

排序

快速排序：轴点

14-A7

邓俊辉

deng@tsinghua.edu.cn

左朱雀之芨芨兮，右苍龙之躍躍

分而治之

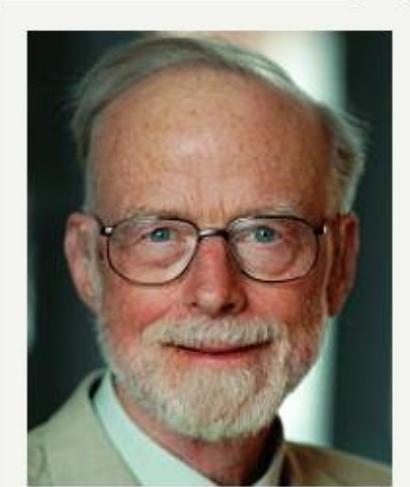
- ❖ pivot: $\max[lo, mi] \leq [mi] \leq \min(mi, hi)$
- ❖ 前缀、后缀各自（递归）排序之后，原序列便自然有序

`sorted(S) = sorted(SL) + pivot + sorted(SR)`

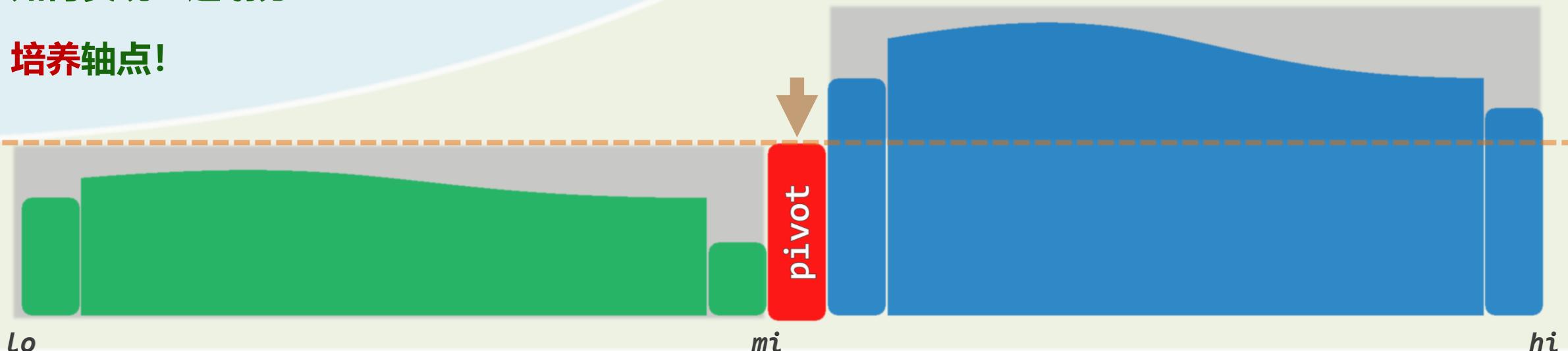
- ❖ mergesort难点在于合，而quicksort在于分

- ❖ 如何实现上述划分？

培养轴点！

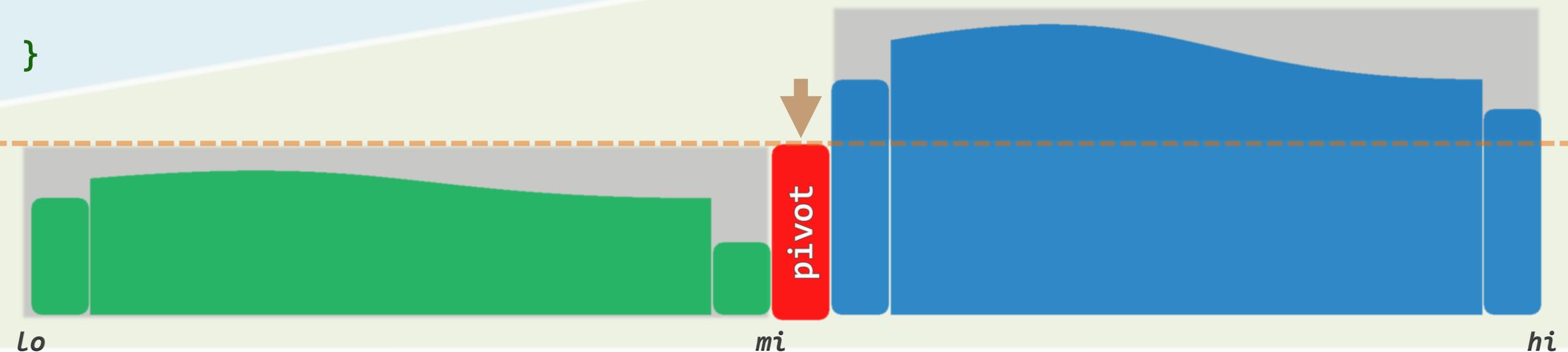


C. A. R. Hoare
(1934 ~)
Turing Award, 1980



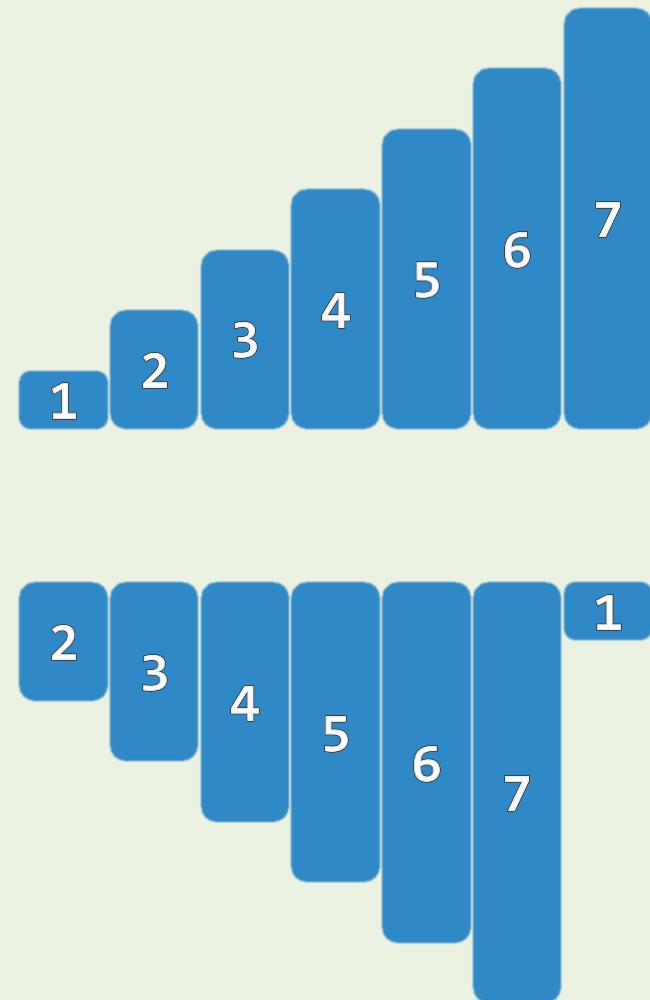
快速排序

```
template <typename T> void Vector<T>::quickSort( Rank lo, Rank hi ) {  
    if ( hi - lo < 2 ) return;  
  
    Rank mi = partition( lo, hi ); //能否足够高效?  
    quickSort( lo, mi );  
    quickSort( mi + 1, hi );  
}
```



轴点

- ❖ 必要条件: 轴点必定已然就位 //尽管反之不然
- ❖ 特别地: 在有序序列中, 所有元素皆为轴点
反之亦然
- ❖ 快速排序: 就是将所有元素逐个转换为轴点的过程
- ❖ 坏消息: 在原始序列中, 轴点未必存在...
- ❖ derangement: 任何元素都不在原位
比如, 顺序序列循环移位
- ❖ 好消息: 不需很多交换, 即可使任一元素转为轴点
- ❖ 问题: 如何交换? 成本多高?



14 - A2

排序

快速排序：快速划分：LUG版

邓俊辉

deng@tsinghua.edu.cn

减而治之，相向而行

❖ 任取一个候选者（如 $[θ]$ ）

❖ L + U + G

❖ 交替地向内移动lo和hi

❖ 逐个检查当前元素：

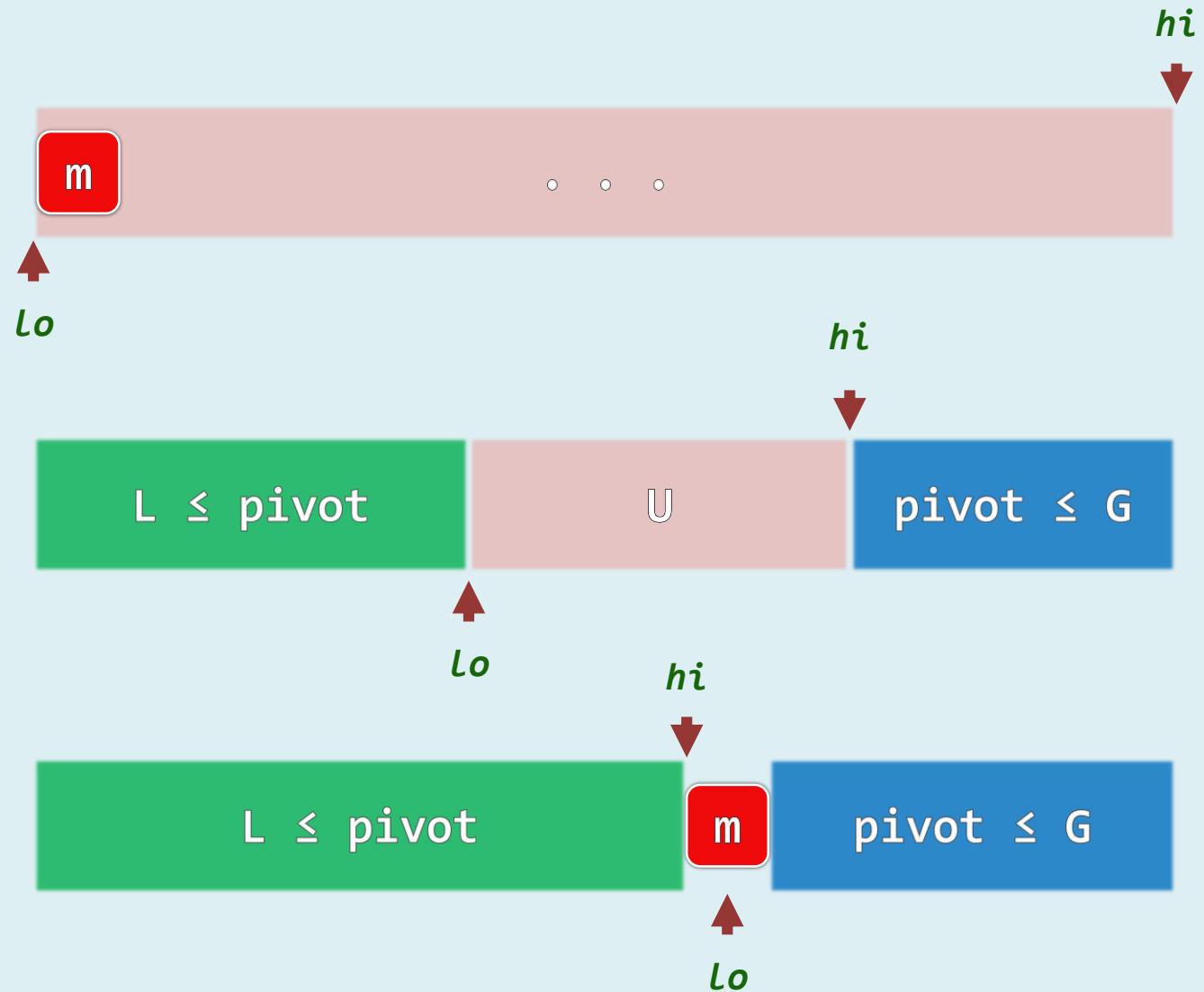
若更小/大，则转移归入L/G

❖ 当 $lo = hi$ 时，只需

将候选者嵌入于L、G之间，即成轴点！

❖ 各元素最多移动一次（候选者两次）

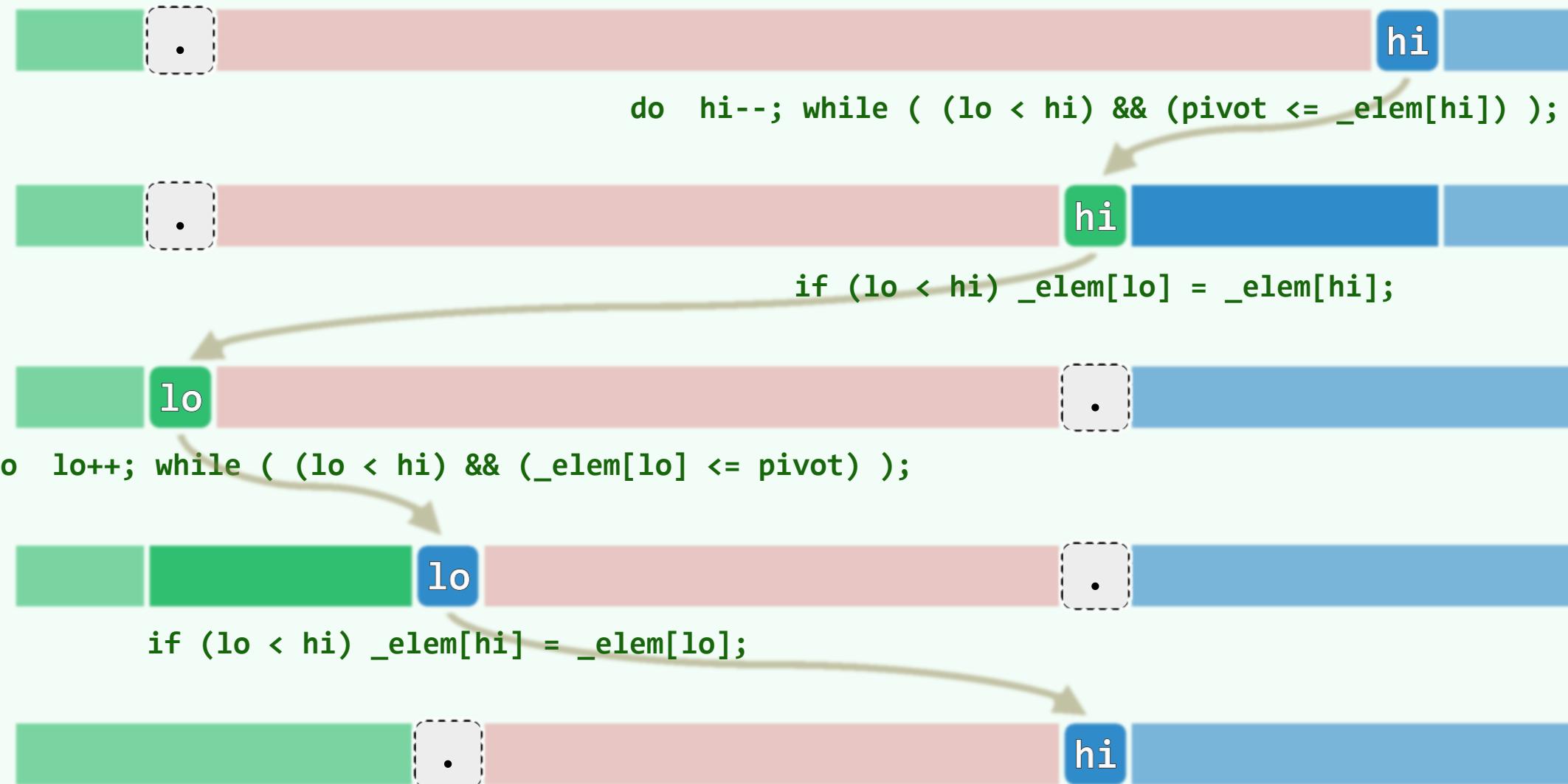
——累计 $O(n)$ 时间、 $O(1)$ 辅助空间



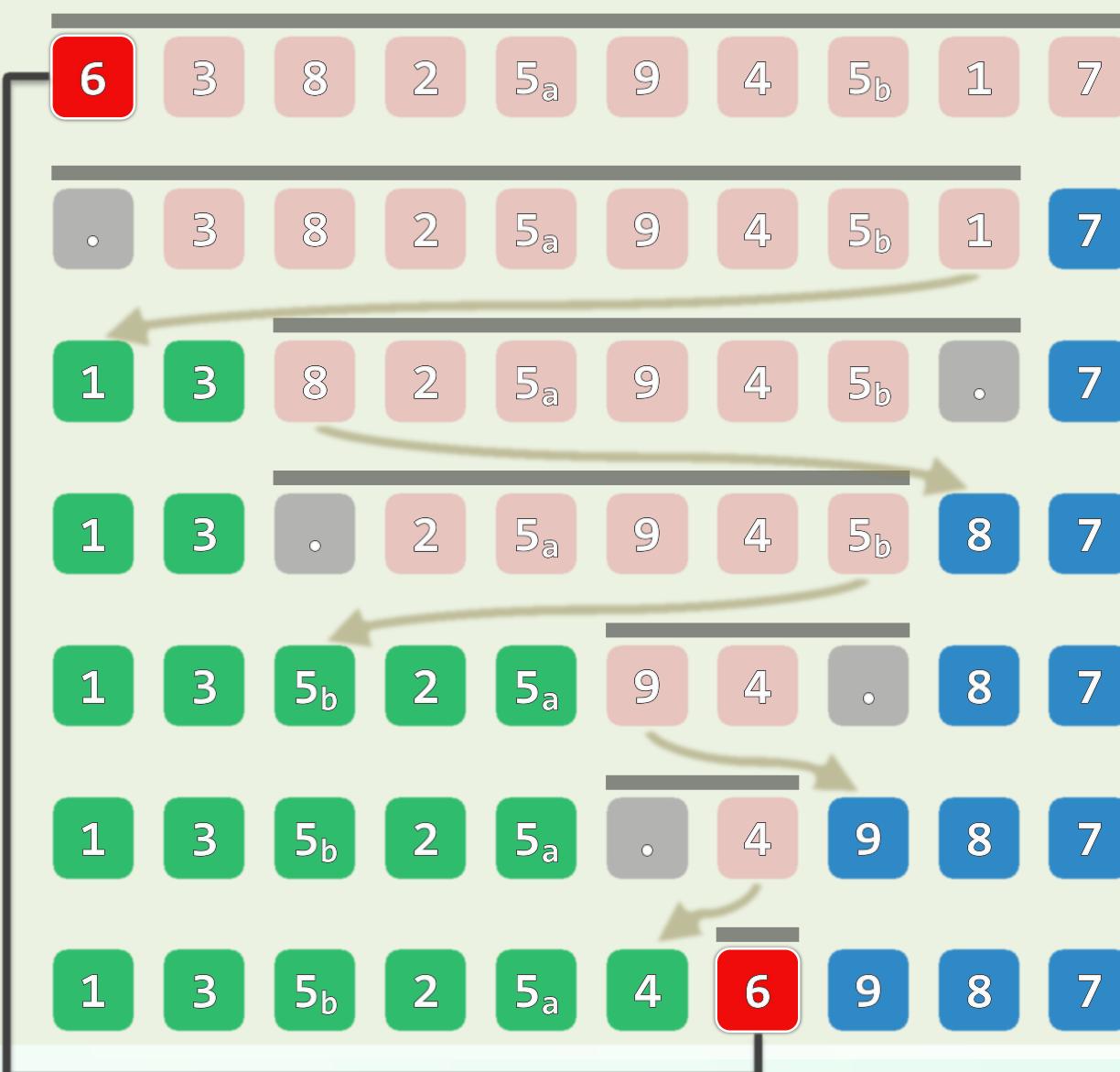
快速划分：LUG版

```
template <typename T> Rank Vector<T>::partition( Rank lo, Rank hi ) { // [lo, hi)
    swap( _elem[lo], _elem[lo + rand() % (hi-lo)] ); // 随机交换
    T pivot = _elem[lo]; // 经以上交换，等效于随机选取候选轴点
    while ( lo < hi ) { // 从两端交替地向中间扫描，彼此靠拢
        do hi--; while ( (lo < hi) && (pivot <= _elem[hi]) ); // 向左拓展G
        if (lo < hi) _elem[lo] = _elem[hi]; // 凡 小于 轴点者，皆归入L
        do lo++; while ( (lo < hi) && (_elem[lo] <= pivot) ); // 向右拓展L
        if (lo < hi) _elem[hi] = _elem[lo]; // 凡 大于 轴点者，皆归入G
    } // assert: lo == hi or hi+1
    _elem[hi] = pivot; return hi; // 候选轴点归位；返回其秩
```

不变性 + 单调性: $L \leq \text{pivot} \leq G$; $U = [\text{lo}, \text{hi})$ 中, $[\text{lo}]$ 和 $[\text{hi}]$ 交替空闲



实例



❖ 线性时间

- 尽管lo、hi交替移动
- 累计移动距离不过 $\Theta(n)$

❖ in-place

- 只需 $\Theta(1)$ 附加空间

❖ unstable

- lo/hi的移动方向相反
- 左/右侧的大/小重复元素

可能前/后颠倒

排序

快速排序：迭代、贪心与随机

14-A3

邓俊辉

deng@tsinghua.edu.cn

瑕不掩瑜，瑜不掩瑕，忠也

察一叶而知天下秋

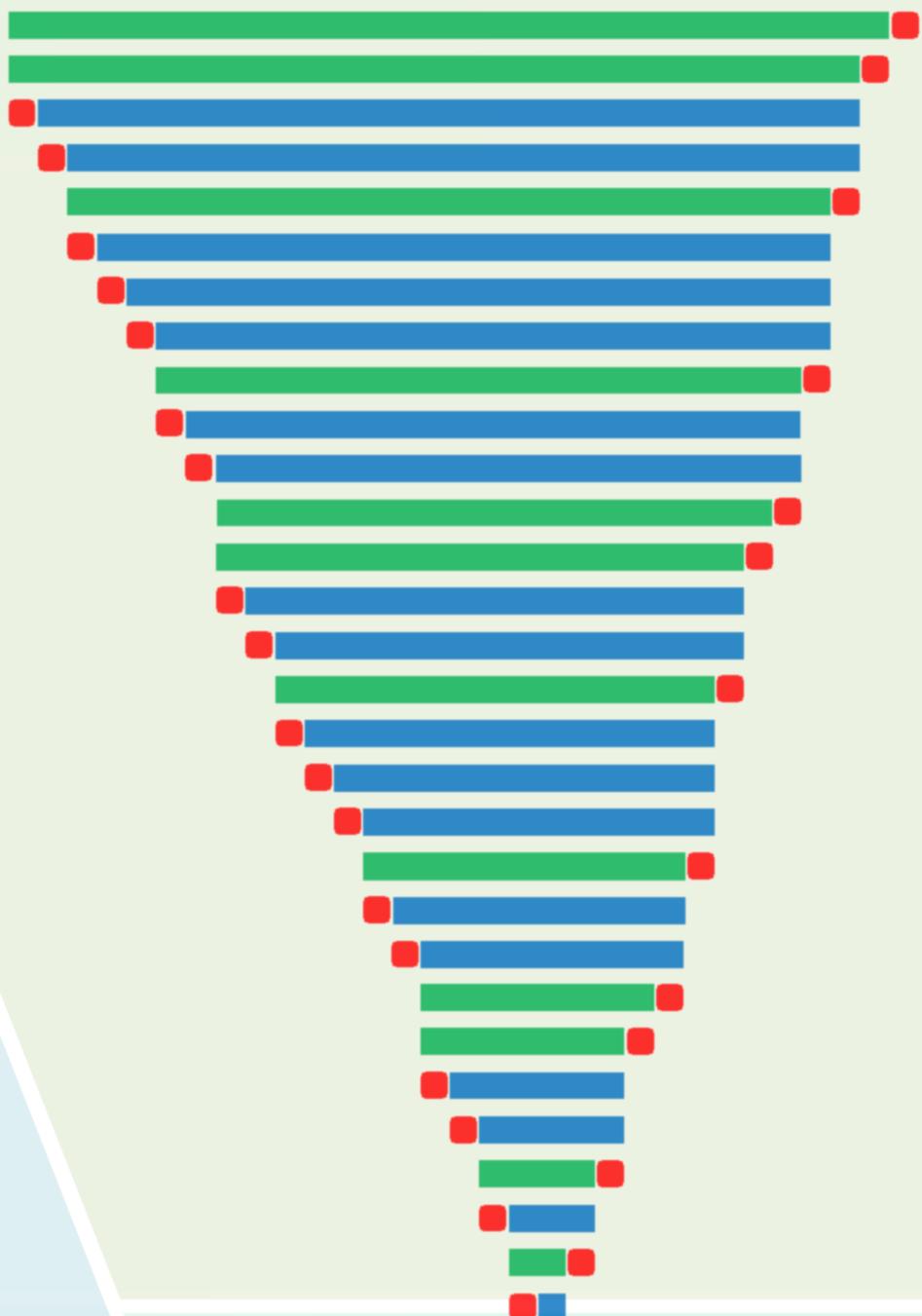
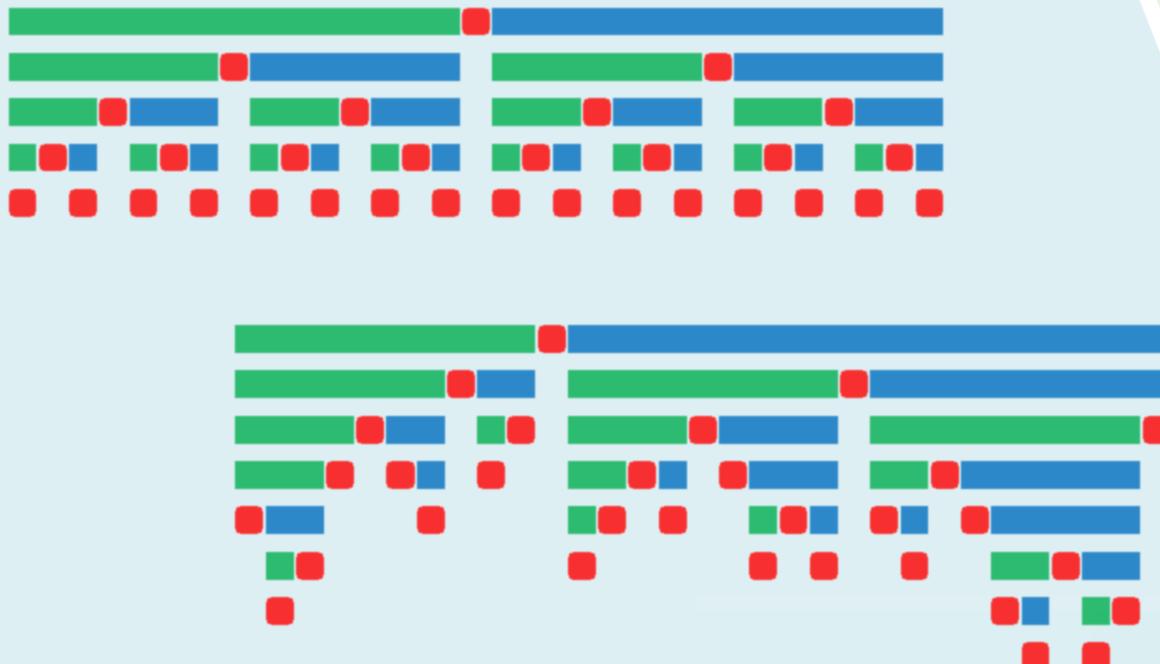
空间复杂度 ~ 递归深度

❖ 最好：划分总均衡 $\mathcal{O}(\log n)$

最差：划分皆偏侧 $\mathcal{O}(n)$

平均：均衡不致太少 $\mathcal{O}(\log n)$

❖ 可否避免最坏情况？如何避免？



非递归 + 贪心

```
#define Put( K, s, t ) { if ( 1 < (t) - (s) ) { K.push(s); K.push(t); } }

#define Get( K, s, t ) { t = K.pop(); s = K.pop(); }

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

Stack<Rank> Task; Put( Task, lo, hi ); //等效于对递归树的先序遍历

while ( !Task.empty() ) {

    Get( Task, lo, hi ); Rank mi = partition( lo, hi );

    if ( mi-lo < hi-mi ) { Put( Task, mi+1, hi ); Put( Task, lo, mi ); }

    else                  { Put( Task, lo, mi ); Put( Task, mi+1, hi ); }

} //大|小任务优先入|出栈，可保证（辅助栈）空间不过 $O(\log n)$ 

}
```

时间性能 + 随机

❖ 最好情况：每次划分都（接近）平均，轴点总是（接近）中央 —— 到达下界！

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

❖ 最坏情况：每次划分都极不均衡（比如，轴点总是最小/大元素）—— 与起泡排序相当！

$$T(n) = T(n-1) + T(0) + \mathcal{O}(n) = \mathcal{O}(n^2)$$

❖ 采用随机选取 (Randomization)、三者取中 (Sampling) 之类的策略

只能降低最坏情况的概率，而无法杜绝 —— 既如此，为何还称作快速排序？

排序

快速排序：递归深度

14 - A4

今夫盲者行于道，人谓之左则左，谓之右则右。遇君子则得其平易，遇小人则蹈于沟壑。

邓俊辉

deng@tsinghua.edu.cn

居中 + 偏侧

❖ 最坏情况 ($\Omega(n)$ 递归深度), 概率极低

平均情况 ($\mathcal{O}(\log n)$ 递归深度), 概率极高

$(1 - \lambda) / 2$

width = λ = Pr.

$(1 - \lambda) / 2$

❖ 实际上: 除非过于侧偏的pivot, 都会有效地缩短递归深度

❖ 准居中: pivot落在宽度为 $\lambda \cdot n$ 的居中区间

(λ 也是这种情况出现的概率)

❖ 每一递归路径上, 至多出现 $\log_{\frac{2}{1+\lambda}} n$ 个准居中的pivots ...



期望深度

- 每递归一层，都有 $\lambda (1 - \lambda)$ 的概率准居中（准偏侧）
- 深入 $\frac{1}{\lambda} \cdot \log_{\frac{2}{1+\lambda}} n$ 层后，即可期望出现 $\log_{\frac{2}{1+\lambda}} n$ 次准居中，且有极高的概率出现

(1 - λ) / 2

width = λ = Pr.

(1 - λ) / 2

相反情况的概率 < $(1 - \lambda)^{(\frac{1}{\lambda} - 1) \cdot \log_{\frac{2}{1+\lambda}} n} = n^{(\frac{1}{\lambda} - 1) \cdot \log_{\frac{2}{1+\lambda}} (1-\lambda)}$

且随着 λ 增加而下降，比如 $\lambda > 1/3$ 之后，即...

至少有 $1 - n^{2 \cdot \log_{\frac{3}{2}} (\frac{2}{3})} = 1 - n^{-2}$ 的概率，使得

递归深度不超过 $\frac{1}{\lambda} \cdot \log_{\frac{2}{1+\lambda}} n = 3 \cdot \log_{\frac{3}{2}} n \approx 5.129 \cdot \log n$



排序

快速排序：比较次数

时间究竟是什么

假使人家不问我，我像很明了

假使要我解释起来，我就茫无头绪

庇拉尔·特尔内拉在这场造梦运动中出力最多，她

成功地将纸牌算命从推演未来应用到追溯过往。

14 - A5

邓俊辉

deng@tsinghua.edu.cn

递推分析 (1/2)

❖ 记期望的比较次数为 $T(n)$: $T(1) = 0, T(2) = 1, \dots$

❖ 可以证明: $T(n) = \mathcal{O}(n \log n)$...



$$T(n) = (n-1) + \frac{1}{n} \cdot \sum_{k=0}^{n-1} [T(k) + T(n-k-1)] = (n-1) + \frac{2}{n} \cdot \sum_{k=0}^{n-1} T(k)$$

$$n \cdot T(n) = n \cdot (n-1) + 2 \times \sum_{k=0}^{n-1} T(k)$$

$$(n-1) \cdot T(n-1) = (n-1) \cdot (n-2) + 2 \times \sum_{k=0}^{n-2} T(k)$$

递推分析 (2/2)

$$n \cdot T(n) - (n-1) \cdot T(n-1) = 2 \cdot (n-1) + 2 \times T(n-1)$$

$$n \cdot T(n) - (n+1) \cdot T(n-1) = 2 \cdot (n-1)$$



$$\frac{T(n)}{n+1} - \frac{T(n-1)}{n} = \frac{4}{n+1} - \frac{2}{n}$$

$$\frac{T(n)}{n+1} = \frac{T(n)}{n+1} - \frac{T(1)}{2} = 4 \cdot \sum_{k=2}^n \frac{1}{k+1} - 2 \cdot \sum_{k=2}^n \frac{1}{k} = 2 \cdot \sum_{k=1}^{n+1} \frac{1}{k} + \frac{2}{n+1} - 4 \approx 2 \cdot \ln n$$

$$T(n) \approx 2 \cdot n \cdot \ln n = (2 \cdot \ln 2) \cdot n \log n \approx 1.386 \cdot n \log n$$

后向分析 (1/2)

❖ 设经排序后得到的输出序列为：

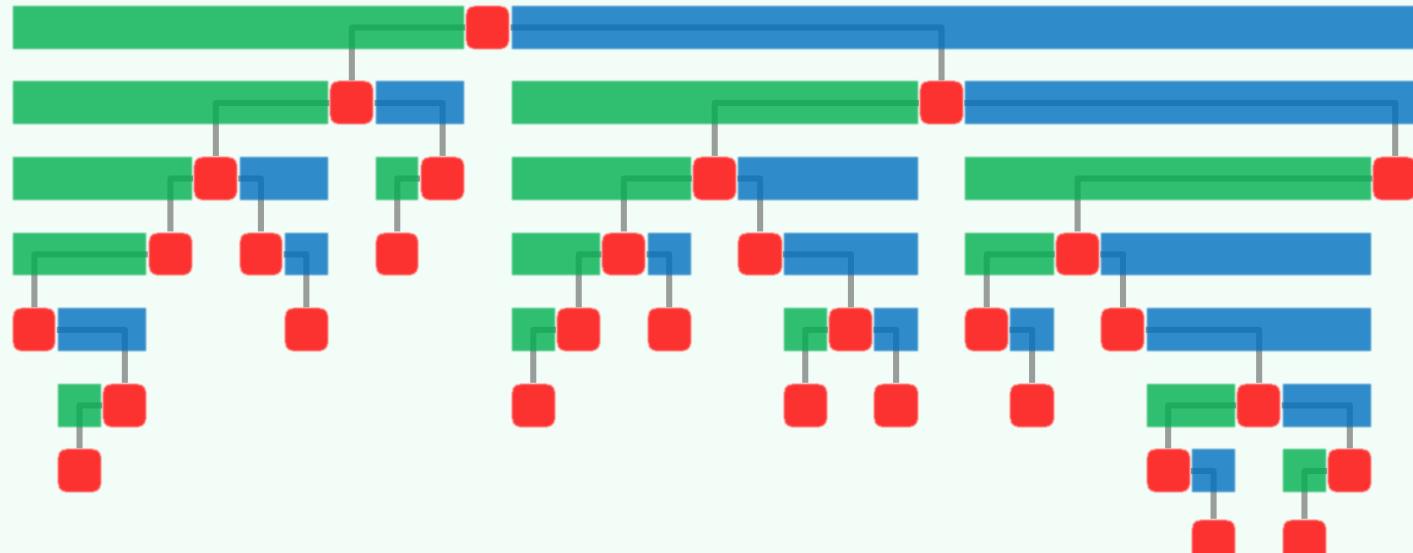
$$\{ a_0, a_1, a_2, \dots, a_i, \dots, a_j, \dots, a_{n-1} \}$$

❖ 这一输出与具体使用何种算法无关

故可使用Backward Analysis

❖ 比较操作的期望次数应为

$$T(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} Pr(i, j)$$



亦即，每一对 $\langle a_i, a_j \rangle$ 在排序过程中接受比较之概率的总和

后向分析 (2/2)

❖ quickSort的过程及结果，可理解为：按某种次序，将各元素逐个确认为pivot

❖ 若 $k \in [0, i) \cup (j, n]$ ，则

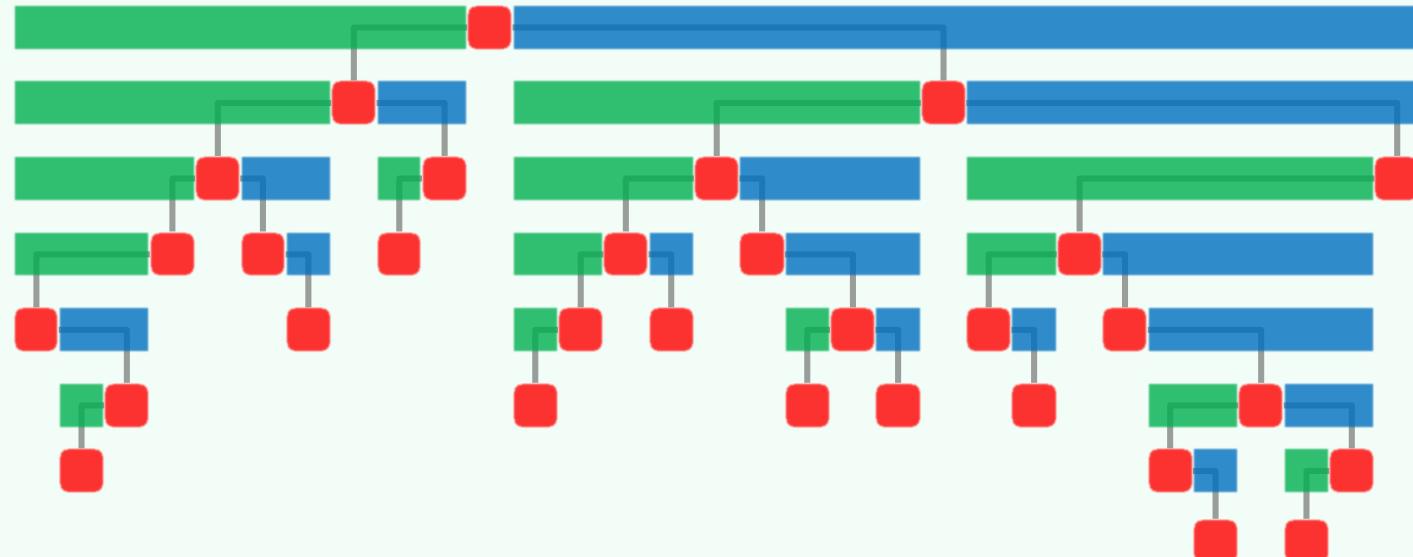
a_k 早于或晚于 a_i 和 a_j 被确认

均与 $Pr(i, j)$ 无关

❖ 实际上， $\langle a_i, a_j \rangle$ 接受比较，当且仅当

在 $\{ a_i, a_{i+1}, a_{i+2}, \dots, a_{j-2}, a_{j-1}, a_j \}$ 中， a_i 或 a_j 率先被确认

$$T(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} Pr(i, j) = \sum_{j=1}^{n-1} \sum_{i=0}^{j-1} Pr(i, j) = \sum_{j=1}^{n-1} \sum_{d=1}^j \frac{2}{d+1} \approx \sum_{j=1}^{n-1} 2 \cdot (\ln j - 1) \leq 2 \cdot n \cdot \ln n$$



对比

Vector sorter	#compare	#move	cache-friendly
Insertionsort	$\mathcal{O}(n) \sim \mathcal{O}(n^2)$ expected- $\mathcal{O}(n^2)$	$\mathcal{O}(n) \sim \mathcal{O}(n^2)$ expected- $\mathcal{O}(n^2)$	
Selectionsort	$\Theta(n^2)$	$\mathcal{O}(n)$	
Heapsort	$2.0 * n \log n$	$1.0 * n \log n$	percolateDown(): $x[i]$ $x[j]$
Mergesort	$1.0 * n \log n$	$1.5 * n \log n$	merge(): $\checkmark[i]$ $\checkmark[j]$ $\checkmark[k]$
Quicksort	expected- $1.386 * n \log n$ w.h.p.	expected- $(1.386/2 = 0.694) * n \log n$	\checkmark pivot $\checkmark[lo]$ $\checkmark[hi]$

14 - A6

排序

快速排序：快速划分：DUP版

邓俊辉

deng@tsinghua.edu.cn

左见兮鸣鶡，右睹兮呼枭

重复元素

❖ 有大量元素与轴点雷同时

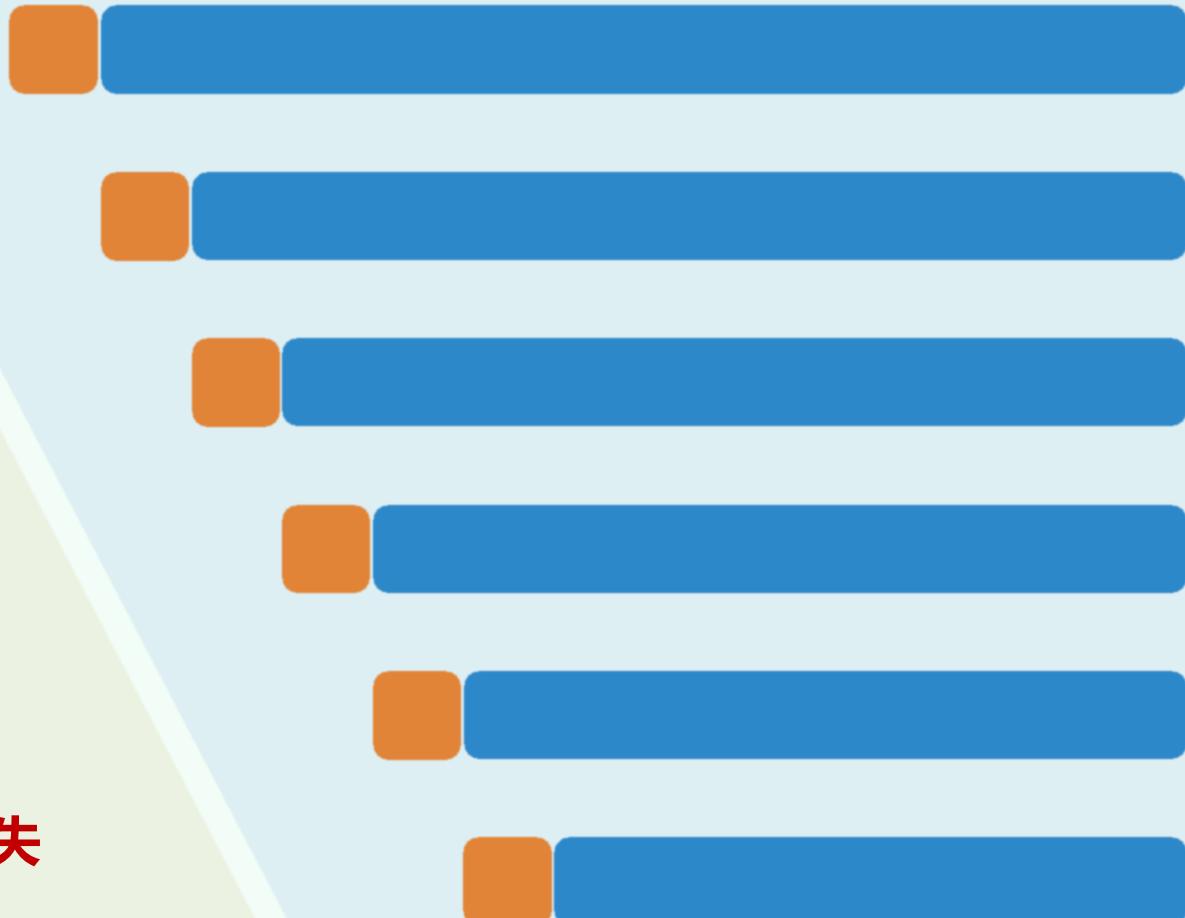
- 切分点将接近于 lo
- 划分极度失衡
- 递归深度接近于 $\theta(n)$
- 运行时间接近于 $\theta(n^2)$

❖ 移动 lo 和 hi 的过程中，同时比较相邻元素

若属于相邻的重复元素，则不再深入递归

❖ 但一般情况下，如此计算量反而增加，得不偿失

❖ 实际上，LUG版略做调整，即可解决问题...



快速划分：LUG版

```
template <typename T> Rank Vector<T>::partition( Rank lo, Rank hi ) { // [lo, hi)
    swap( _elem[lo], _elem[lo + rand() % (hi-lo)] ); // 随机交换
    T pivot = _elem[lo]; // 经以上交换，等效于随机选取候选轴点
    while ( lo < hi ) { // 从两端交替地向中间扫描，彼此靠拢
        do hi--; while ( (lo < hi) && (pivot <= _elem[hi]) ); // 向左拓展G
        if (lo < hi) _elem[lo] = _elem[hi]; // 凡 小于 轴点者，皆归入L
        do lo++; while ( (lo < hi) && (_elem[lo] <= pivot) ); // 向右拓展L
        if (lo < hi) _elem[hi] = _elem[lo]; // 凡 大于 轴点者，皆归入G
    } // assert: lo == hi or hi+1
    _elem[hi] = pivot; return hi; // 候选轴点归位；返回其秩
```

快速划分：DUP版

```
template <typename T> Rank Vector<T>::partition( Rank lo, Rank hi ) { // [lo, hi)
    swap( _elem[lo], _elem[lo + rand() % (hi-lo)] ); // 随机交换
    T pivot = _elem[lo]; // 经以上交换，等效于随机选取候选轴点
    while ( lo < hi ) { // 从两端交替地向中间扫描，彼此靠拢
        do hi--; while ( (lo < hi) && (pivot < _elem[hi]) ); // 向左拓展G
        if (lo < hi) _elem[lo] = _elem[hi]; // 凡不大于轴点者，皆归入L
        do lo++; while ( (lo < hi) && (_elem[lo] < pivot) ); // 向右拓展L
        if (lo < hi) _elem[hi] = _elem[lo]; // 凡不小于轴点者，皆归入G
    } // assert: lo == hi or hi+1
    _elem[hi] = pivot; return hi; // 候选轴点归位；返回其秩
```

性能

❖ 可以正确地处理一般情况

同时复杂度并未实质增高

❖ 遇到连续的重复元素时

- lo和hi会交替移动
- 切分点接近于 $(lo+hi)/2$

❖ 由LUG版的勤于拓展、懒于交换

转为懒于拓展、勤于交换

❖ 交换操作有所增加，“更”不稳定



排序

快速排序：快速划分：LGU版

14-A>

The Great walks with the Small without fear.

The Middle keeps aloof.

邓俊辉

deng@tsinghua.edu.cn

不变性: $S = [lo, hi) = [lo] + (lo, mi] + (mi, k) + [k, hi) = pivot + L + G + U$

$$L < pivot \leq G$$



单调性：[k]不小于轴点 ? 直接G拓展 : G滚动后移，L拓展

`pivot <= S[k] ? k++ : swap(S[++mi], S[k++])`



快速划分：LGU版

```
template <typename T> Rank Vector<T>::partition( Rank lo, Rank hi ) { // [lo, hi)
    swap( _elem[ lo ], _elem[ lo + rand() % ( hi - lo ) ] ); // 随机交换
    T pivot = _elem[ lo ]; Rank mi = lo;
    for ( Rank k = lo + 1; k < hi; k++ ) // 自左向右考查每个[k]
        if ( _elem[ k ] < pivot ) // 若[k]小于轴点，则将其
            swap( _elem[ ++mi ], _elem[ k ] ); // 与[mi]交换，L向右扩展
    swap( _elem[ lo ], _elem[ mi ] ); // 候选轴点归位（从而名副其实）
    return mi; // 返回轴点的秩
}
```

实例



排序

选取：众数

14B7

善钩，从众。夫善，众之主也。三卿为主，可谓众矣。从之，不亦可乎？！

诚若为今立计，所当稽求既往，相度方来，培物质而张灵明，任个人而排众数。

然而，在现代文明社会里，居有其所的家庭却不到一半；在文明特别发达的大城市里，拥有住房的人只占全体居民的极小部分。

邓俊辉

deng@tsinghua.edu.cn

选取 + 中位数

❖ **k-selection** 在任意一组可比较大小的元素中，如何由小到大，找到次序为 k 者？

亦即，在这组元素的非降排序序列 S 中，找出 $S[k]$

// Excel: large(range, rank)

❖ **median** 长度为 n 的有序序列 S 中，元素 $S[\lfloor n/2 \rfloor]$ 称作中位数 // 数值上可能有重复

在任意一组可比较大小的元素中，如何找到中位数？ // Excel: median(range)



❖ 中位数是 k -选取的一个特例；稍后将看到，也是其中难度最大者

Majority

❖ 无序向量中，若有一半以上元素同为 m ，则称之为众数

- 在 { $\boxed{3}$, 5, 2, $\boxed{3}$, $\boxed{3}$ } 中，众数为3；然而
- 在 { $\boxed{3}$, 5, 2, $\boxed{3}$, $\boxed{3}$, 0 } 中，却无众数

❖ 平凡算法 排序 + 扫描

但进一步地 若限制时间不超过 $O(n)$ ，附加空间不超过 $O(1)$ 呢？

❖ 必要性 众数若存在，则亦必中位数

❖ 事实上 只要能够找出中位数，即不难验证它是否众数

```
template <typename T> bool majority( Vector<T> A, T & maj )  
{ return majEleCheck( A, maj = median( A ) ); }
```

必要条件

❖ 然而 在高效的中位数算法未知之前，如何确定众数的候选呢？

❖ mode 众数若存在，则亦必频繁数 //Excel: mode(range)

```
template <typename T> bool majority( Vector<T> A, T & maj )  
{ return majEleCheck( A, maj = mode( A ) ); }
```

❖ 同样地 mode() 算法难以兼顾时间、空间的高效

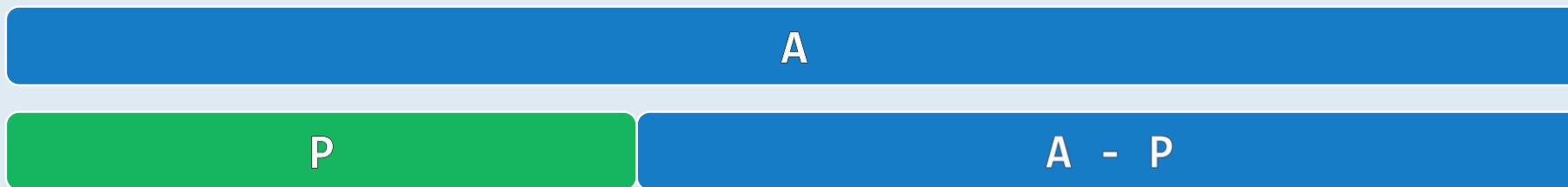
❖ 可行思路 借助更弱但计算成本更优的必要条件，选出唯一的候选者

```
template <typename T> bool majority( Vector<T> A, T & maj )  
{ return majEleCheck( A, maj = majEleCandidate( A ) ); }
```

减而治之

❖ 若在向量A的前缀P (|P|为偶数) 中，元素x出现的次数恰占半数，则

A有众数，仅当对应的后缀A - P有众数m，且m就是A的众数



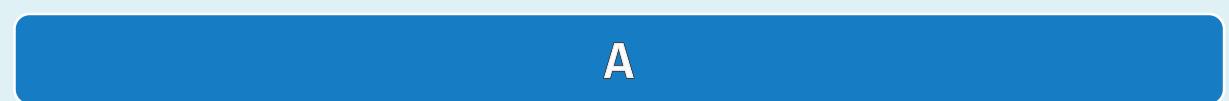
❖ 既然最终总要花费 $O(n)$ 时间做验证，故而只需考虑A的确含有众数的两种情况：

1. 若 $x = m$ ，则在排除前缀P之后，m与其它元素在数量上的差距保持不变
2. 若 $x \neq m$ ，则在排除前缀P之后，m与其它元素在数量上的差距不致缩小

❖ 若将众数的标准从“一半以上”改作“至少一半”，算法需做什么调整？

算法

```
template <typename T> T majCandidate( Vector<T> A ) {  
    T maj;  
  
    for ( Rank c = 0, i = 0; i < A.size(); i++ )  
        if ( 0 == c ) {  
            maj = A[i]; c = 1;  
        } else  
            maj == A[i] ? c++ : c--;  
  
    return maj;  
}
```



排序

选取：中位数

14-B2

中也者，天下之大本也；和也者，天下之达道也

德性是两种恶——过度与不及——的中间。在感情与实践中，恶要么达不到正确，要么超过正确。德性则找到并且选取那个正确。所以虽然从其本质或概念来说德性是适度，从最高善的角度来说，它是一个极端

邓俊辉

deng@tsinghua.edu.cn

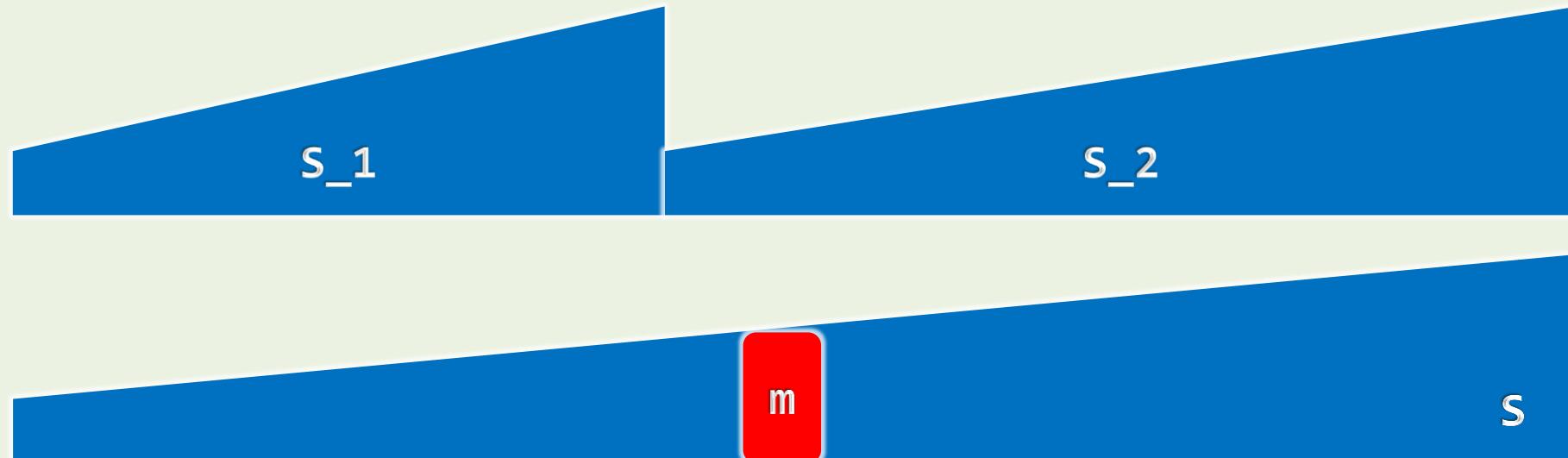
归并向量的中位数

❖ 任给有序向量 S_1 和 S_2 ，长度 n_1 和 n_2
如何快速找出 $S = S_1 \cup S_2$ 的中位数？

❖ 蛮力：经归并得到有序向量 S
取 $S[(n_1 + n_2)/2]$ 即是

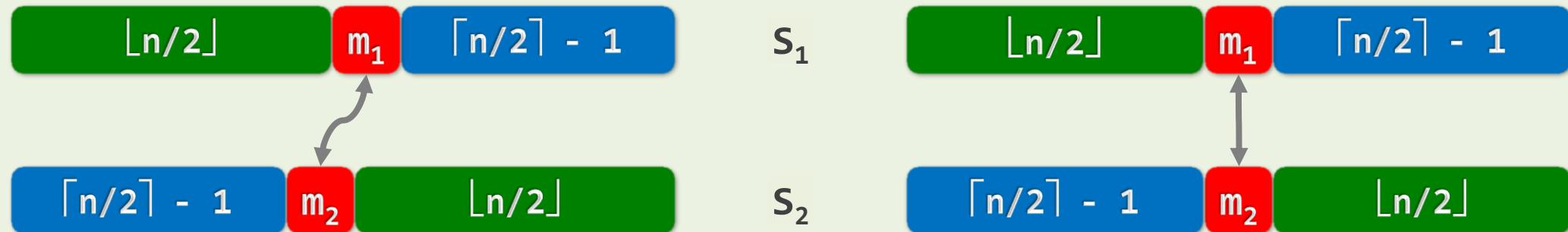
❖ 如此，共需 $\mathcal{O}(n_1 + n_2)$ 时间
但毕竟未能充分利用 S_1 和 S_2 的有序性

❖ 以下，先解决 $n_1 = n_2$ 的情况
依然采用减而治之策略...



等长子向量：构思

❖ 考查: $m_1 = S_1[\lfloor n/2 \rfloor]$ 和 $m_2 = S_2[\lceil n/2 \rceil - 1] = S_2[\lfloor (n-1)/2 \rfloor]$



❖ 若 $m_1 = m_2$ ，则它们同为 S_1 、 S_2 和 S 的中位数

❖ 若 $m_1 < m_2$ ，则 n 无论偶奇，必有:

$$\text{median}(S_1 \cup S_2) = \text{median}(S_1.\text{suffix}(\lceil n/2 \rceil) \cup S_2.\text{prefix}(\lceil n/2 \rceil))$$

这意味着，如此减除（一半规模）之后，中位数不变

❖ $m_1 > m_2$ 时同理

等长子向量：实现

```
template <typename T> //尾递归，可改写为迭代形式

T median( Vector<T> & S1, Rank lo1, Vector<T> & S2, Rank lo2, Rank n ) {
    if ( n < 3 ) return trivialMedian( S1, lo1, n, S2, lo2, n ); //递归基
    Rank mi1 = lo1 + n/2, mi2 = lo2 + (n - 1)/2; //长度减半
    if ( S1[ mi1 ] < S2[ mi2 ] ) //取S1右半、S2左半
        return median( S1, mi1, S2, lo2, n + lo1 - mi1 );
    else if ( S1[ mi1 ] > S2[ mi2 ] ) //取S1左半、S2右半
        return median( S1, lo1, S2, mi2, n + lo2 - mi2 );
    else
        return S1[ mi1 ];
}
```

任意子向量：实现 (1/2)

```
template <typename T>

T median ( Vector<T>& S1, Rank lo1, Rank n1, Vector<T>& S2, Rank lo2, Rank n2 ) {

    if ( n1 > n2 )

        return median( S2, lo2, n2, S1, lo1, n1 ); //确保n1 <= n2

    if ( n2 < 6 )

        return trivialMedian( S1, lo1, n1, S2, lo2, n2 ); //递归基

    if ( 2 * n1 < n2 )

        return median( S1, lo1, n1, S2, lo2 + (n2-n1-1)/2, n1+2-(n2-n1)%2 );
```

任意子向量：实现 (2/2)

```
Rank mi1 = lo1 + n1/2, mi2a = lo2 + (n1 - 1)/2, mi2b = lo2 + n2 - 1 - n1/2;

if ( s1[ mi1 ] > s2[ mi2b ] ) //取s1左半、s2右半
    return median( s1, lo1, n1 / 2 + 1, s2, mi2a, n2 - (n1 - 1) / 2 );
else if ( s1[ mi1 ] < s2[ mi2a ] ) //取s1右半、s2左半
    return median( s1, mi1, (n1 + 1) / 2, s2, lo2, n2 - n1 / 2 );
else //s1保留，s2左右同时缩短
    return median( s1, lo1, n1, s2, mi2a, n2 - (n1 - 1) / 2 * 2 );

} //O( log(min(n1,n2)) )——可见，实际上等长版本才是难度最大的
```

排序

选取: QuickSelect

14-B3

邓俊辉

deng@tsinghua.edu.cn

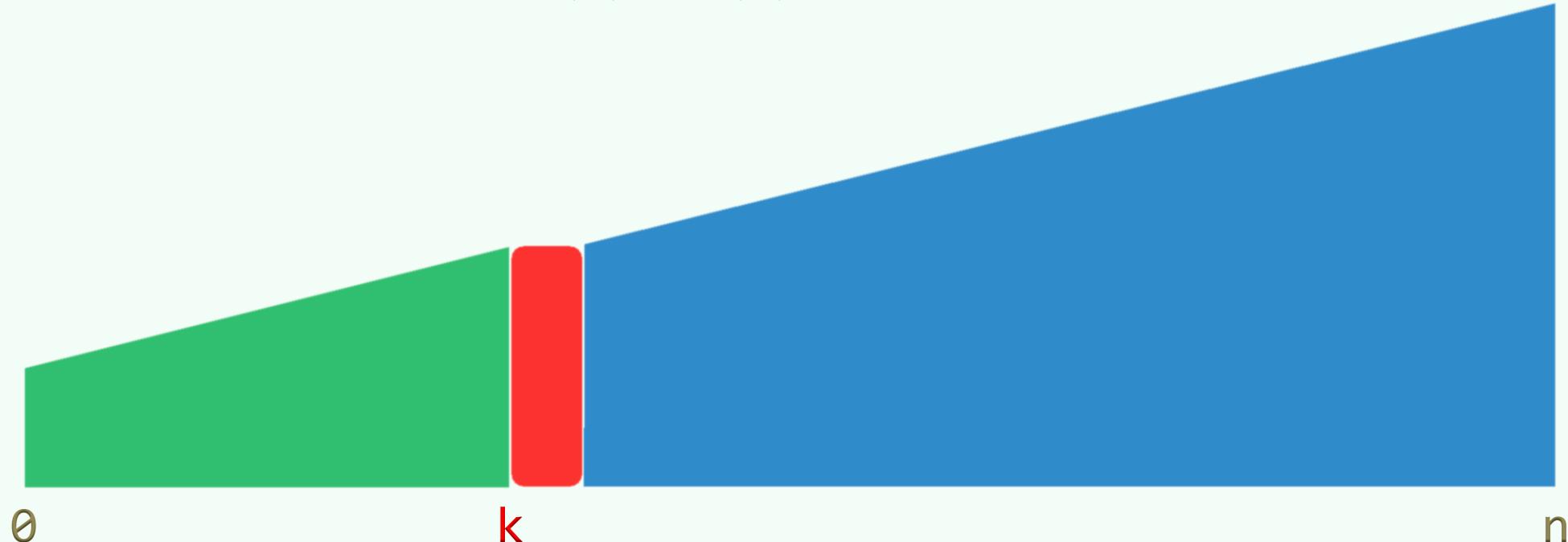
大胆猜测，小心求证

他们在一起谈了一下之后，就转过身来向我表示敬意，对此，我的老师微微一笑；此外，他们还给了我更多的荣誉，因为他们把我列入他们的行列，结果，我就是这样赫赫有名的智者中的第六位。

尝试：蛮力

❖ 对A排序 // $\theta(n \log n)$

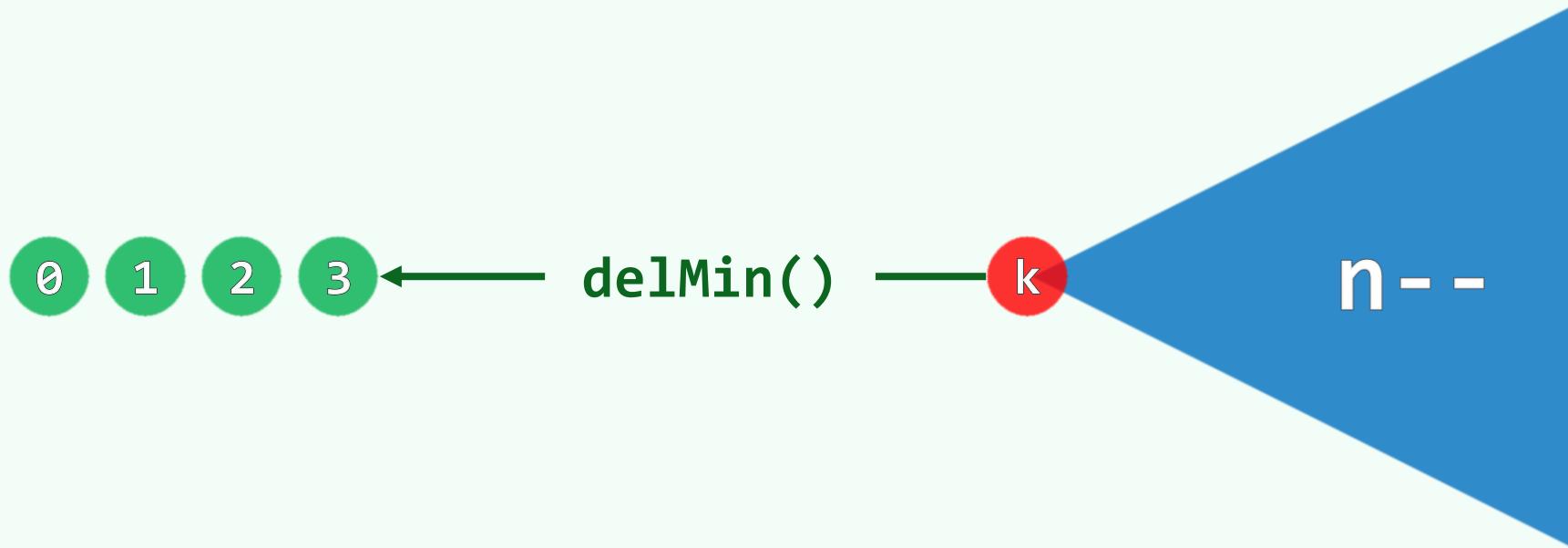
从首元素开始，向后行进k步 // $\theta(k) = \theta(n)$



尝试：堆 (A)

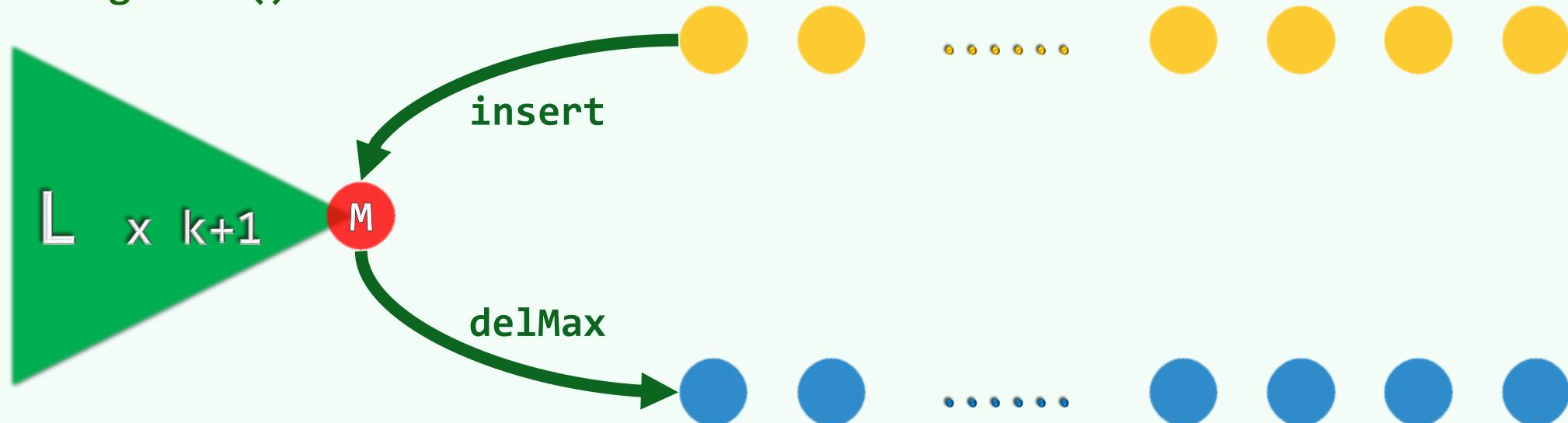
❖ 将所有元素组织为小顶堆 $// O(n)$

连续调用 $k+1$ 次`delMin()` $// O(k \log n)$



尝试：堆 (B)

```
❖ L = heapify( A[0, k] ) //任选 k+1 个元素，组织为大顶堆:  $\mathcal{O}(k)$   
❖ for each i in (k, n) // $\mathcal{O}(n - k)$   
    L.insert( A[i] ) // $\mathcal{O}(\log k)$   
  
    L.delMax() // $\mathcal{O}(\log k)$   
  
return L.getMax()
```



尝试：堆 (C)

❖ 将输入任意划分为规模为 k 、 $n-k$ 的子集

分别组织为大、小顶堆

$//\mathcal{O}(k + (n-k)) = \mathcal{O}(n)$

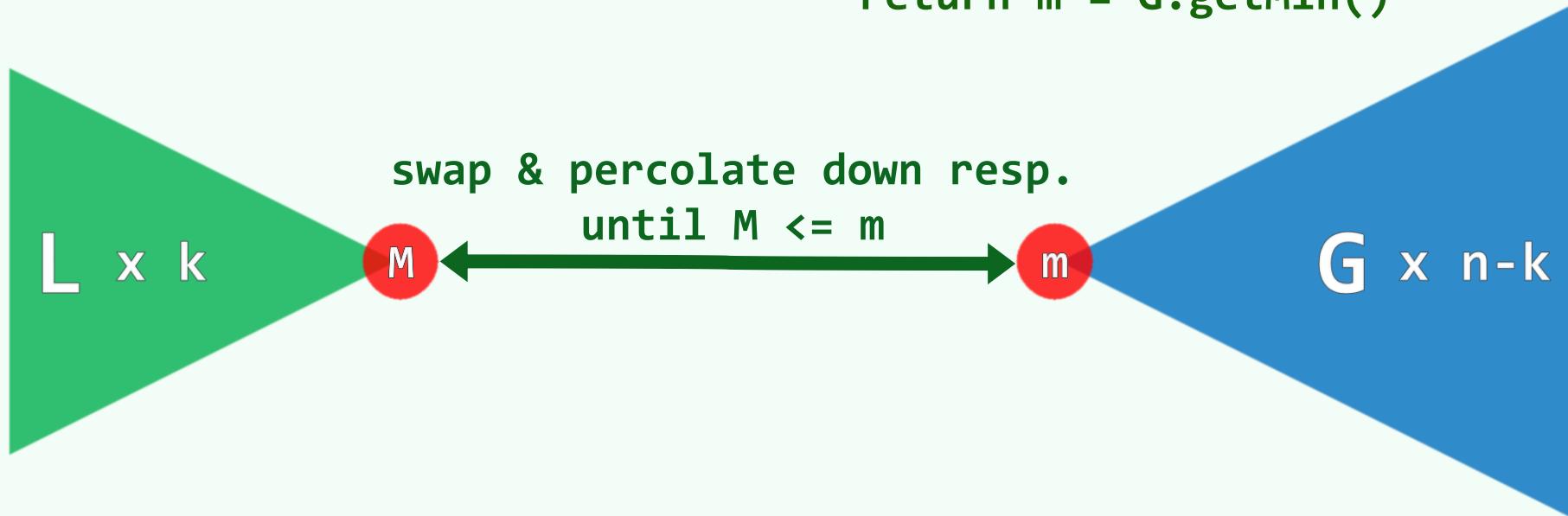
❖ while ($m < M$) $//\mathcal{O}(\min(k, n - k))$

swap(m, M)

L.percolateDown() $//\mathcal{O}(\log k)$

G.percolateDown() $//\mathcal{O}(\log(n - k))$

return $m = G.getMin()$



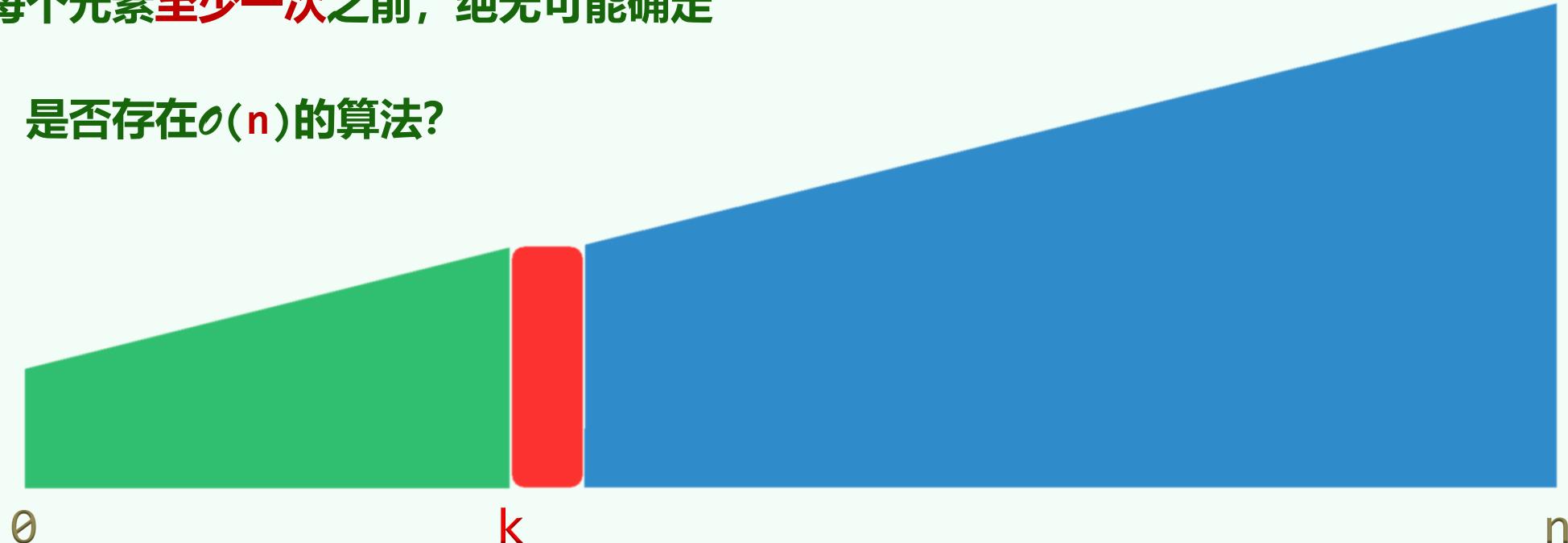
下界与最优

❖ 是否存在**更快的算法**? 当然, 最快也不至于快过 $\Omega(n)$!

❖ 所谓**第k小**, 是相对于序列**整体**而言, 所以...

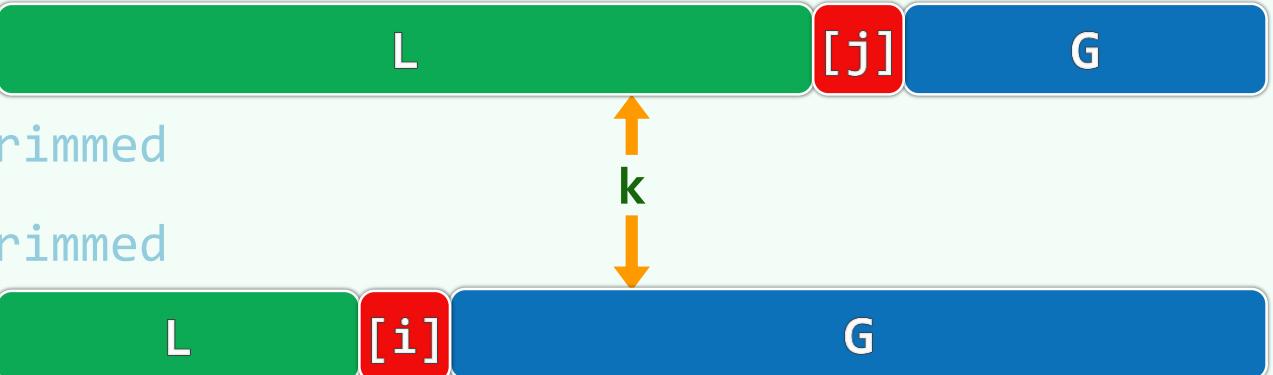
在访问每个元素**至少一次**之前, 绝无可能确定

❖ 反过来, 是否存在 $\mathcal{O}(n)$ 的算法?



quickSelect()

```
template <typename T> void quickSelect( Vector<T> & A, Rank k ) {  
    for ( Rank lo = 0, hi = A.size(); lo < hi; ) {  
        Rank i = lo, j = hi; T pivot = A[lo]; //大胆猜测  
        while ( i < j ) { //小心求证:  $\mathcal{O}(hi - lo + 1) = \mathcal{O}(n)$   
            do j--; while ( (i < j) && (pivot <= A[j]) ); if (i < j) A[i] = A[j];  
            do i++; while ( (i < j) && (A[i] <= pivot) ); if (i < j) A[j] = A[i];  
        } //assert: quit with i == j or j+1  
        A[j] = pivot;  
        if ( k <= j ) hi = j; //suffix trimmed  
        if ( i <= k ) lo = i; //prefix trimmed  
    } //A[k] is now a pivot  
}
```



期望性能

❖ 记期望的比较次数为 $T(n)$

$$T(1) = 0, T(2) = 1, \dots$$

❖ 可以证明: $T(n) = \mathcal{O}(n) \dots$

$$T(n) = (n-1) + \frac{1}{n} \times \sum_{k=0}^{n-1} \max\{T(k), T(n-k-1)\} \leq (n-1) + \frac{2}{n} \times \sum_{k=n/2}^{n-1} T(k)$$

❖ 事实上, 不难验证: $T(n) < 4 \cdot n \dots$

$$T(n) \leq (n-1) + \frac{2}{n} \times \sum_{k=n/2}^{n-1} 4k \leq (n-1) + 3n < 4n$$

排序

选取：LinearSelect

14-B4

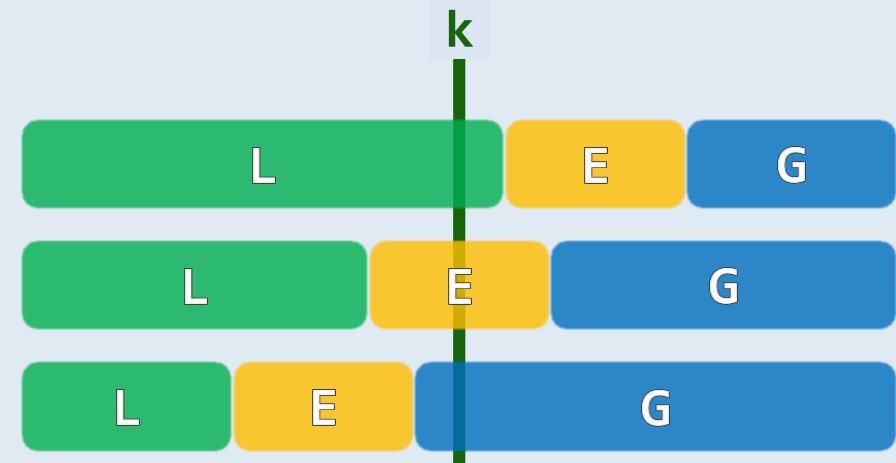
世兄的才名，弟所素知的。在世兄是数万人里头选出来最清最雅的，至于弟乃庸庸碌碌一等愚人，忝附同名，殊觉玷辱了这两个字。

为埃皮奈先生生产的水果，尽管给人偷掉了四分之三，还比他在舍弗莱特的那片大菜园要多。

邓俊辉
deng@tsinghua.edu.cn

linearSelect(A, n, k)

Let Q be a small constant



复杂度

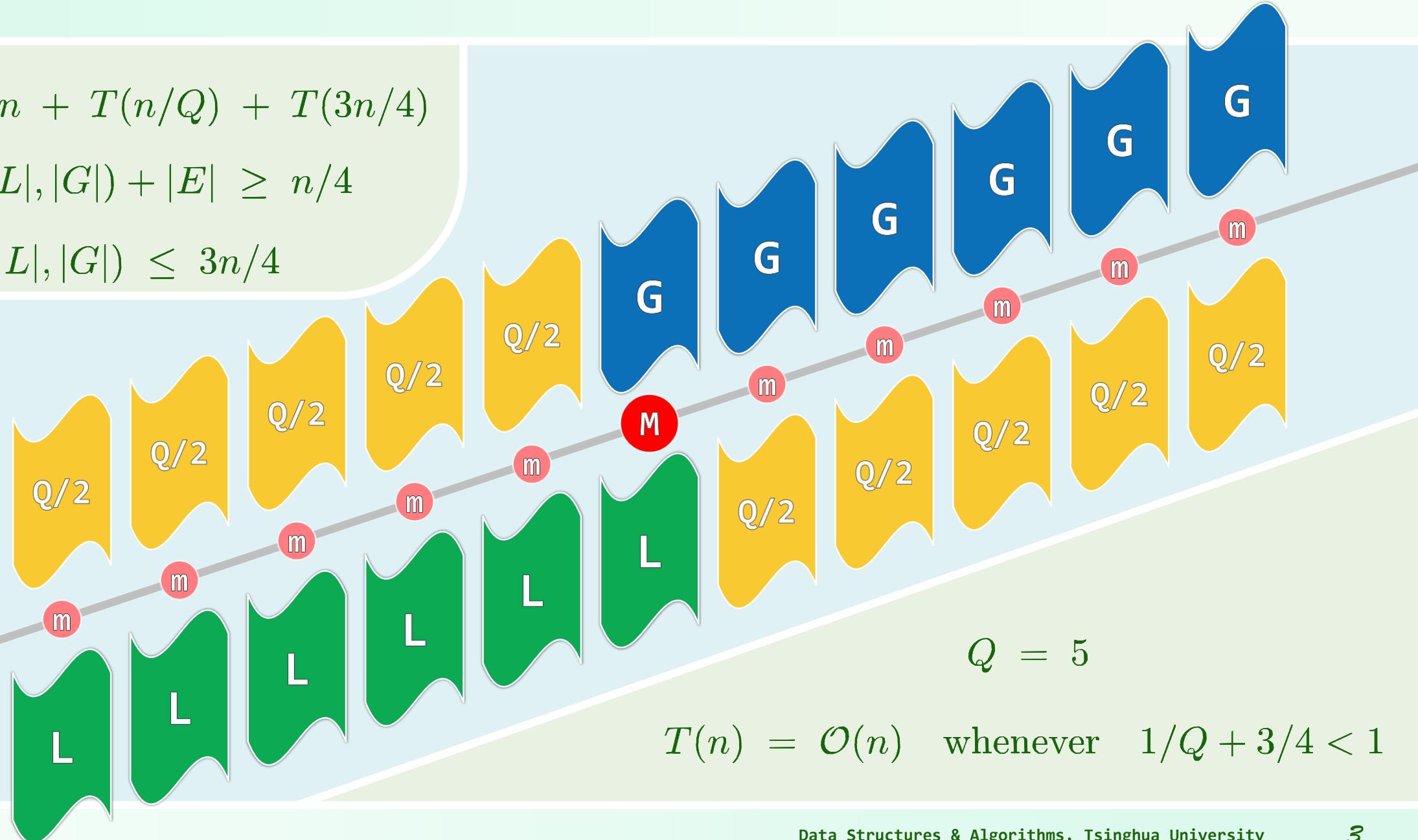
- ❖ 将`linearSelect()`算法的运行时间记作 $T(n)$
- ❖ 第0步: $\mathcal{O}(1) = \mathcal{O}(Q \log Q)$ //递归基: 序列长度 $|A| \leq Q$
- ❖ 第1步: $\mathcal{O}(n)$ //子序列划分
- ❖ 第2步: $\mathcal{O}(n) = Q^2 \times n/Q$ //子序列各自排序, 并找到中位数
- ❖ 第3步: $T(n/Q)$ //从 n/Q 个中位数中, 递归地找到全局中位数
- ❖ 第4步: $\mathcal{O}(n)$ //划分子集L/E/G, 并分别计数 —— 一趟扫描足矣
- ❖ 第5步: $T(3n/4)$ //为什么...

复杂度

$$T(n) = c \cdot n + T(n/Q) + T(3n/4)$$

$$\min(|L|, |G|) + |E| \geq n/4$$

$$\max(|L|, |G|) \leq 3n/4$$



排序

希尔排序：框架+实例

14 - C7

瓜熟蒂落，水到渠成

今有物不知其数，三三数之剩二；五五数之剩三；七七数之剩二。问物几何

邓俊辉

deng@tsinghua.edu.cn

Shellsort

❖ D. L. Shell: 将整个序列视作一个矩阵, 逐列各自排序

❖ 递减增量 (diminishing increment)

- 由粗到细: 重排矩阵, 使其更窄, 再次逐列排序 (**h-sorting/h-sorted**)
- 逐步求精: 如此往复, 直至矩阵变成一列 (**1-sorting/1-sorted**)

❖ 步长序列 (step sequence) : 由各矩阵宽度逆向排列而成的序列

$$\mathcal{H} = \{ h_1 = 1, h_2, h_3, \dots, h_k, \dots \}$$

❖ 正确性: 最后一次迭代, 等同于全排序

1-sorted = ordered



实例： $h_5 = 8$

80 23 19 40 85 1 18 92 71 8 96 46 12

80	23	19	40	85	1	18	92
71	8	96	46	12			

71	8	19	40	12	1	18	92
80	23	96	46	85			

71 8 19 40 12 1 18 92 80 23 96 46 85

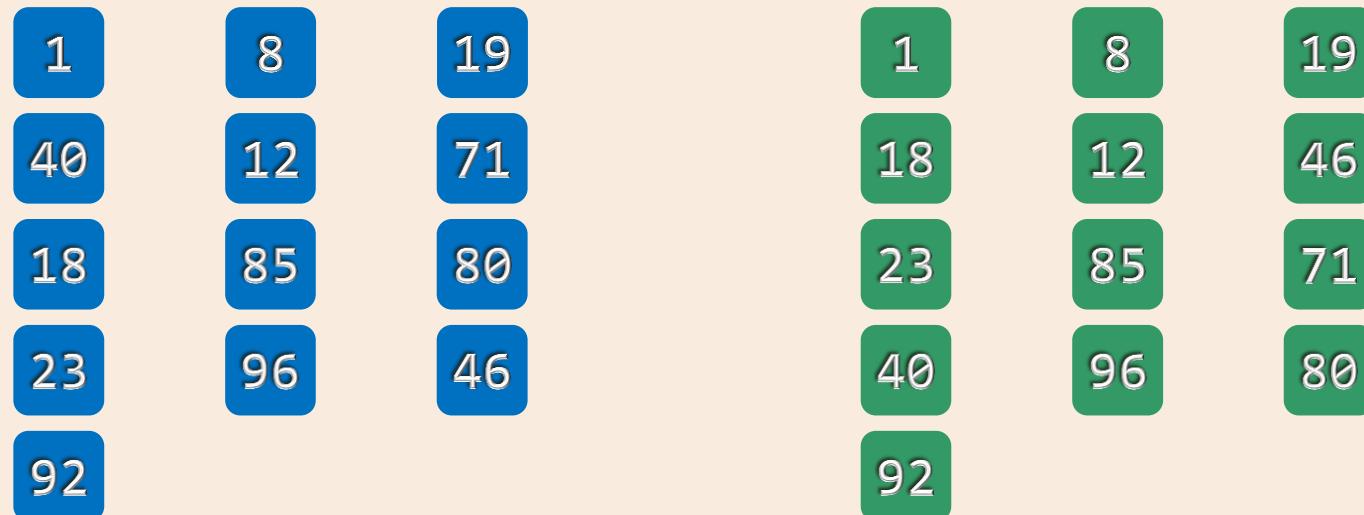
实例： $h_4 = 5$

1 8 19 40 12 71 18 85 80 23 96 46 92

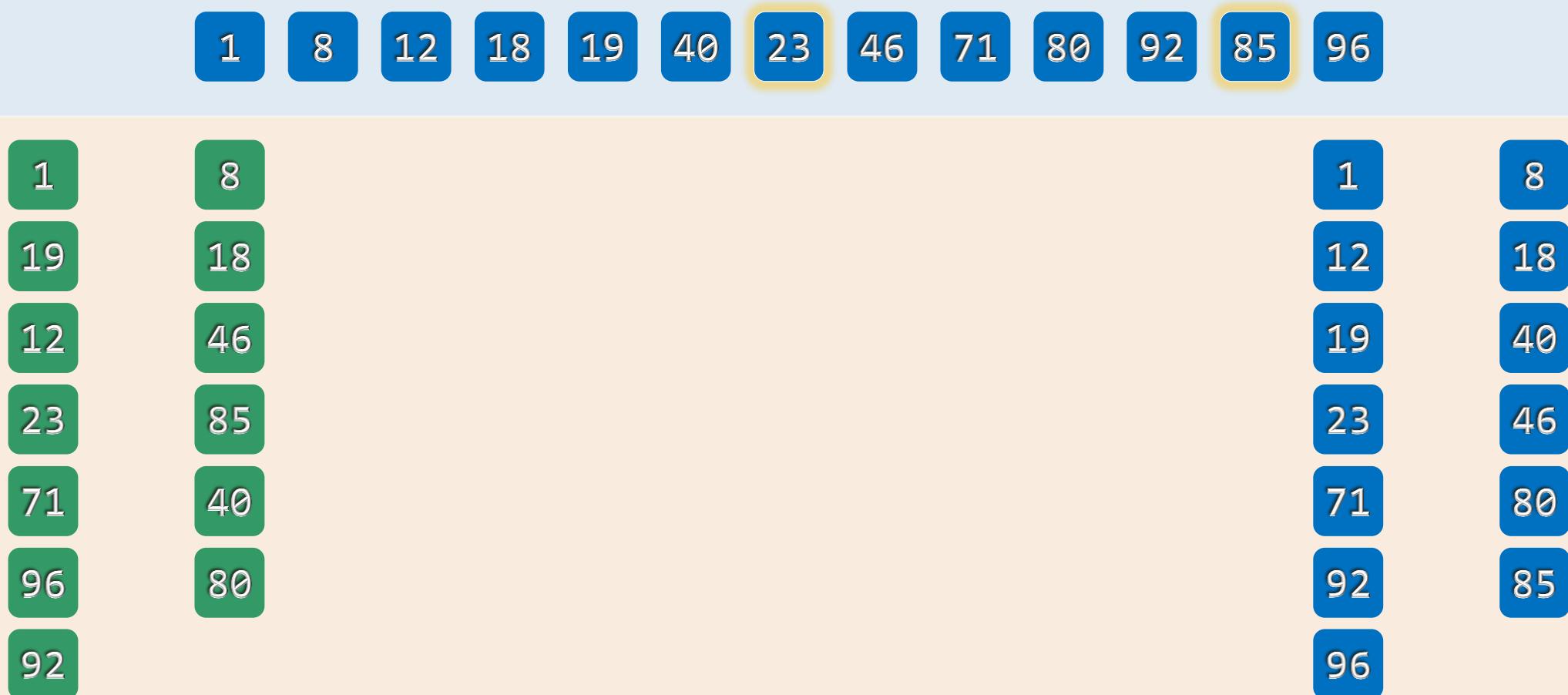
71	8	19	40	12	1	8	19	40	12
1	18	92	80	23	71	18	85	80	23
96	46	85			96	46	92		

71 8 19 40 12 1 18 92 80 23 96 46 85

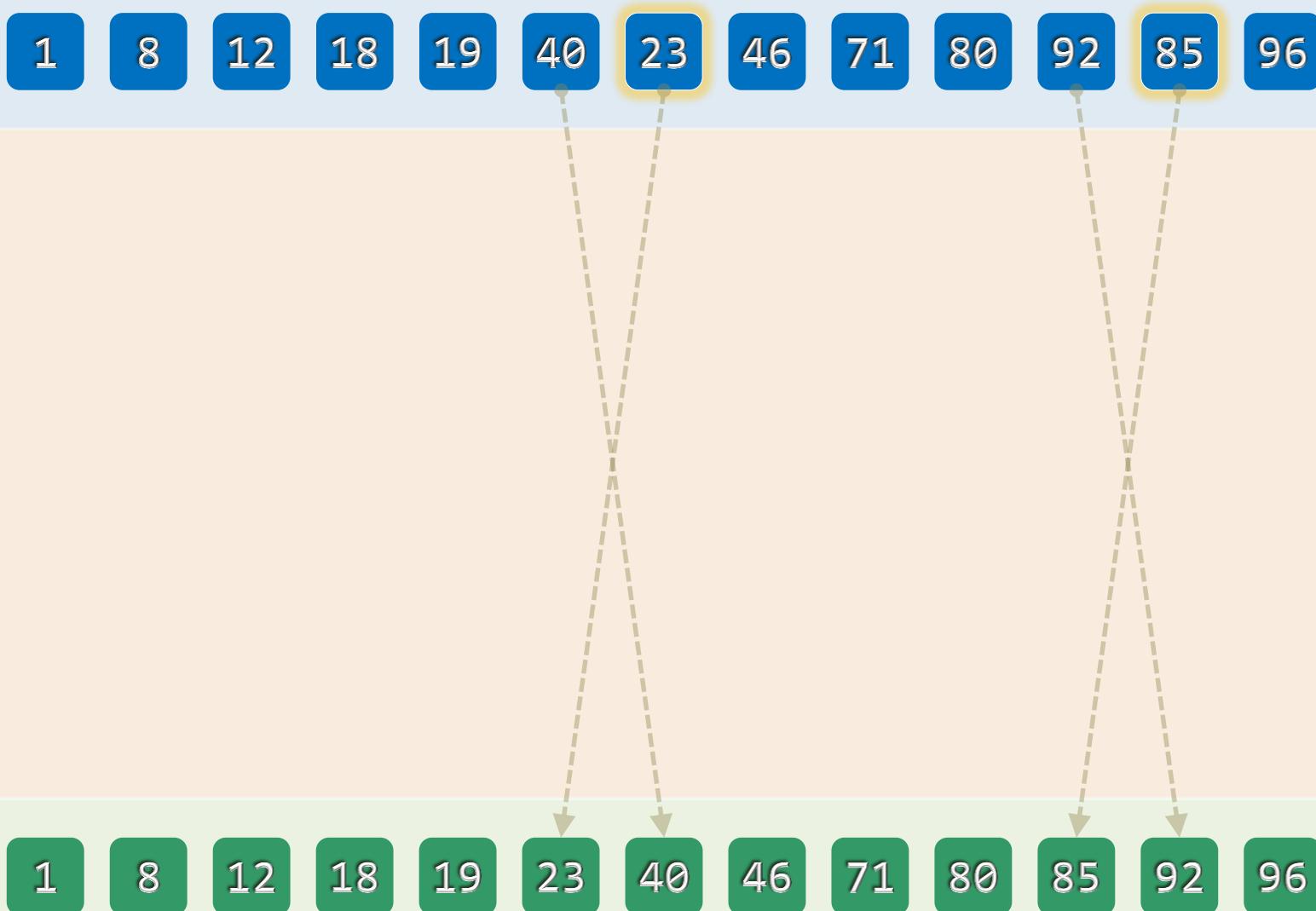
实例： $h_3 = 3$



实例： $h_2 = 2$



实例： $h_1 = 1$



Call-by-rank

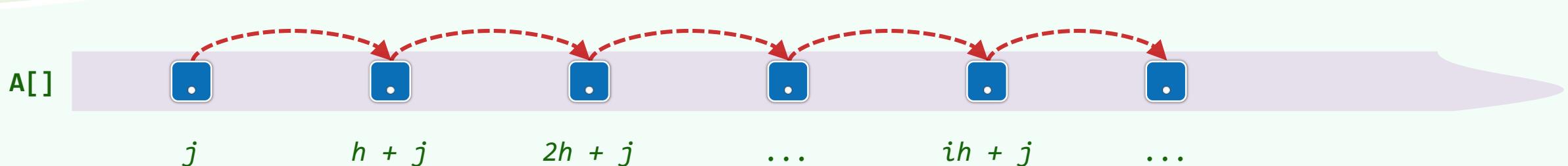
❖ 如何实现矩阵重排？莫非，需要使用二维向量？

❖ 实际上，借助一维向量足矣

❖ 在每步迭代中，若当前的矩阵宽度为 h ，则

$$B[i][j] = A[i \cdot h + j]$$

或 $A[k] = B[k/h][k \% h]$



实现

```
template <typename T> void Vector<T>::shellSort( Rank lo, Rank hi ) {  
    for ( Rank d = 0x7FFFFFFF; 0 < d; d >>= 1 ) //PS Sequence: 1, 3, 7, 15, 31, ...  
        for ( Rank j = lo + d; j < hi; j++ ) { //for each j in [lo+d, hi)  
            T x = _elem[j]; Rank i = j; //within the prefix of the subsequence of [j]  
            while ( (lo + d <= i) && (x < _elem[i-d]) ) //find the appropriate  
                _elem[i] = _elem[i-d]; i -= d; //predecessor [i]  
            _elem[i] = x; //where to insert [j]  
        }  
    } //0 <= lo < hi <= size <= 2^31
```

排序

希尔排序：Shell序列 + 输入敏感性

14 - C2

邓俊辉

deng@tsinghua.edu.cn

让上帝的归上帝，凯撒的归凯撒

Shell's Sequence

- ❖ Shell 1959: $\mathcal{H}_{shell} = \{ 1, 2, 4, 8, 16, 32, 64, \dots, 2^k, \dots \}$
- ❖ 实际上，采用 \mathcal{H}_{shell} ，在最坏情况下需要运行 $\Omega(n^2)$ 时间...
- ❖ 考查由子序列 $A = \text{unsort}[0, 2^{N-1})$ 和 $B = \text{unsort}[2^{N-1}, 2^N)$ 交错而成的序列



- ❖ 在做2-sorting时，A、B各成一列；故此后必然各自有序

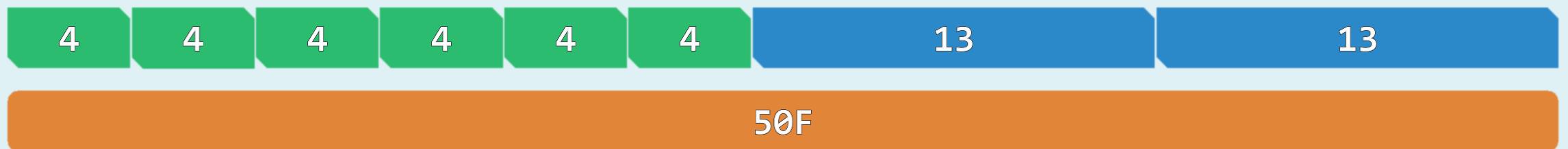


- ❖ 然而其中的逆序对依然很多，最后的1-sorting仍需 $1 + 2 + 3 + \dots + 2^{N-1} = \Omega(n^2/4)$ 时间
- ❖ 根源在于， \mathcal{H}_{shell} 中各项并不互素，甚至相邻项也非互素

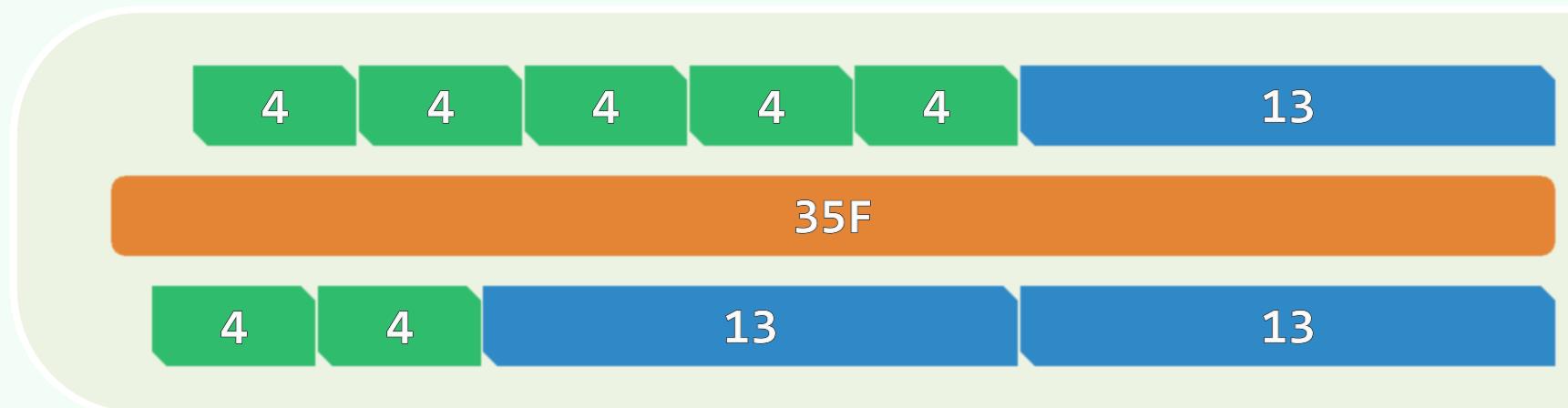
Postage Problem

❖ The postage for a letter is **50F**, and a postcard **35F**

But there are only stamps of **4F** and **13F** available



❖ Possible to stamp
the letter and
the postcard
EXACTLY?



❖ Given a postage P , determine whether $P \in \{ n \cdot 4 + m \cdot 13 \mid n, m \in \mathcal{N} \}$

Linear Combination

- ❖ Let $g, h \in \mathcal{N}$
- ❖ For any $n, m \in \mathcal{N}$, $n \cdot g + m \cdot h$ is called a **linear combination** of g and h
- ❖ Denote $\mathbf{C}(g, h) = \{ ng + mh \mid n, m \in \mathcal{N} \}$
 $\mathbf{N}(g, h) = \mathcal{N} \setminus \mathbf{C}(g, h)$ //numbers that are NOT combinations of g and h
 $\mathbf{x}(g, h) = \max\{ \mathbf{N}(g, h) \}$ //always exists?

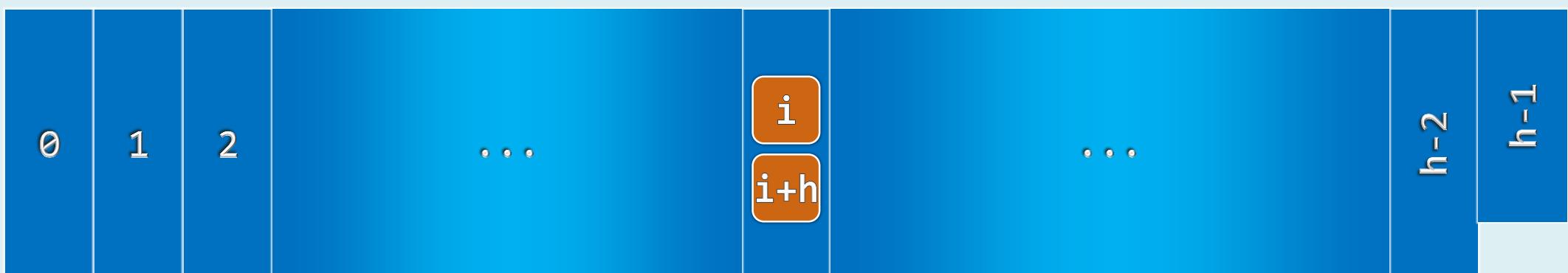
❖ Theorem: when g and h are **relatively prime**, we have

$$\mathbf{x}(g, h) = (g - 1) \cdot (h - 1) - 1 = gh - g - h$$

e.g. $\mathbf{x}(3, 7) = 11$, $\mathbf{x}(4, 9) = 23$, $\mathbf{x}(\boxed{4}, \boxed{13}) = \boxed{35}$, $\mathbf{x}(5, 14) = 51$

h-sorting & h-ordered

- ❖ A sequence $S[0, n)$ is called **h-ordered** if $S[i] \leq S[i + h], \forall 0 \leq i < n - h$
- ❖ A **1-ordered** sequence is sorted
- ❖ **h-sorting:** an h-ordered sequence is obtained by
 - arranging S into a 2D matrix with **h** columns and
 - sorting each column respectively

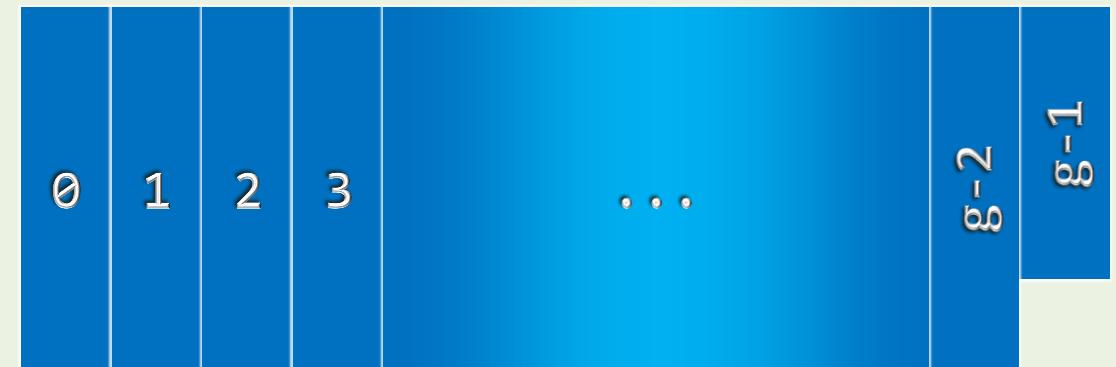
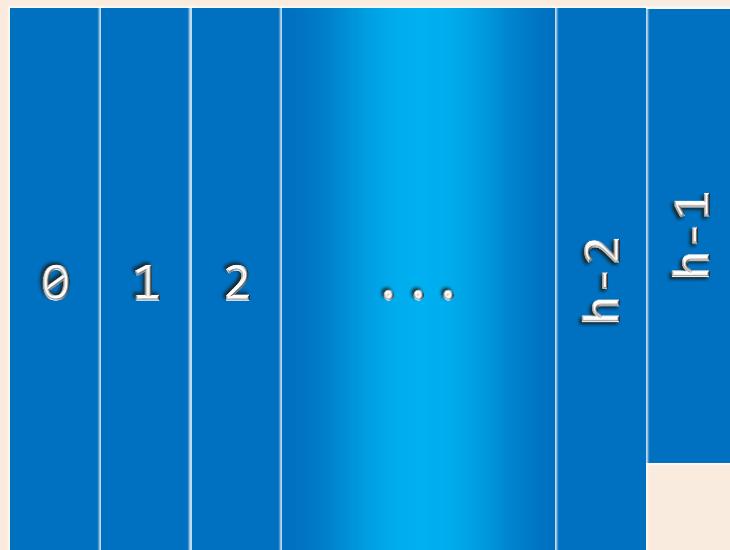


Theorem K

❖ [Knuth, ACP Vol.3 p.90]

// 习题解析[12-12, 12-13]

A **g**-ordered sequence REMAINS **g**-ordered after being **h**-sorted.

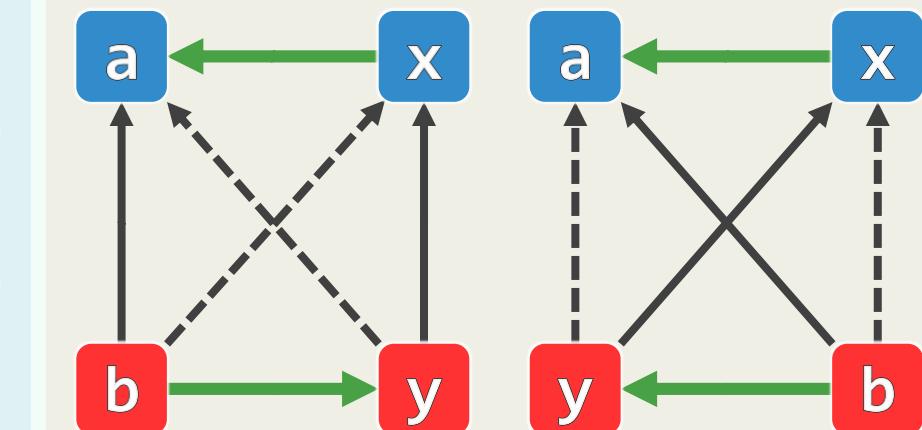
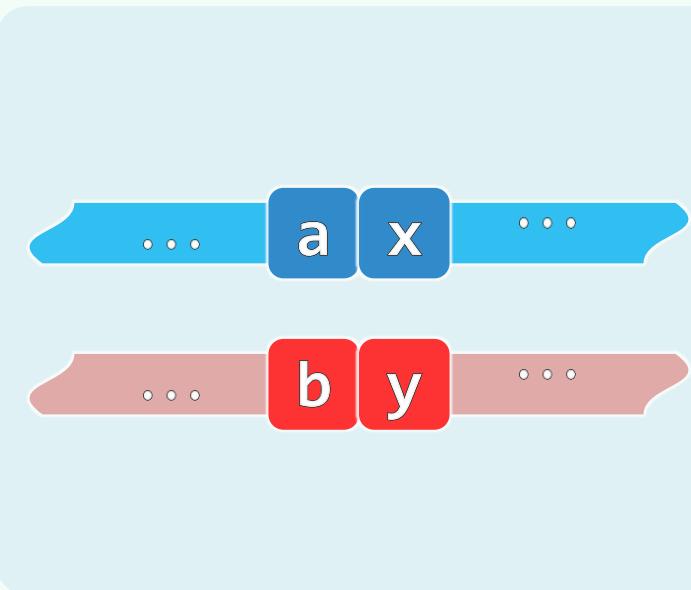
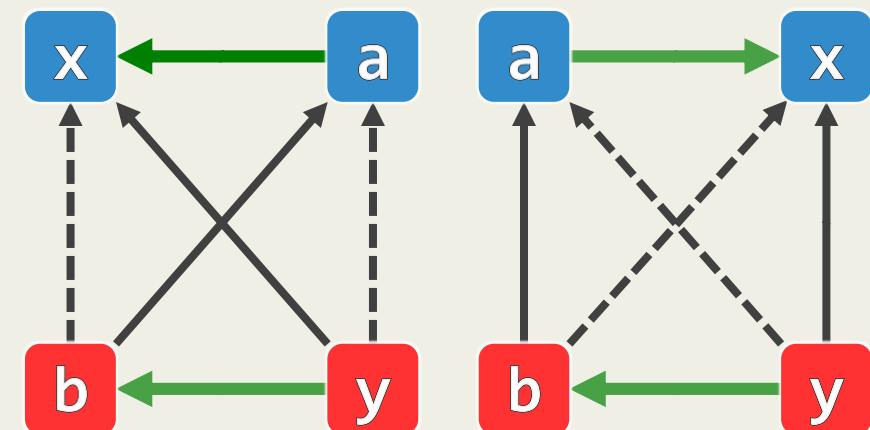


Order Preservation

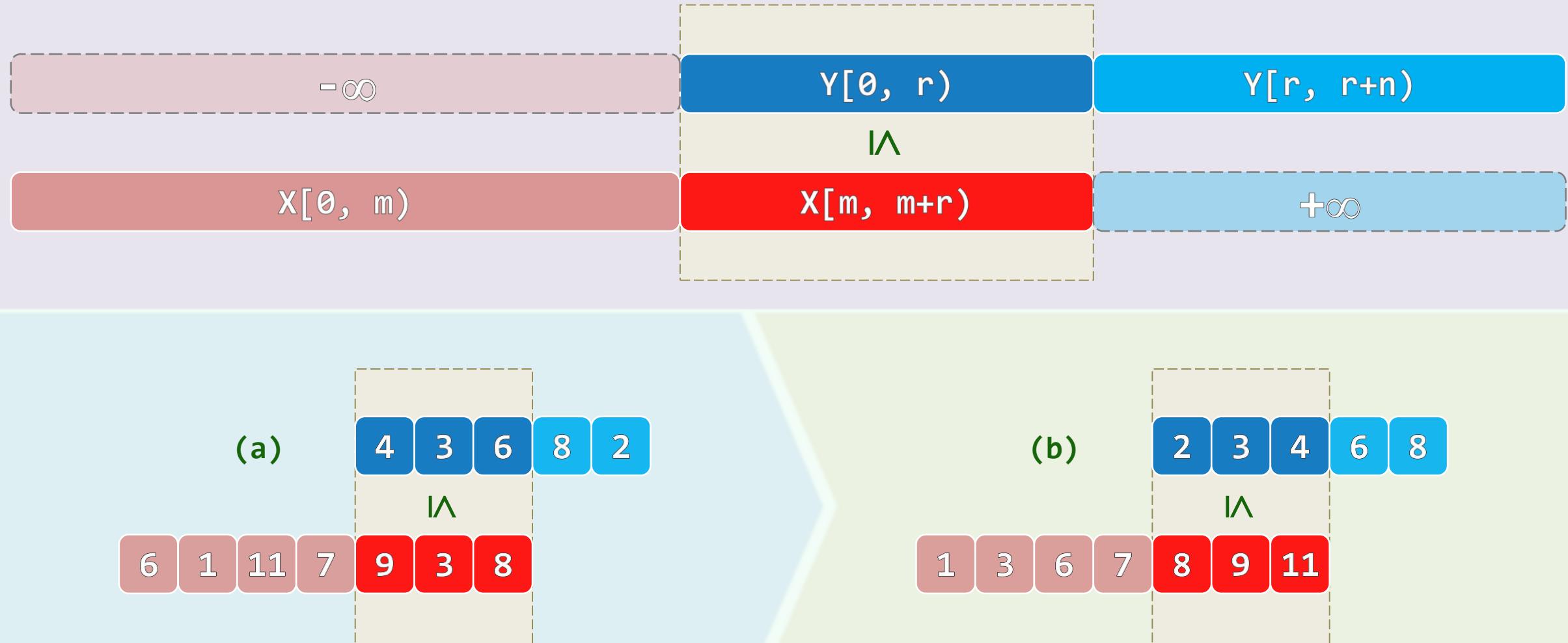
5	8	7	3	5
1	5	2	8	8
0	9	4	6	2
6	3	1	4	7

0	3	1	3	2
1	5	2	4	5
5	8	4	6	7
6	9	7	8	8

0	1	2	3	3
1	2	4	5	5
4	5	6	7	8
6	7	8	8	9



Lemma L

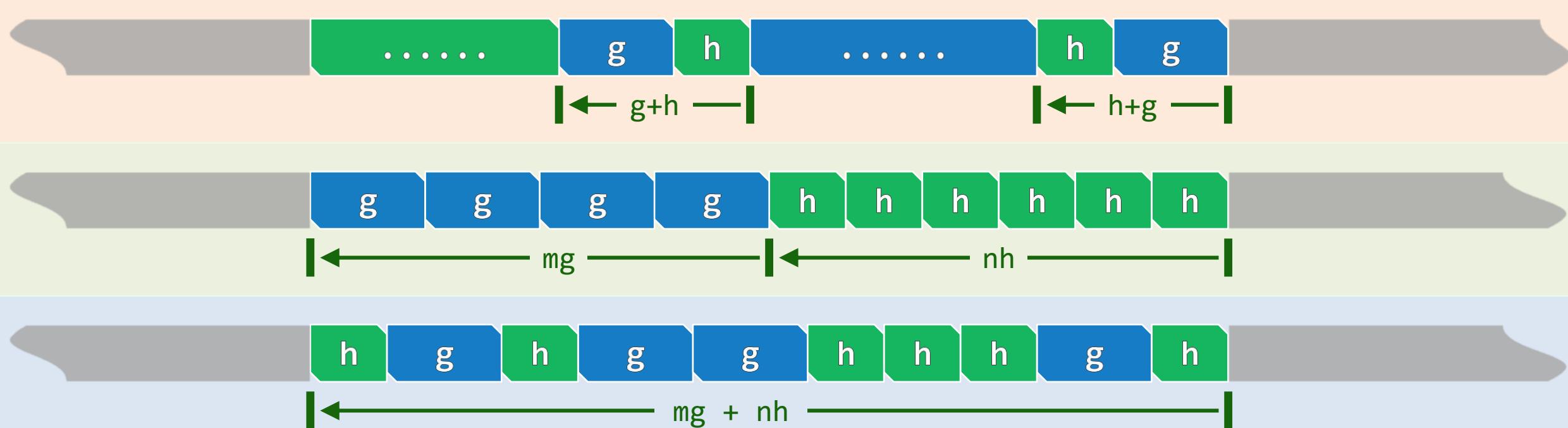


Linear Combination

A sequence that is both **g**-ordered and **h**-ordered

is called **(g,h)-ordered**, which must be both

(g+h)-ordered and **(mg+nh)-ordered** for any $m, n \in \mathbb{N}$

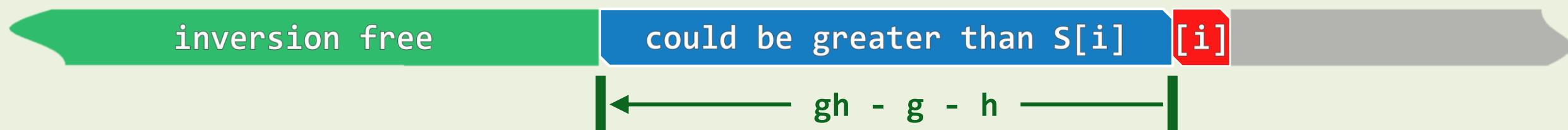


Inversion

- ❖ Let $S[0, n)$ be a (g, h) -ordered sequence, where g and h are **relatively prime**
- ❖ Then for all elements $S[j]$ and $S[i]$, we have

$$i - j > x(g, h) \quad \text{only if} \quad S[j] \leq S[i]$$

- ❖ This implies that to the **LEFT** of each element,
only the previous $x(g, h)$ elements could be **GREATER**



- ❖ There would be no more than $n \cdot x(g, h)$ **INVERSIONs** altogether

排序

希尔排序：PS序列

14 - C3

邓俊辉

deng@tsinghua.edu.cn

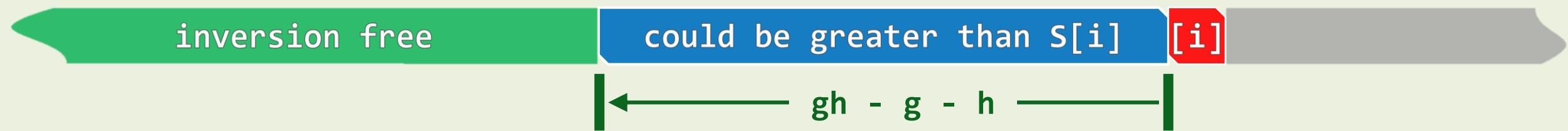
They are like the leaves which a tempest whirls up and
scatters in every direction and then allows to fall.

d-Sorting an $\mathcal{O}(d)$ -Ordered Sequence in $\mathcal{O}(dn)$ Time

❖ If g and h are relatively prime and are both in $\mathcal{O}(d)$

we can d-sort the sequence in $\mathcal{O}(dn)$ time ...

- re-arrange the sequence as a 2D matrix with d columns
- each element is swapped with $\mathcal{O}((g - 1) \cdot (h - 1)/d) = \mathcal{O}(d)$ elements



❖ Since this holds for all elements, $\mathcal{O}(dn)$ steps are enough

PS Sequence

❖ Papernov & Stasevic, 1965 //also called Hibbard's sequence

$$\mathcal{H}_{PS} = \mathcal{H}_{Shell} - 1 = \{ 2^k - 1 \mid k \in \mathcal{N} \} = \{ 1, 3, 7, 15, 31, 63, 127, 255, \dots \}$$

❖ Different items **MAY NOT** be relatively prime, e.g., $h_{2k} = h_k \cdot (h_k + 2)$

But ADJACENT items **MUST** be, since $h_{k+1} - 2 \cdot h_k \equiv 1$

❖ Shellsort with \mathcal{H}_{PS} needs

- $\mathcal{O}(\log n)$ outer iterations and
- $\mathcal{O}(n^{3/2})$ time to sort a sequence of length n //Why ...

$t < k$

- ❖ Let h_t be the h closest to \sqrt{n} and hence $h_t \approx \sqrt{n} = \Theta(n^{1/2})$

1) Consider those iterations for $\{ h_k \mid t < k \} = \{ \overbrace{h_{t+1}, h_{t+2}, \dots, h_m}^{\text{ }} \}$

\therefore there would be $\mathcal{O}(n/h_k)$ elements in each of the h_k columns

\therefore we can **insertionsort** each column in $\mathcal{O}((n/h_k)^2)$ time

\therefore each h_k -sorting costs $\mathcal{O}(n^2/h_k)$ time

\therefore all these iterations cost time of

$$\mathcal{O}(2 \times n^2/h_t) = \mathcal{O}(n^{3/2})$$

$$\begin{aligned} k &= t \\ h_k &= h_t \end{aligned}$$

$$\begin{aligned} t &< k \\ h_t &< h_k \end{aligned}$$

t h_t
VI VI
 \nwarrow h_k

$k \leq t$

2) Consider those iterations for $\{ h_k \mid k \leq t \} = \{ \overleftarrow{h_1, h_2, \dots, h_t} \}$

$\therefore h_{k+1}$ and h_{k+2} are relatively prime and are both in $\mathcal{O}(h_k)$

\therefore each h_k -sorting costs $\mathcal{O}(n \times h_k)$ time

\therefore all these iterations cost $\mathcal{O}(n \times 2 \cdot h_t) = \mathcal{O}(n^{3/2})$ time

- ❖ This upper bound is TIGHT
- ❖ What about the average cases?
 - $\mathcal{O}(n^{5/4})$ based on simulations
 - but not proved yet

$$\begin{aligned}t &< k \\ h_t &< h_k\end{aligned}$$

$$\begin{aligned}k &= t \\ h_k &= h_t\end{aligned}$$

t h^t
 $\vee\!\!/\!$ $\vee\!\!/\!$
 \blacktriangleright h_k

14 - C4

排序

希尔排序：Pratt序列

邓俊辉

deng@tsinghua.edu.cn

聚沙成塔，集腋成裘

Pratt's Sequence, 1971

$$\mathcal{H}_{pratt} = \{ 2^p \cdot 3^q \mid p, q \in \mathcal{N} \}$$

$$= \{ 1, 2, 3, 4, 6, 8, 9, 12, 16, 18, 24, 27, 32, 36, \dots \}$$

❖ Note that

- adjacent items are NOT always relatively prime and
- there are $\mathcal{O}(\log^2 n)$ items no greater than n

❖ With \mathcal{H}_{pratt} ,

Shellsort sorts a sequence of length n in $\mathcal{O}(n \cdot \log^2 n)$ time ...

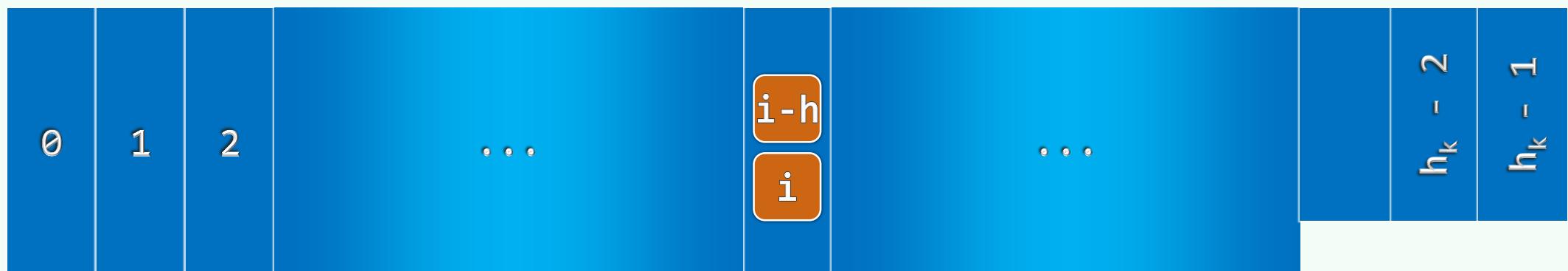
From $(2,3)$ -ordered to 1-ordered

$$\therefore x(2,3) = 2 \cdot 3 - 2 - 3 = 1$$

\therefore To the **LEFT** of each element in a $(2,3)$ -ordered sequence,

only the **NEXT** element can be smaller

\therefore It costs $\mathcal{O}(n)$ time to sort such a sequence



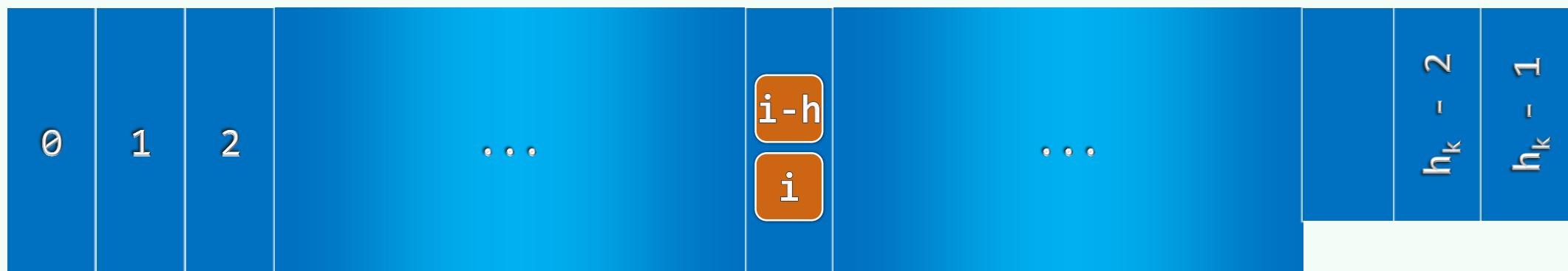
From $(2*h_k, 3*h_k)$ -ordered to h_k -ordered

❖ Divide S into h_k subsequences, each of which is $(2,3)$ -ordered

∴ it costs altogether $\mathcal{O}(n)$ time to sort them resp.

❖ ∵ there are altogether $\mathcal{O}(\log^2 n)$ iterations

∴ we need $\mathcal{O}(n \cdot \log^2 n)$ time



14 - C5

排序

希尔排序：Sedgewick序列

邓俊辉

deng@tsinghua.edu.cn

Sedgewick's Sequence

- ❖ Pratt's sequence requires **too many iterations** and hence is not good for pre-sorted sequences
- ❖ Sedgewick's sequence: a combination of PS's sequence with Pratt's

$$\{ 1 \quad 5 \quad 19 \quad 41 \quad 109 \quad 209 \quad 505 \quad 929 \quad 2161 \quad 3905 \quad 8929 \quad 16001 \quad 36289 \quad 64769 \quad \dots \}$$

$$\{ 9 \times 4^k - 9 \times 2^k + 1 \mid k \geq 0 \} \cup \{ 4^k - 3 \times 2^k + 1 \mid k \geq 2 \}$$

- worst $\mathcal{O}(n^{4/3})$ & average $\mathcal{O}(n^{7/6})$
- best performance in practice

- ❖ Is there a step sequence with $\mathcal{O}(n \cdot \log n)$ worst-case performance?