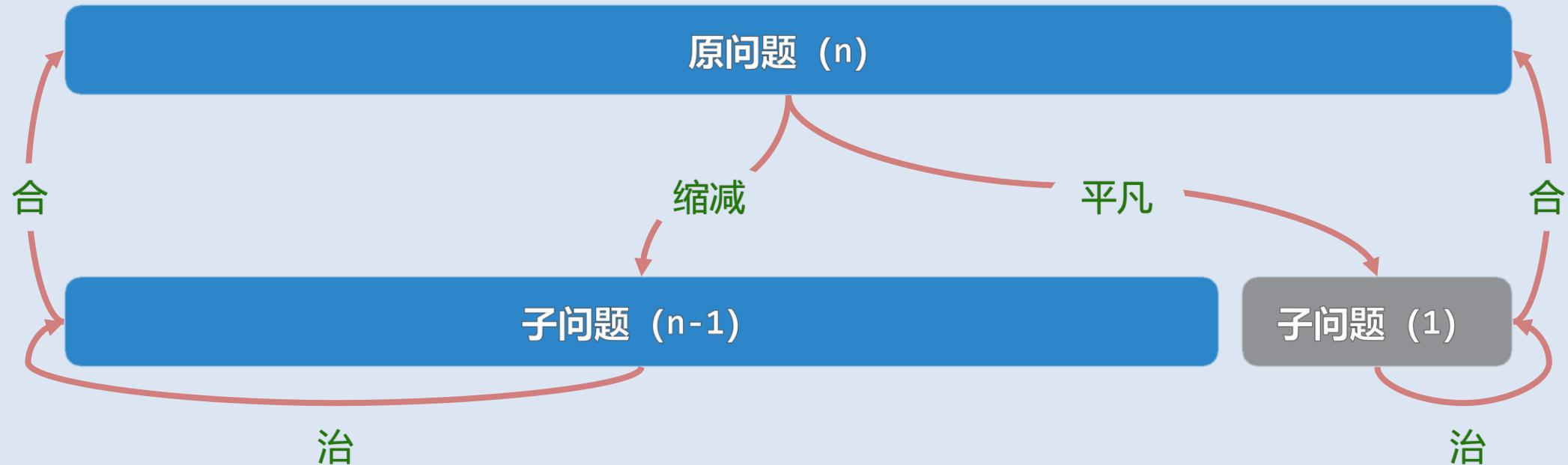
```
int SumI( int A[], int n ) {  
  
    int sum = 0; //O(1)  
  
    for ( int i = 0; i < n; i++ ) //O(n)  
        sum += A[i]; //O(1)  
  
    return sum; //O(1)  
}
```

❖ 无论A[]内容如何，都有：

$$\begin{aligned} T(n) &= 1 + n \cdot 1 + 1 \\ &= n + 2 \\ &= O(n) \\ &= \Omega(n) \\ &= \Theta(n) \end{aligned}$$

❖ 空间呢？

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** //单调性
- 分别求解子问题；再由子问题的解，得到原问题的解

Linear Recursion: Trace

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

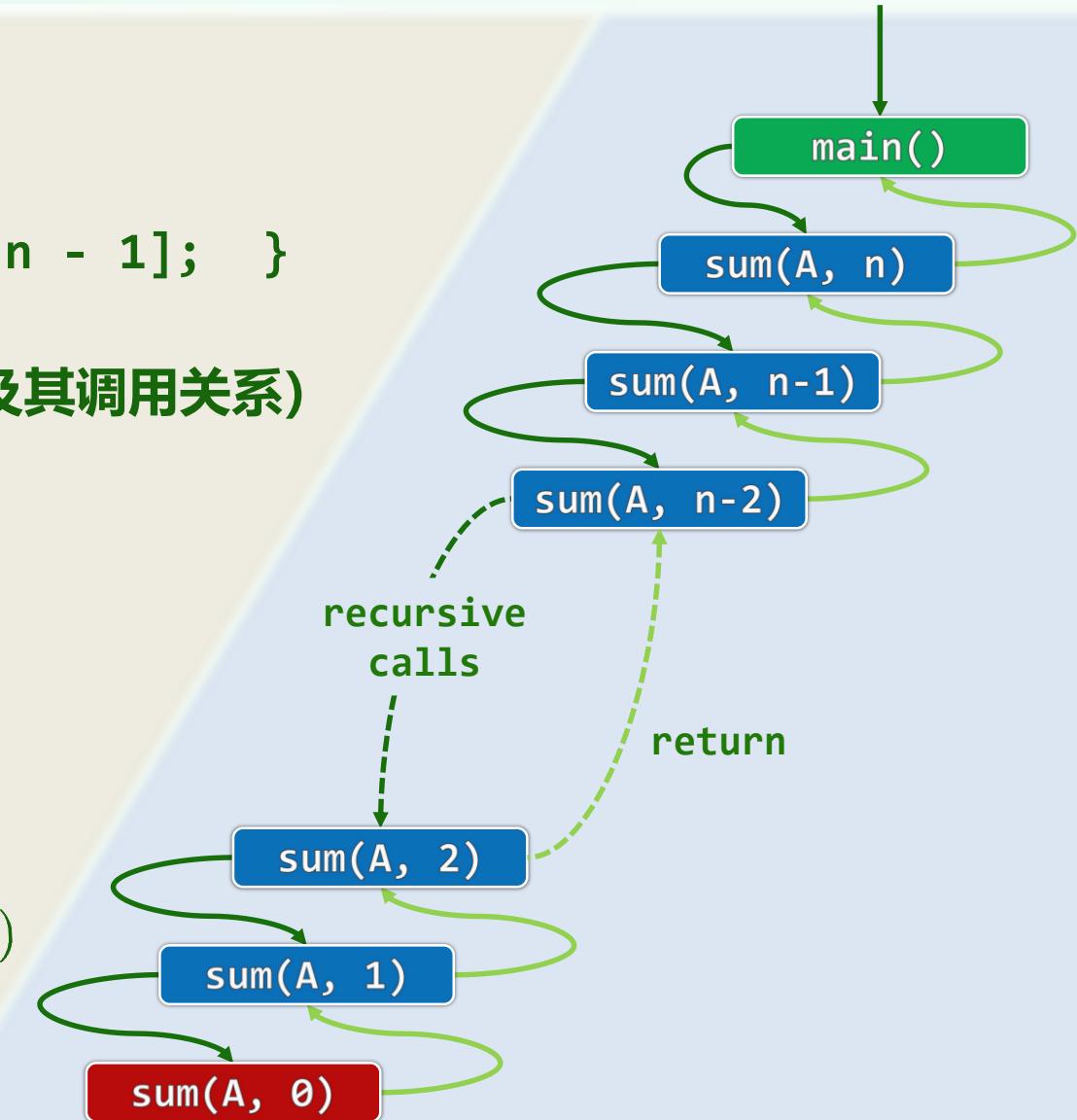
❖ 递归跟踪：绘出计算过程中出现过的所有递归实例（及其调用关系）

- 它们各自所需时间之总和，即为整体运行时间
- （调用操作本身的成本，如何处理？）

❖ 本例中，共计 $n+1$ 个递归实例，各自只需 $\mathcal{O}(1)$ 时间

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

❖ 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 $\text{sum}(A, n)$ ，需 // $T(n)$

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

❖ 递推方程： $T(n) = T(n - 1) + \mathcal{O}(1)$ //recurrence

$$T(0) = \mathcal{O}(1) \quad //\text{base: } \text{sum}(A, 0)$$

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

Reverse

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

//将数组中的区间A[lo,hi]前后颠倒

❖ **减治:** $Rev(lo, hi) = [hi] + Rev(lo + 1, hi - 1) + [lo]$

❖ `if (lo < hi) { //递归版`

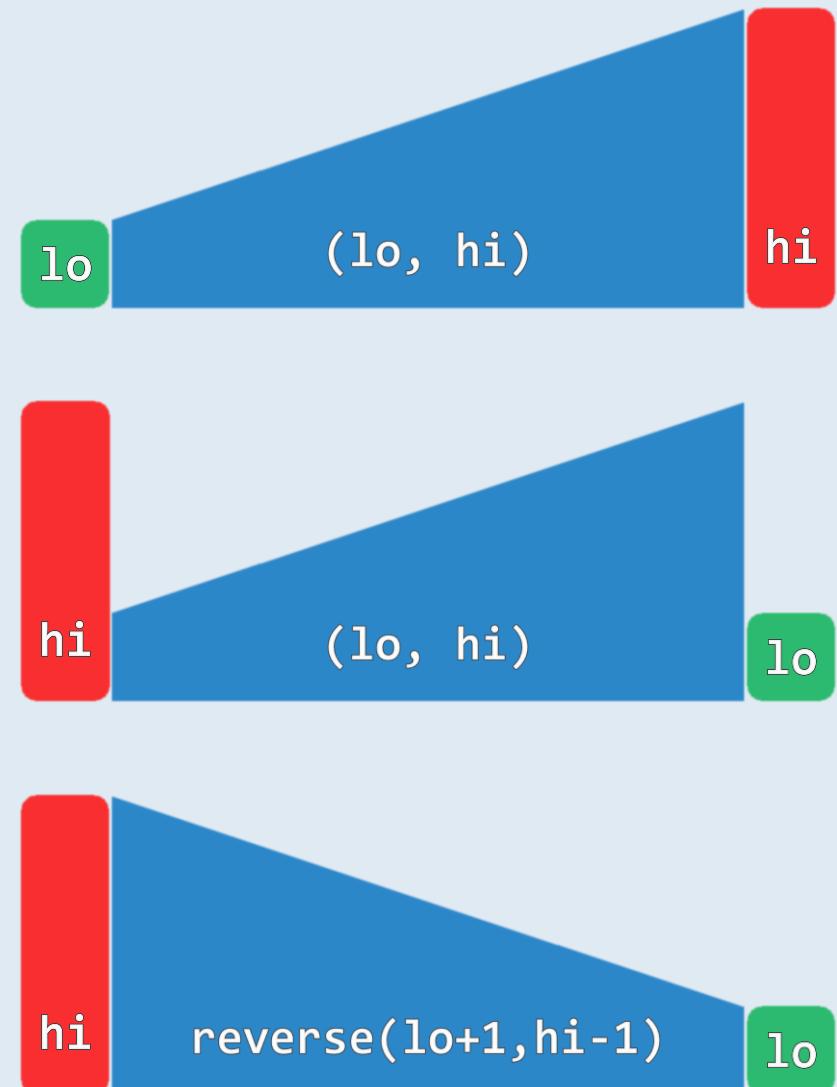
`swap(A[lo], A[hi]);`

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

`} //线性递归 (尾递归) , $\mathcal{O}(n)$`

❖ `while (lo < hi) //迭代版`

`swap(A[lo++], A[hi--]); //亦是 $\mathcal{O}(n)$`



绪论

迭代与递归：分而治之



邓俊辉

deng@tsinghua.edu.cn

凡治众如治寡，分数是也

Divide-and-Conquer

❖ 为求解一个大规模的问题，可以...

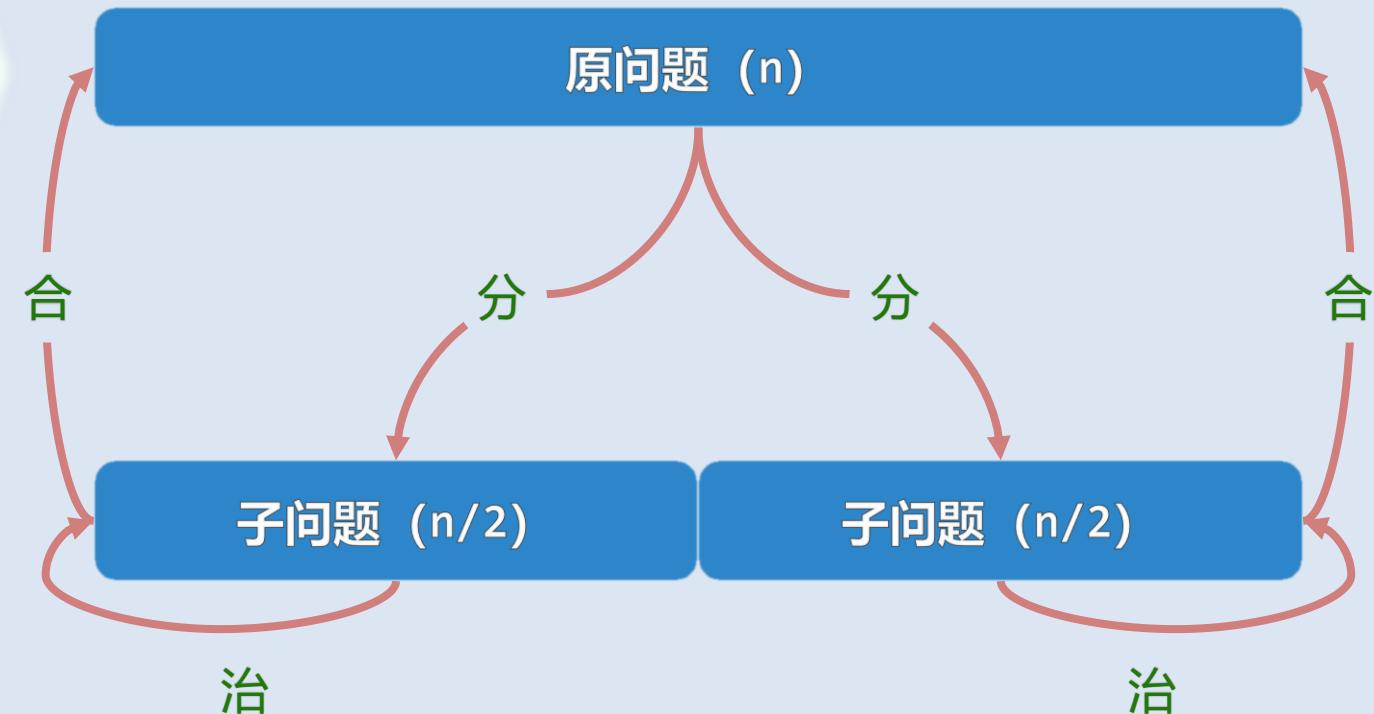
❖ 将其划分为若干子问题

(通常两个，且规模**大体相当**)

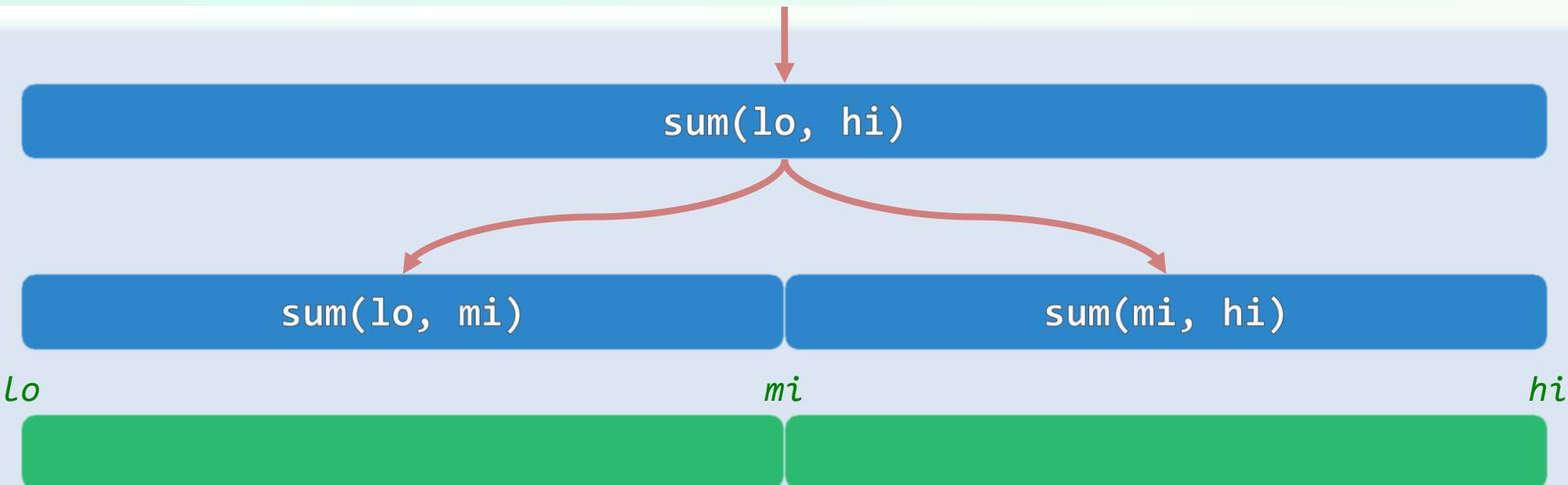
❖ 分别求解子问题

❖ 由子问题的解

合并得到原问题的解

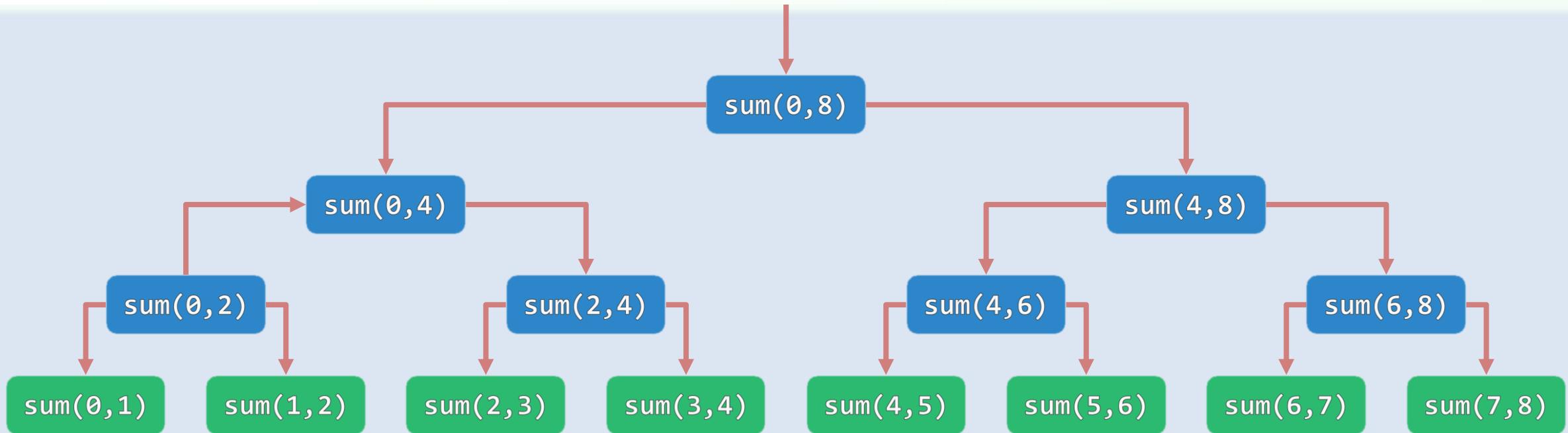


Binary Recursion



```
sum( int A[], int lo, int hi ) { //区间范围A[lo, hi)  
    if ( hi - lo < 2 ) return A[lo];  
  
    int mi = (lo + hi) >> 1;    return sum( A, lo, mi ) + sum( A, mi, hi );  
} //入口形式为sum( A, 0, n )
```

Binary Recursion: Trace



$T(n) = \text{各层递归实例所需时间之和} \quad // \text{递归跟踪}$

$$= \mathcal{O}(1) \times (2^0 + 2^1 + 2^2 + \dots + 2^{\log n})$$

$$= \mathcal{O}(1) \times (2^{1+\log n} - 1) = \mathcal{O}(n) \quad // \text{更快捷地, 作为几何级数...}$$

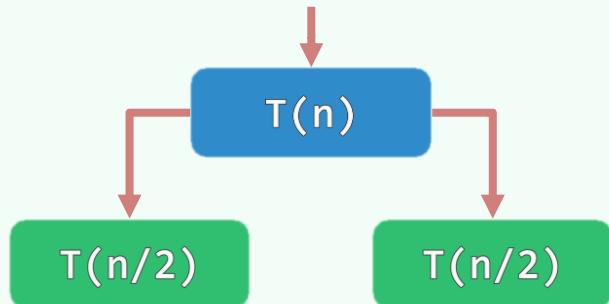
Binary Recursion: Recurrence

❖ 从递推的角度看，为求解 $\text{sum}(A, \text{lo}, \text{hi})$ ，需要

- 递归求解 $\text{sum}(A, \text{lo}, \text{mi})$ 和 $\text{sum}(A, \text{mi}+1, \text{hi})$ ，进而 $//2*T(n/2)$

- 将子问题的解累加 $//\mathcal{O}(1)$

❖ 递推方程： $T(n) = 2 \cdot T(n/2) + \mathcal{O}(1)$



$$T(1) = \mathcal{O}(1) \quad //\text{base: } \text{sum}(A, k, k)$$

❖ 求解： $T(n) = 4 \cdot T(n/4) + \mathcal{O}(3) = 8 \cdot T(n/8) + \mathcal{O}(7) = 16 \cdot T(n/16) + \mathcal{O}(15) = \dots$

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

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、解的合并总共耗时 $f(n)$ ）

❖ 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)**: $T(n) = 1 \cdot T(n/2) + \mathcal{O}(n) = \mathcal{O}(n)$

Multiplication: Naive + DAC

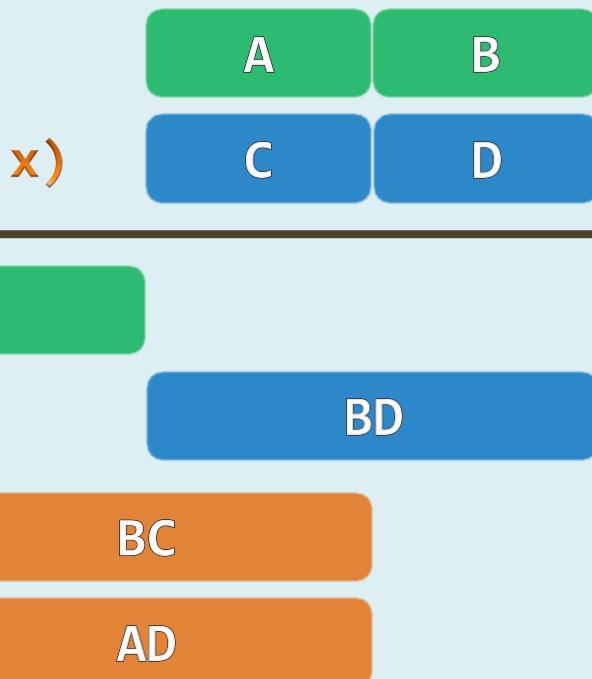
$$2n = n \times n$$

$$T(n) = 4 \cdot T(n/2) + \mathcal{O}(n)$$

$$= \mathcal{O}(n^{\log_2 4})$$

$$= \mathcal{O}(n^2)$$

$$B \times C + A \times D = \dots$$



Multiplication: Optimal

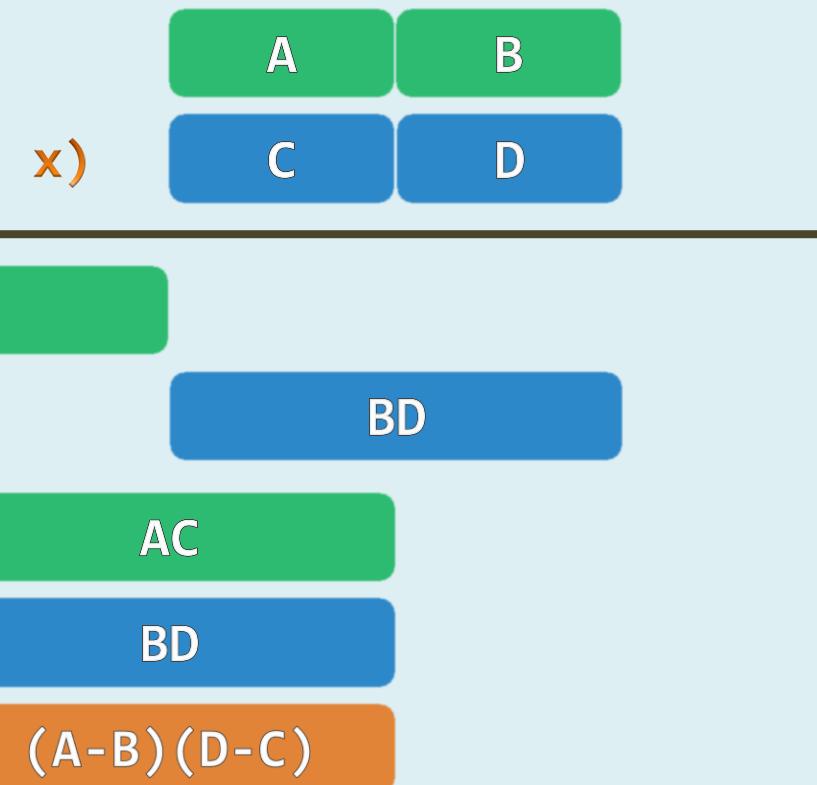
$$2n = n \times n$$

$$T(n) = 3 \cdot T(n/2) + \mathcal{O}(n)$$

$$= \mathcal{O}(n^{\log_2 3})$$

$$\approx \mathcal{O}(n^{1.585})$$

$$B \times C + A \times D = A \times C + B \times D + (A - B) \times (D - C)$$



绪论

迭代与递归：总和最大区段

θ₁ - E₃

这时，公共智慧的结果便产生理智与意志在社会体中的结合，也才有了各个部分的密切配合，以及最后全体的最大力量。

邓俊辉

deng@tsinghua.edu.cn

问题 + 蛮力算法

❖ 从整数序列中，找出总和最大的区段（有多个时，短者、靠后者优先）

$$A[0, 19) = \{ 1, -2, \underline{7, 2, 6}, -9, 5, 6, -12, -8, \underline{13, 0, -3}, 1, -2, \underline{8, 0}, -5, 3 \}$$

```
int gs_BF( int A[], int n ) { //蛮力策略:  $\Theta(n^3)$ 
    int gs = A[0]; //当前已知的最大和
    for ( int i = 0; i < n; i++ )
        for ( int j = i; j < n; j++ ) { //枚举所有的 $\Theta(n^2)$ 个区段!
            int s = 0;
            for ( int k = i; k <= j; k++ ) s += A[k]; //用 $\Theta(n)$ 时间求和
            if ( gs < s ) gs = s; //择优、更新
        }
    return gs;
}
```



递增策略

```
int gs_IC( int A[], int n ) { //Incremental Strategy:  $\mathcal{O}(n^2)$ 
```

```
    int gs = A[0]; //当前已知的最大和
```

```
    for ( int i = 0; i < n; i++ ) { //枚举所有起始于i
```

```
        int s = 0;
```

```
        for ( int j = i; j < n; j++ ) { //终止于j的区间
```

```
            s += A[j]; //递增地得到其总和:  $\mathcal{O}(1)$ 
```

```
            if ( gs < s ) gs = s; //择优、更新
```

```
}
```

```
}
```

```
return gs;
```



分而治之：前缀 + 后缀： $\mathcal{A}[lo, hi) = \mathcal{A}[lo, mi) \cup \mathcal{A}[mi, hi) = \mathcal{P} \cup \mathcal{S}$

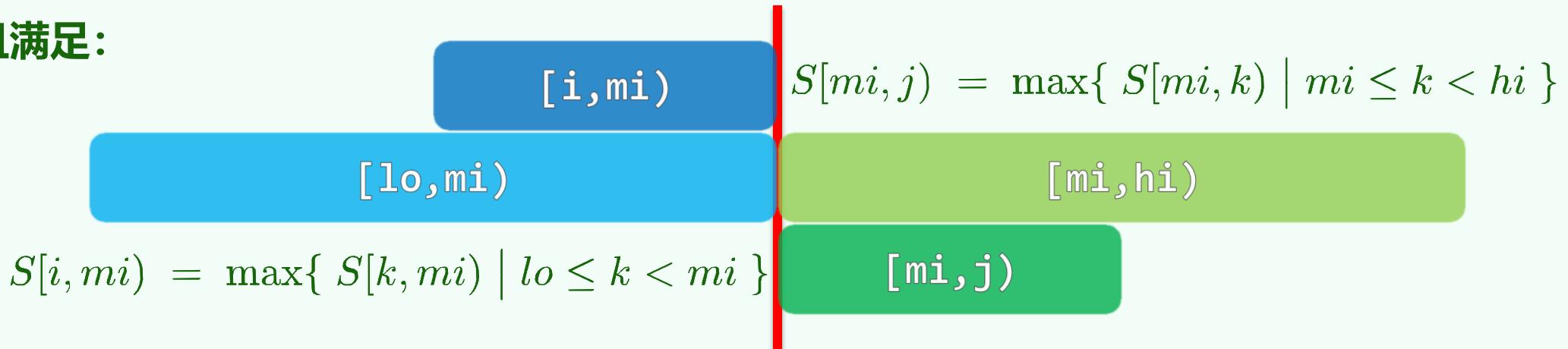
❖ 借助递归，便可求得 \mathcal{P} 、 \mathcal{S} 内部的GS；而剩余的实质任务无非是...

考察那些**跨越切分线**的区段...

❖ 沿着切分线，这类区段必被分割为**非空**的前缀、后缀：

$$\mathcal{A}[i, j) = \mathcal{A}[i, mi) + \mathcal{A}[mi, j), \quad i < mi < j$$

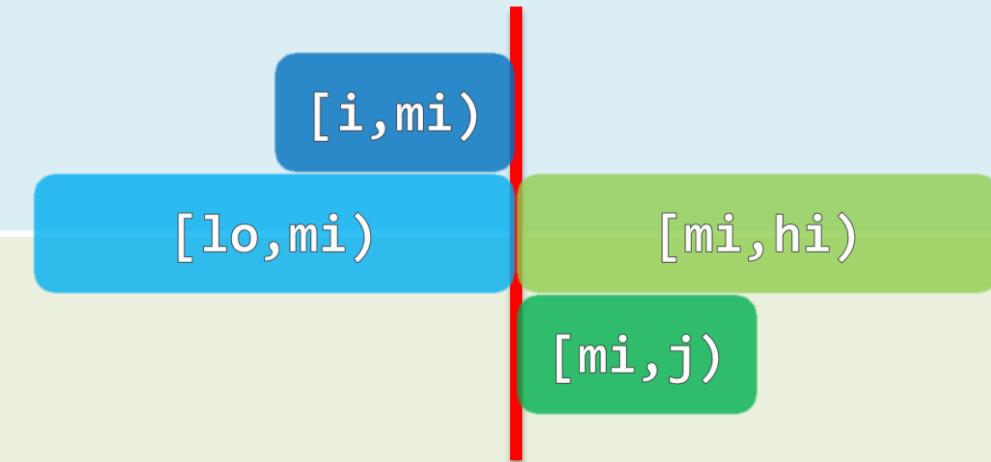
而且满足：



❖ 可见，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 $\mathcal{O}(n \cdot \log n)$

分治策略：实现

```
int gs_DC( int A[], int lo, int hi ) { //Divide-And-Conquer: O(n*logn)
    if ( hi - lo < 2 ) return A[lo]; //递归基
    int mi = (lo + hi) / 2; //在中点切分
    int gsL = A[mi-1], sL = 0, i = mi; //枚举
    while ( lo < i-- ) //所有[i, mi)类区段
        if ( gsL < (sL += A[i]) ) gsL = sL; //更新
    int gsR = A[mi], sR = 0, j = mi-1; //枚举
    while ( ++j < hi ) //所有[mi, j)类区段
        if ( gsR < (sR += A[j]) ) gsR = sR; //更新
    return max( gsL + gsR, max( gs_DC(A, lo, mi), gs_DC(A, mi, hi) ) ); //递归
}
```



减治策略：最短的总和非正的后缀 ~ 总和最大区段

$$\text{suffix}(k) = \mathcal{A}[k, hi], k = \max\{lo \leq i < hi \mid \text{sum}[i, hi] \leq 0\}$$

$$GS(lo, hi) = \mathcal{A}[i, j]$$

❖ 后者要么是前者的（真）后缀，要么与前者无交

lo [lo, hi) hi

❖ [反证]

假若二者确有非空的公共部分：

$$\mathcal{A}[k, j), k < j < hi$$

i k j

$$GS(lo, hi) = \mathcal{A}[i, j)$$

$$\text{sum}(k, hi) \leq 0$$

$$\text{sum}(k, j) > 0$$

$$\text{sum}(j, hi) < 0$$

❖ 由 $GS[lo, hi]$ 的最大、最短性，必有

$\text{sum}[k, j) > 0$ ，即 $\text{sum}[j, hi) < 0$ ——这与 $\mathcal{A}[k, hi)$ 的最短性矛盾

❖ 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...

减治策略：实现

```
int gs_LS( int A[], int n ) { //Linear Scan:  $\Theta(n)$ 
```

```
    int gs = A[0], s = 0, i = n;
```

```
    while ( 0 < i-- ) { //在当前区间内
```

```
        s += A[i]; //递增地累计总和
```

```
        if ( gs < s ) gs = s; //并择优、更新
```

```
        if ( s <= 0 ) s = 0; //剪除负和后缀
```

```
}
```

```
return gs;
```

```
}
```

to be scanned

[θ, n)

绪论

动态规划：记忆法

01

-F1

圣人不记事，所以常记得；今人忘事，以其记事

有人建议不妨置备一本签名簿，供来访者留下自己的名字，就像怀特山那样；可是，天哪！我的记性非常好，用不着那个玩意儿。

邓俊辉

deng@tsinghua.edu.cn

fib(): 递归

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



```
int fib(n) { return (2 > n) ? n : fib(n - 1) + fib(n - 2); } //为何这么慢?
```

❖ 复杂度: $T(0) = T(1) = 1$; $T(n) = T(\boxed{n - 1}) + T(\boxed{n - 2}) + 1$, $\forall n > 1$

- 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 $\phi = (1 + \sqrt{5})/2 \approx 1.618$

封底估算

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

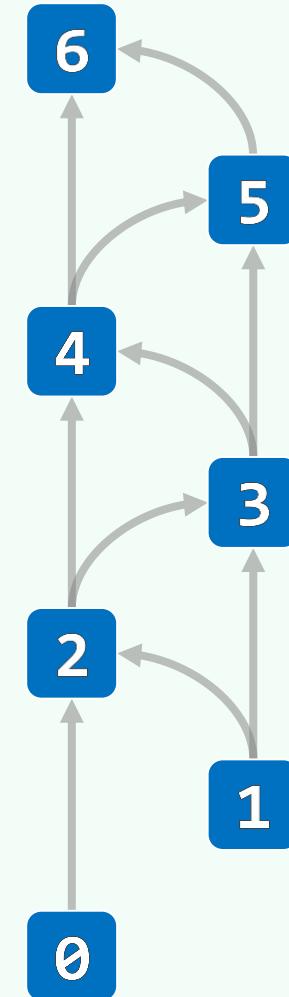
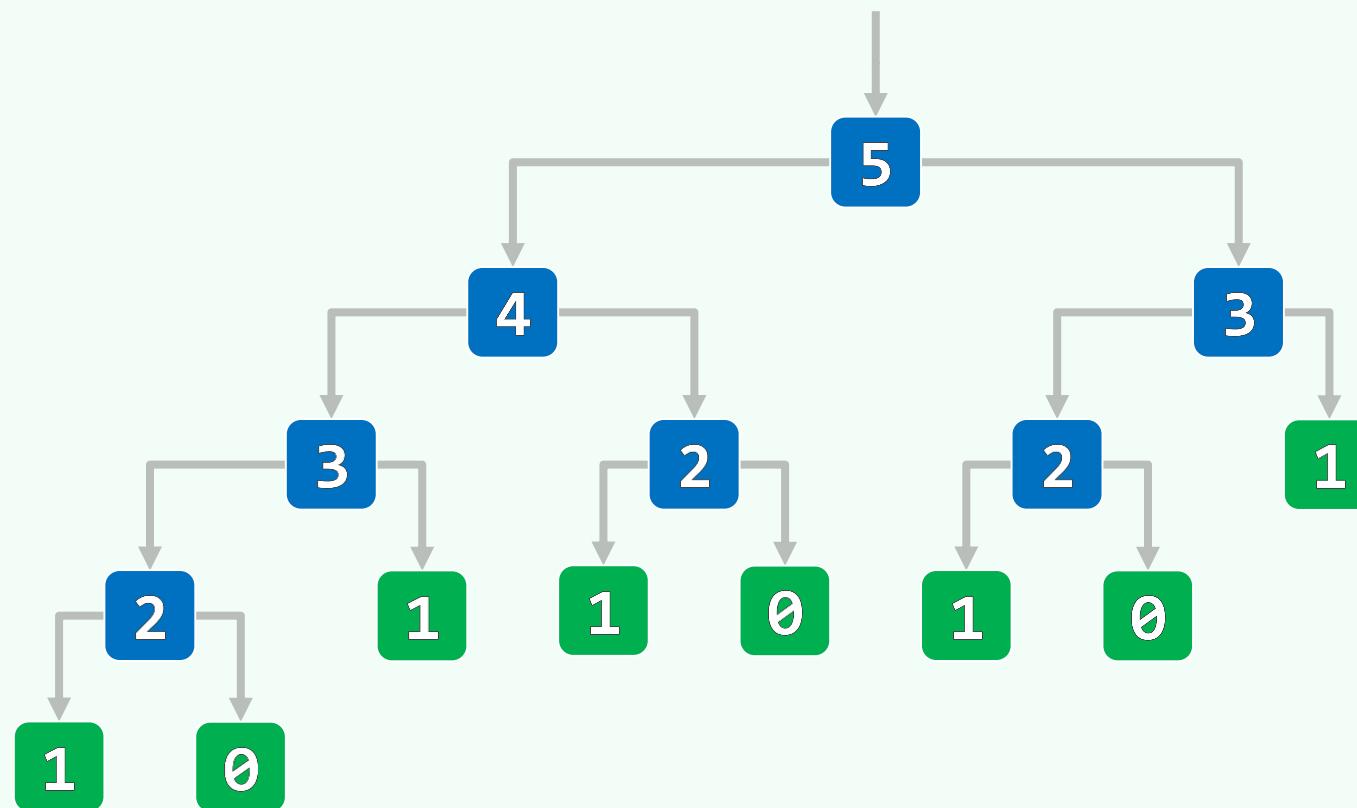
$$\phi^5 \approx 10 \quad \phi^{67} \approx 10^{14} \text{ flo} = 10^5 \text{ sec} \approx 1 \text{ day}$$

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

递归

❖ 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



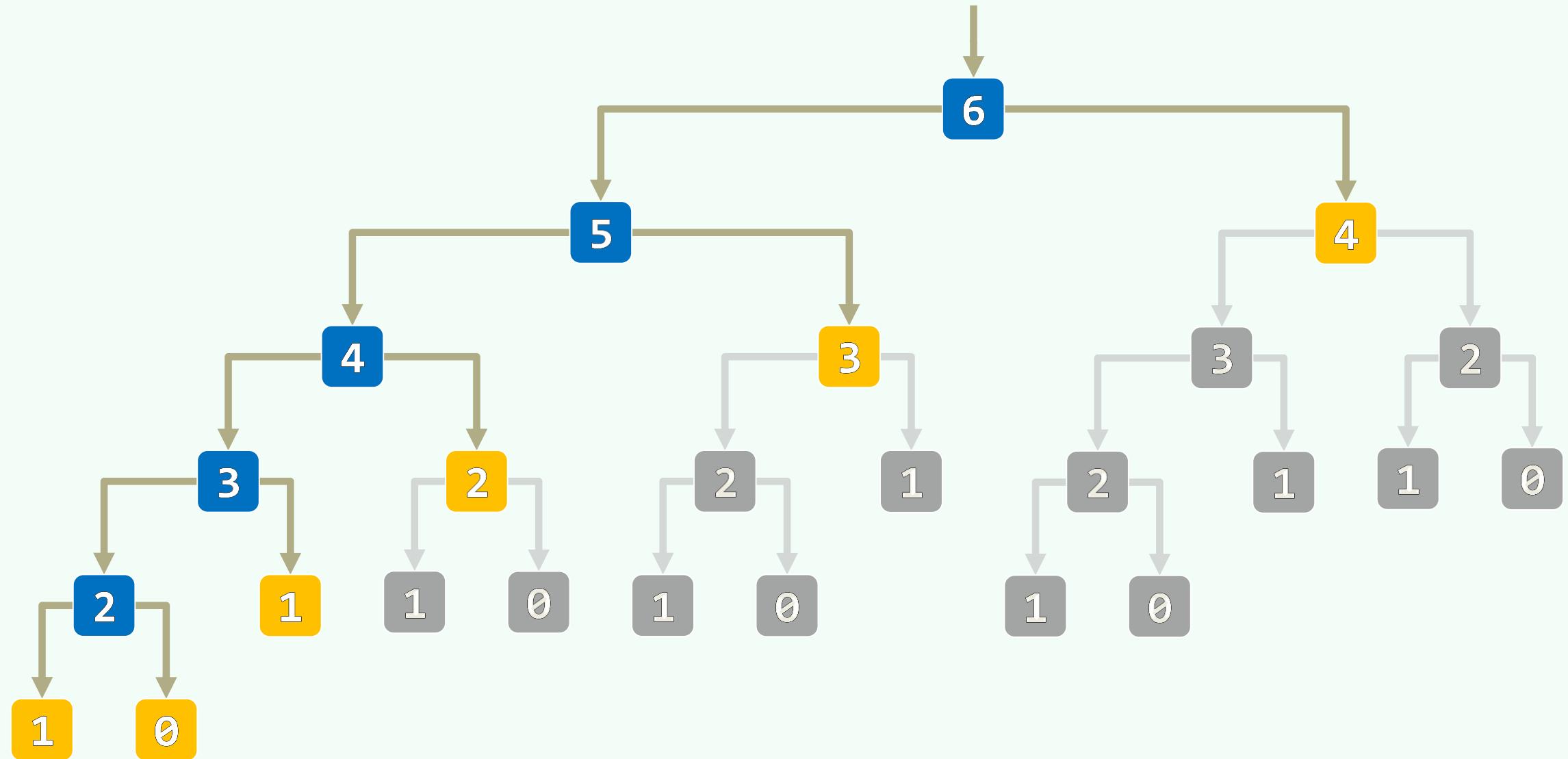
Memoization

```
def f(n)
    if ( n < 1 ) return trivial( n );
    return f(n-X) + f(n-Y)*f(n-Z);
```

```
T M[ N ]; #init. with UNDEFINED

def f(n)
    if ( n < 1 ) return trivial( n );
    # recur 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];
```

Memoization



动态规划

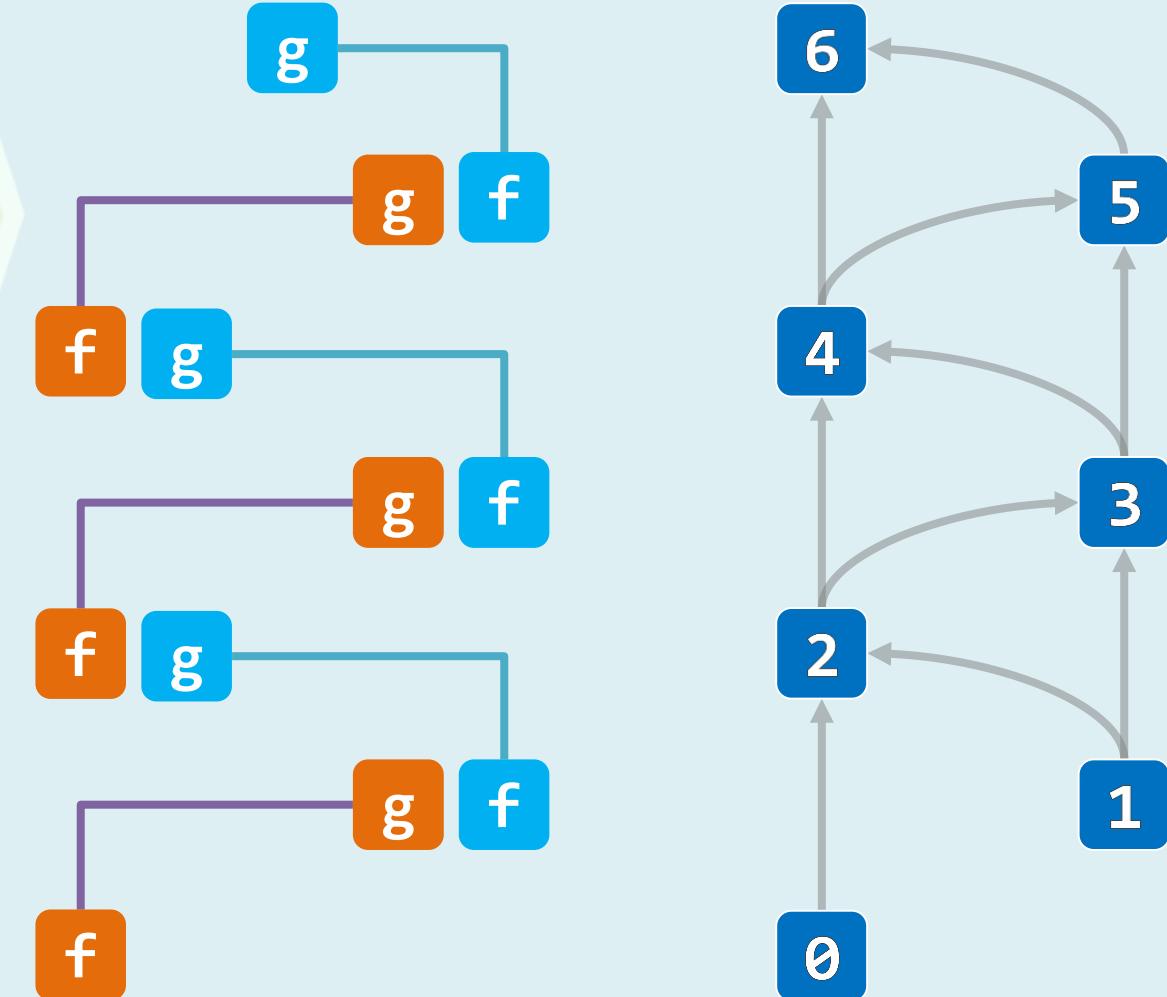
❖ Dynamic programming, 颠倒计算方向:

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;
```

❖ $T(n) = \Theta(n)$, 而且仅需 $\Theta(1)$ 空间!



绪论

e1

-F2

世上一切都无独有偶，为什么你与我却否？

Make it work, make it right, make it fast.

- Kent Beck

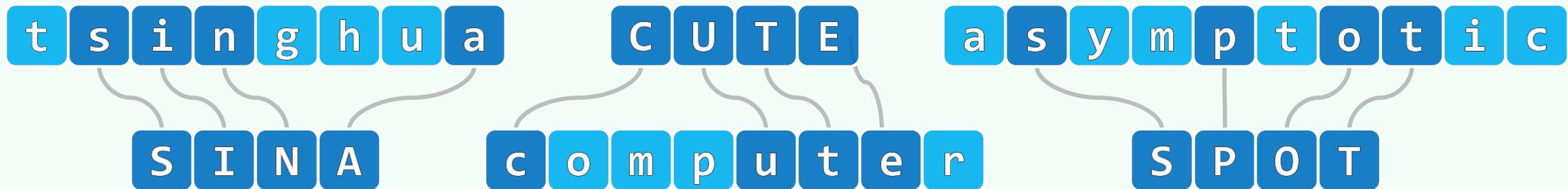
动态规划：最长公共子序列

邓俊辉

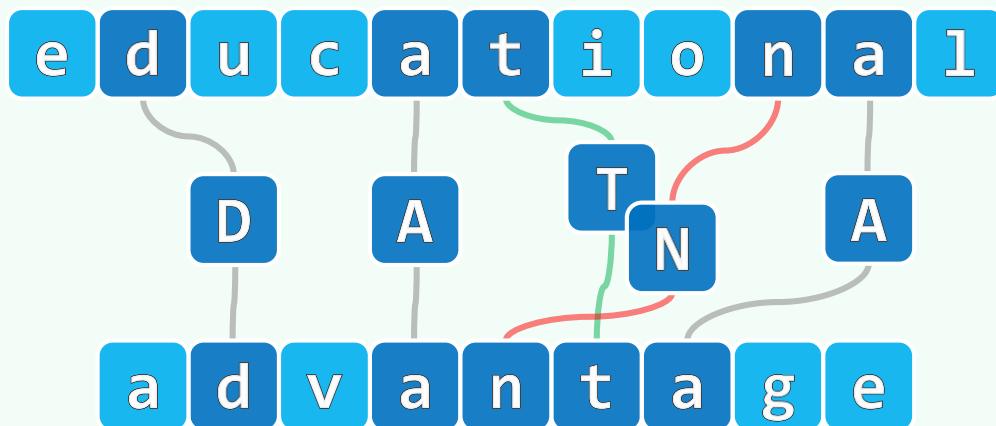
deng@tsinghua.edu.cn

问题定义

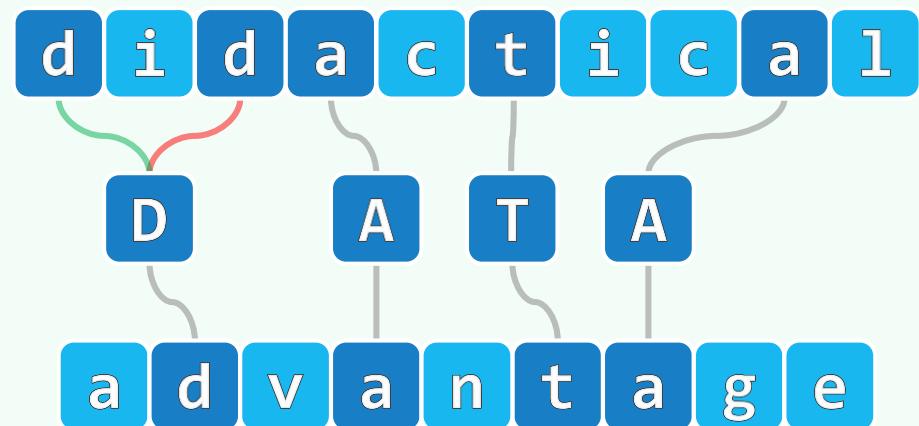
❖ 子序列 (Subsequence) : 由序列中若干字符, 按原相对次序构成



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

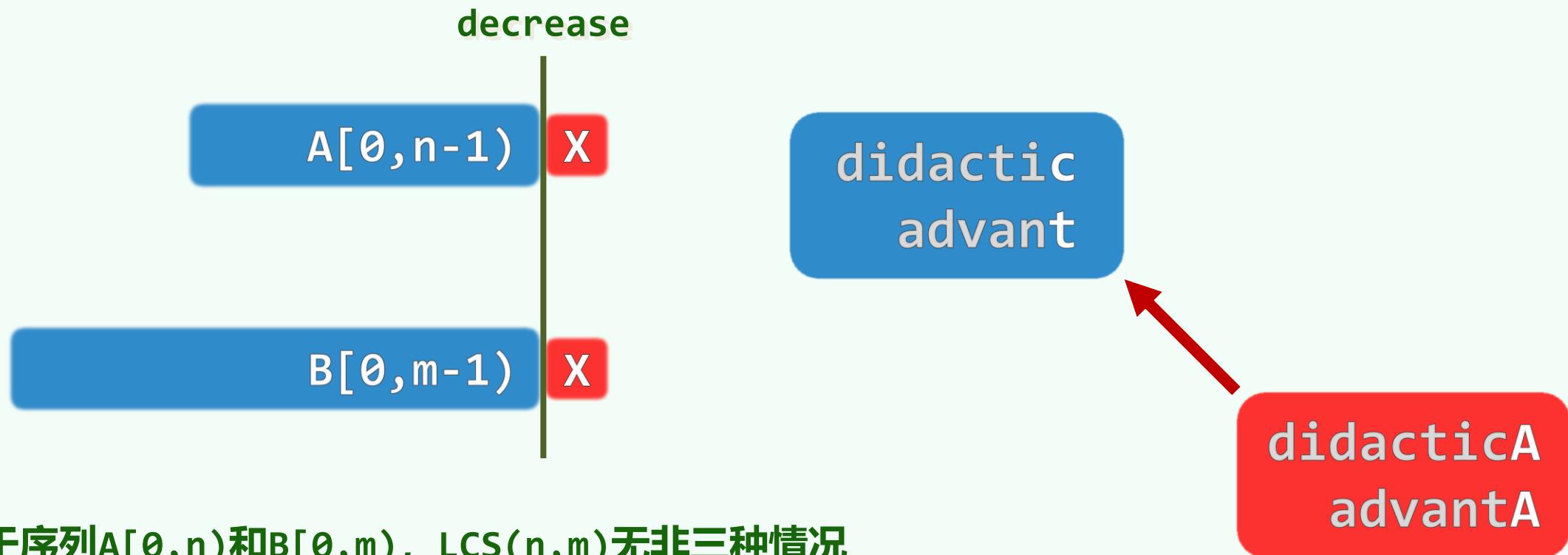


可能有多个



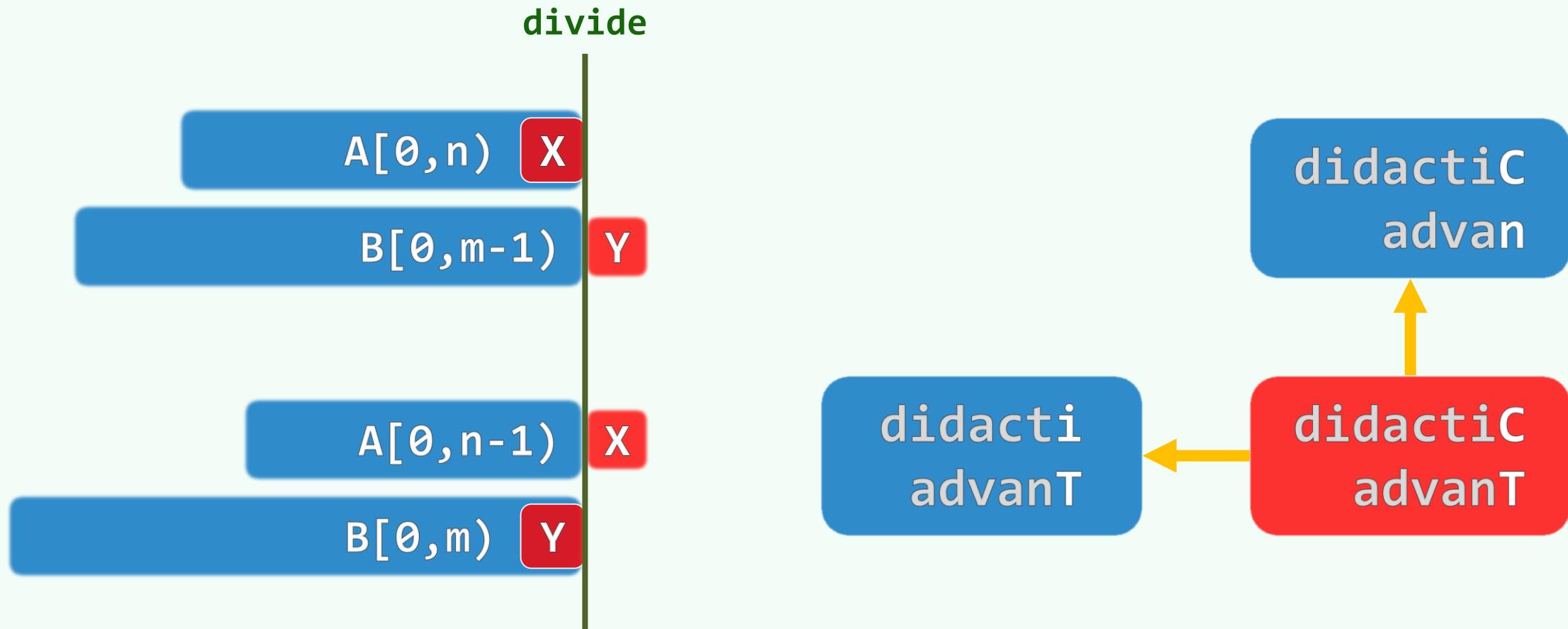
可能有歧义

减治递归



- 0) 若 $n = 0$ 或 $m = 0$, 则取作空序列 (长度为零) // 递归基: 必然总能抵达
- 1) 若 $A[n-1] = 'X' = B[m-1]$, 则取作: $LCS(n-1, m-1) + 'X'$

分治递归



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

描述：伪代码

Input: two strings A and B of length n and m resp.,

Output: (the length of) the longest common subsequence of A and B

lcs(A[], n, B[], m)

Compare the last characters of A and B, i.e., A[n-1] and B[m-1]

If A[n-1] = B[m-1]

Compute x = **lcs(A, n-1, B, m-1)** recursively and return 1 + x

Else

Compute x = **lcs(A, n-1, B, m)** & y = **lcs(A, n, B, m-1)** and return max(x, y)

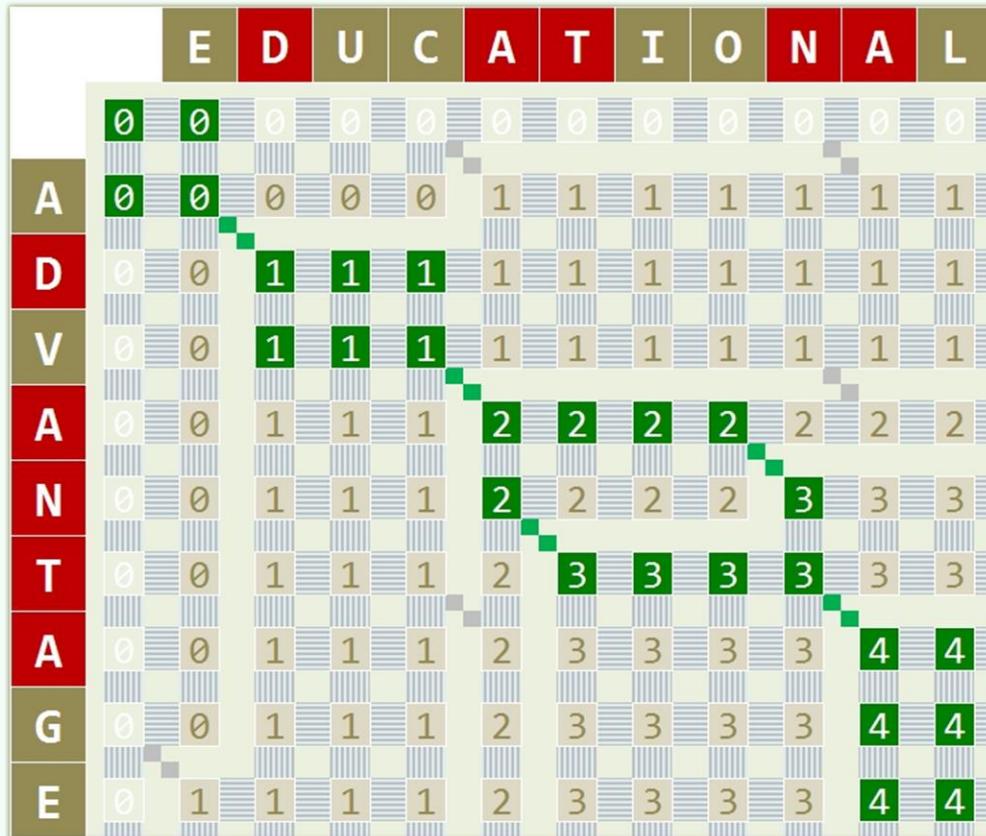
As the recursion base, return 0 when either n or m is 0

实现：递归版

```
unsigned int lcs( char const * A, int n, char const * B, int m ) {  
    if (n < 1 || m < 1) //trivial cases  
        return 0;  
  
    else if ( A[n-1] == B[m-1] ) //decrease & conquer  
        return 1 + lcs(A, n-1, B, m-1);  
  
    else //divide & conquer  
        return max( lcs(A, n-1, B, m), lcs(A, n, B, m-1) );  
}
```

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



多解



歧义

复杂度

◆ 单调性：每经一次比对，至少一个序列的长度缩短一个单位

◆ 最好情况，只需 $\mathcal{O}(n + m)$ 时间 // 比如...

◆ 然而最坏情况下，子问题数量不仅会增加，且可能大量雷同

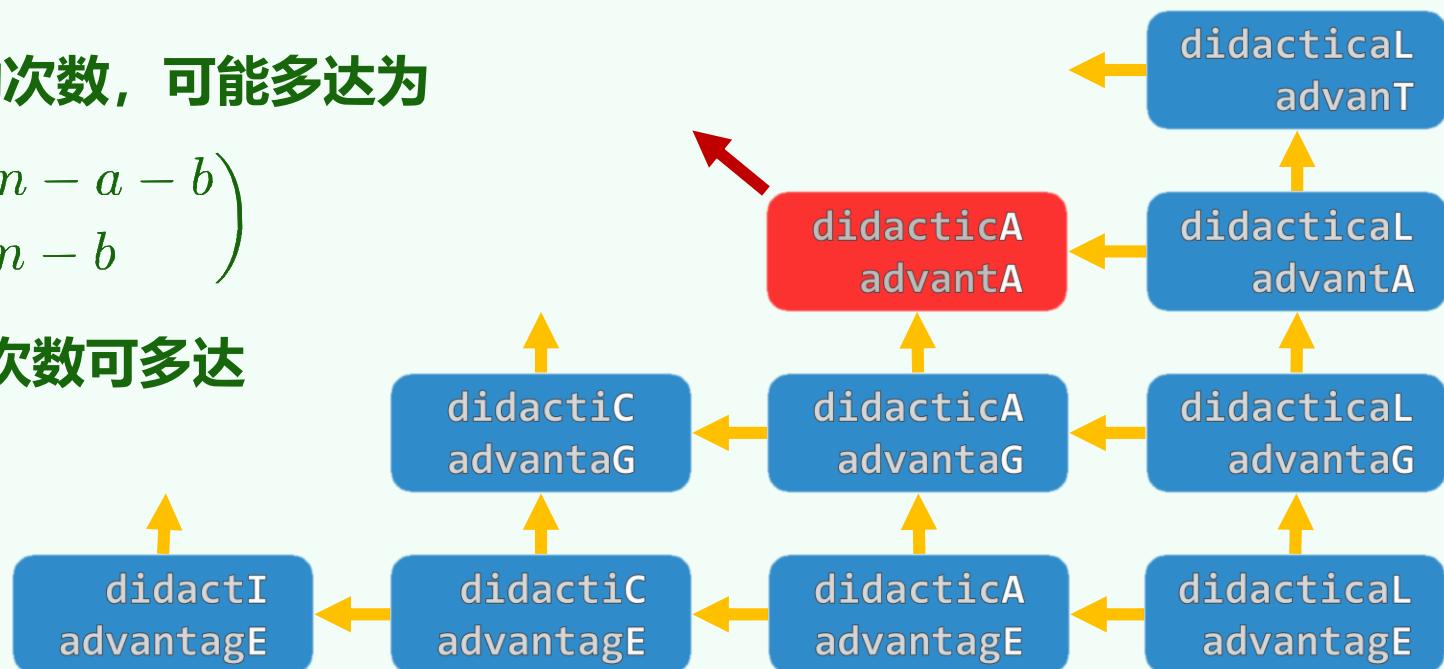
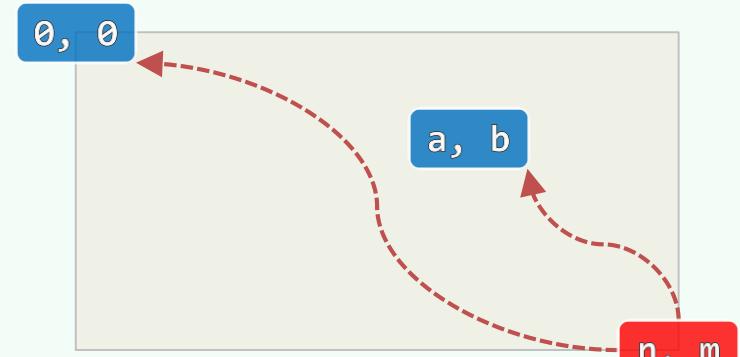
子任务 $\text{LCS}(A[a], B[b])$ 重复的次数，可能多达为

$$\binom{n+m-a-b}{n-a} = \binom{n+m-a-b}{m-b}$$

特别地， $\text{LCS}(A[0], B[0])$ 的次数可多达

$$\binom{n+m}{n} = \binom{n+m}{m}$$

当 $n = m$ 时，为 $\Omega(2^n)$



实现：记忆化版 (1/2)

```
unsigned int lcsMemo(char const* A, int n, char const* B, int m) {  
  
    unsigned int * lcs = new unsigned int[n*m]; //lookup-table of sub-solutions  
  
    memset(lcs, 0xFF, sizeof(unsigned int)*n*m); //initialized with n*m UINT_MAX's  
  
    unsigned int solu = lcsM(A, n, B, m, lcs, m);  
  
    delete[] lcs;  
  
    return solu;  
}
```

实现：记忆化版 (2/2)

```
unsigned int lcsM( char const * A, int n, char const * B, int m,  
                      unsigned int * const lcs, int const M ) {  
  
    if (n < 1 || m < 1) return 0; //trivial cases  
  
    if (UINT_MAX != lcs[(n-1)*M + m-1]) return lcs[(n-1)*M + m-1]; //recursion stops  
  
    else return lcs[(n-1)*M + m-1] =  
            (A[n-1] == B[m-1]) ?  
                1 + lcsM(A, n-1, B, m-1, lcs, M)  
                max( lcsM(A, n-1, B, m, lcs, M), lcsM(A, n, B, m-1, lcs, M) );  
}
```

动态规划

❖ 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

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

- 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)

实现：迭代（动态规划）版

```
unsigned int lcs(char const * A, int n, char const * B, int m) {  
    if (n < m) { swap(A, B); swap(n, m); } //make sure m <= n  
    unsigned int* lcs1 = new unsigned int[m+1]; //the current two rows are  
    unsigned int* lcs2 = new unsigned int[m+1]; //buffered alternatively  
    memset(lcs1, 0x00, sizeof(unsigned int) * (m+1));  
    memset(lcs2, 0x00, sizeof(unsigned int) * (m+1));  
    for (int i = 0; i < n; swap(lcs1, lcs2), i++)  
        for (int j = 0; j < m; j++)  
            lcs2[j+1] = (A[i] == B[j]) ? 1 + lcs1[j] : max(lcs2[j], lcs1[j+1]);  
    unsigned int solu = lcs1[m]; delete[] lcs1; delete[] lcs2; return solu;  
}
```