



快速排序

- 快速排序算法
- 时空复杂度分析
- 优化策略

数据之法
结构之美
算法之道

快速排序算法的提出者



Charles Antony Richard Hoare

图灵奖获得者

英国皇家科学院院士

英国皇家工程院院士

牛津大学教授

微软剑桥研究院首席研究员

There are two ways of constructing a software design: One way is to make it so simple that there are obviously no errors, and the other way is to make it so complicated that there are no obvious errors. The first method is far more difficult.

快速排序

Algorithms

ALGORITHM 64
QUICKSORT

C. A. R. HOARE

Elliott Brothers Ltd., Borehamwood, Hertfordshire, Eng.

procedure quicksort (A,M,N); **value** M,N;

array A; **integer** M,N;

comment Quicksort is a very fast and convenient method of sorting an array in the random-access store of a computer. The entire contents of the store may be sorted, since no extra space is required. The average number of comparisons made is $2(M-N) \ln(N-M)$, and the average number of exchanges is one sixth this amount. Suitable refinements of this method will be desirable for its implementation on any actual computer;

begin **integer** I,J;
 if M < N **then begin** partition (A,M,N,I,J);
 quicksort (A,M,J);
 quicksort (A, I, N)

end

end quicksort

- 1960年提出，1961年实现，1962年论文发表。
- 被IEEE评为20世纪十大算法之一。



IEEE

快速排序的基本思想

- ① 选取待排序数组中的某个元素 K （例如第1个元素）作为**基准元素**（也称为主元、轴点，Pivot），按照其关键词的大小，对数组重新排列，使整个数组划分为左右两个子数组：
- ✓ 左侧子数组中元素的值都小于或等于 K ；
 - ✓ 右侧子数组中元素的值都大于 K ；
 - ✓ 基准元素 K 排在两个子数组中间。
- 此时 K 已就位，其已放在排完序后它应该所在正确位置
- ② 分别对两个子数组重复上述方法，直至所有元素都排在正确位置上。

 $\leq K$ K $> K$

快速排序算法简要描述

```
void QuickSort(int R[], int n) {  
    if(R的长度大于1) {  
        将R划分为两个子数组LeftR 和 RightR;  
        QuickSort(LeftR);  
        QuickSort(RightR);  
    }  
}
```

操作①: Partition
分划、分割、划分

将R划分为两个子数组LeftR 和 RightR; // ①
QuickSort(LeftR); // ②
QuickSort(RightR); // ③

LeftR ($\leq K$)

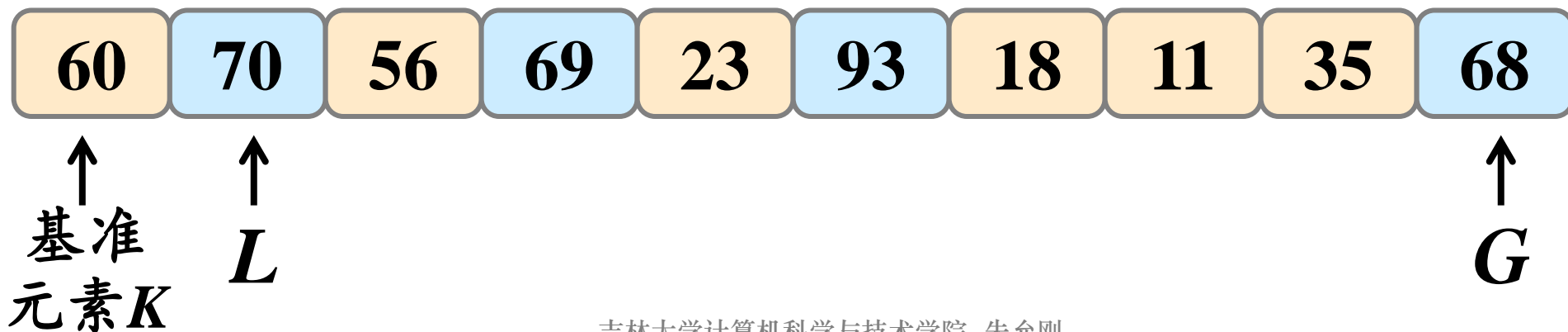
K

RightR ($> K$)

特点：一趟分划把一个记录放在最终的位置上。

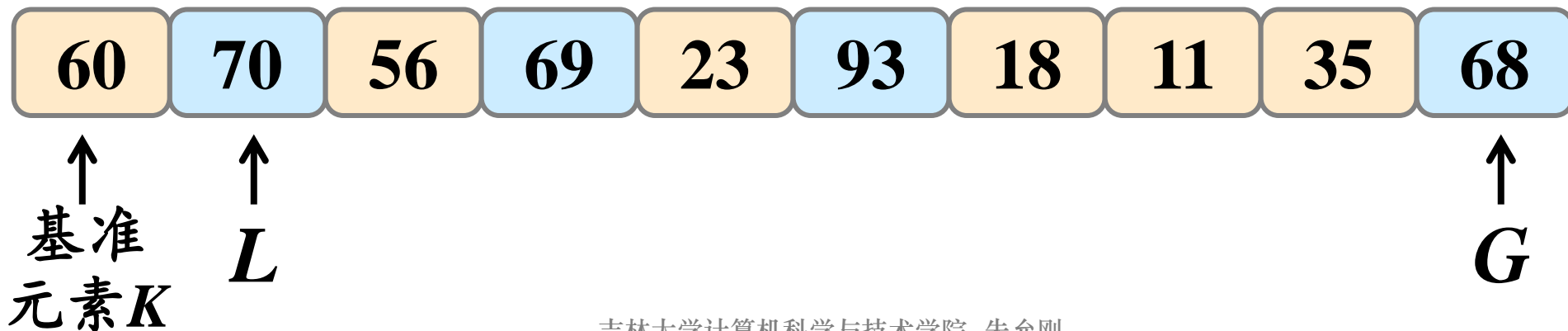
Partition操作算法思想

- 思想：把比 K 大的元素移到数组右边，把比 K 小的元素移到数组左边。
- 引入两个指针 L 和 G 相向而行：指针 L 从左向右扫描数组找第一个 $>K$ 的元素，指针 G 从右向左找第一个 $\leq K$ 的元素，交换 $R[L]$ 和 $R[G]$ 。重复如此，不断把 $\leq K$ 的元素换到数组左边，大于 K 的元素换到数组右边，直至指针 L 和 G 相遇。

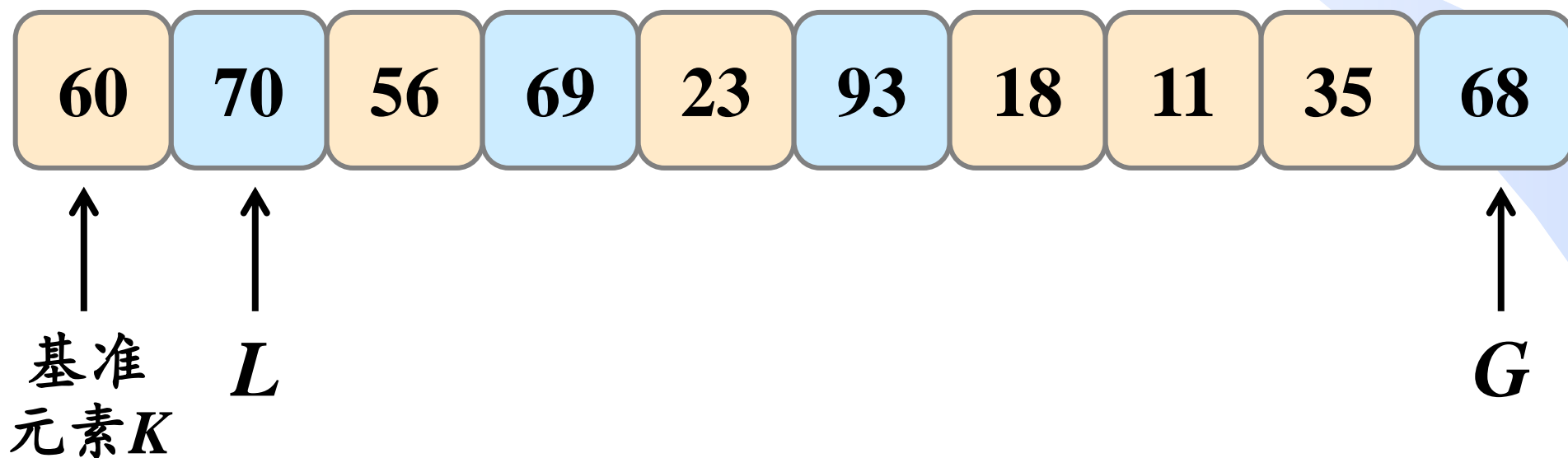


Partition操作算法思想

- 指针 L (Less) : 喜欢比 K 小的元素, 讨厌比 K 大的元素。从左向右扫描时, 遇到比 K 小的元素就继续扫描, 从而把比 K 小的元素留在自己这一边, 当遇到比 K 大的元素就把它换到另一边。
- 指针 G (Greater) : 喜欢比 K 大的元素, 讨厌比 K 小的元素。从右向左扫描时, 遇到比 K 大的元素就继续扫描, 从而把比 K 大的元素留在自己这一边, 当遇到比 K 小的元素就把它换到另一边。



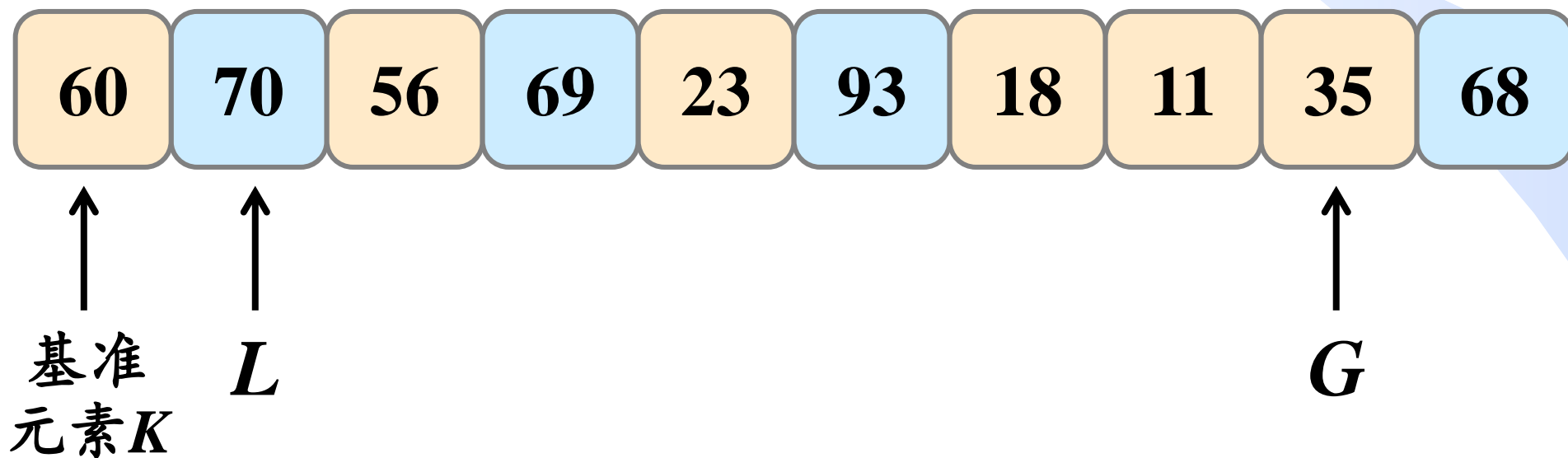
分划过程—例子



指针 L : 从左向右找第一个大于 K 的元素

指针 G : 从右向左找第一个小于等于 K 的元素

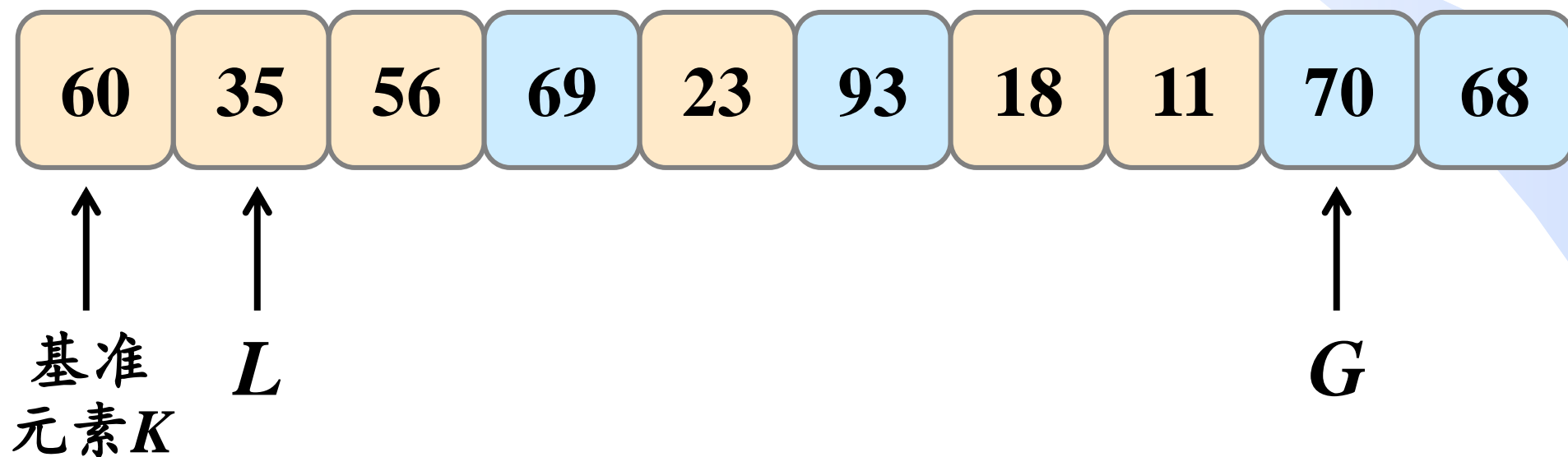
分划过程—例子



指针 L : 从左向右找第一个大于 K 的元素

指针 G : 从右向左找第一个小于等于 K 的元素

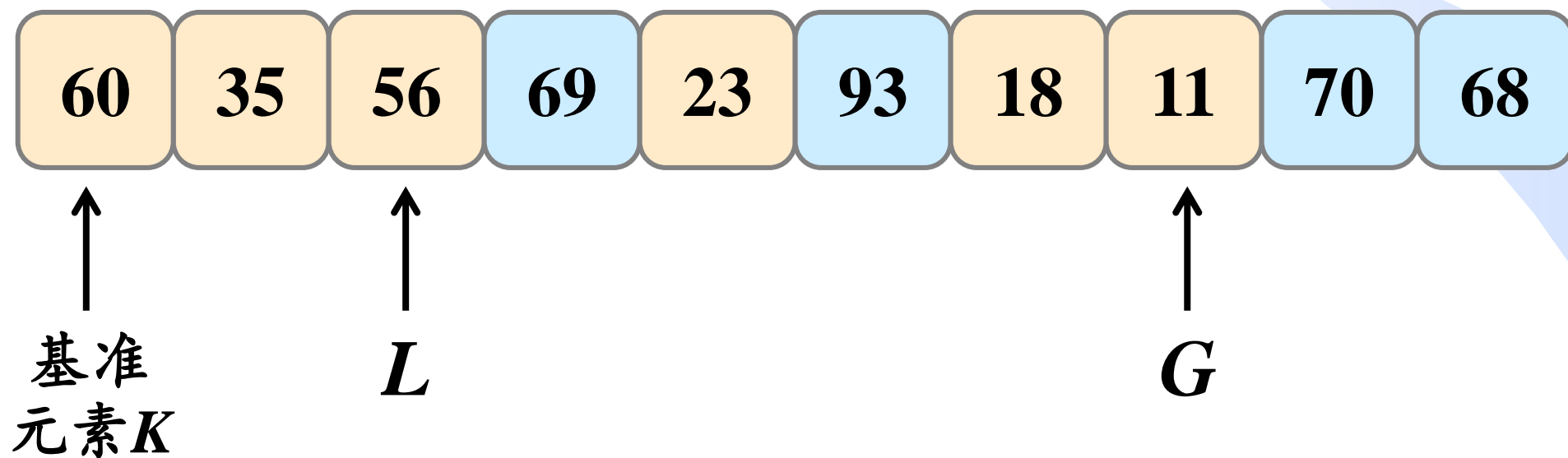
分划过程—例子



指针 L : 从左向右找第一个大于 K 的元素

指针 G : 从右向左找第一个小于等于 K 的元素

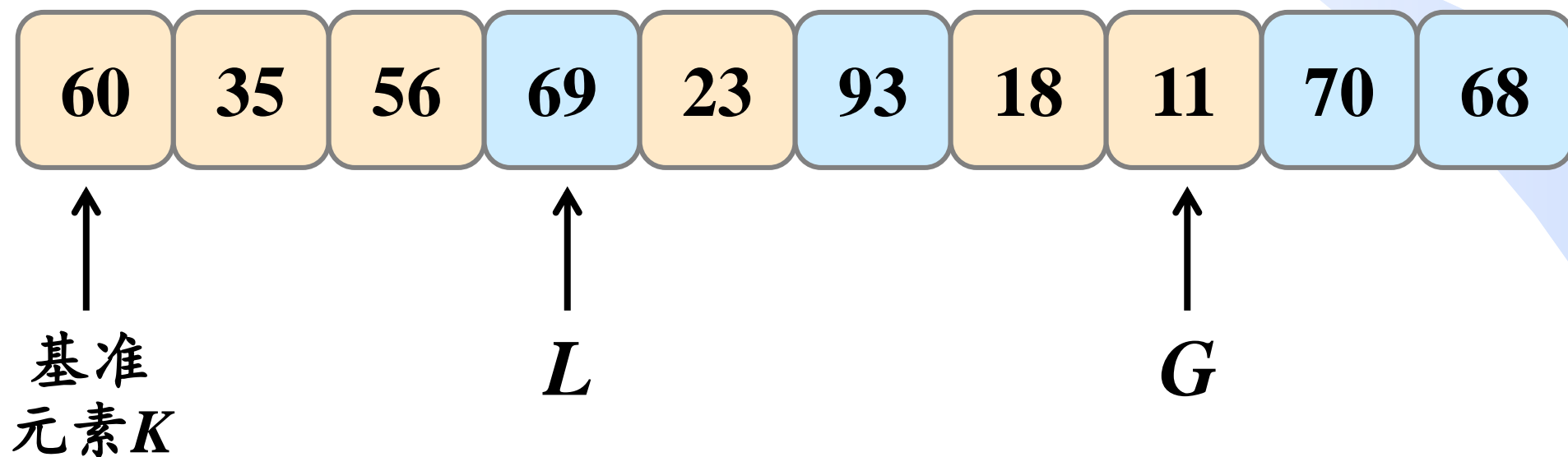
分划过程—例子



指针 L : 从左向右找第一个大于 K 的元素

指针 G : 从右向左找第一个小于等于 K 的元素

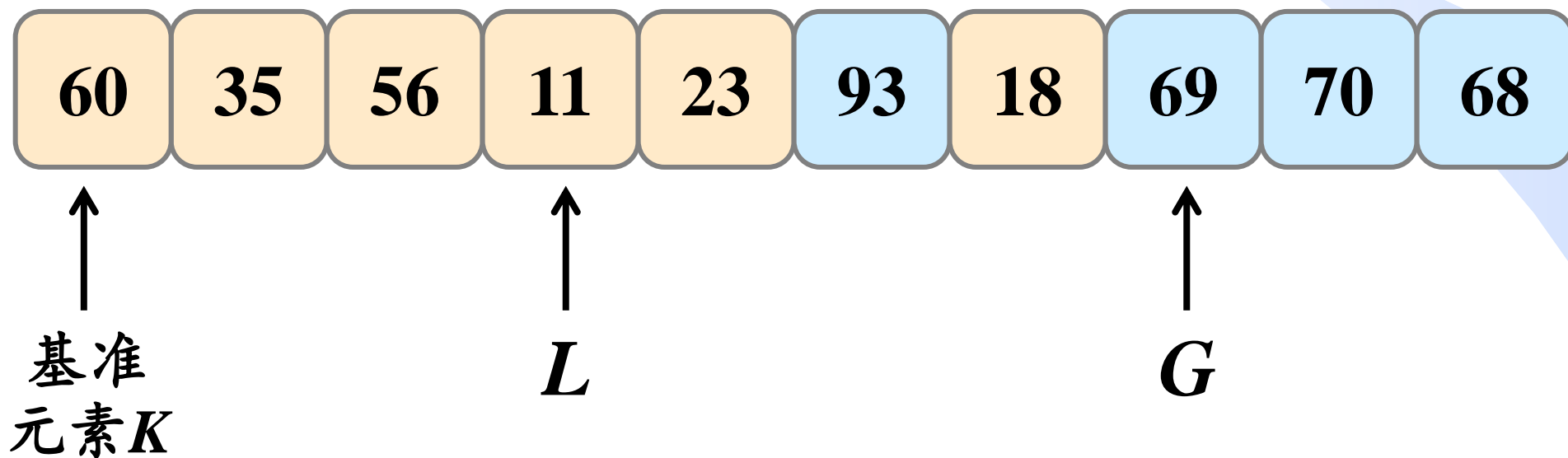
分划过程一例子



指针 L : 从左向右找第一个大于 K 的元素

指针 G : 从右向左找第一个小于等于 K 的元素

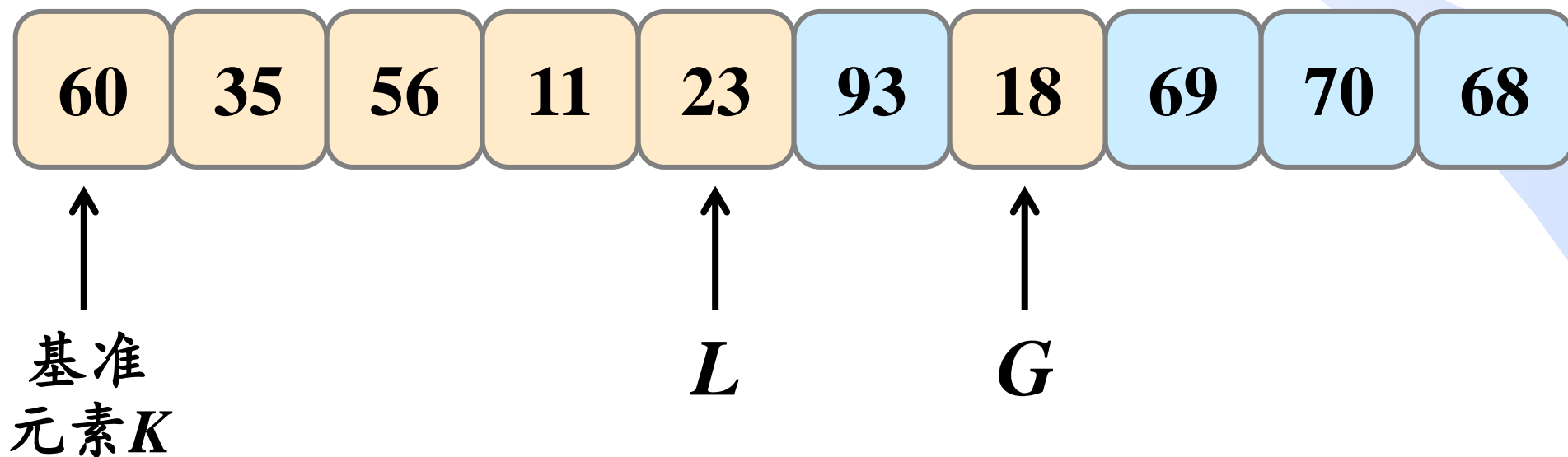
分划过程一例子



指针 L : 从左向右找第一个大于 K 的元素

指针 G : 从右向左找第一个小于等于 K 的元素

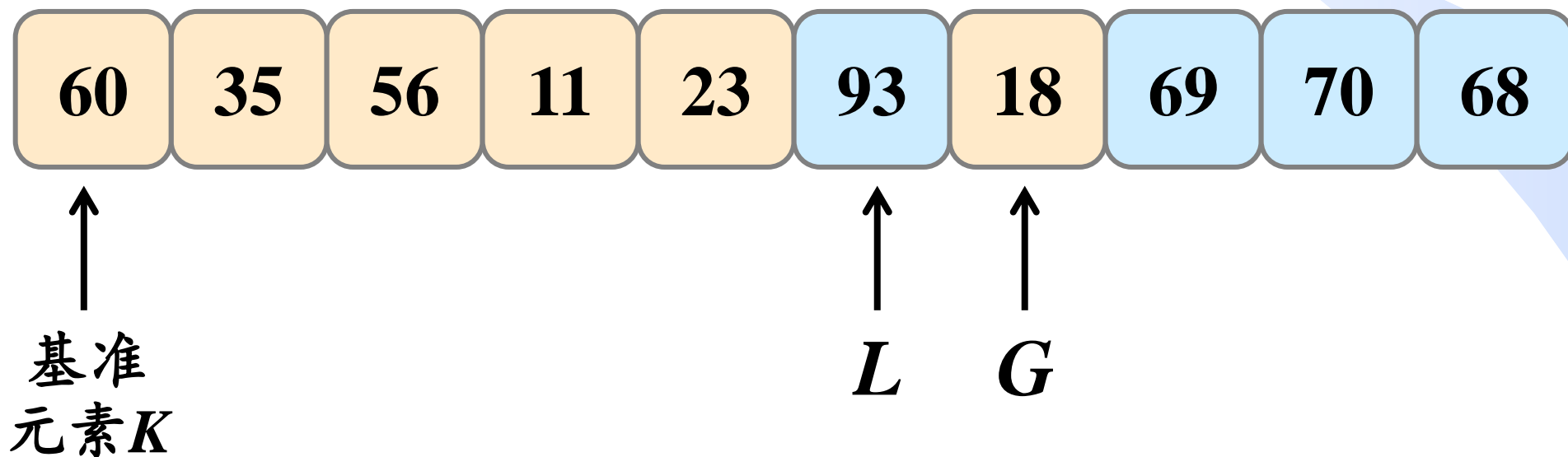
分划过程—例子



指针 L : 从左向右找第一个大于 K 的元素

指针 G : 从右向左找第一个小于等于 K 的元素

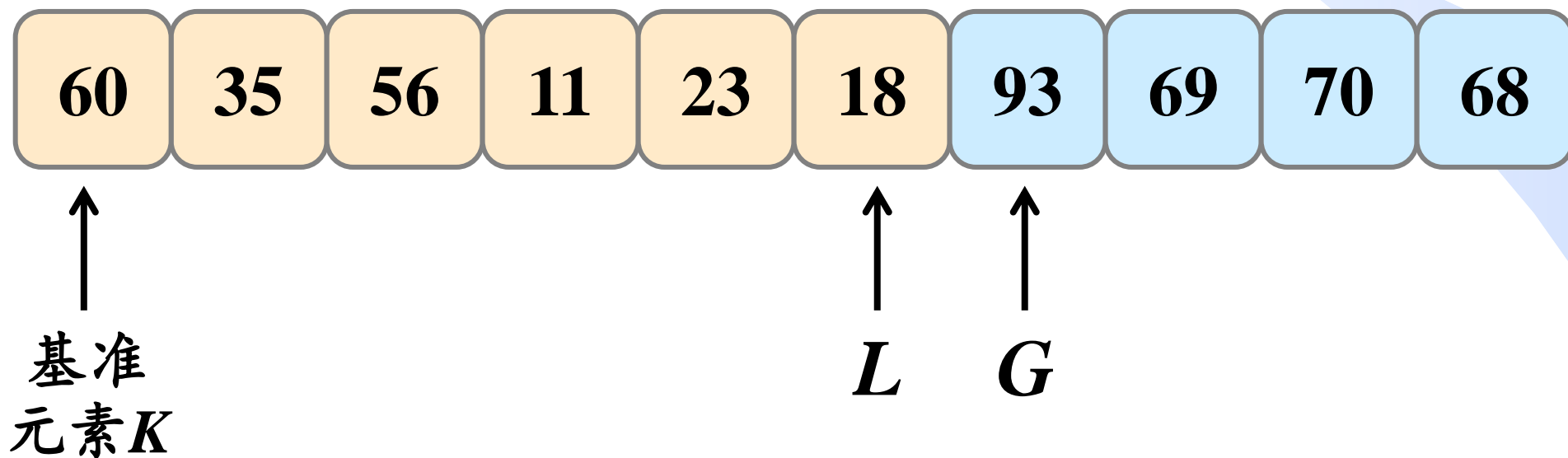
分划过程—例子



指针 L : 从左向右找第一个大于 K 的元素

指针 G : 从右向左找第一个小于等于 K 的元素

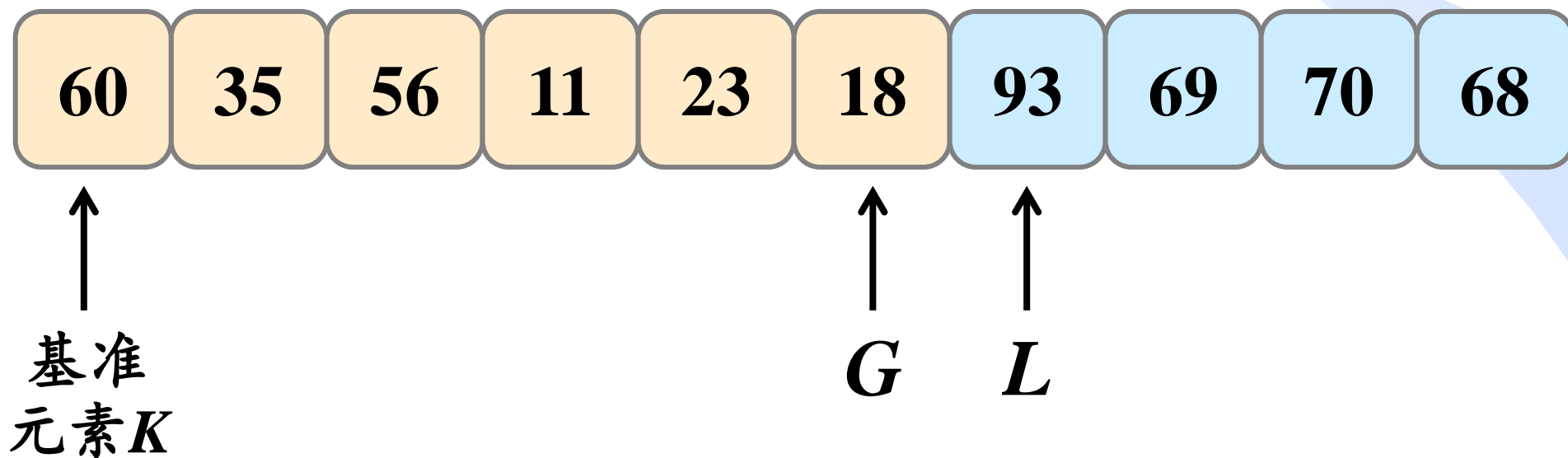
分划过程—例子



指针 L : 从左向右找第一个大于 K 的元素

指针 G : 从右向左找第一个小于等于 K 的元素

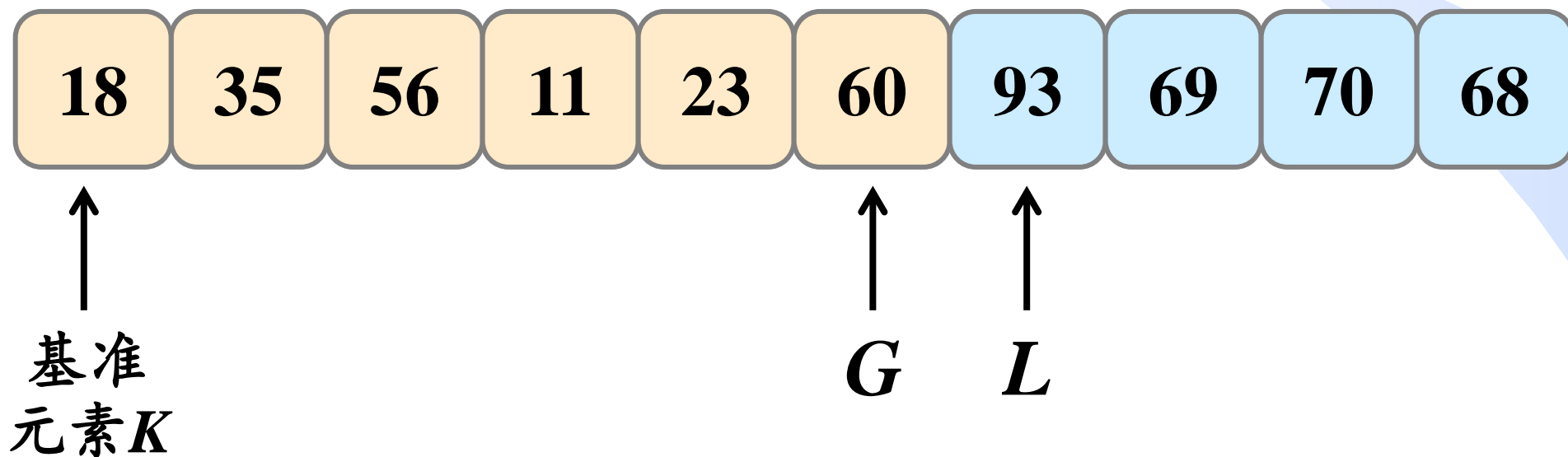
分划过程—例子



指针 L : 从左向右找第一个大于 K 的元素

指针 G : 从右向左找第一个小于等于 K 的元素

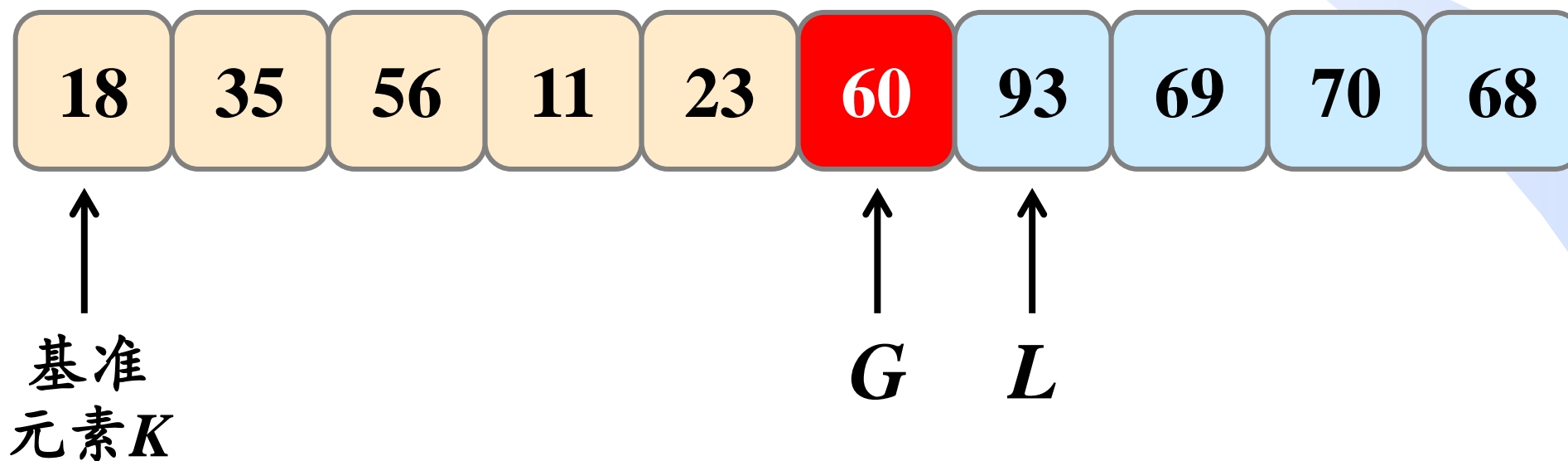
分划过程—例子



指针 L : 从左向右找第一个大于 K 的元素

指针 G : 从右向左找第一个小于等于 K 的元素

分划过程一例子



指针 L : 从左向右找第一个大于 K 的元素

指针 G : 从右向左找第一个小于等于 K 的元素

Partition算法

```

int Partition(int R[], int m, int n){ //对子数组 $R_m \dots R_n$ 分划
    int K=R[m], L=m+1, G=n;          // $R_m$ 为基准元素
    while(L<=G) {
        while(L<=n && R[L]<=K) L++; //从左向右找第一个 $>K$ 的元素
        while(R[G]>K) G--;           //从右向左找第一个 $\leq K$ 的元素
        if(L<G) {swap(R[L],R[G]); L++; G--;}
    }
    swap(R[m],R[G]);
    return G;
}

```

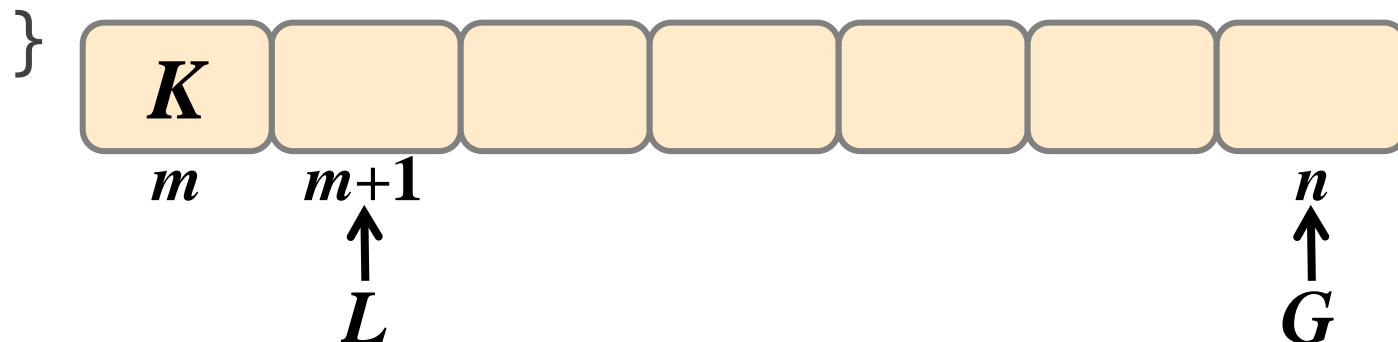
为何G不考虑越界?

时间复杂度
 $O(n)$

```

void swap(int &a,int &b){
    int temp = a;
    a = b;
    b = temp;
}

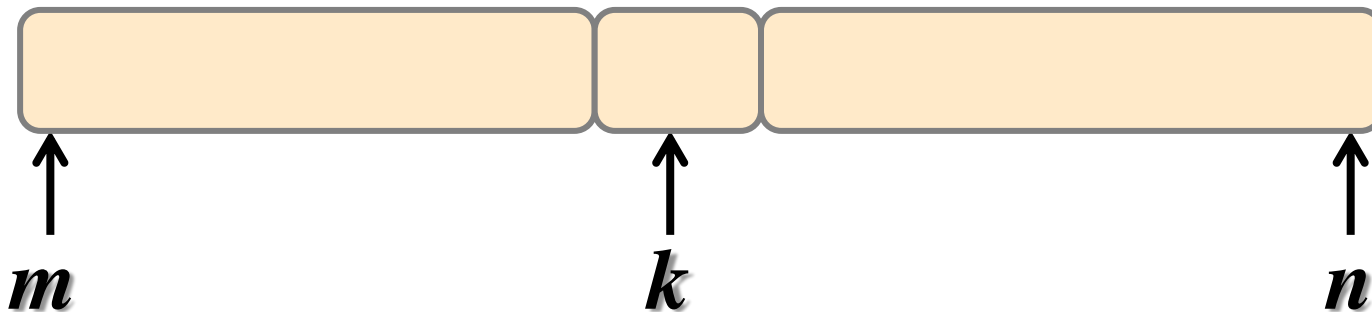
```



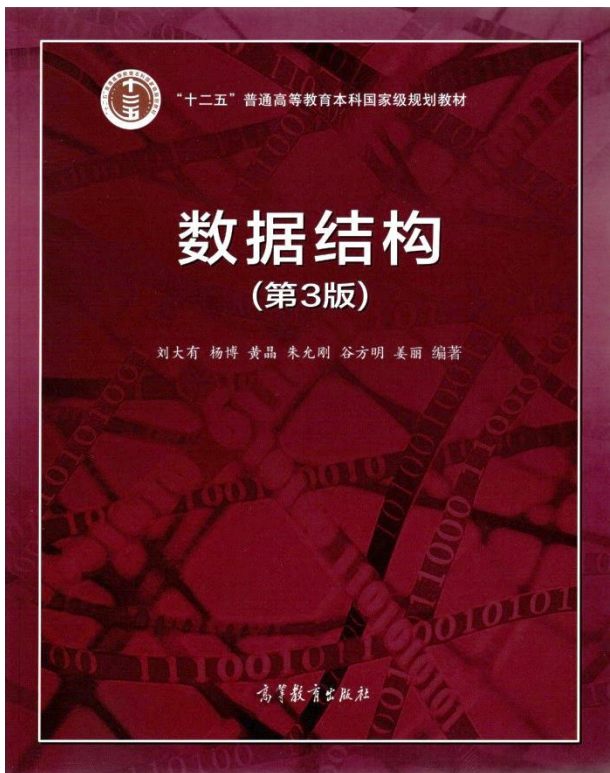
快速排序算法

```
void QuickSort(int R[], int m, int n){ //对 $R_m \dots R_n$ 递增排序
    if(m < n){
        int k=Partition(R, m, n);
        QuickSort(R, m, k-1);
        QuickSort(R, k+1, n);
    }
}
```

初始调用: QuickSort(R, 1, n)



分治法



快速排序

- 快速排序算法
- 时空复杂度分析
- 优化策略

数据之法
结构之美
算法之道

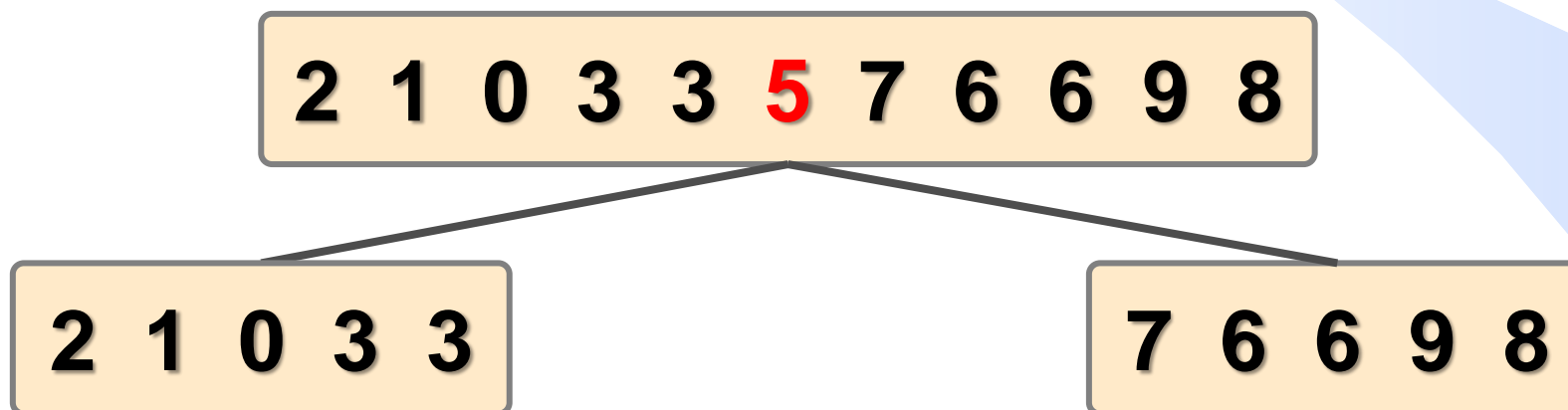
时空复杂度—最好情况

每次分划发生在最中间，把子数组分成长度相等的两半

5 1 0 3 3 2 7 6 6 9 8

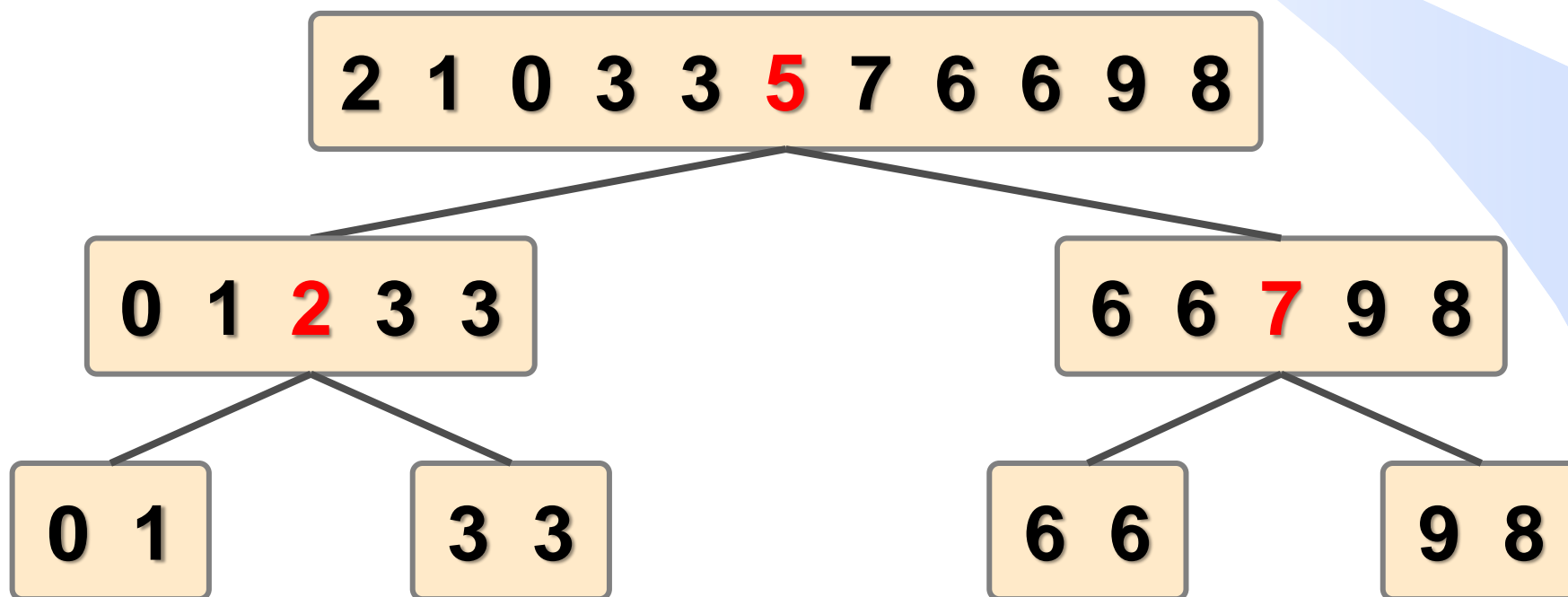
时空复杂度—最好情况

每次分划发生在最中间，把子数组分成长度相等的两半



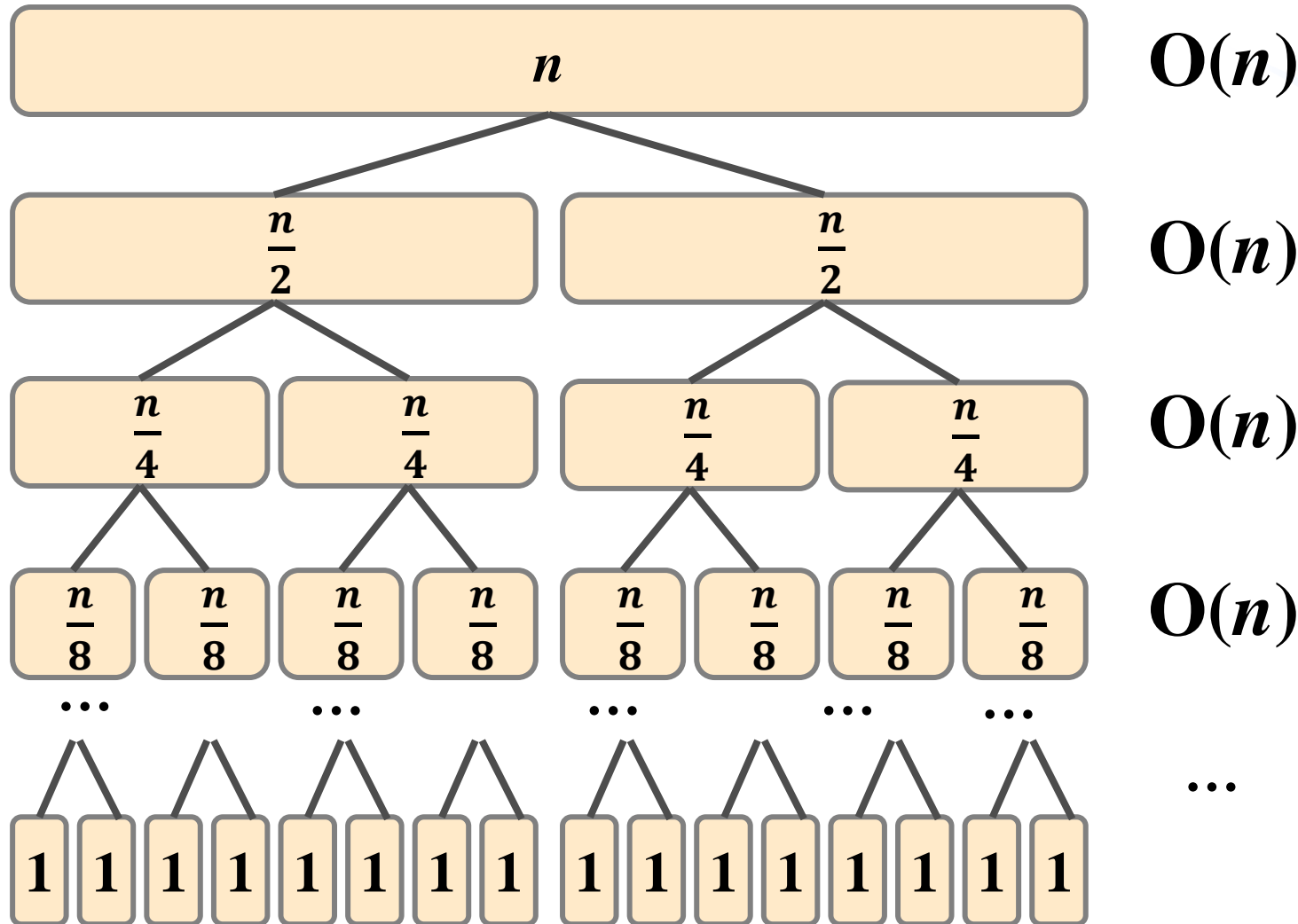
时空复杂度—最好情况

每次分划发生在最中间，把子数组分成长度相等的两半



时空复杂度—最好情况

A



每次分划发生在最中间，把子数组分成长度相等的两半

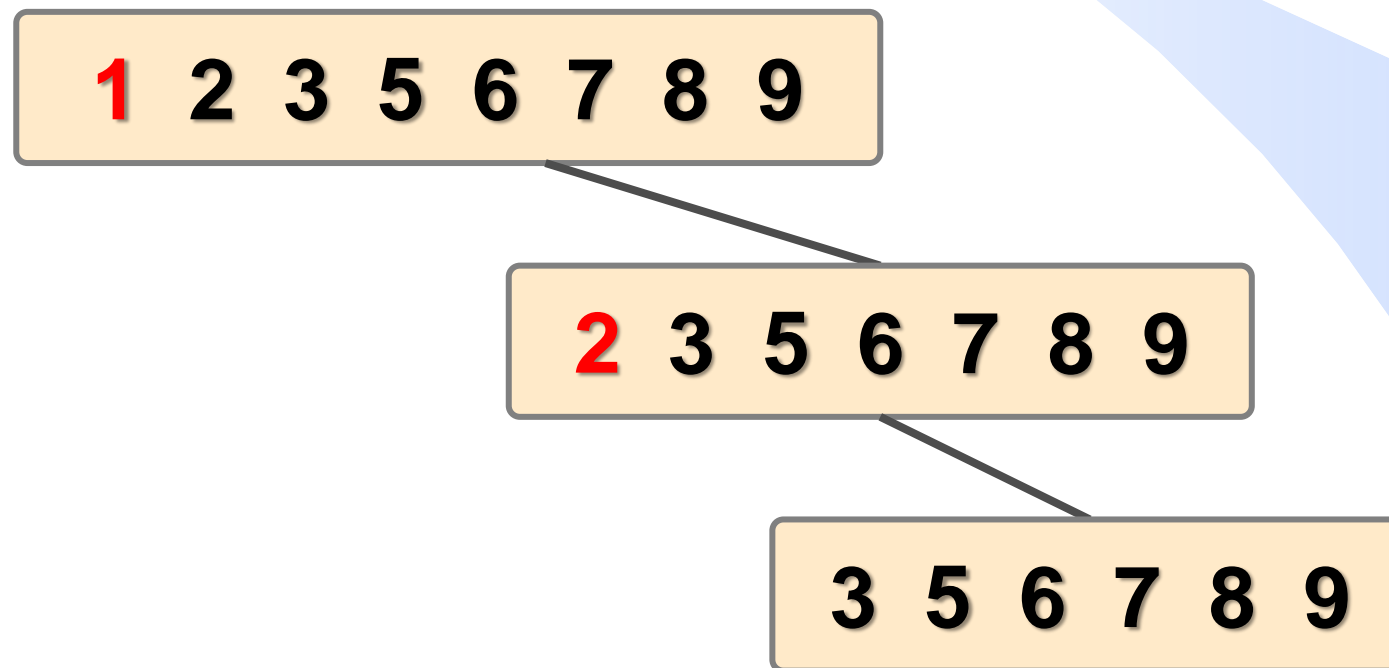
$\log n$

时间复杂度
 $O(n \log n)$

空间复杂度
 $O(\log n)$

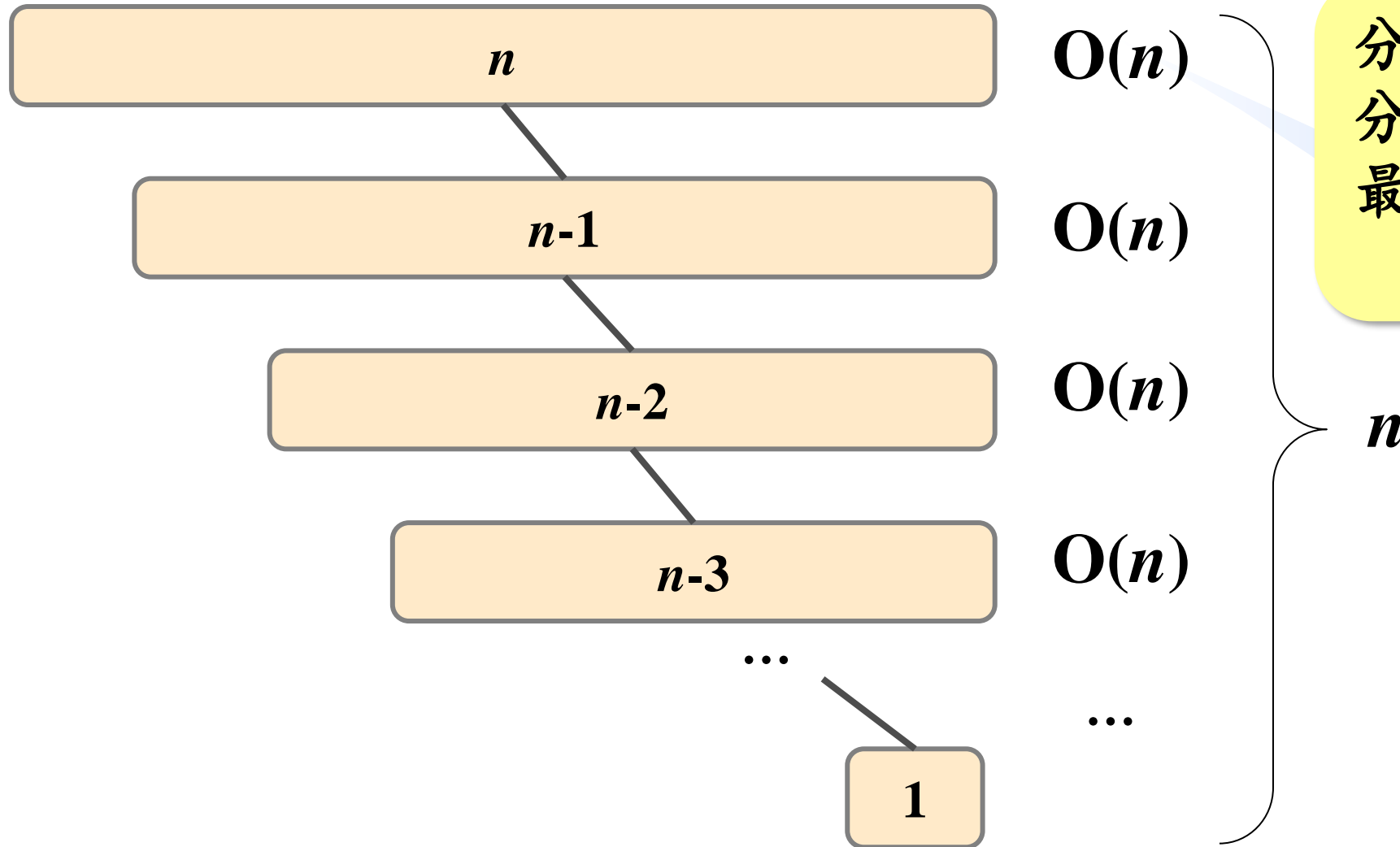
时空复杂度—最坏情况

分划极不平衡，每次分划都发生在子数组最边上，如初始时已排好序或逆序



时空复杂度—最坏情况

A

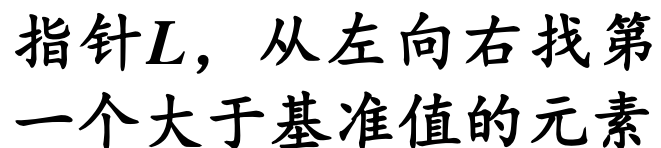


分划极不平衡，每次分划都发生在子数组最边上，如初始时已排好序或逆序

时间复杂度
 $O(n^2)$

空间复杂度
 $O(n)$

最多 $n+1$ 次

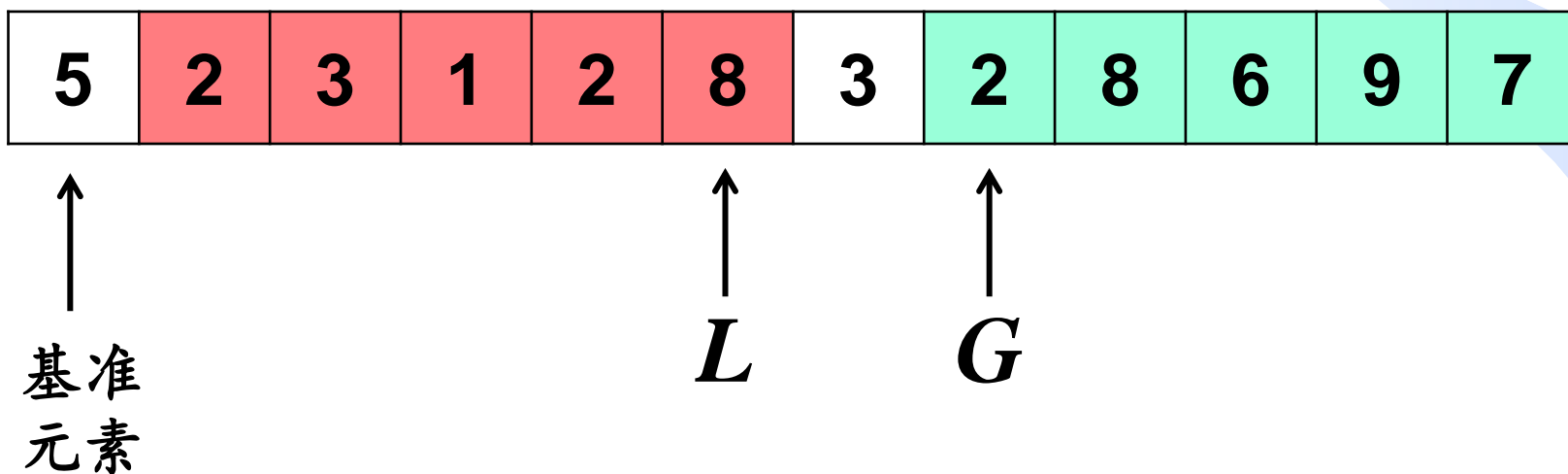


指针G，从右向左找第一个小于等于基准值的元素

时间复杂度—平均情况

Partition操作关键词比较次数

最多 $n+1$ 次



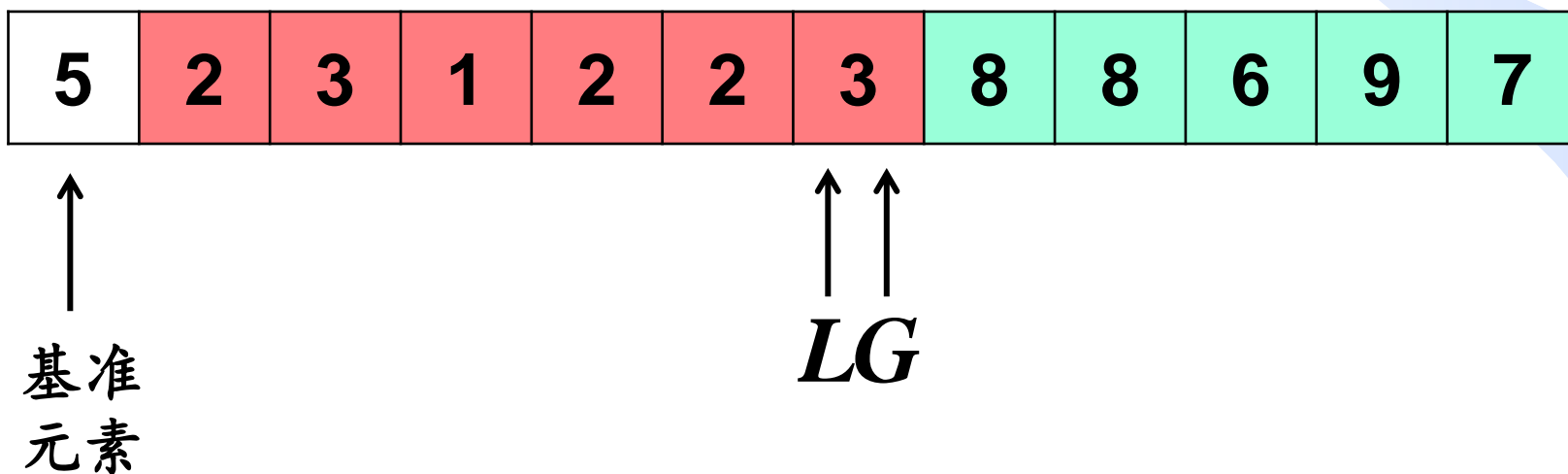
指针 L ，从左向右找第一个大于基准值的元素

指针 G ，从右向左找第一个小于等于基准值的元素

时间复杂度—平均情况

Partition操作关键词比较次数

最多 $n+1$ 次



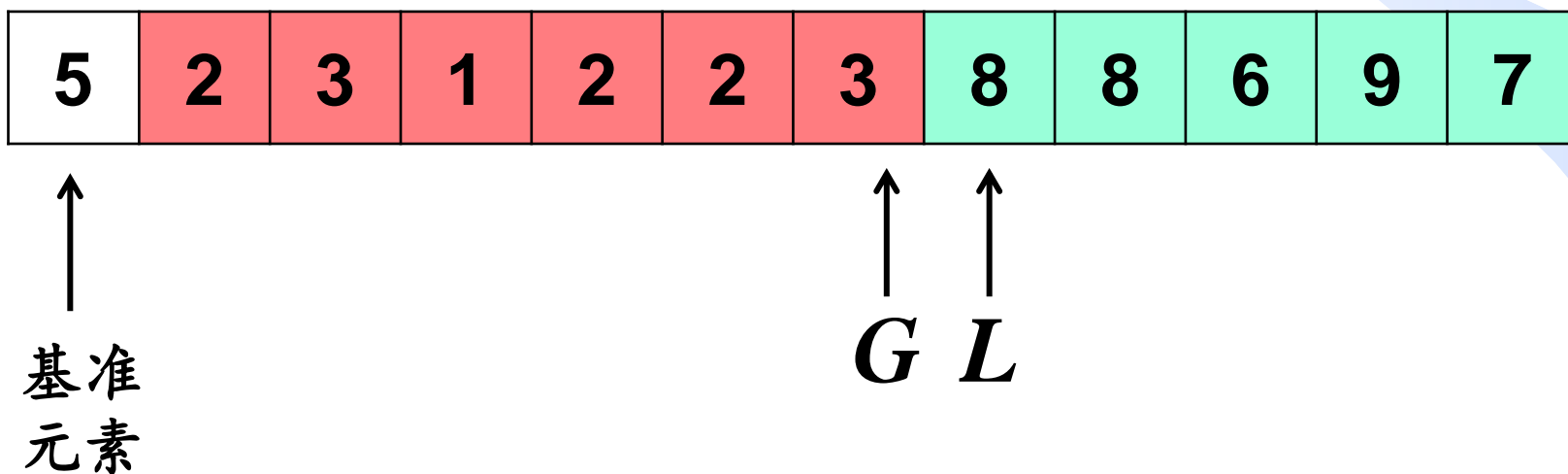
指针 L ，从左向右找第一个大于基准值的元素

指针 G ，从右向左找第一个小于等于基准值的元素

时间复杂度—平均情况

Partition操作关键词比较次数

最多 $n+1$ 次

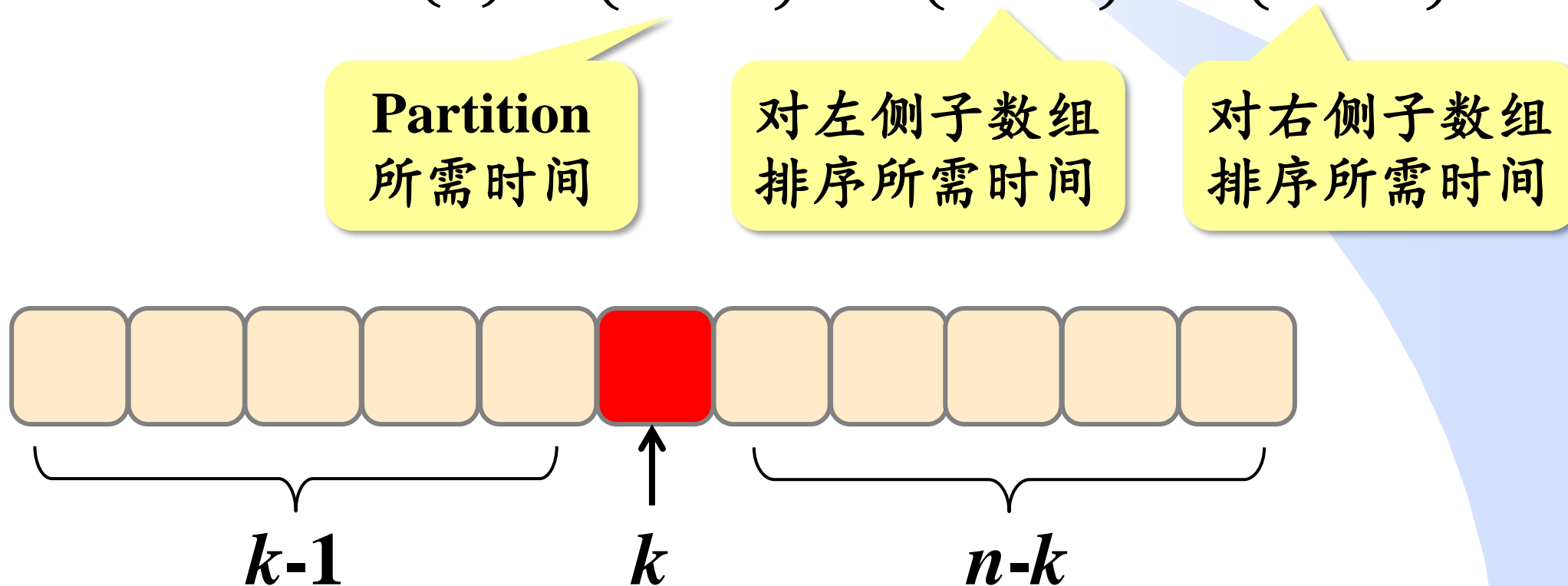


指针 L ，从左向右找第一个大于基准值的元素

指针 G ，从右向左找第一个小于等于基准值的元素

时间复杂度—平均情况

- 假定经过Partition之后，基准元素最终放在数组第 k 个位置，即Partition函数返回值为 k
- 关键词比较次数： $T(n) = (n + 1) + T(k - 1) + T(n - k)$



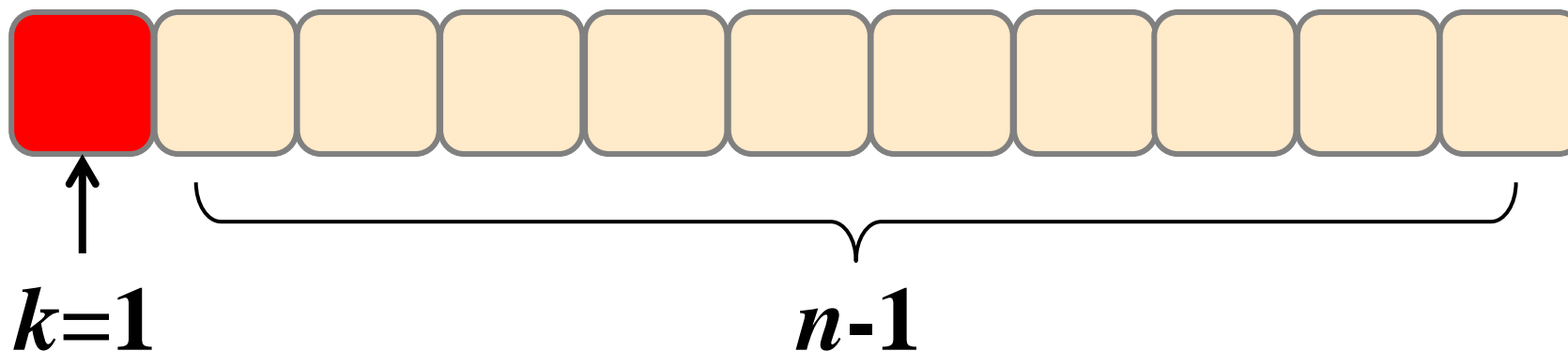
时间复杂度—平均情况

- $k=1$: 经过Partition之后, 基准元素最终放在数组第1个位置, 即Partition函数返回值为1
- 关键词比较次数: $T(n) = (n + 1) + T(0) + T(n - 1)$

Partition
所需时间

对左侧子数组
排序所需时间

对右侧子数组
排序所需时间



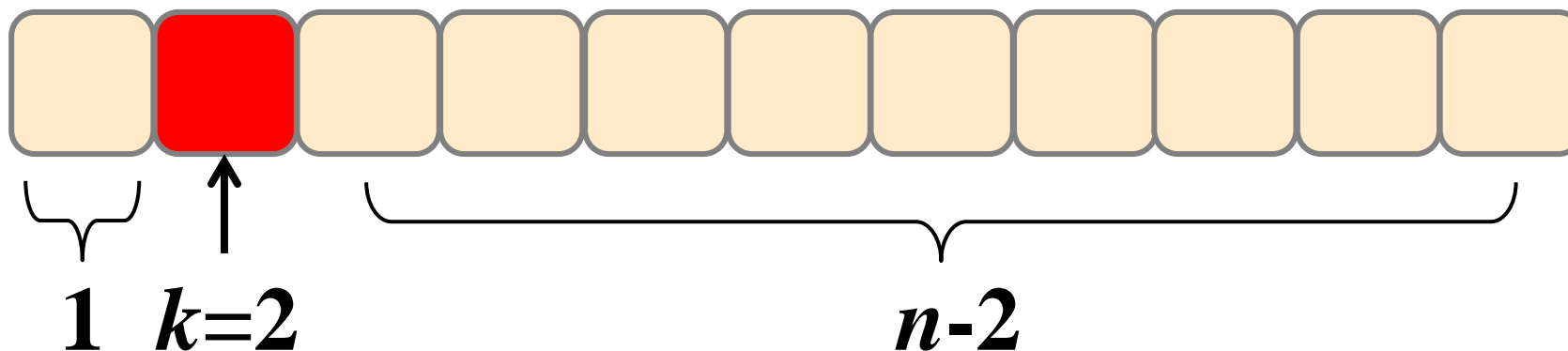
时间复杂度—平均情况

- $k=2$: 经过Partition之后, 基准元素最终放在数组第2个位置, 即Partition函数返回值为2
- 关键词比较次数: $T(n) = (n + 1) + T(1) + T(n - 2)$

Partition
所需时间

对左侧子数组
排序所需时间

对右侧子数组
排序所需时间



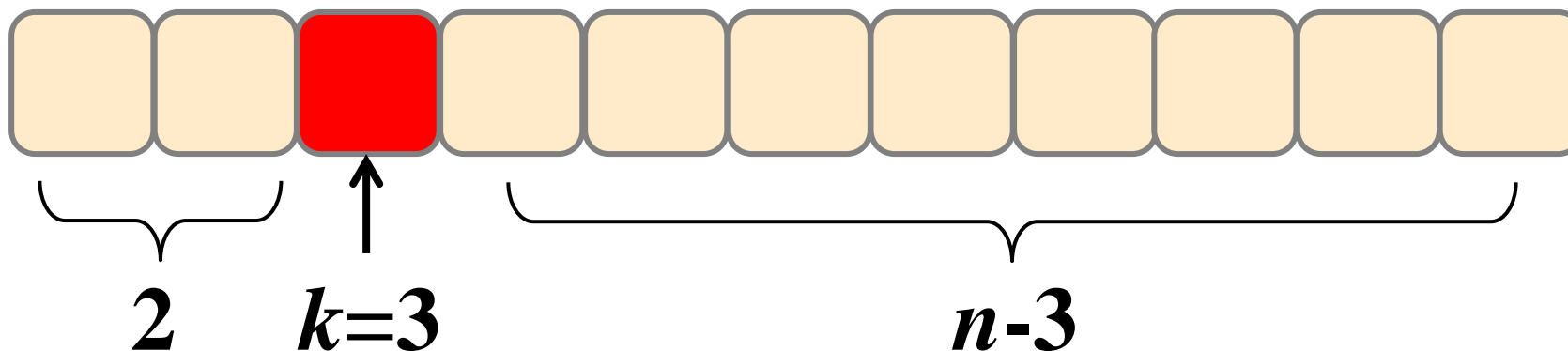
时间复杂度—平均情况

- $k=3$: 经过Partition之后, 基准元素最终放在数组第3个位置, 即Partition函数返回值为3
- 关键词比较次数: $T(n) = (n + 1) + T(2) + T(n - 3)$

Partition
所需时间

对左侧子数组
排序所需时间

对右侧子数组
排序所需时间



时间复杂度—平均情况

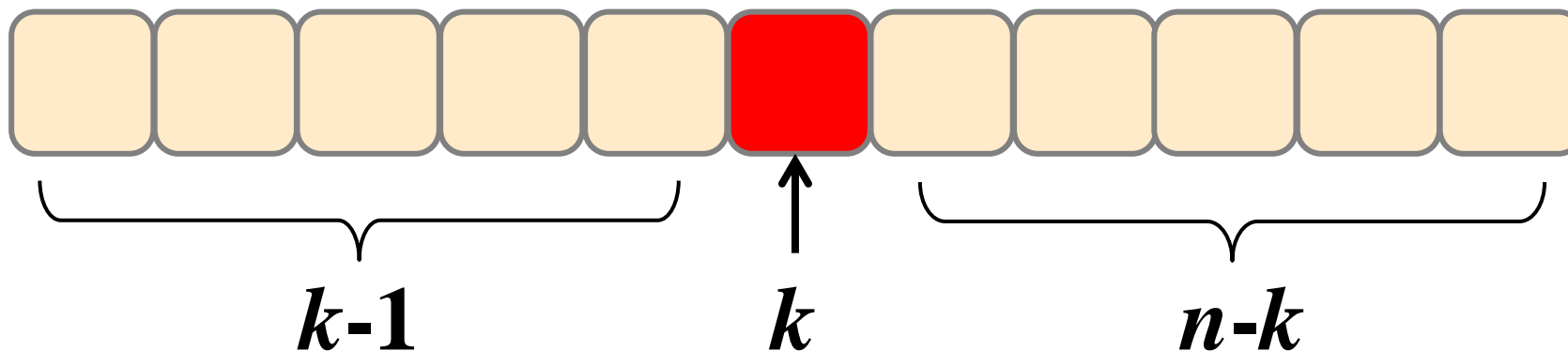
- k 可能有 n 种取值 ($k = 1 \dots n$)，对应于 n 种分划结果，亦对应算法的 n 种可能输入；
- 假定元素随机分布，故每种输入出现的概率相等，为 $1/n$ 。

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

对所有输入求和

第 k 种输入出现的概率

第 k 种输入下元素比较次数



时间复杂度—平均情况

$$\begin{aligned} T(n) &= \sum_{k=1}^n \frac{1}{n} [(n+1) + T(k-1) + T(n-k)] \\ &= \sum_{k=1}^n \frac{1}{n} (n+1) + \sum_{k=1}^n \frac{1}{n} [T(k-1) + T(n-k)] \\ &= (n+1) + \frac{1}{n} \sum_{k=1}^n [T(k-1) + T(n-k)] \end{aligned}$$

时间复杂度—平均情况

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

$T(0)+T(1)+\dots T(n-1)$

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

等式两边乘以 n :

$$nT(n) = (n+1)n + 2 \sum_{k=0}^{n-1} T(k) \quad \text{①}$$

对上式，用 $n-1$ 替换 n :

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

用①式减②式:

$$nT(n) - (n-1)T(n-1) = 2n + 2T(n-1)$$

快速排序平均情况时间复杂度分析

$$nT(n) - (n-1)T(n-1) = 2n + 2T(n-1)$$

$$nT(n) - (n+1)T(n-1) = 2n$$

等式两边除以 $n(n+1)$ ：

$$S_n \left(\frac{T(n)}{n+1} - \frac{T(n-1)}{n} \right) = \frac{2}{n+1}$$

$$S_n - S_1 = 2\left(\frac{1}{3} + \dots + \frac{1}{n+1}\right)$$

$$\frac{T(n)}{n+1} = 2\left(\frac{1}{3} + \dots + \frac{1}{n+1}\right) \approx 2 \int_3^{n+1} \frac{1}{x} \approx 2 \ln n$$

$$T(n) = O(2n \ln n) = O(2n \ln 2 \log_2 n) = O(1.386 n \log_2 n) = O(n \log_2 n)$$

$$S_n - S_{n-1} = \frac{2}{n+1}$$

$$S_{n-1} - S_{n-2} = \frac{2}{n}$$

$$S_{n-2} - S_{n-3} = \frac{2}{n-1}$$

⋮

$$S_2 - S_1 = \frac{2}{3}$$

快速排序的时空复杂度和稳定性

排序算法	时间复杂度			空间复杂度	稳定性
	最好	平均	最坏		
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n) \sim O(n)$	不稳定

课下验证

$3, 3^*, 2 \rightarrow 2, 3^*, 3$

课下思考

给定一个含有正数和负数的数组，编写程序对数组进行重新排列，使得所有正整数在数组左侧、负数在数组右侧，要求时间复杂度为 $O(n)$ 、空间复杂度 $O(1)$ 。

```
void Rearrange(int R[], int n){    //类似于分划操作
    int i=1, j=n;
    while(i<j){
        while(i<=n && R[i]>0) i++;    //从左往右找负数
        while(j>=1 && R[j]<0) j--;    //从右往左找正数
        if(i<j) swap(R[i], R[j]);
    }
}
```


课下思考

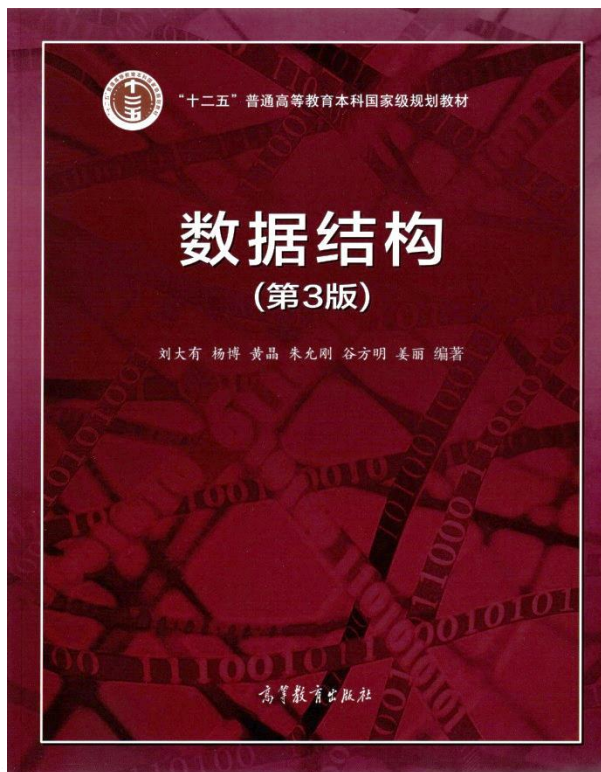
如果一个整数序列中一半为奇数，一半为偶数，编写一个时间复杂度 $O(n)$ 、空间复杂度 $O(1)$ 的算法，重新排列这些整数，使得奇数在前，偶数在后。【字节跳动、华为、腾讯、快手、微软、美团面试题】

课下思考

给定一个非负整数数组 R ，其中一半整数是奇数，一半整数是偶数。编写时间复杂度 $O(n)$ 、空间复杂度 $O(1)$ 的算法，对数组重新排列，使得奇数放在数组的奇数位，偶数放在数组的偶数位。【微软、字节跳动、谷歌面试题，哈尔滨工业大学期末考试题】

- 指针 i 从左向右扫描数组奇数位，找第一个偶数；
- 指针 j 从左向右扫描数组偶数位，找第一个奇数；
- 交换 $R[i]$ 和 $R[j]$ ，从而把偶数换到偶数位，奇数换到奇数位。

```
void ReArrange(int R[], int n){
    int i=1,j=0;
    while(i<n && j<n){
        while(i<n && R[i]%2==1) i+=2;
        while(j<n && R[j]%2==0) j+=2;
        if (i<n && j<n){
            swap(R[i],R[j]);
            i+=2; j+=2;
        }
    }
}
```



快速排序

- 快速排序算法
- 时空复杂度分析
- 优化策略

数据之法
结构之美
算法之道



(1) 处理小数据时结合插入排序

- 当待排序元素很少时，为极小的子数组产生许多的递归调用，得不偿失，此时快速排序反而没有插入排序快。
- 策略1：递归过程中，当前处理的子数组长度 \leq 某个阈值（一般取16左右）时，直接对当前子数组插入排序。

```
const int threshold=16;
void QuickSort(int R[], int m, int n){
    if(n-m+1 > threshold){
        int j=Partition(R, m, n);
        QuickSort(R, m, j-1);
        QuickSort(R, j+1, n);
    }
    else InsertionSort(R, m, n);
}
```

(1) 处理小数据时结合插入排序

- 策略2：递归过程中，当前处理的子数组长度 \leq 某个阈值（一般取16左右）时，对当前子数组什么也不做，直接退出本层递归。待最后所有递归都退出后，再对整个大数组做一次插入排序（插入排序在处理“接近有序”的序列时效率高）。

```
void QuickSort(int R[], int m, int n){  
    if(n-m+1 > threshold){  
        int j=Partition(R, m, n);  
        QuickSort(R, m, j-1);  
        QuickSort(R, j+1, n);  
    }  
}
```

主函数：QuickSort(R, 1, n);
InsertionSort(R, 1, n); //最后统一做1次插入排序

(2) 随机选取基准元素

- 快速排序达最坏情况的主要原因：数组有序，每次选的基准元素（第1个元素）恰好是当前子数组的最小元素。
- 在当前子数组中**随机选择**一个元素作为基准元素：
`int k=rand()%(n-m+1)+m; //返回m到n间的随机数.`
`swap(R[m],R[k]);`
再把R[m]作为基准元素
- 显著降低最坏情况发生的概率，但无法杜绝。
- 实际应用中，采用该策略遇到最坏情况的概率极低，可获得很好的性能。

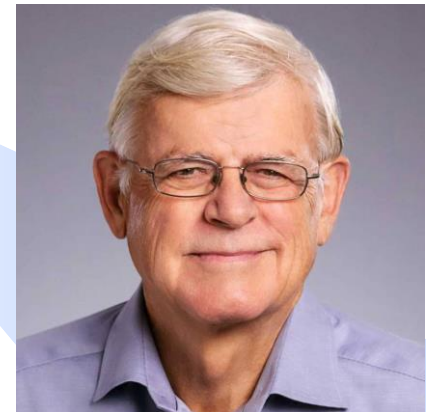
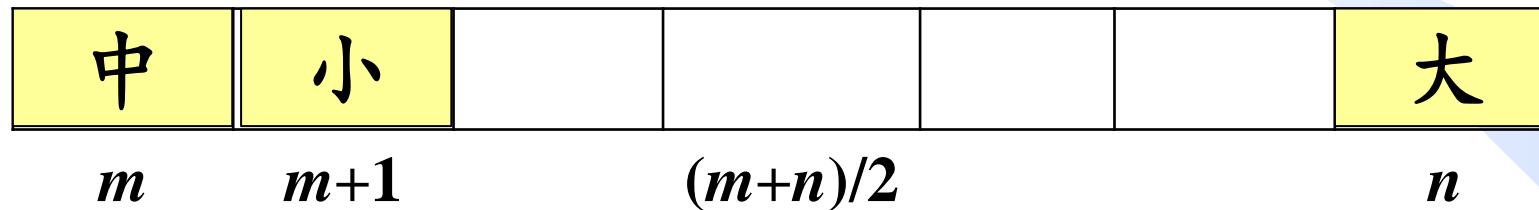
随机算法简介

- 如果一个算法的行为不仅由输入决定，而且也由随机数决定，则可称此类算法为随机算法。与之对应，未使用随机数做决策的算法可称为确定型算法。
- 对于确定型算法，可能存在特定的输入数据，其总能导致最坏情况的运行时间。而对于随机算法，以相同的输入数据运行两次，往往会得到不同的运行时间。
- 对于一些问题，当算法在执行过程中面临某种选择时，随机性选择往往比最优选择省时，这种情况下将使得随机算法的效率优于确定型算法。
- 而对于另一些问题，某些算法往往在平均情况下时间复杂度较低，仅在最坏情况下时间复杂度高，此时可以结合随机策略，降低最坏情况发生的概率。

(3) 三数取中(Median of Three)选基准元素

B

- 选取 $R[m]$ 、 $R[(m+n)/2]$ 和 $R[n]$ 的中位数作为基准元素。
- 保证选出的基准元素不是子数组的最小元素，也不是最大元素，分划肯定不会分到最边上。



Sedgwick、Singleton建议如下实现：

```
swap(R[(m+n)/2], R[m+1]);
```

```
if(R[m+1] > R[n]) swap(R[m+1], R[n]); //  $R_{m+1}$  和  $R_n$  的较大者移到  $R_n$  处
```

```
if(R[m] > R[n]) swap(R[m], R[n]); //  $R_m$  和  $R_n$  的较大者移到  $R_n$  处
```

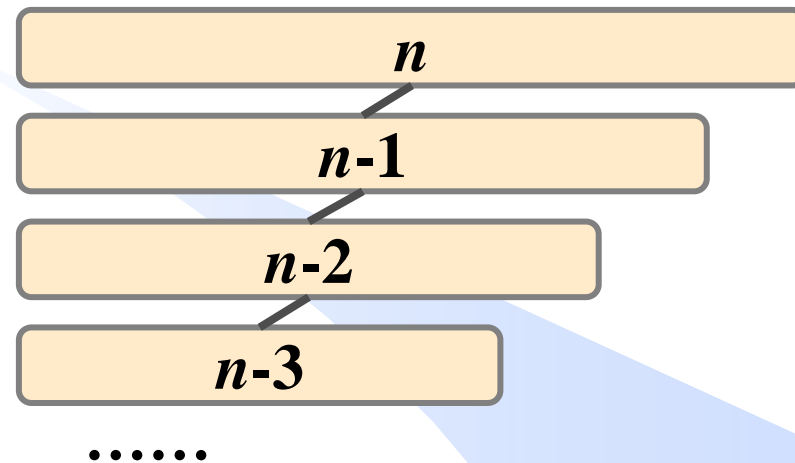
```
if(R[m+1] > R[m]) swap(R[m+1], R[m]); //  $R_{m+1}$  和  $R_m$  的较大者移到  $R_m$  处
```

- 降低最坏情况发生的概率，但无法杜绝。

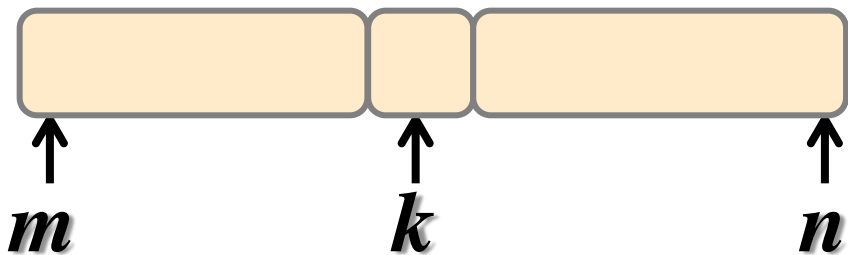
(4) 尾递归转化为循环

将尾递归转为循环，减少递归调用所需的系统栈空间。

```
void QuickSort(int R[],int m,int n){  
    if(m < n){  
        int k=Partition(R, m, n);  
        QuickSort(R, m, k-1);  
        QuickSort(R, k+1, n);  
    }  
}
```



最坏情况空间仍为 $O(n)$
每次分划都分到最右边
使左区间长度 $n-1, n-2, \dots, 1$



```
void QuickSort(int R[],int m,int n){  
    while(m < n){  
        int k=Partition(R, m, n);  
        QuickSort(R, m, k-1);  
        m=k+1;  
    }  
}
```

先递归处理左区间
后循环处理右区间

(5) 尾递归转循环 + 先处理短区间

先处理短区间，最坏空间复杂度（递归深度）可降为 $O(\log n)$ 。

```
void QuickSort(int R[], int m, int n){
```

```
    while(m < n){
```

```
        int k=Partition(R,m,n);
```

```
        if(k-m < n-k){ //左区间短
```

```
            QuickSort(R,m,k-1);
```

```
            m=k+1;
```

```
        }else{ //右区间短
```

```
            QuickSort(R,k+1,n);
```

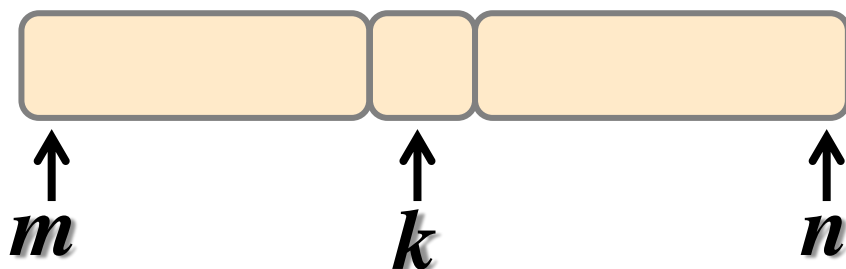
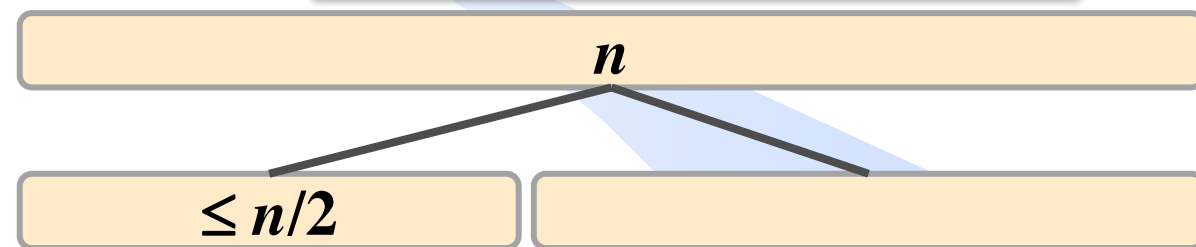
```
            n=k-1;
```

```
        }
```

```
    }
```

```
}
```

先递归处理短区间
后循环处理长区间



(5) 尾递归转循环 + 先处理短区间

先处理短区间，最坏空间复杂度（递归深度）可降为 $O(\log n)$ 。

```
void QuickSort(int R[], int m, int n){
```

先递归处理短区间
后循环处理长区间

```
    while(m < n){
```

```
        int k=Partition(R,m,n);
```

```
        if(k-m < n-k){ //左区间短
```

```
            QuickSort(R,m,k-1);
```

```
            m=k+1;
```

```
        }else{ //右区间短
```

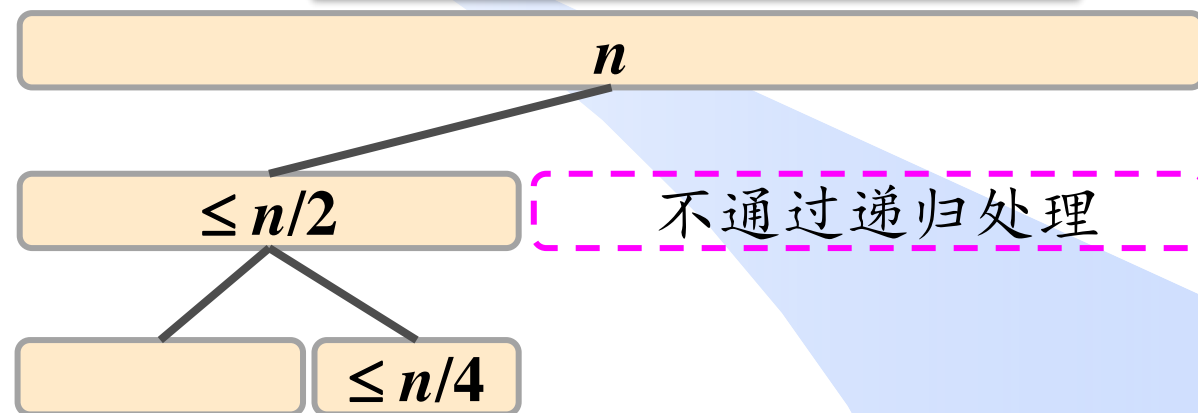
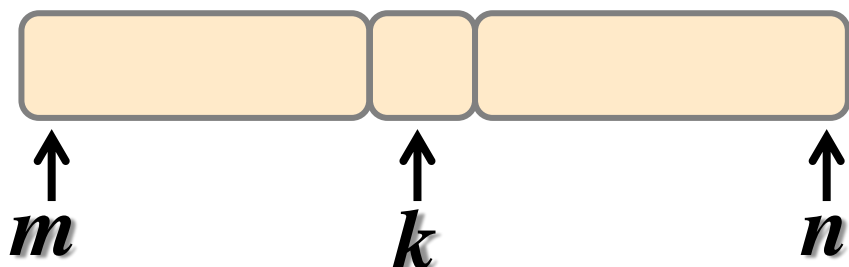
```
            QuickSort(R,k+1,n);
```

```
            n=k-1;
```

```
        }
```

```
    }
```

```
}
```



(5) 尾递归转循环 + 先处理短区间

先处理短区间，最坏空间复杂度（递归深度）可降为 $O(\log n)$ 。

```
void QuickSort(int R[], int m, int n){
```

先递归处理短区间
后循环处理长区间

```
    while(m < n){
```

```
        int k=Partition(R,m,n);
```

```
        if(k-m < n-k){ //左区间短
```

```
            QuickSort(R,m,k-1);
```

```
            m=k+1;
```

```
        }else{ //右区间短
```

```
            QuickSort(R,k+1,n);
```

```
            n=k-1;
```

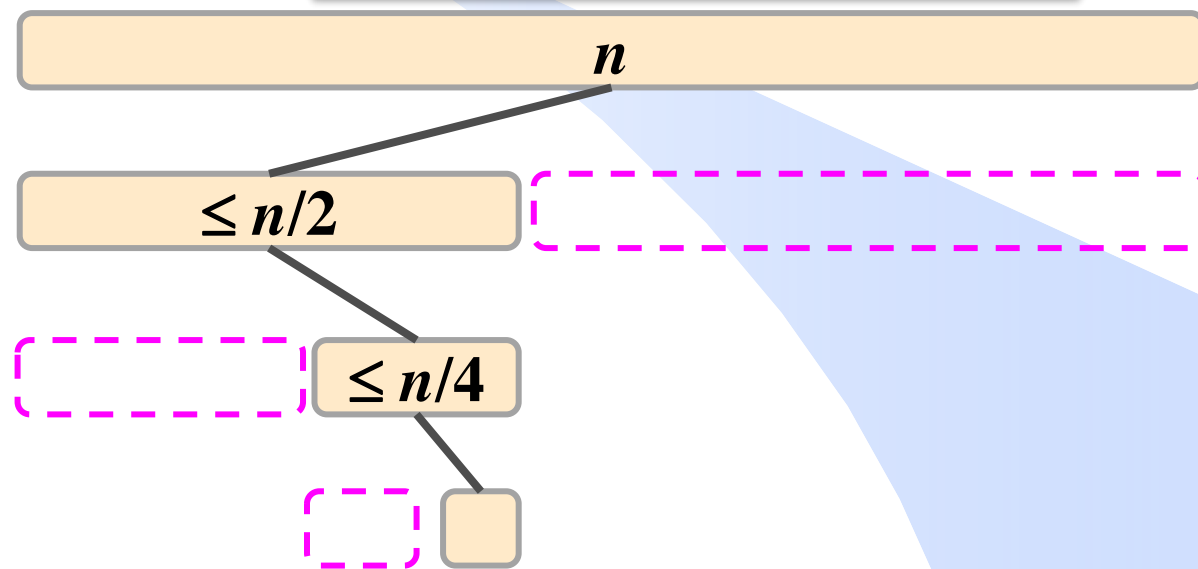
```
    }
```



↑
 m

↑
 k

↑
 n



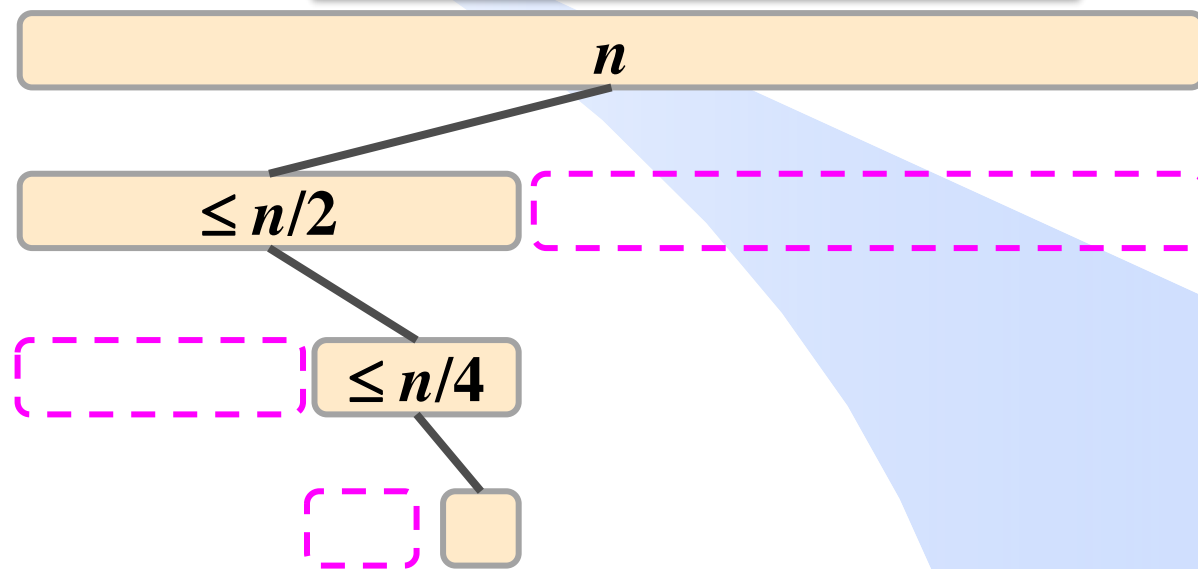
每次递归处理的子数组长度至少
缩减一半，故递归深度为 $O(\log n)$

(5) 尾递归转循环 + 先处理短区间

先处理短区间，最坏空间复杂度（递归深度）可降为 $O(\log n)$ 。

```
void QuickSort(int R[], int m, int n){
    while(m < n){
        int k=Partition(R,m,n);
        if(k-m < n-k){ //左区间短
            QuickSort(R,m,k-1);
            m=k+1;
        }else{ //右区间短
            QuickSort(R,k+1,n);
            n=k-1;
        }
    }
}
```

先递归处理短区间
后循环处理长区间



如果每次分划都分到最边上
空间复杂度甚至可降为 $O(1)$

课下思考：使用该策略能使最坏时间复杂度降为 $O(n \log n)$ 么？

(6) 利用栈消除所有递归

```
void QuickSort(int R[], int m, int n){
```

```
    Stack s; //栈需预先实现
```

```
    s.Push(m, n);
```

```
    while(!s.Empty()){
```

```
        s.Pop(m, n); //出栈的二元组分别存入变量m和n
```

```
        int k=Partition(R, m, n);
```

```
        s.Push(k+1, n);
```

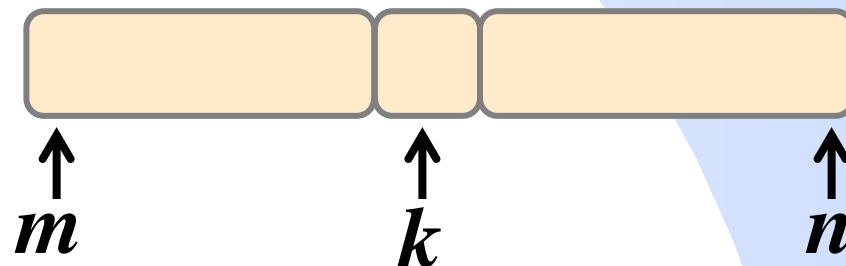
```
        s.Push(m, k-1);
```

```
    }
```

```
}
```

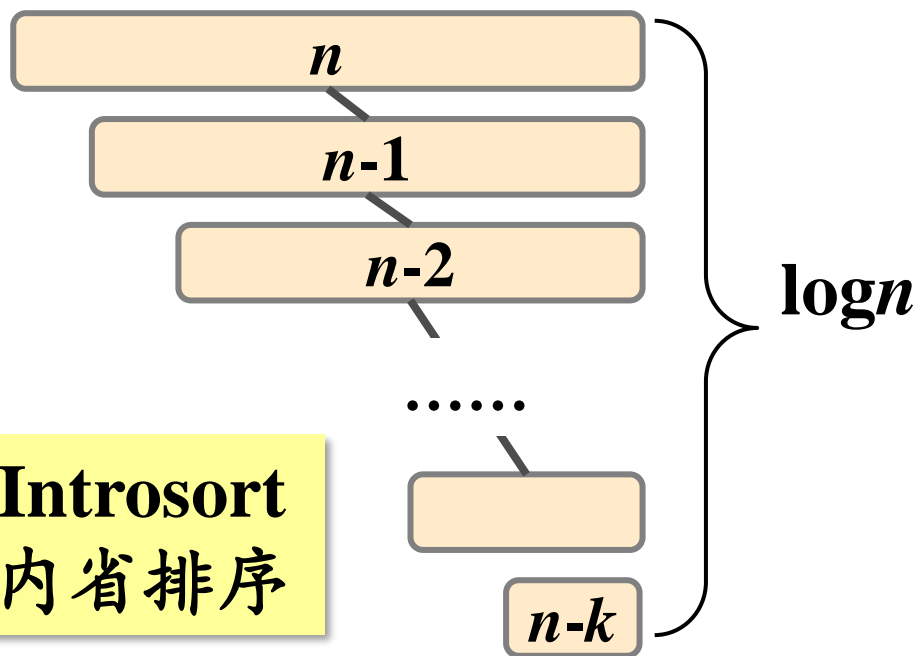
初始调用: QuickSort(R, 1, n)

栈内元素：二元组，标识当前处理的区间的起止下标

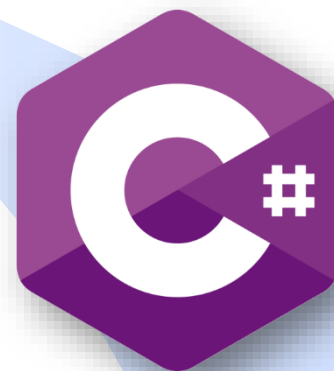


(7) 侦测递归深度，适时转为堆排序

- 侦测快速排序的递归深度，当递归深度达到 $O(\log n)$ 层时，强行停止递归，转而对当前处理的子数组进行**堆排序**。
- **真正使最坏情况时间复杂度降为 $O(n \log n)$!**



David R. Musser



(8) 三路分划 (3-Way-Partition)

- 当重复元素很多时，传统快速排序效率较低。

5 1 5 2 5 3 5 9 5 8 5 5 7 5 5

5 1 5 2 5 3 5 5 5 5 5 5 7 8 9

- 修改Partition操作，把当前数组划分为三部分：小于基准元素 K 的元素放在左边，等于 K 的元素放在中间，大于 K 的元素在右边。

1 2 3 5 5 5 5 5 5 5 5 5 7 8 9

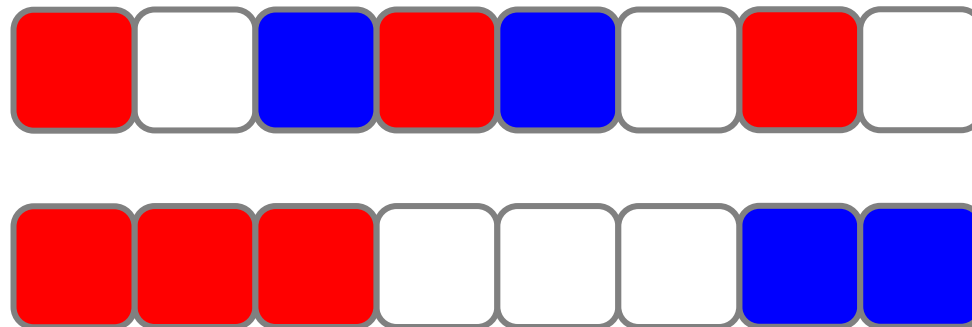
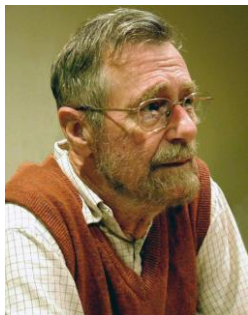
- 进一步递归时，仅对小于 K 的左半部分子数组和大于 K 的右半部分子数组进行递归排序。

荷兰国旗问题(Dijkstra提出)



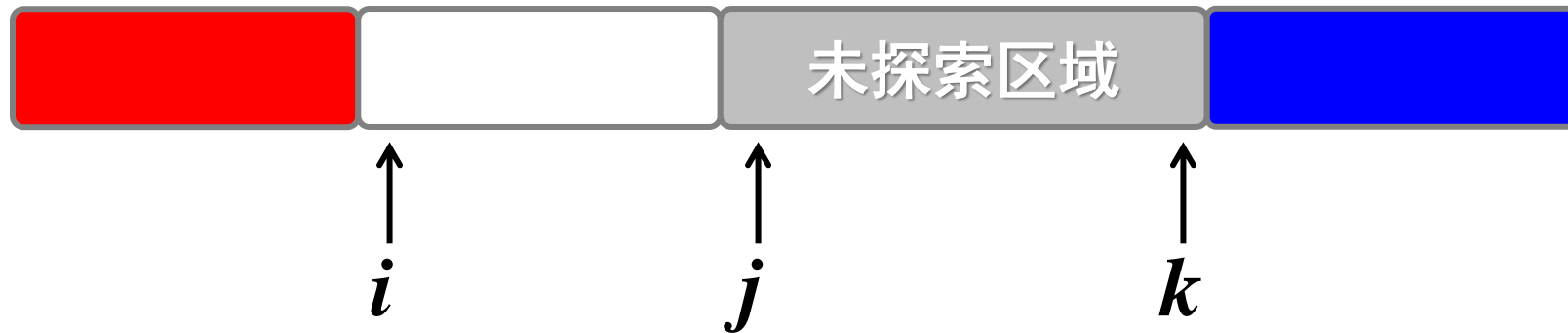
B

给定 n 粒石头，每粒石头的颜色是红，白，蓝之一。要求重新排列这些石头，使得所有**红色石头在前**，所有白色石头居中，所有**蓝色石头居后**。【腾讯、华为、小米、字节跳动、苹果、微软、谷歌、滴滴、快手面试题，浙大考研题】



三路分划

- 设置3个指针，前指针 i ，中指针 j ，后指针 k ；
- 初始时 $i=1, j=1, k=n$ ；指针 j 从左往右扫描数组：
- 若 $R[j]$ 为白，什么也不做继续扫描， $j++$ ；
- 若 $R[j]$ 为红，交换 $R[j]$ 和 $R[i]$ ， $i++$ ， $j++$ ；
- 若 $R[j]$ 为蓝，交换 $R[j]$ 和 $R[k]$ ， $k--$ ；
- 通过 j 的遍历，使红石头换到数组左边，蓝石头换到数组右边。

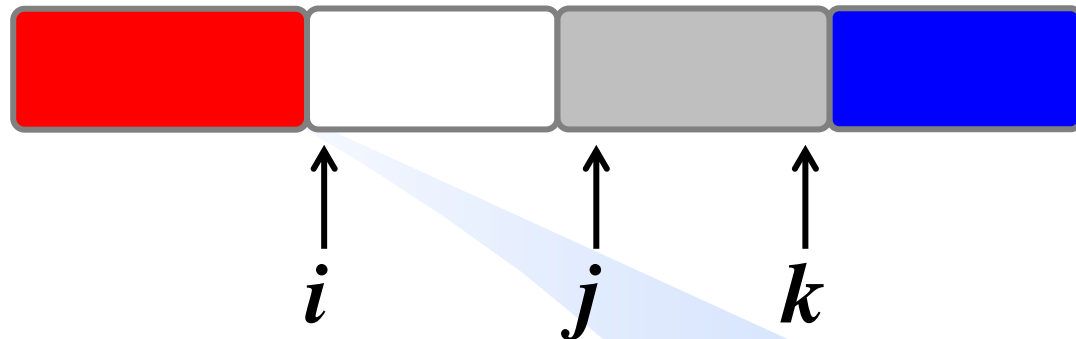


三路分划

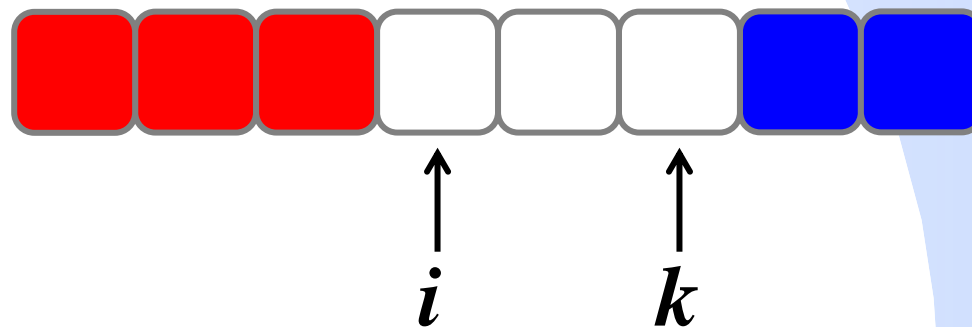
```

while (j<=k){
    if (R[j]=='红'){
        swap(R[j], R[i]);
        j++; i++;
    }
    else if (R[j]=='蓝'){
        swap(R[j], R[k]);
        k--;
    }
    else // R[j]=='白'
        j++;
}

```



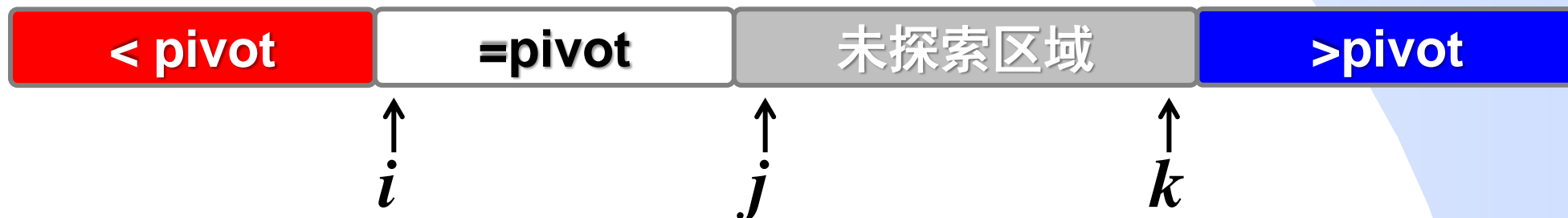
课下思考：
此处为什么不 $j++$ ？



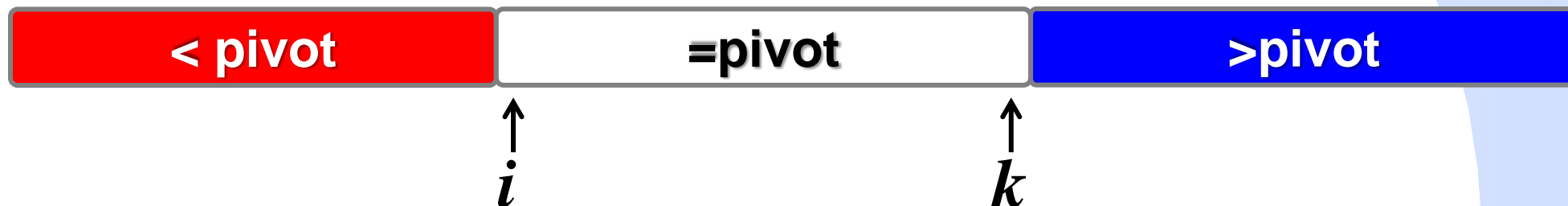
三路分划

```
void Partition3way(int R[], int m, int n, int &i, int &k){
    int j, pivot;
    i=m, j=m, k=n, pivot=R[m];
    while (j<=k){
        if(R[j] < pivot) {swap(R[j],R[i]); j++; i++;}
        else if(R[j] > pivot) {swap(R[j],R[k]); k--;}
        else j++;
    }
}
```

算法
执行中

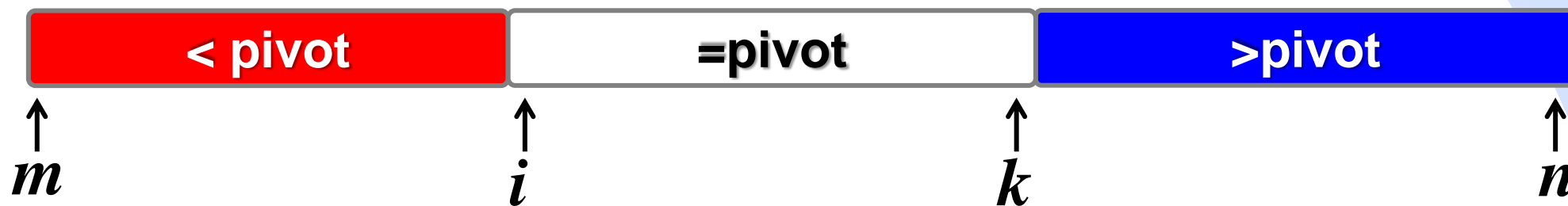


算法
结束后



基于三路分划的快速排序

```
void QuickSort(int R[], int m, int n){  
    if(m>=n) return;  
    int i, k;  
    Partition3way(R, m, n, i, k);  
    QuickSort(R, m, i-1);  
    QuickSort(R, k+1, n);  
}
```



课下思考

- 仅使用交换操作，对只由 0, 1, 2 三种元素构成的整数数组进行排序，概述算法思想并给出算法的时间复杂度【2020年北京大学考研题】。
- 设有一个整数序列 a_1, a_2, \dots, a_n 存放于一个整型数组A中，给定两个整数 x 和 y ($y > x$)，请设计一个时间和空间上尽可能高效的算法，重排该序列，使序列中所有比 x 小的元素都集中到数组的左侧，所有比 y 大的元素都集中到数组的右侧，介于 x 和 y 之间（含等于 x 或 y ）的元素置于数组的中间。【2020级吉林大学期末考试题，15分】

C语言qsort()源码剖析



基于快速排序，并结合如下优化策略：

- 子数组长度小于8转插入排序；
- 三数取中选基准元素Median of Three；
- 三路分划3-Way-Partition；
- 使用栈消除所有递归；
- 先处理短区间。

C++ STL sort() 源码剖析

C

基于快速排序，并结合如下优化策略：

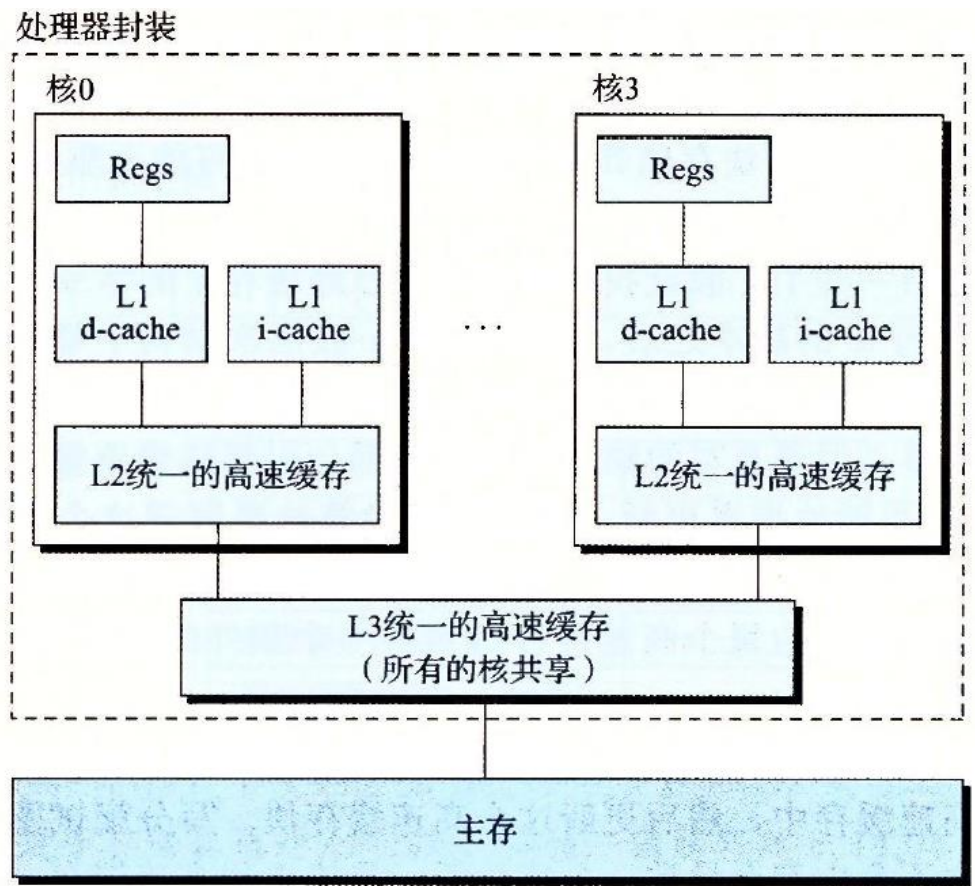
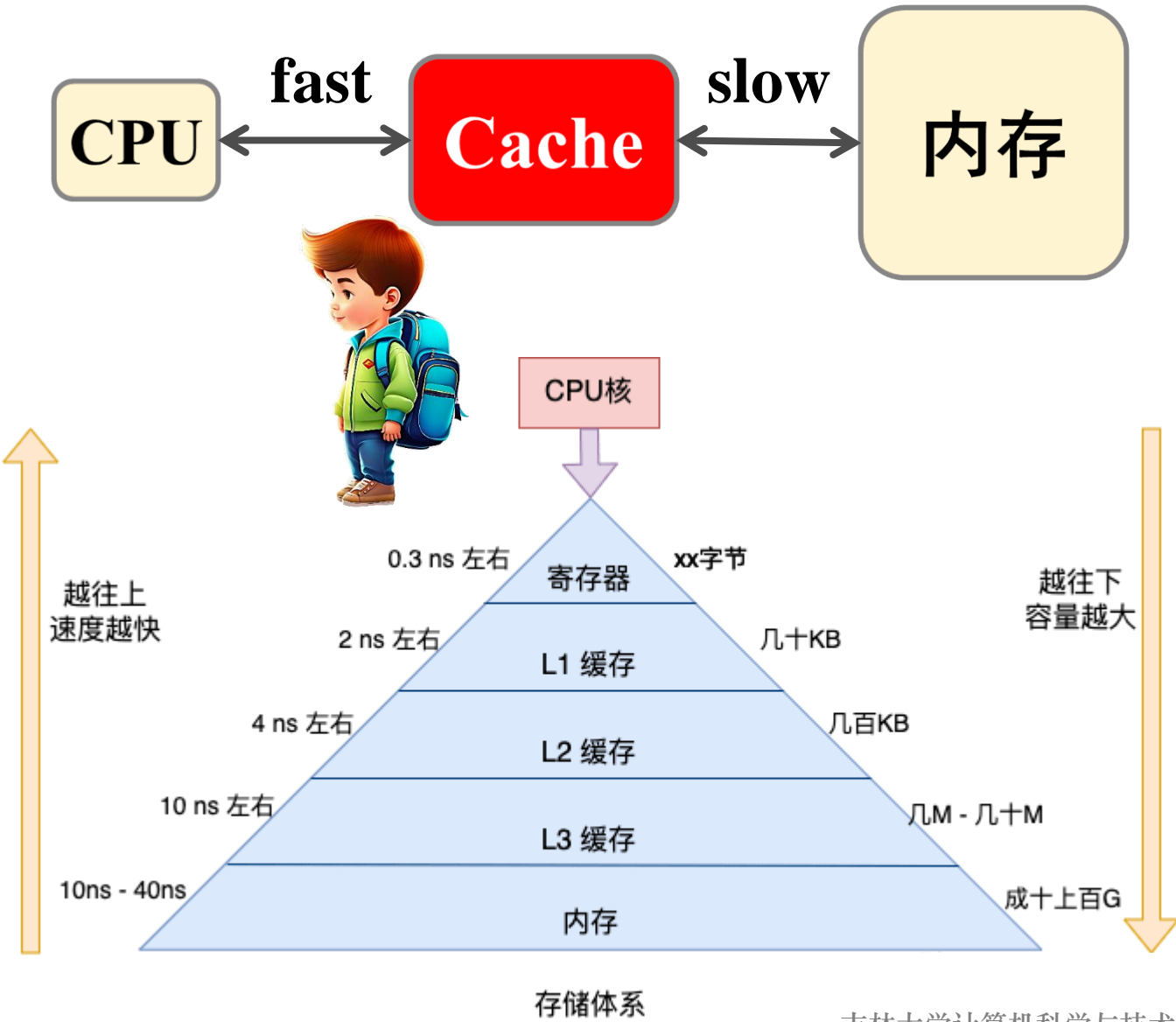
- 子数组长度小于等于16时，直接退出本层递归。待最后快速排序所有递归都退出后，再对整个大数组进行**一次**插入排序；
- 当递归深度达到 $\lfloor 2\log_2 n \rfloor = O(\log n)$ 时，强行停止递归，转而对当前子数组进行**堆排序**；
- 三数取中选基准元素；
- 尾递归转化为循环，先处理短区间；
- 最坏情况时间复杂度 $O(n\log n)$ 。



快速排序总结

- 快速排序方法是基于关键词比较的内排序算法中平均情况下时间最快的。
- 为什么平均情况下**快速排序**比**堆排序**快？
 - ✓ $n\log n$ 的常系数 (1.386 **vs** 2)
 - ✓ 倾向于访问物理上相邻的数据，缓存命中率高

高速缓冲存储器Cache

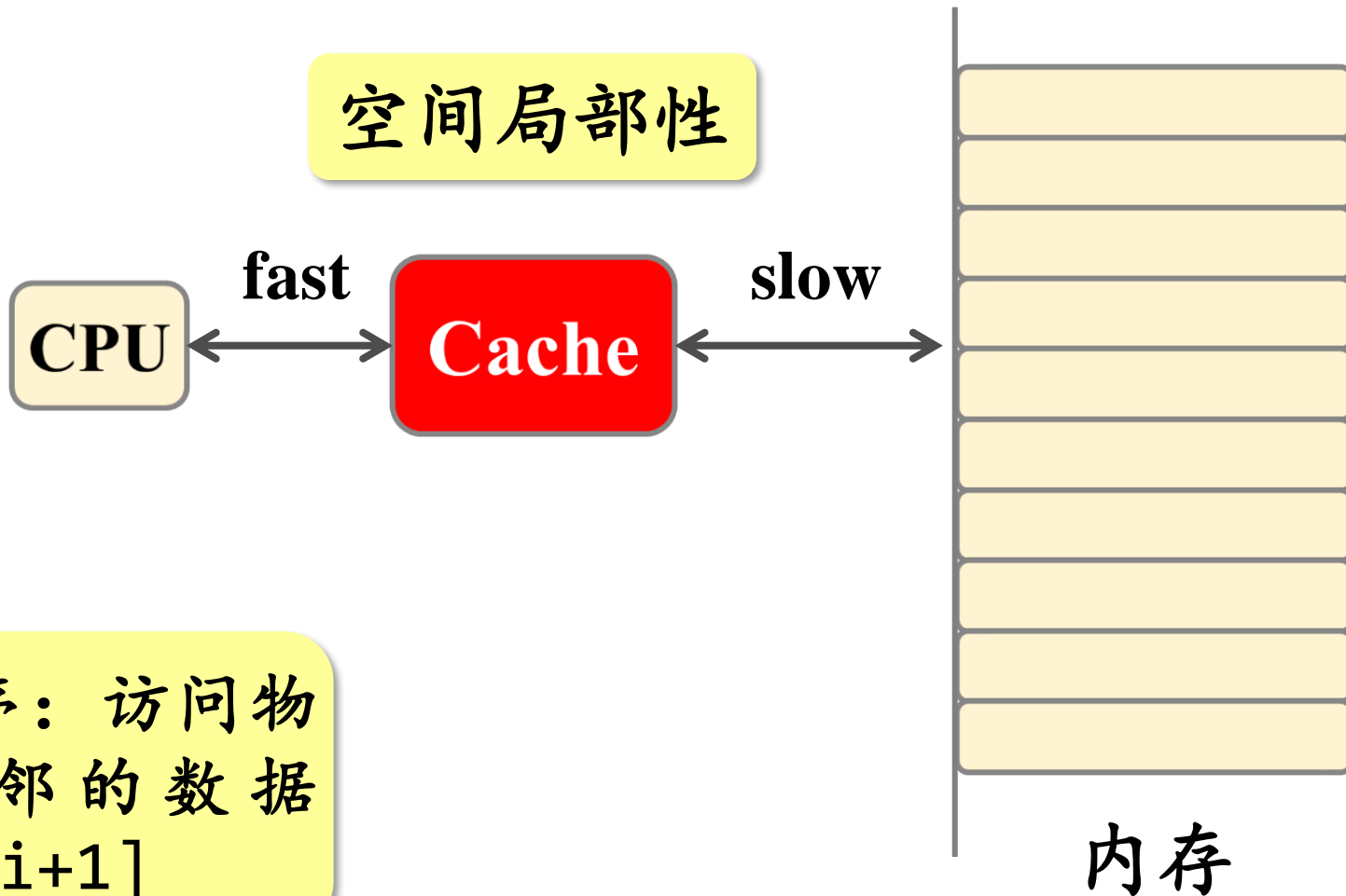


Intel® 酷睿™ i7 处理器的缓存结构



编写缓存友好的代码

空间局部性



缓存友好的程序：
尽量访问物理上
相邻的数据，而
不是跳着访问，
从而使程序具有
较好的空间局部
性和缓存命中率。

堆排序：跳着访问
 $R[i] \sim R[2*i]$