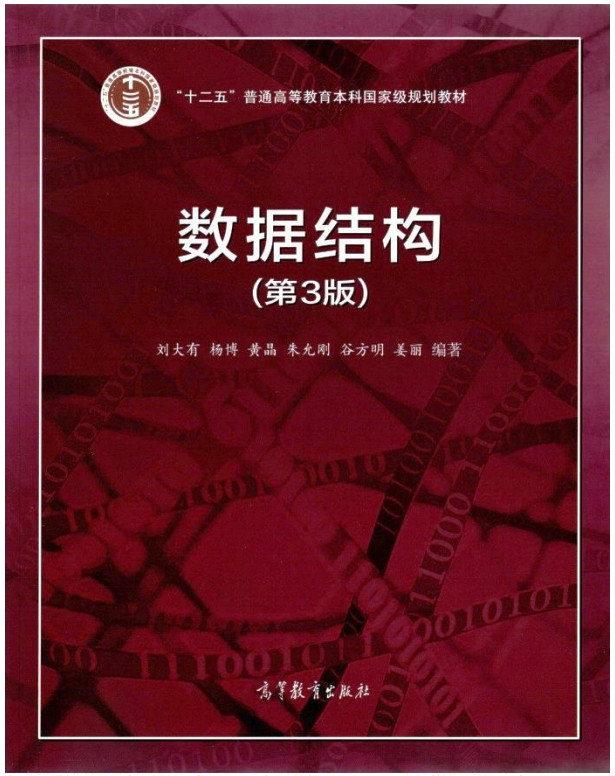


快速排序

- 快速排序算法
- 时空复杂度分析
- 优化策略
- 快速选择算法

数据之法
结构之美
算法之道

Last updated on 2023.11

zhuyungang@jlu.edu.cn

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

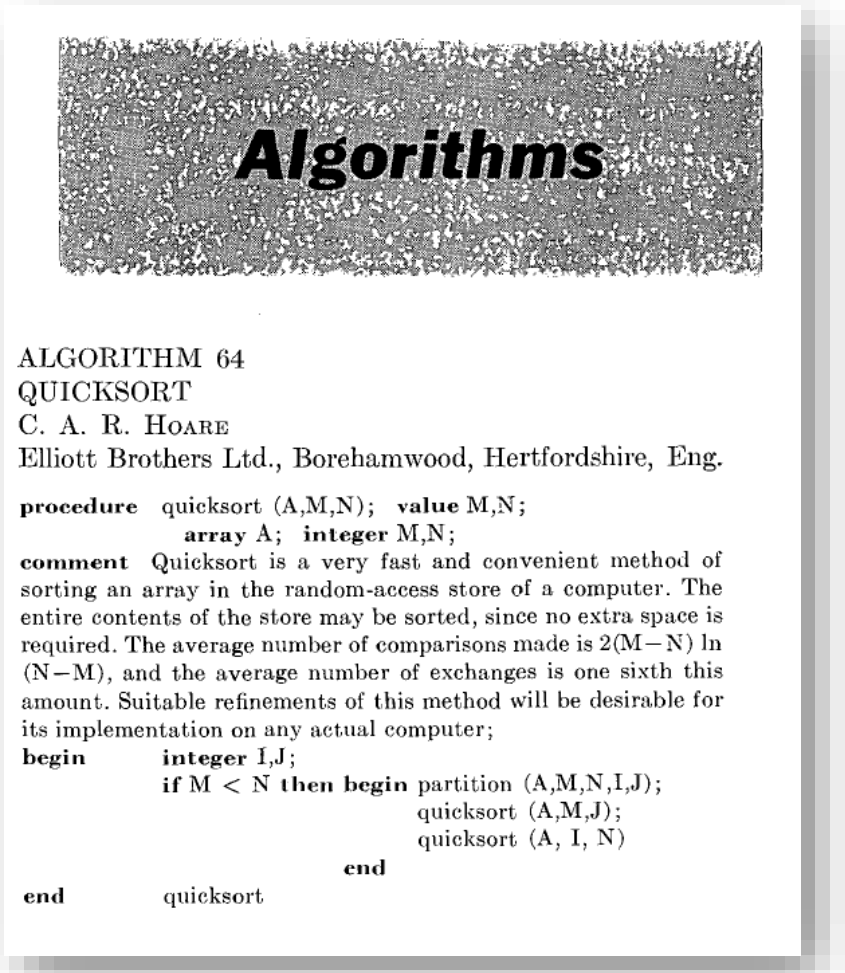
英国皇家工程院院士

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。

快速排序的基本思想

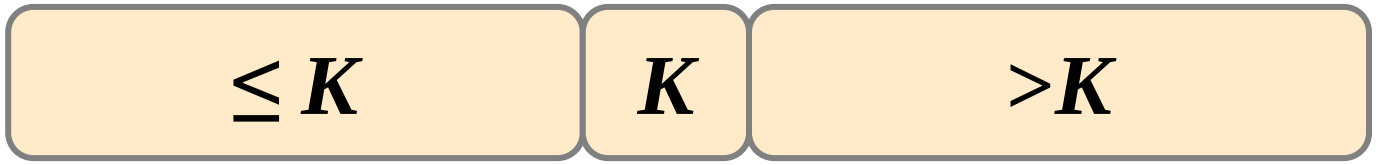
A

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所应该在的正确位置

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。



快速排序算法简要描述

A

```
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; // ①
        QuickSort(LeftR);                    // ②
        QuickSort(RightR);                   // ③
    }
}
```

操作①：Partition
分划、分割、划分

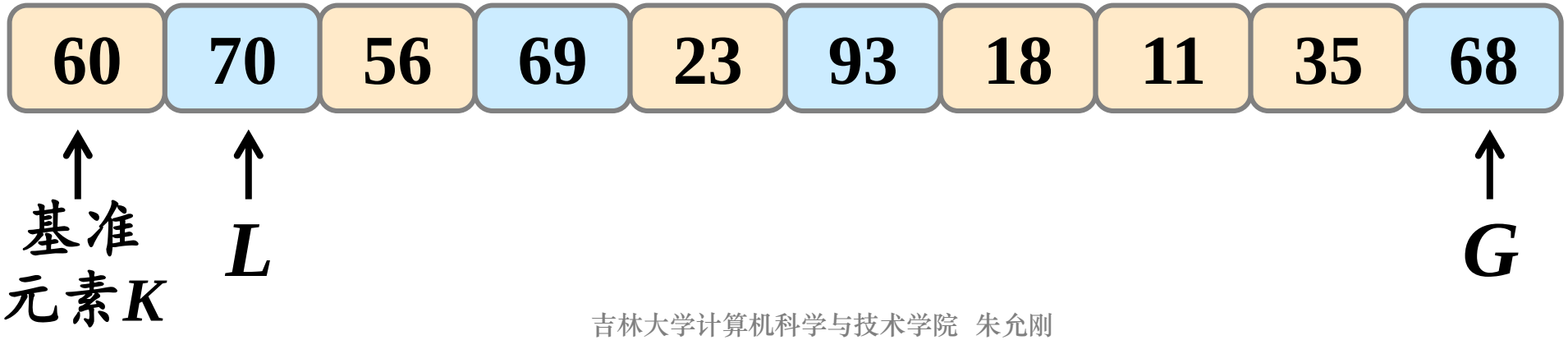
LeftR (≤ *K*)	*K*	RightR(> *K*)

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

A

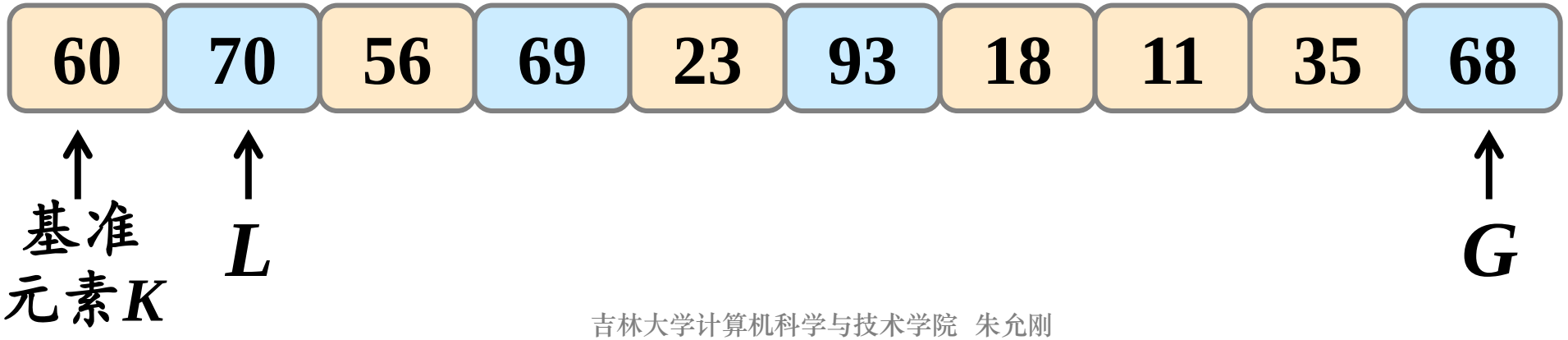
- 思想：把比*K*大的元素移到数组右边，把比*K*小的元素移到数组左边。
- 引入两个指针*L*和*G*相向而行：指针*L*从左向右扫描数组找第一个>*K*的元素，指针*G*从右向左找第一个 ≤ *K*的元素，交换R[*L*]和R[*G*]。重复如此，不断把≤*K*的元素换到数组左边，大于*K*的元素换到数组右边，直至指针*L*和*G*相遇。



Partition操作算法思想

A

- 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。



Partition操作算法

A

```
int Partition(int R[], int m, int n){ //对子数组Rm...Rn分划
    int K=R[m], L=m+1, G=n; // Rm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≤K的元素
        if(L<G) {swap(R[L],R[G]); L++; G--;}
    }
    swap(R[m],R[G]);
    return G;
}
```

时间复杂度
O(*n*)

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

Partition操作算法

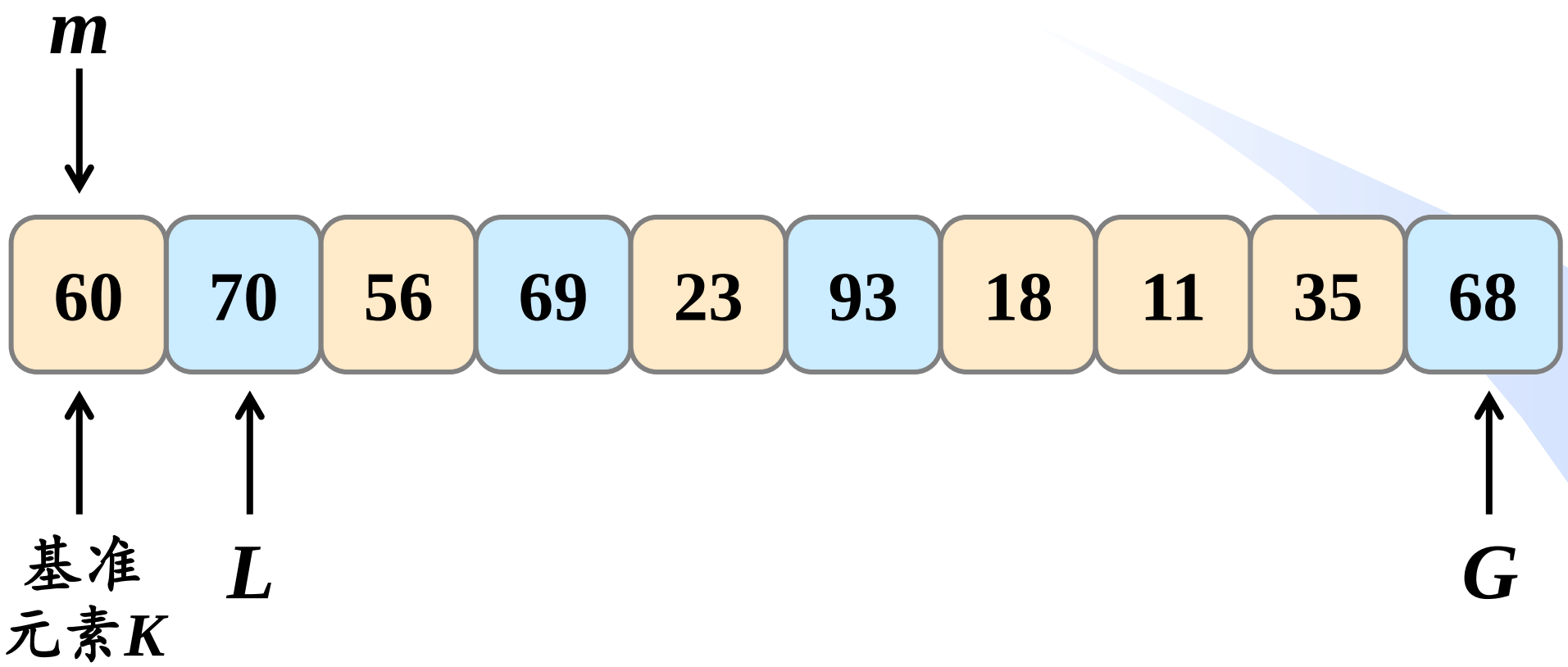
A

```
int Partition(int R[], int m, int n){ //对子数组Rm...Rn分划
    int K=R[m], L=m, G=n+1; // Rm为基准元素
    while(L<G) {
        L++;                          //从左向右找第一个 > K的元素
        while(L<=n && R[L]<=K) L++;
        G--;                          //从右向左找第一个 ≤ K的元素
        while(R[G]>K) G--;  //为什么j不用考虑左边界越界?
        if(L<G) swap(R[L],R[G]);
    }
    swap(R[m],R[G]);
    return G;
}
```

时间复杂度 O(*n*)

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

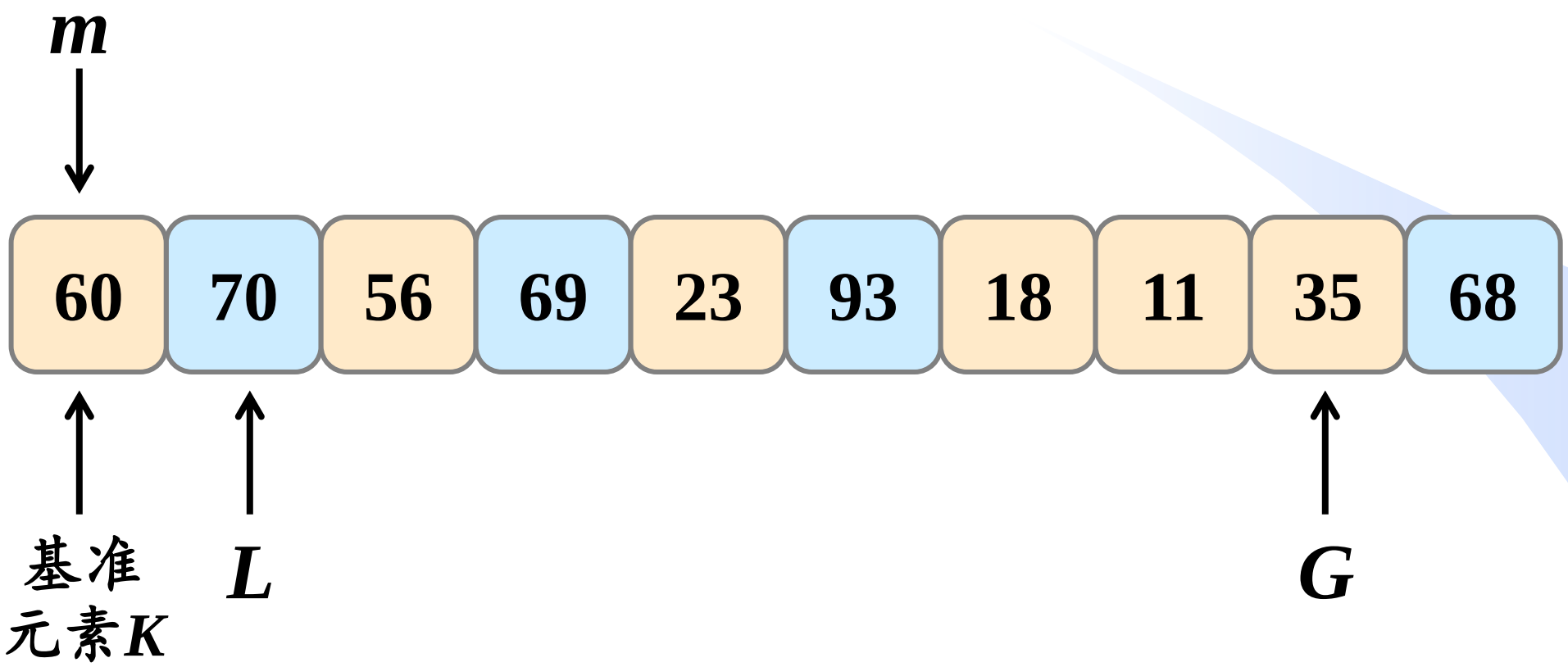
分划过程—例子



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

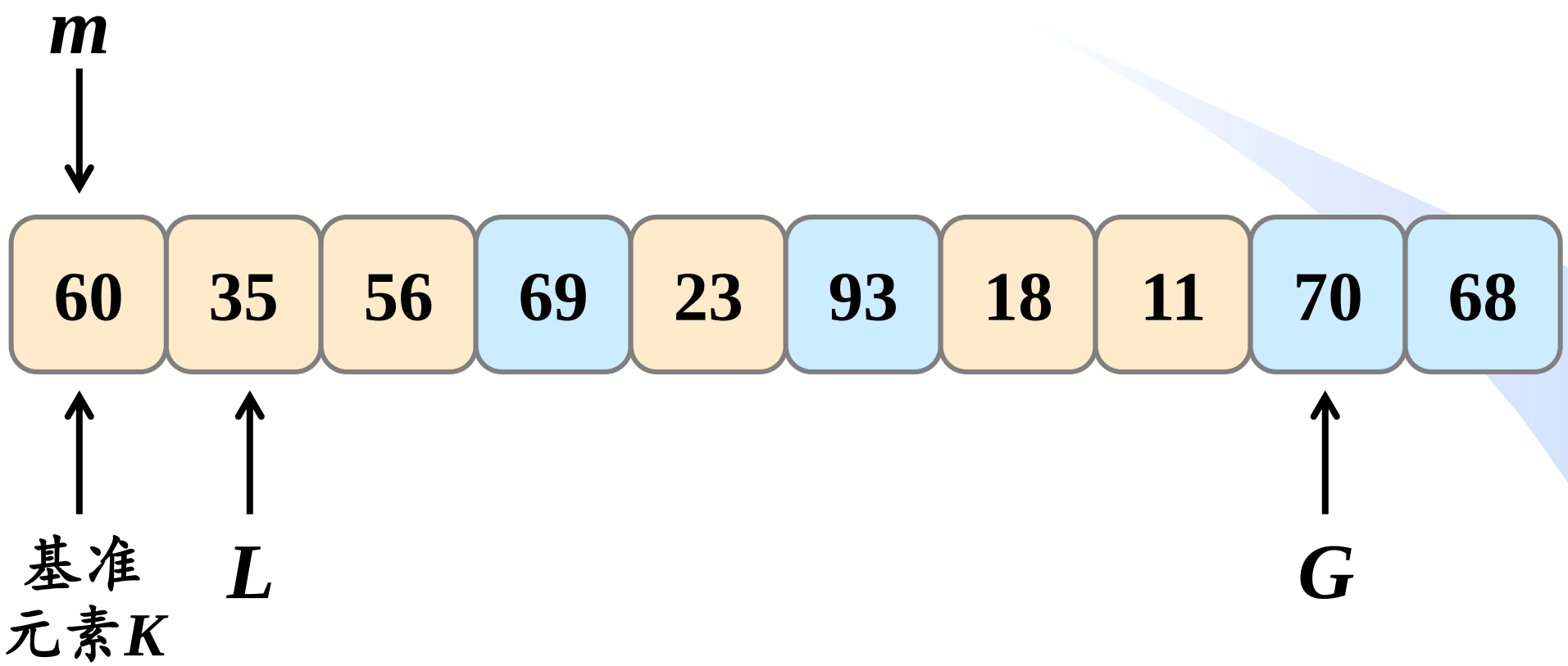
分划过程—例子



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

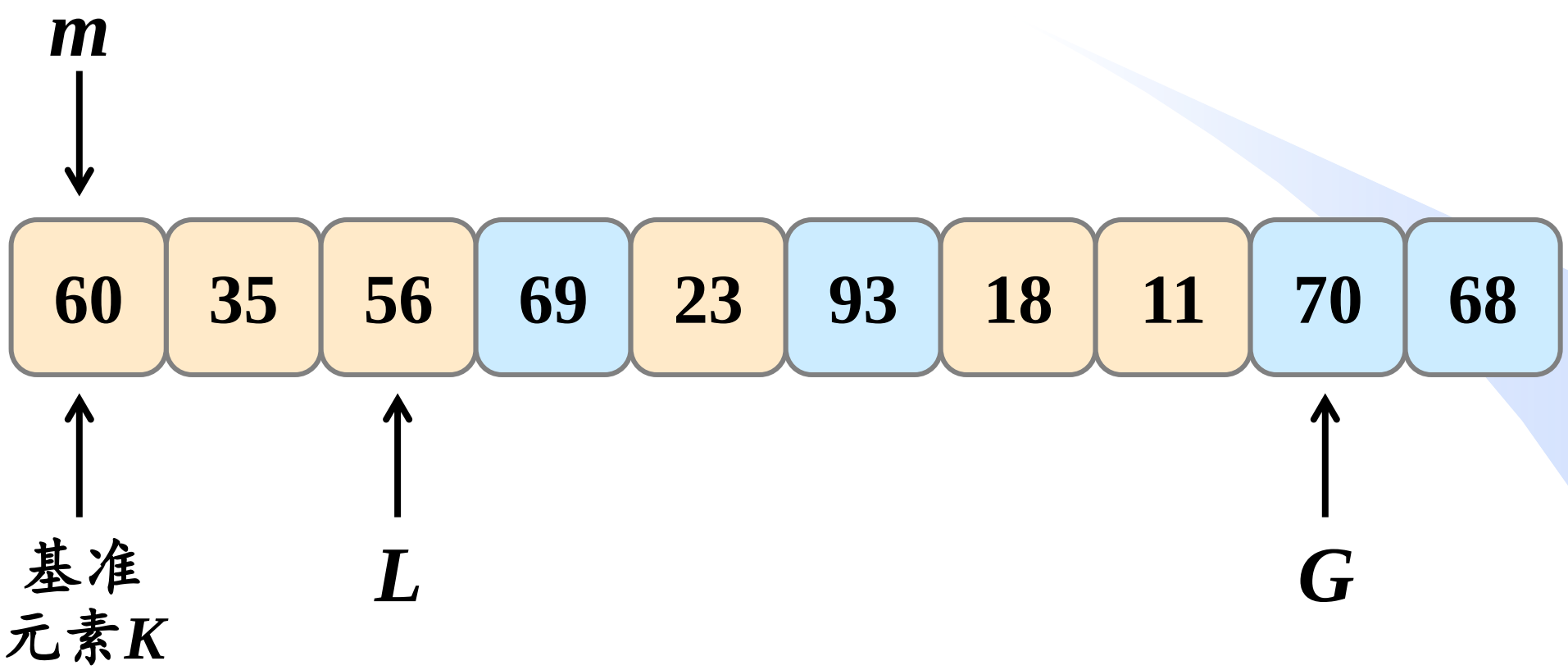
分划过程—例子



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

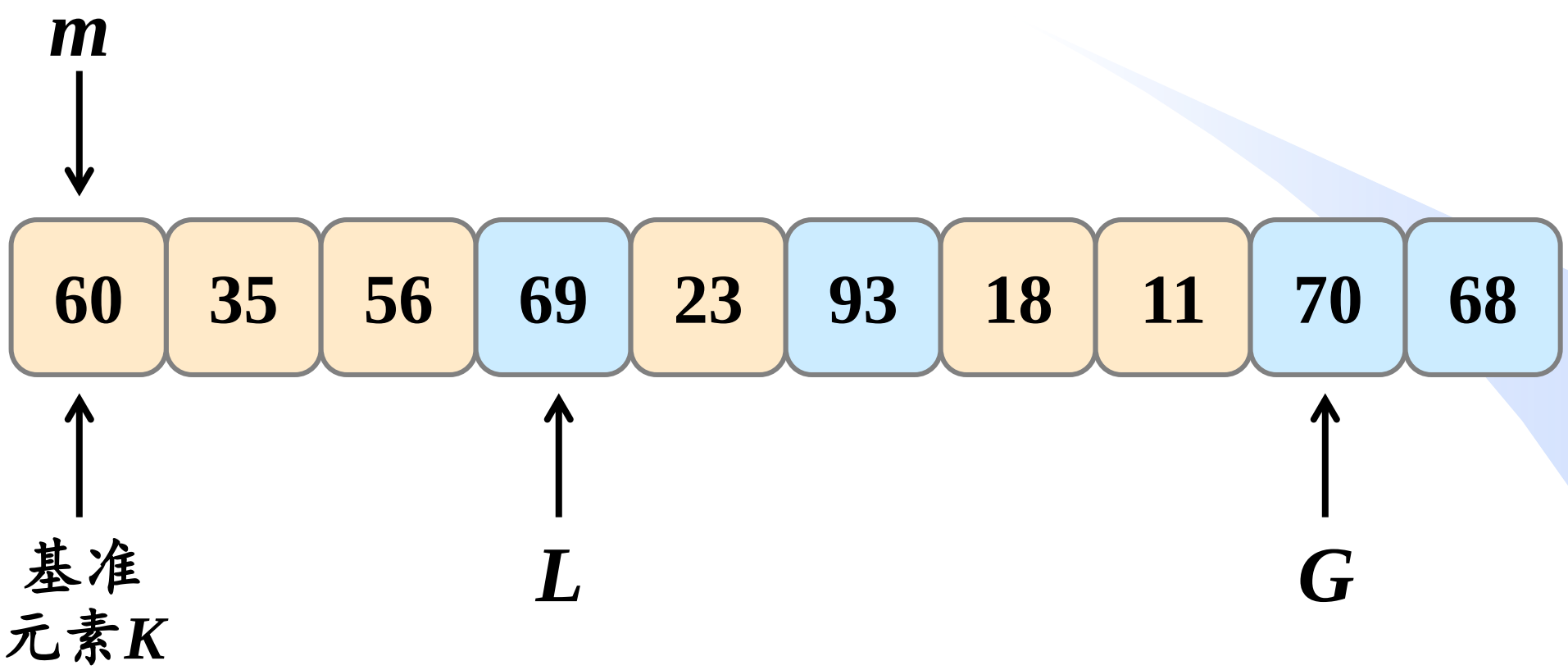
分划过程—例子



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

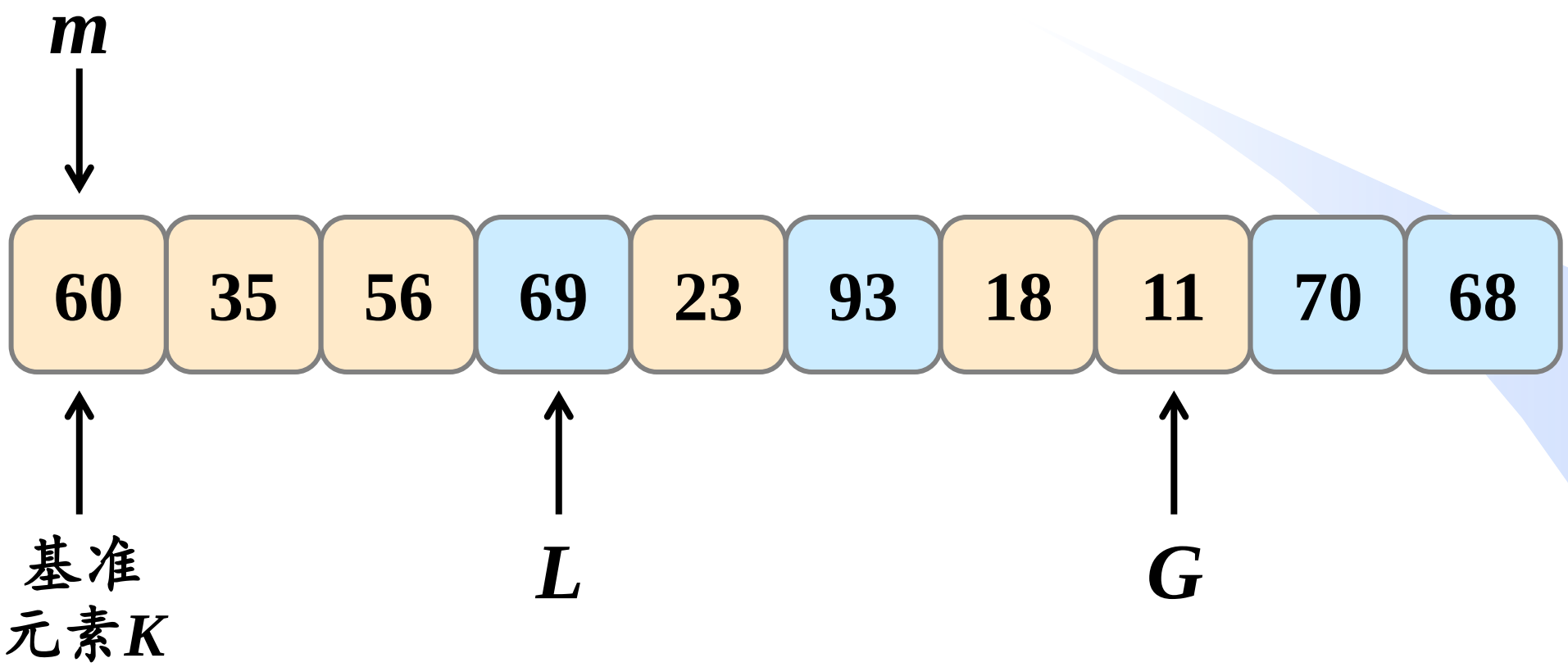
分划过程—例子



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

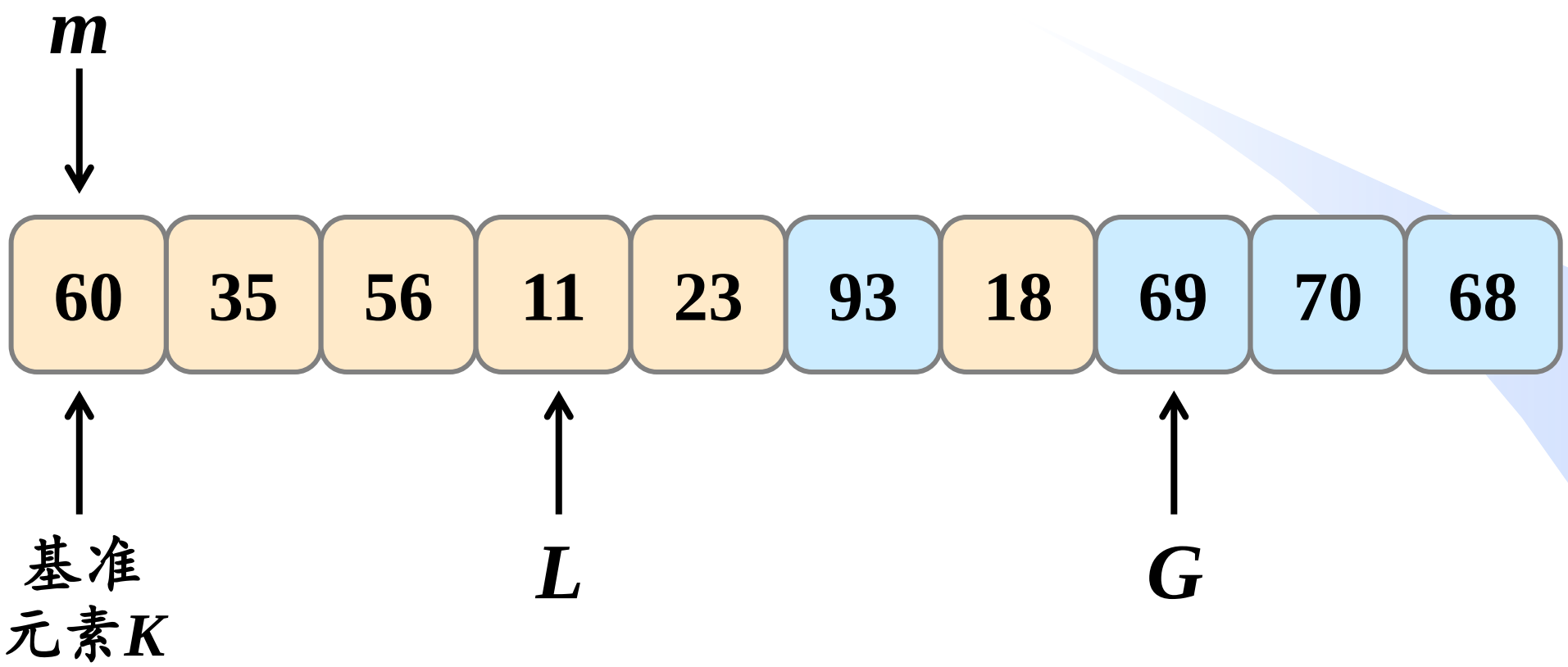
分划过程—例子



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

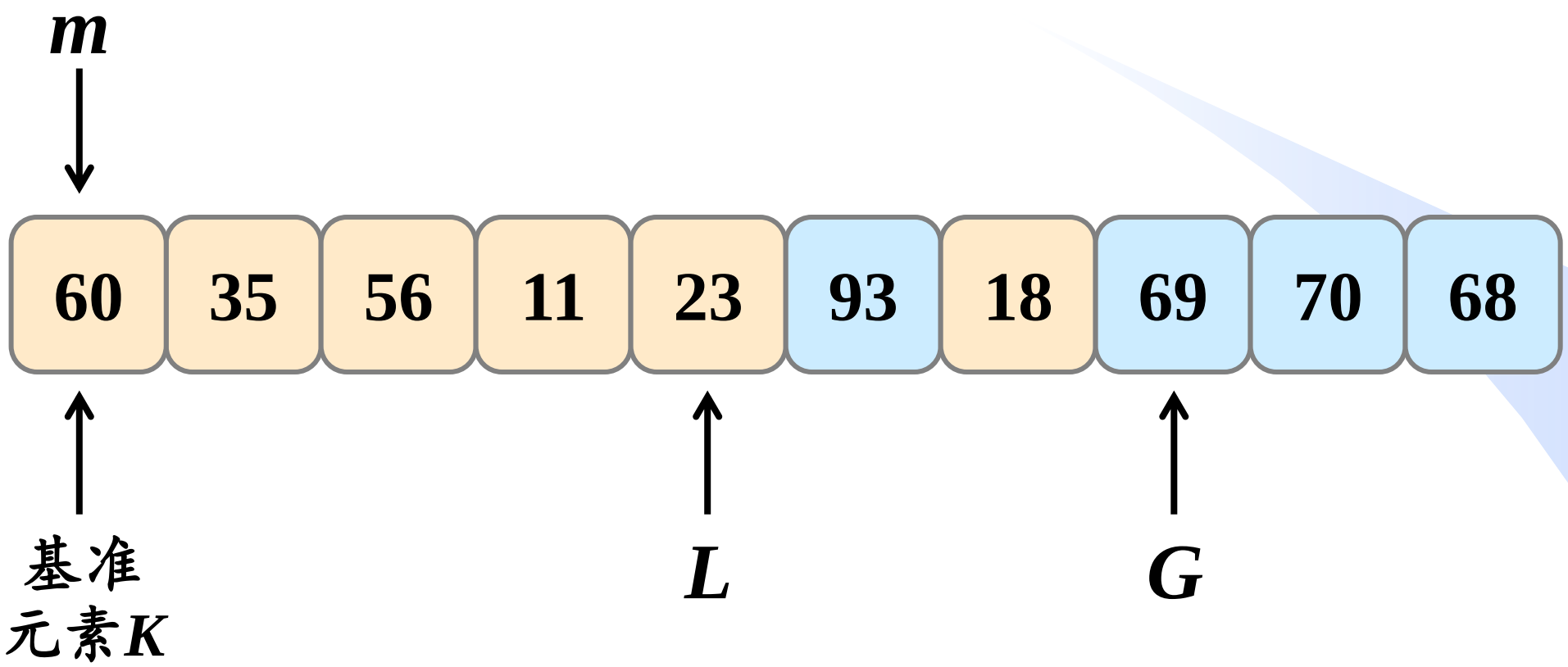
分划过程—例子



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

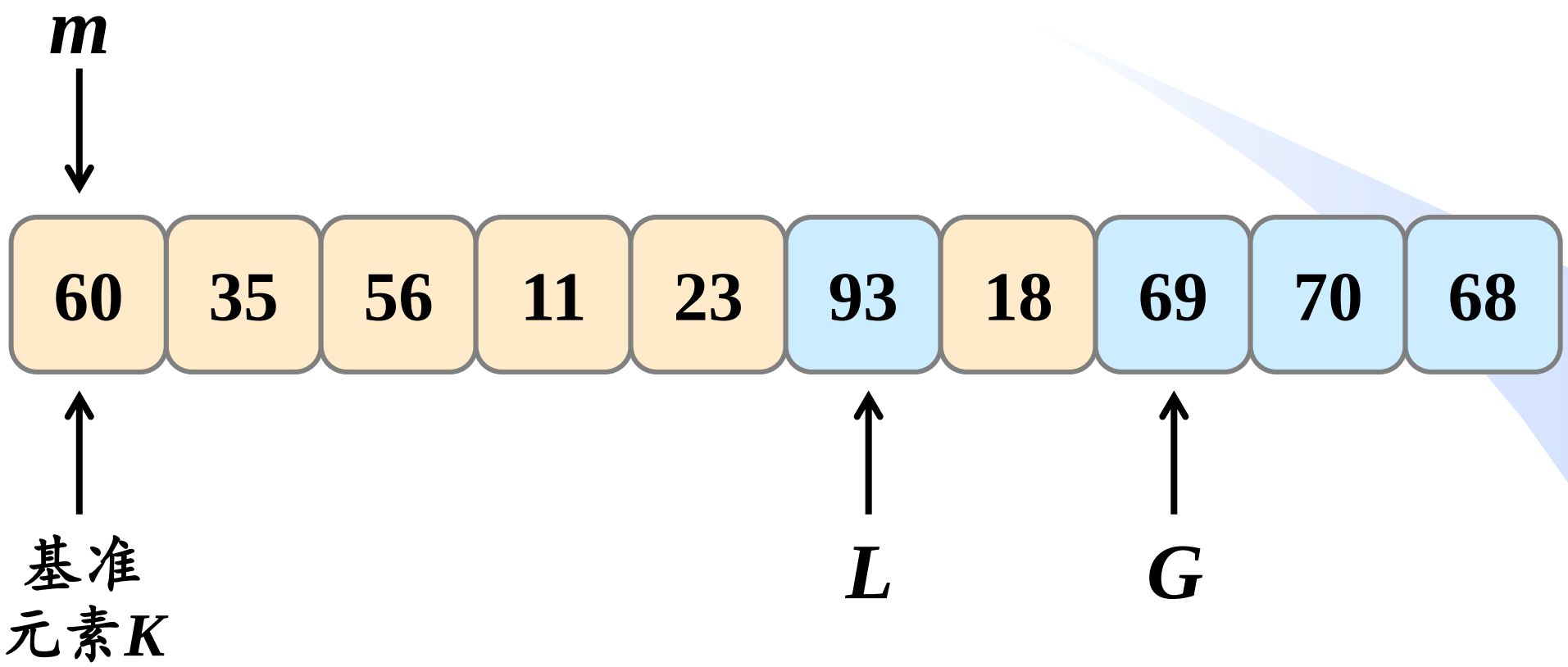
分划过程—例子



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

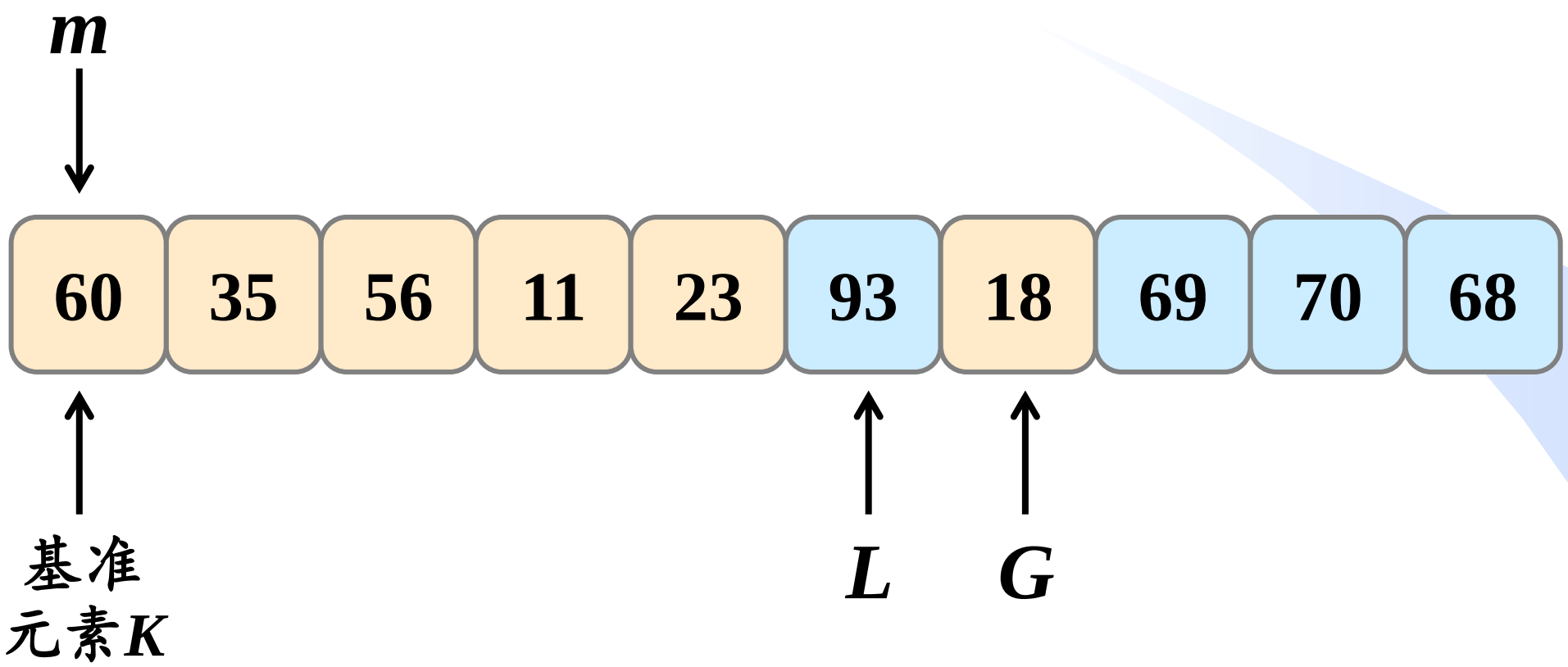
分划过程—例子



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

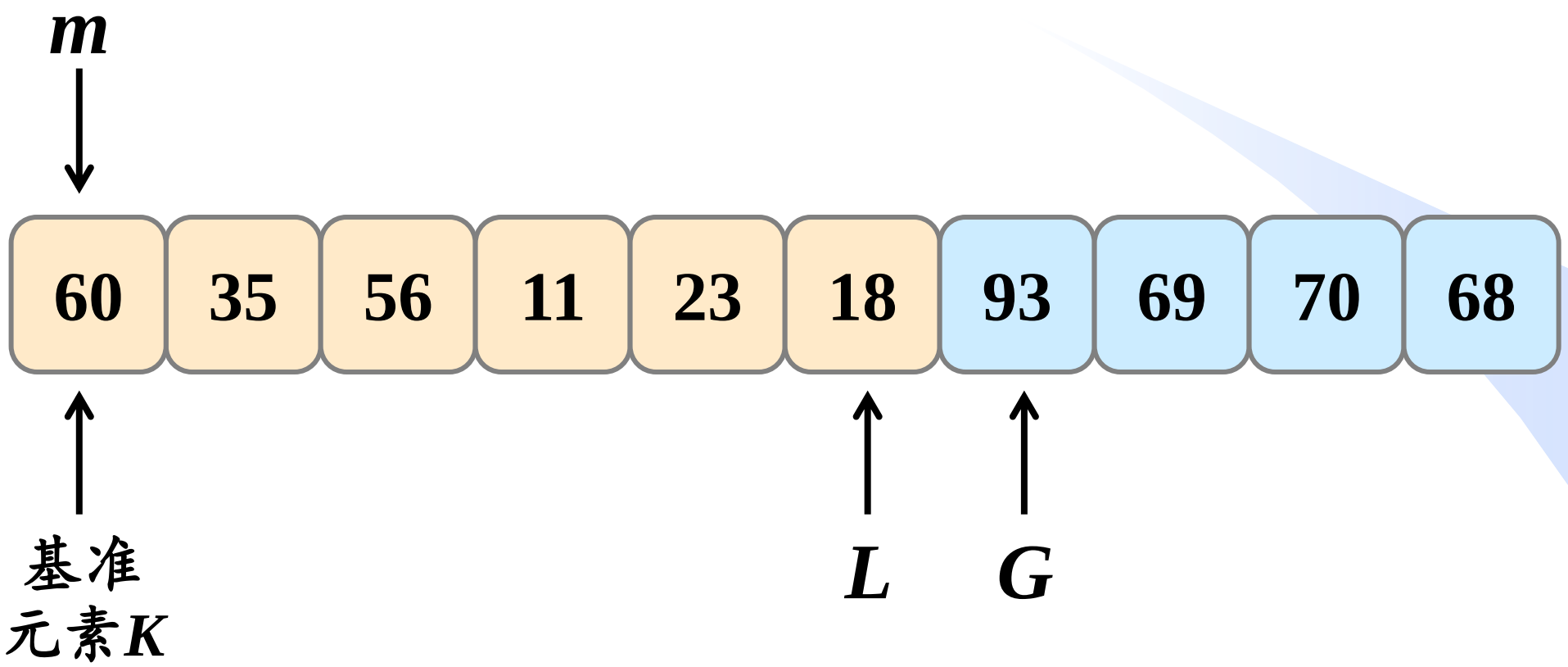
分划过程—例子



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

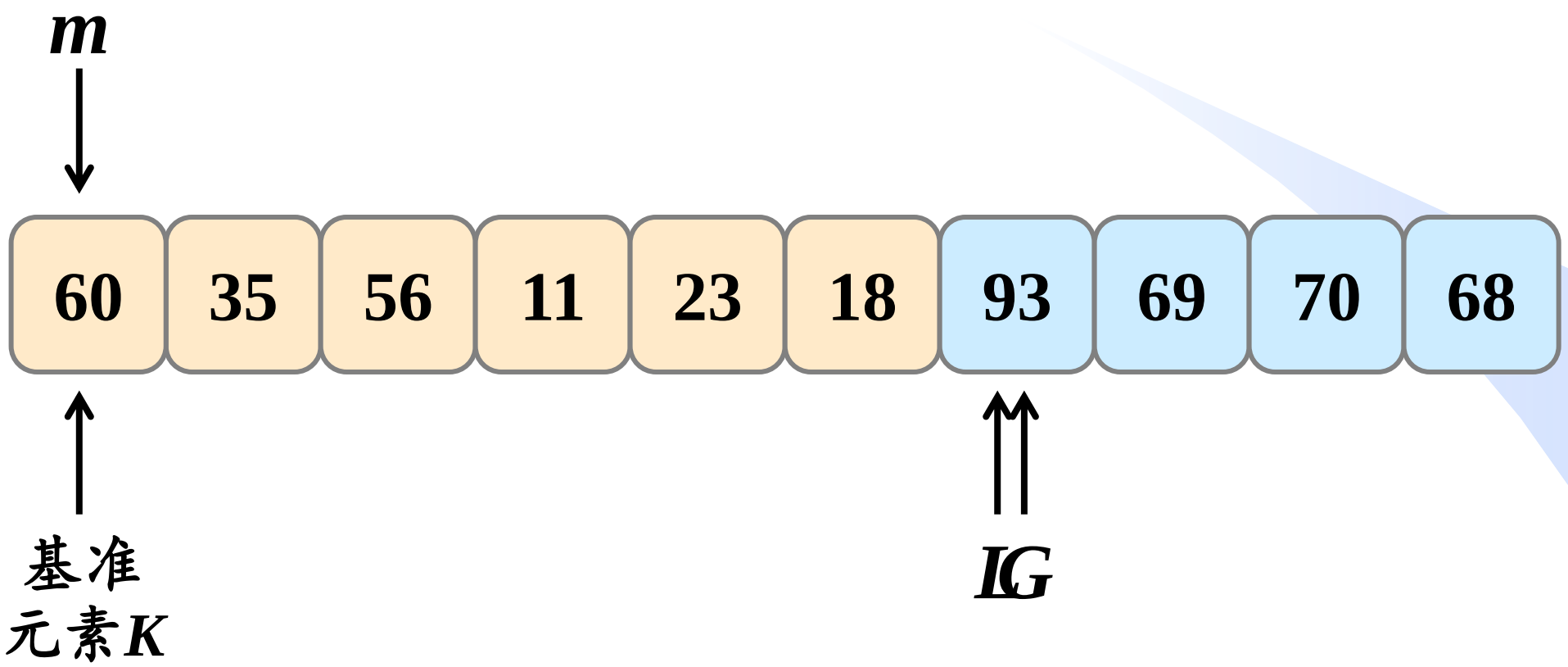
分划过程—例子



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

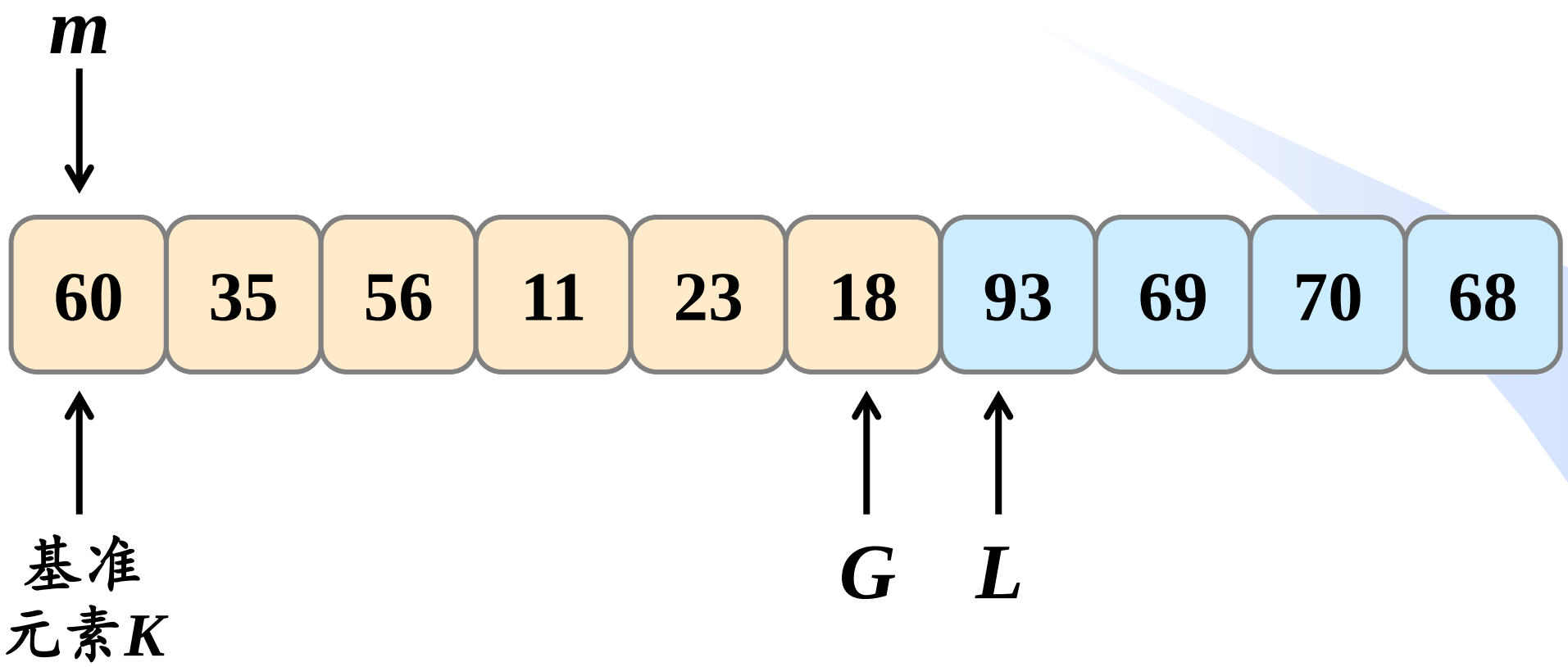
分划过程—例子



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

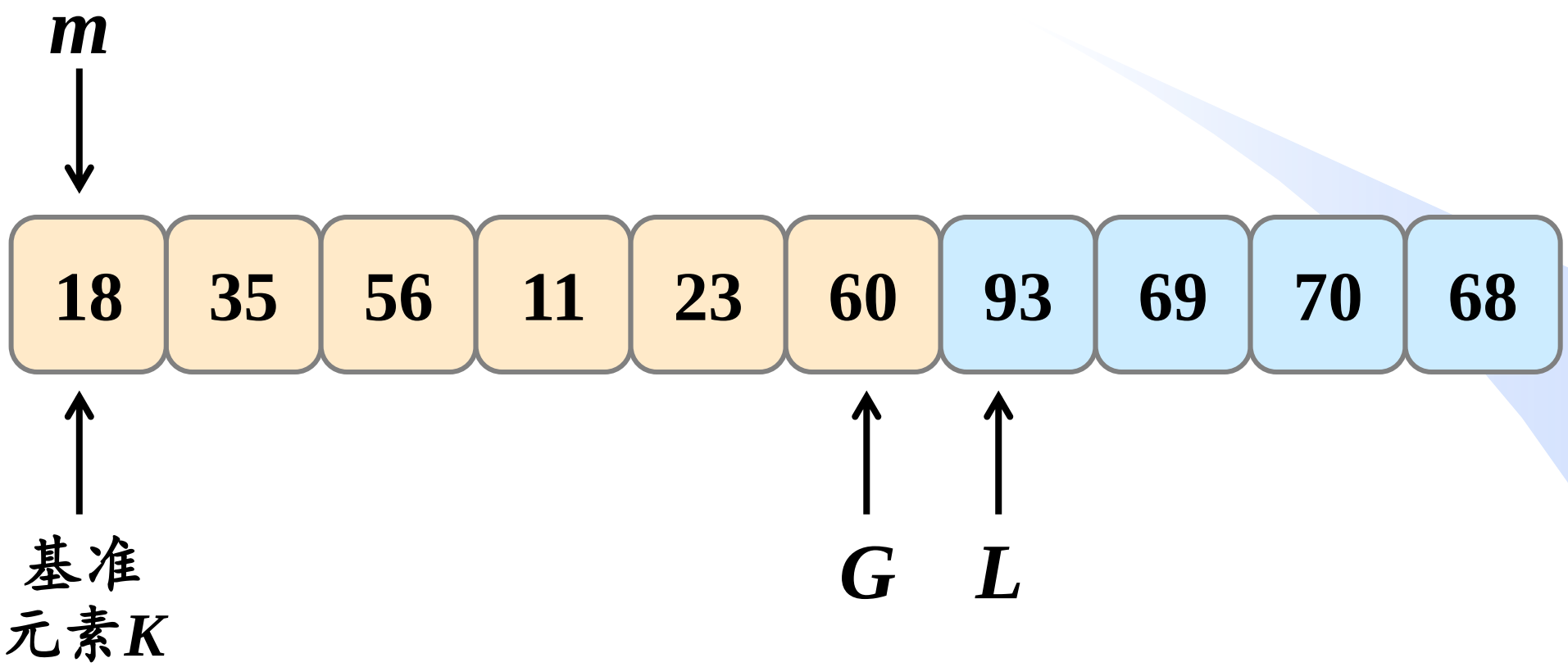
分划过程—例子



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

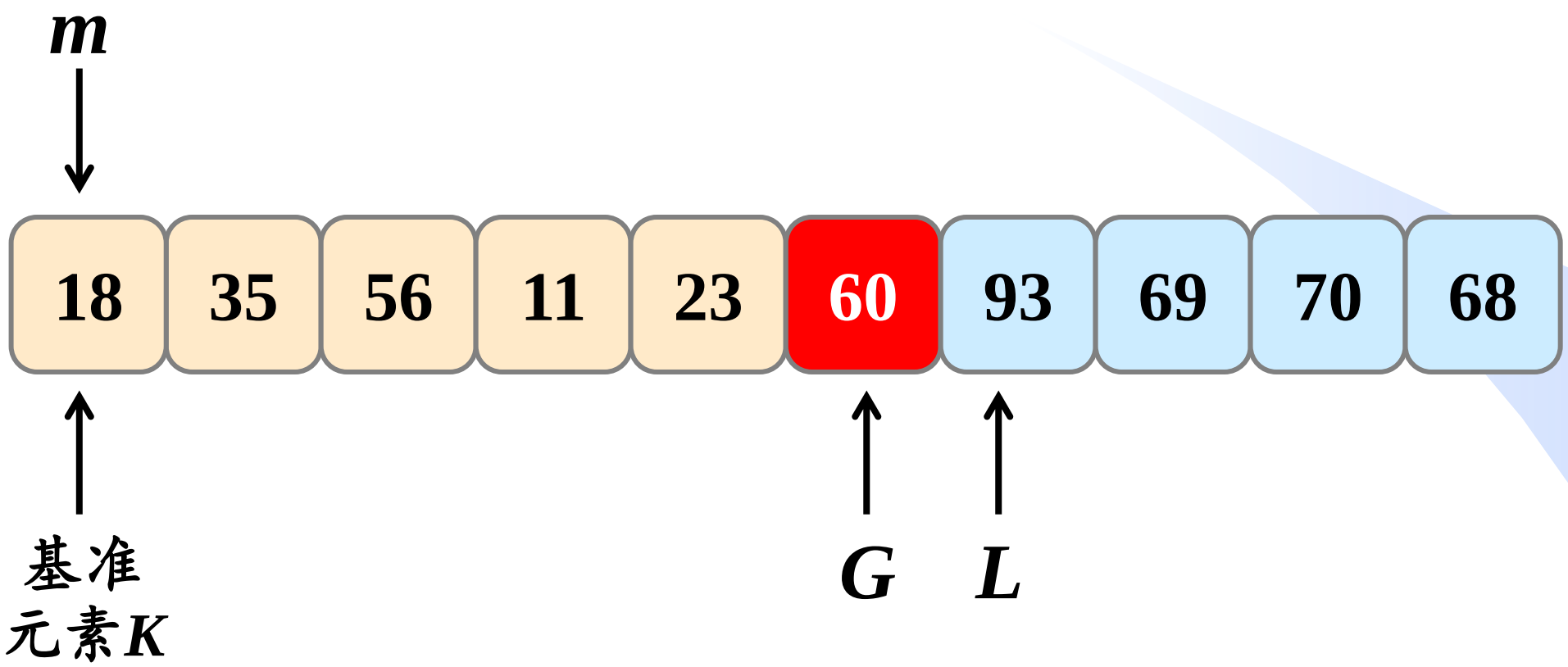
分划过程—例子



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

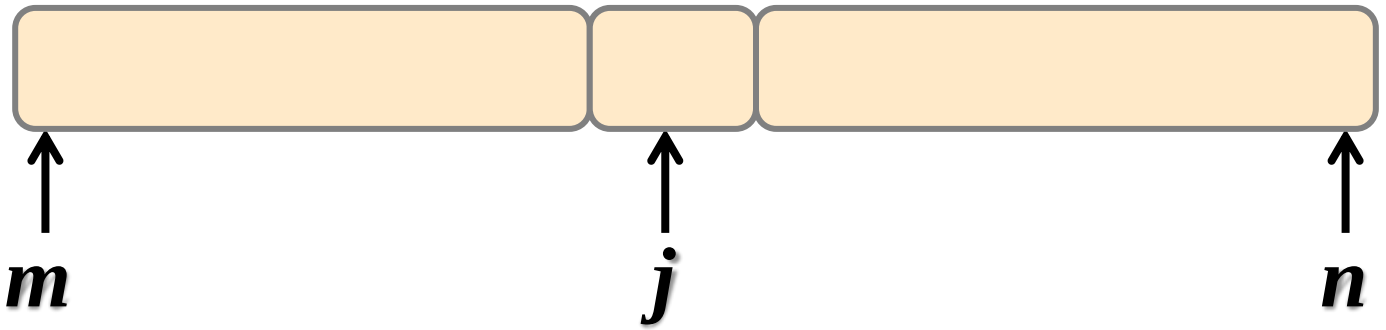
快速排序算法

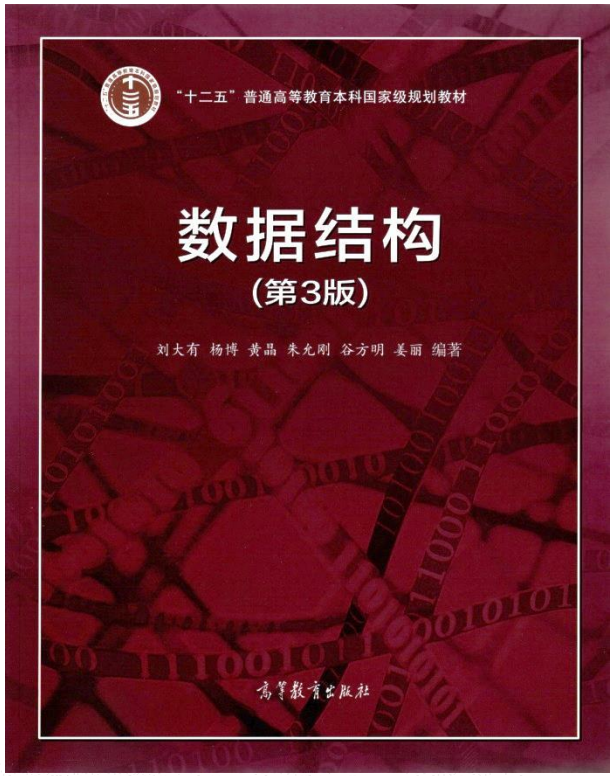
A

```
void QuickSort(int R[], int m, int n){ //对Rm...Rn递增排序
     if(m < n){
          int j=Partition(R, m, n);
          QuickSort(R, m, j-1);
          QuickSort(R, j+1, n);
     }
}
```

分治法

初始调用：QuickSort(R, 1, n)





快速排序

- 快速排序算法
- 时空复杂度分析
- 优化策略
- 快速选择算法

数据之法 结构之美 算法之道

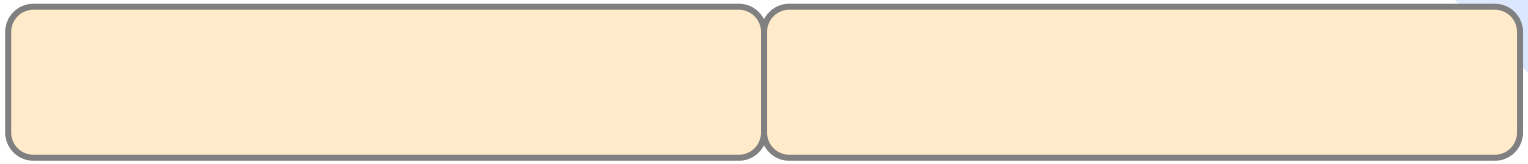
zhuyungang@jlu.edu.cn

Last updated on 2023.11

时空复杂度—最好情况

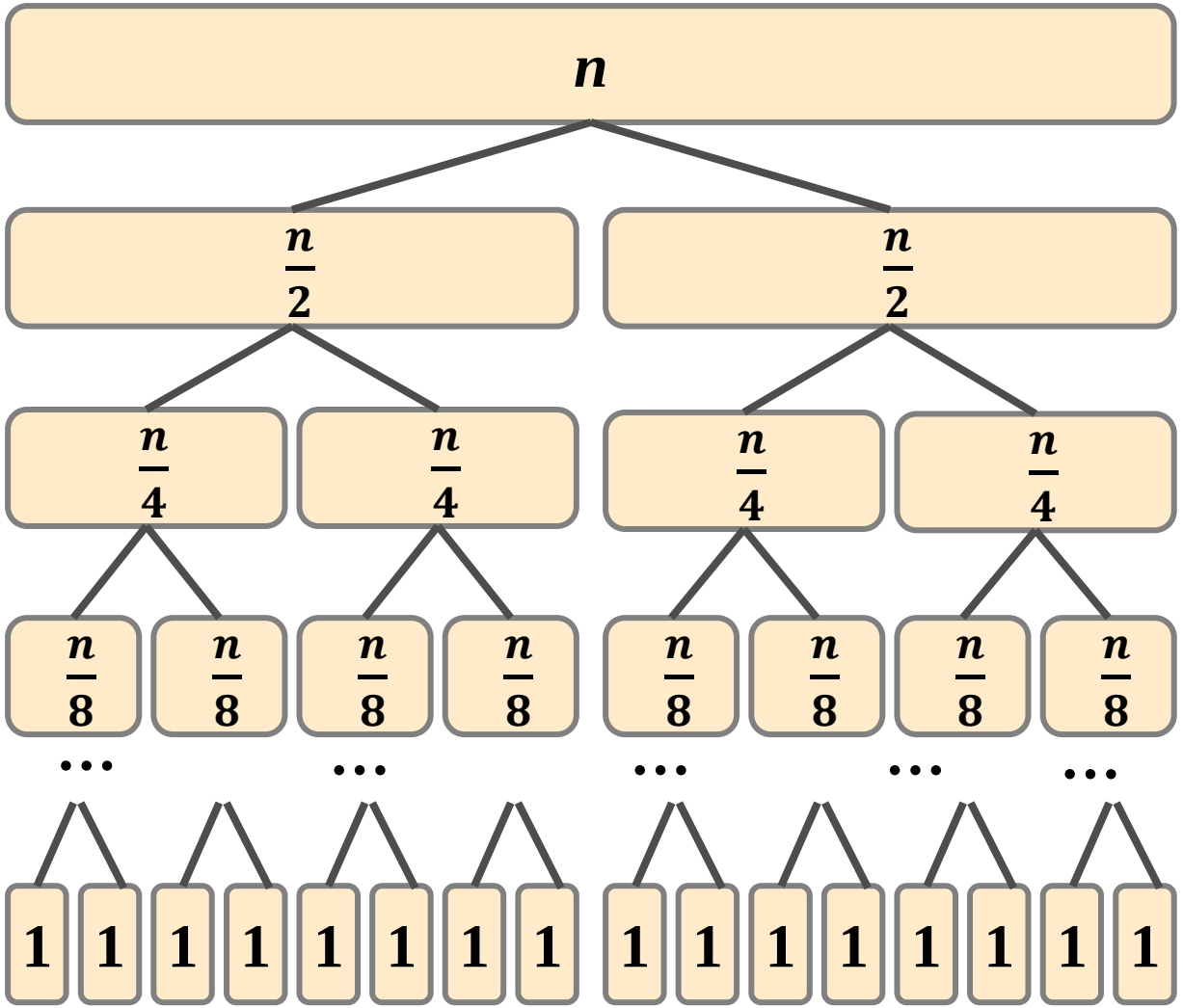
A

每次分划发生在最中间，把子数组分成长度相等的两半



时空复杂度—最好情况

A



O(n)

O(n)

O(n)

O(n)

…

每次分划发生在最中间，把子数组分成长度相等的两半

logn

时间复杂度 O(nlogn)

空间复杂度 O(logn)

时空复杂度—最坏情况

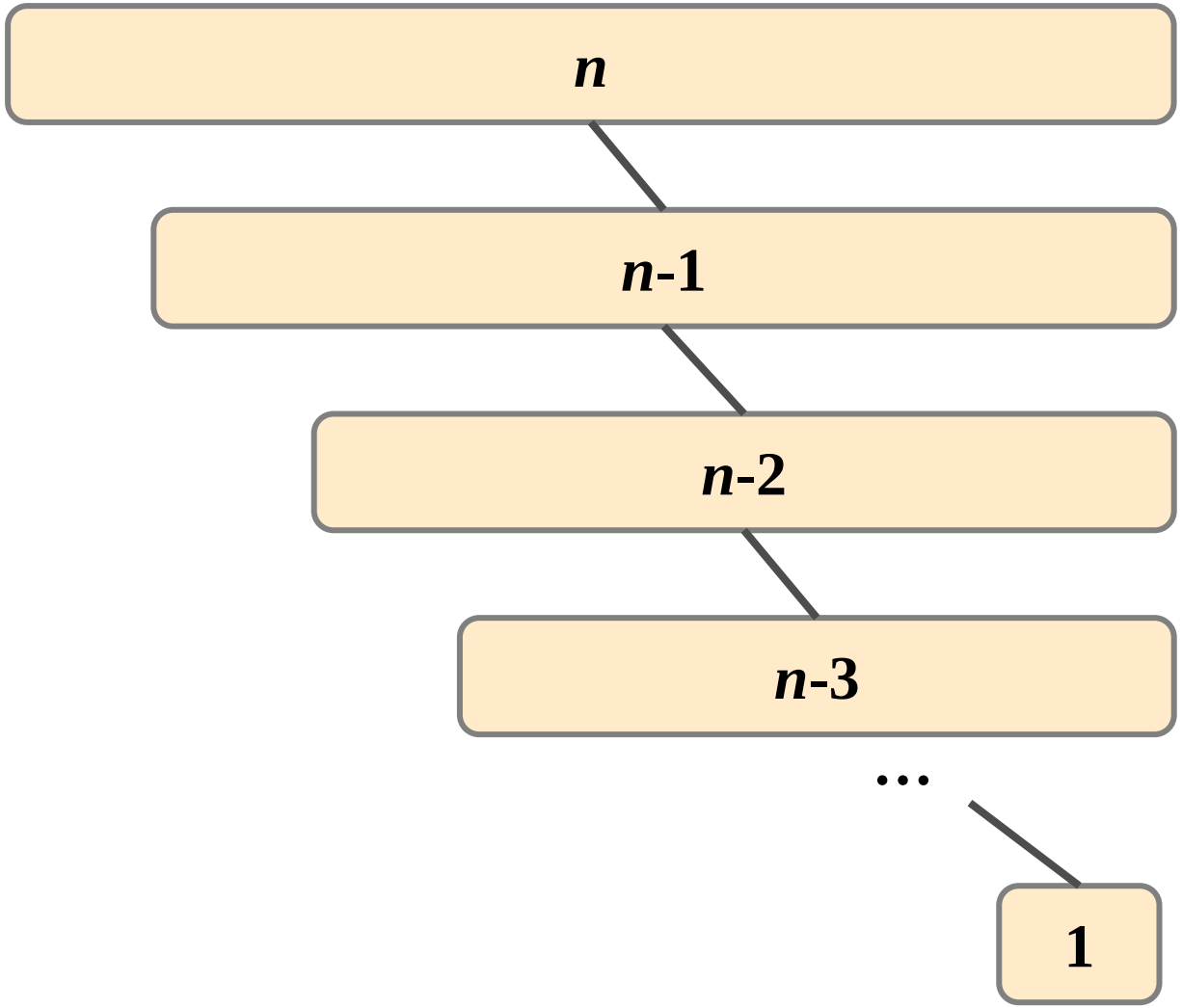
A

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

5	6	8	10	13	18	23	39	50	68

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 $O(n^2)$

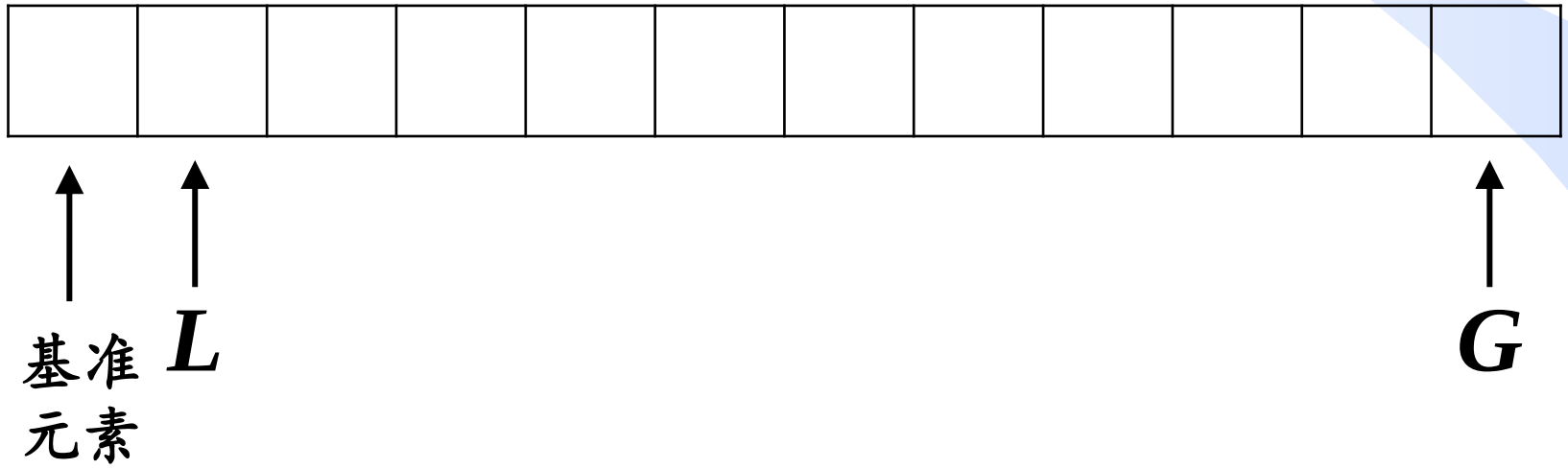
空间复杂度 $O(n)$

时间复杂度—平均情况

C

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

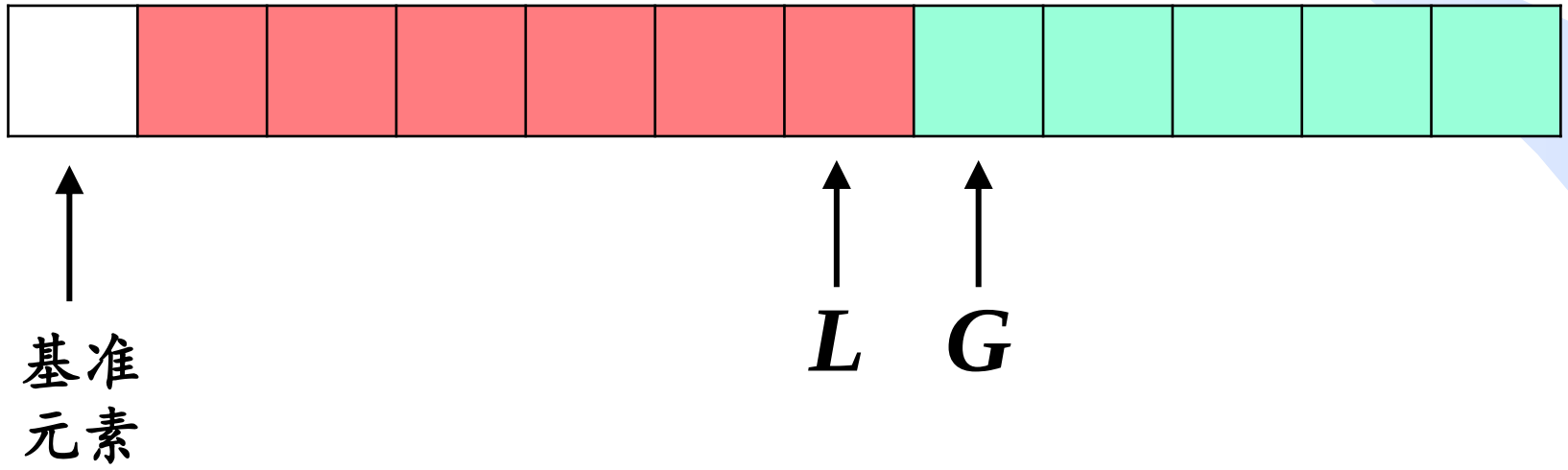
指针G，从右向左找第一个小于等于基准值的元素

时间复杂度—平均情况

C

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

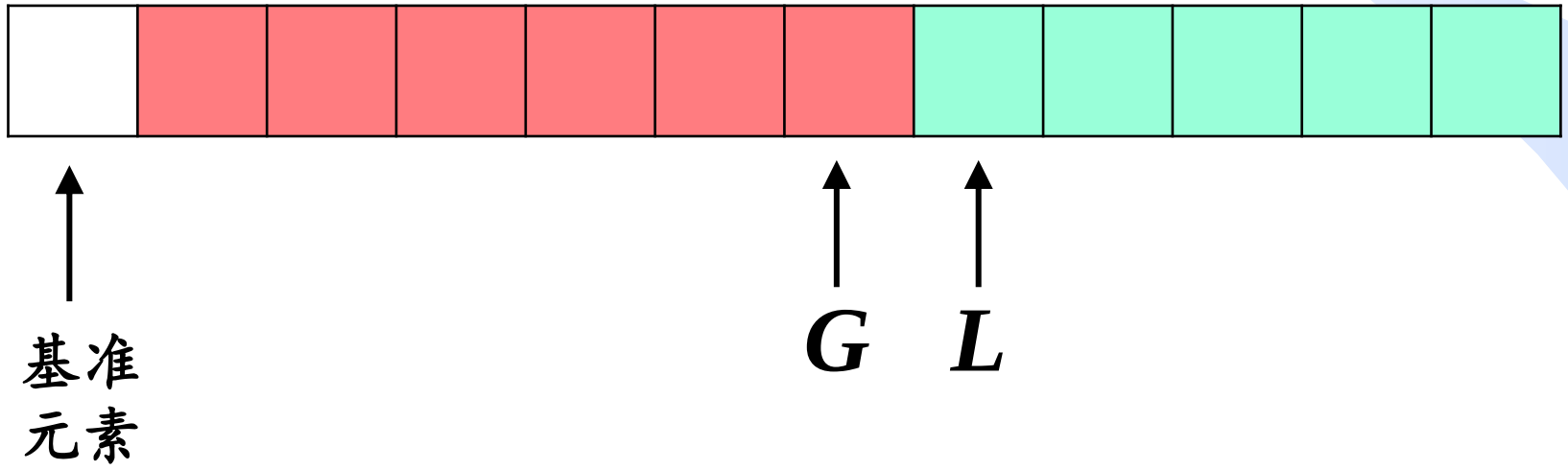
指针G，从右向左找第一个小于等于基准值的元素

时间复杂度—平均情况

C

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

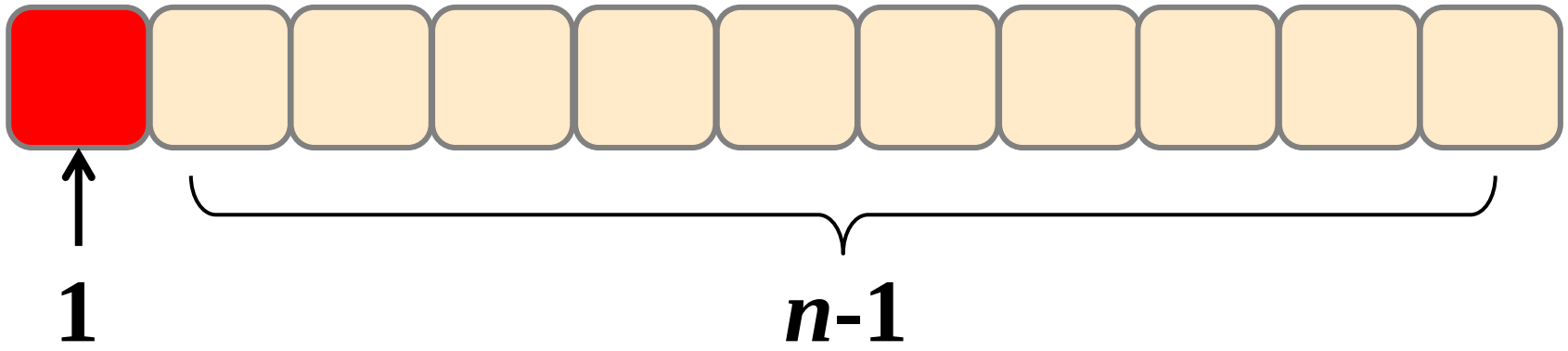
C

- 第1种可能输入：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

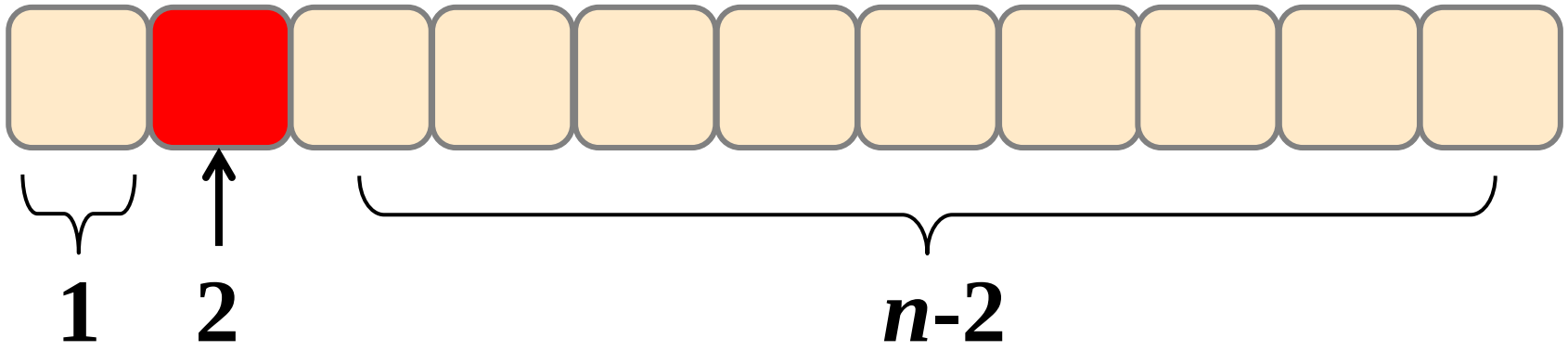
C

- 第2种可能输入：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

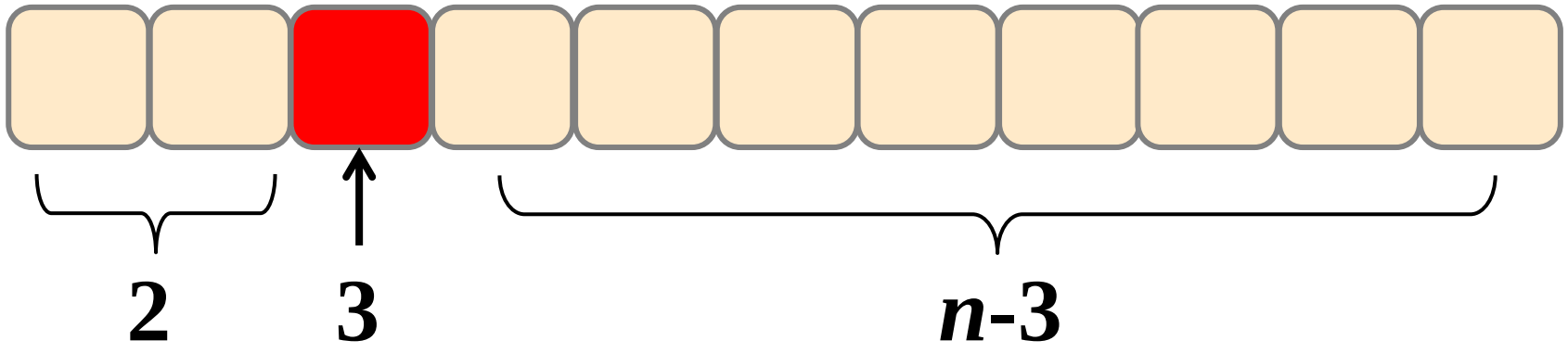
C

- 第3种可能输入：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

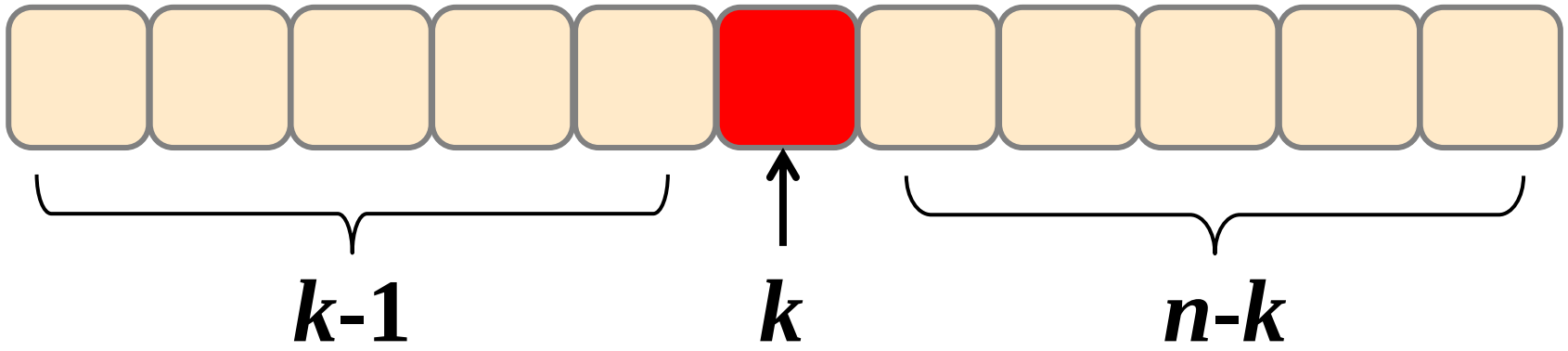
C

- 第k种可能输入：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：$T(n) = (n+1) + T(k-1) + T(n-k)$

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



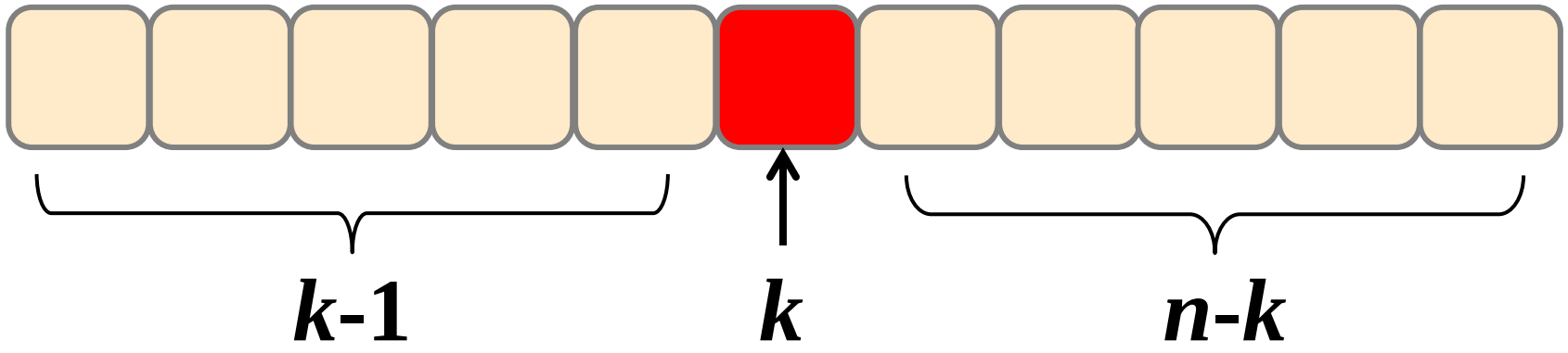
时间复杂度—平均情况

C

- 一共n种可能输入，每种输入出现的概率为$1/n$。

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

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



时间复杂度—平均情况

C

- 一共n种可能输入，每种输入出现的概率为$1/n$。

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

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

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

时间复杂度—平均情况

C

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

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

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

等式两边乘以n：

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

对上式，用$n-1$替换n：

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

用①式减②式：

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

时间复杂度—平均情况

C

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

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

等式两边除以$n(n+1)$：

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

S_n S_{n-1}

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

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

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

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

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

$$\vdots$$

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

时间复杂度—平均情况

C

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

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

$$= O(1.386n\log_2 n)$$

$$= O(n\log n)$$

快速排序的时空复杂度和稳定性

A

排序算法	时间复杂度			空间复杂度	稳定性
	最好	平均	最坏		
排序排序	O(*n*log*n*)	O(*n*log*n*)	O(n^2)	O(log*n*) ~ O(*n*)	不稳定

课下验证

3, 3*, 2 → 2, 3*, 3

课下思考

A

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

课下思考

A

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】

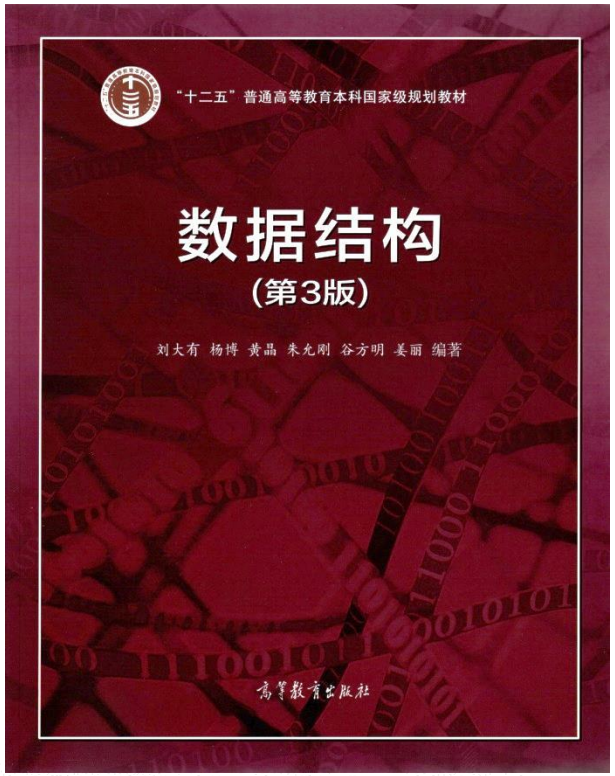
课下思考

B

给定一个非负整数数组 A，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A[*i*]和A[*j*]，从而把偶数换到偶数位，奇数换到奇数位。

```
void ReArrange(int A[], int n){
    int i=1,j=0;
    while(i<n && j<n){
        while(i<n && A[i]%2==1) i+=2;
        while(j<n && A[j]%2==0) j+=2;
        if (i<n && j<n){
            swap(A[i],A[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**
- 快速选择算法

数据之法 结构之美 算法之道

zhuyungang@jlu.edu.cn

Last updated on 2023.11

（1）处理小数据时结合插入排序

A

- 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≤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

A

- 策略2：递归过程中，当前处理的子数组长度≤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

A

- 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

C

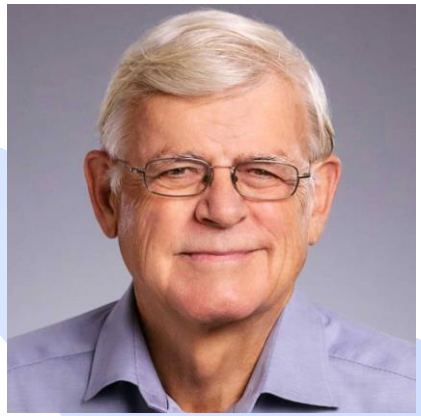
- 如果一个算法的行为不仅由输入决定，而且也由随机数决定，则可称此类算法为随机算法。与之对应，未使用随机数做决策的算法可称为确定型算法。
- 对于确定型算法，可能存在特定的输入数据，其总能导致最坏情况的运行时间。而对于随机算法，以相同的输入数据运行两次，往往会得到不同的运行时间。
- 对于一些问题，当算法在执行过程中面临某种选择时，随机性选择往往比最优选择省时，这种情况下将使得随机算法的效率优于确定型算法。
- 而对于另一些问题，某些算法往往在平均情况下时间复杂度较低，仅在最坏情况下时间复杂度高，此时可以结合随机策略，降低最坏情况发生的概率。

（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。

m　　$m+1$　　$(m+n)/2$　　n



Sedgewick、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]); // Rm+1和Rn的较大者移到Rn处
if(R[m]>R[n]) swap(R[m],R[n]);     // Rm和Rn的较大者移到Rn处
if(R[m+1]>R[m]) swap(R[m+1],R[m]); // Rm+1和Rm的较大者移到Rm处
```

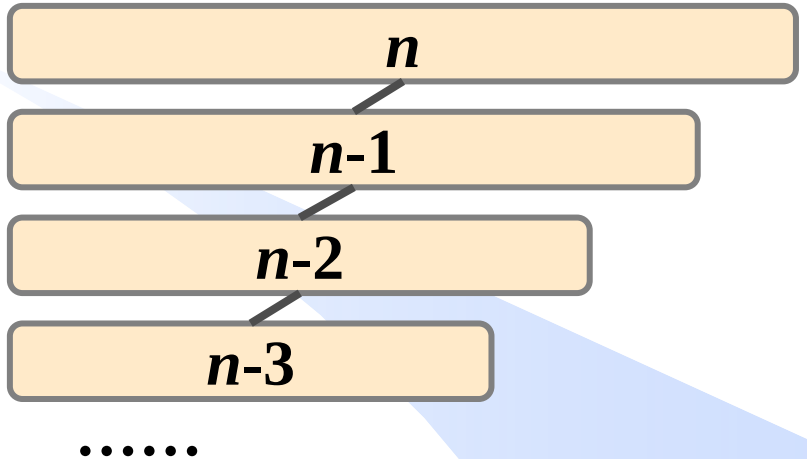
- 降低最坏情况发生的概率，但无法杜绝。

（4）尾递归转化为循环

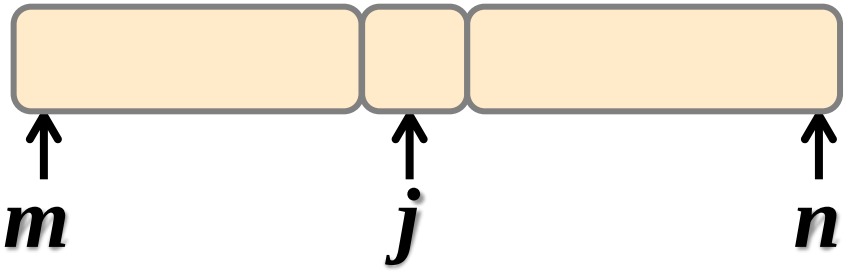
C

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 j=Partition(R, m, n);
      QuickSort(R, m, j-1);
      QuickSort(R, j+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,…,1




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

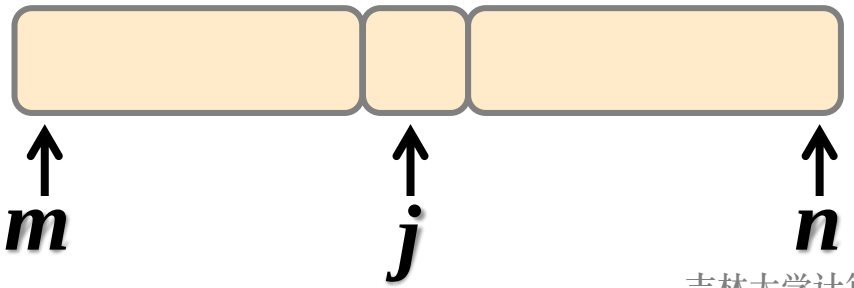
先递归处理左区间
后循环处理右区间

（5）尾递归转循环 + 先处理短区间

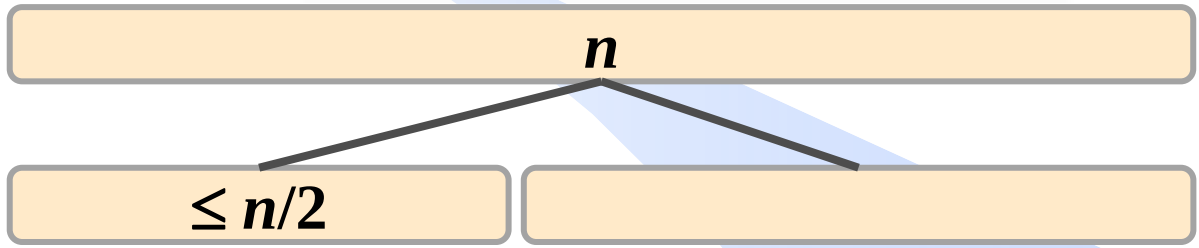
C

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 j=Partition(R,m,n);
        if(j-m < n-j){ //左区间短
            QuickSort(R,m,j-1);
            m=j+1;
        }else{     //右区间短
            QuickSort(R,j+1,n);
            n=j-1;
        }
    }
}
```



先递归处理短区间
后循环处理长区间

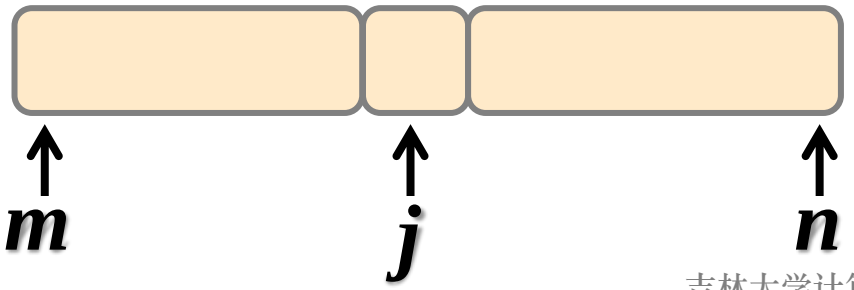


（5）尾递归转循环 + 先处理短区间

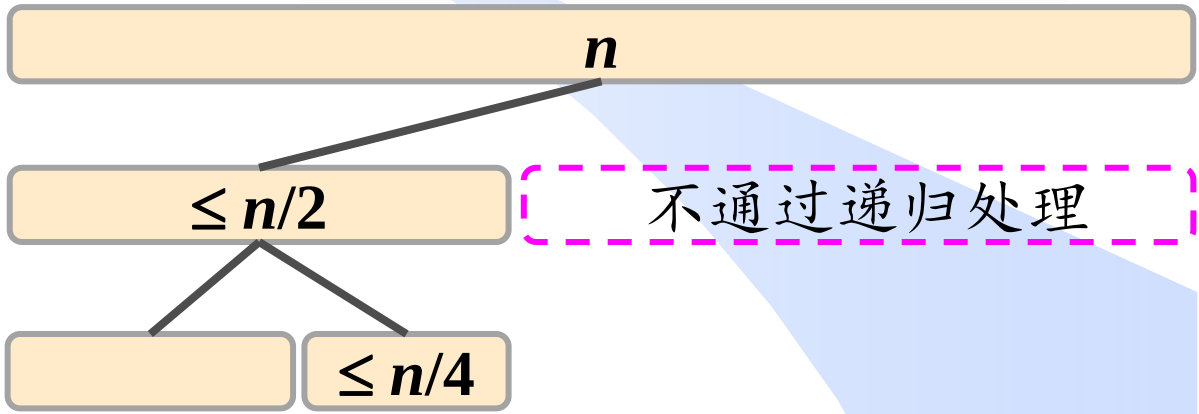
C

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 j=Partition(R,m,n);
        if(j-m < n-j){ //左区间短
            QuickSort(R,m,j-1);
            m=j+1;
        }else{     //右区间短
            QuickSort(R,j+1,n);
            n=j-1;
        }
    }
}
```



先递归处理短区间
后循环处理长区间

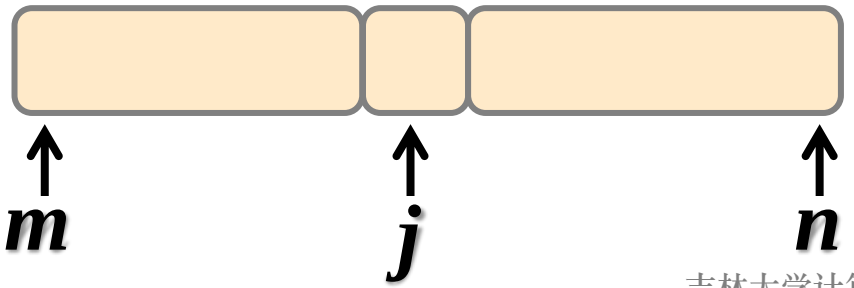


（5）尾递归转循环 + 先处理短区间

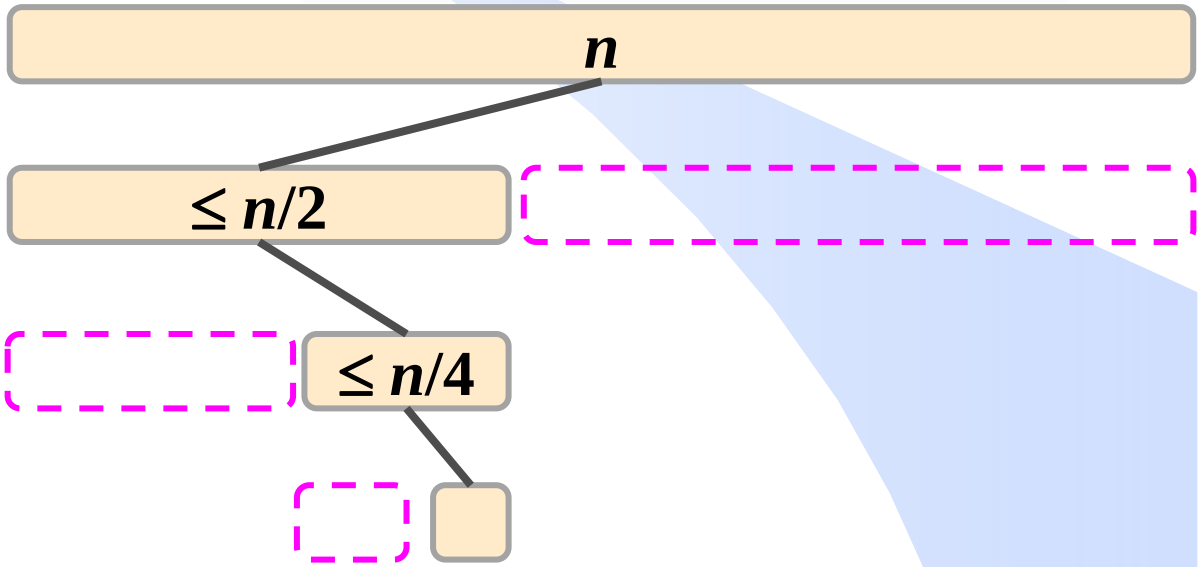
C

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 j=Partition(R,m,n);
        if(j-m < n-j){ //左区间短
            QuickSort(R,m,j-1);
            m=j+1;
        }else{     //右区间短
            QuickSort(R,j+1,n);
            n=j-1;
        }
    }
}
```



先递归处理短区间
后循环处理长区间



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*)

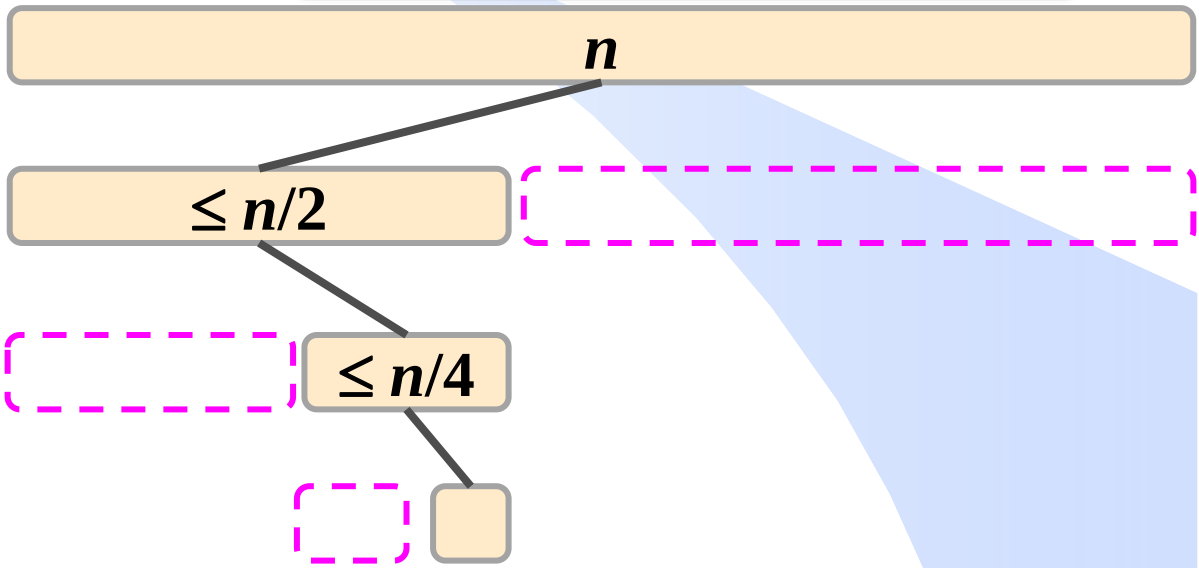
（5）尾递归转循环 + 先处理短区间

C

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 j=Partition(R,m,n);
        if(j-m < n-j){ //左区间短
            QuickSort(R,m,j-1);
            m=j+1;
        }else{     //右区间短
            QuickSort(R,j+1,n);
            n=j-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？

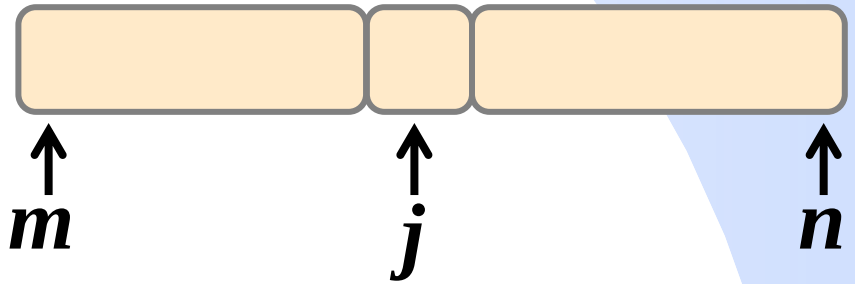
（6）利用栈消除所有递归

B

```
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 j=Partition(R, m, n);
        s.Push(m, j-1);
        s.Push(j+1, n);
    }
}
```

初始调用： QuickSort(R, 1, n)

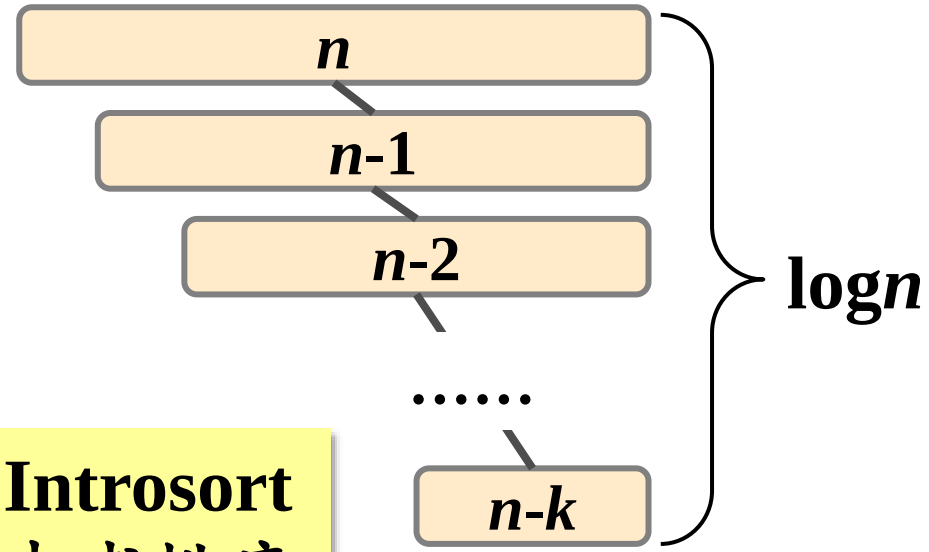
栈内元素：二元组，标识当前处理的区间的起止下标



（7）侦测递归深度，适时转为堆排序

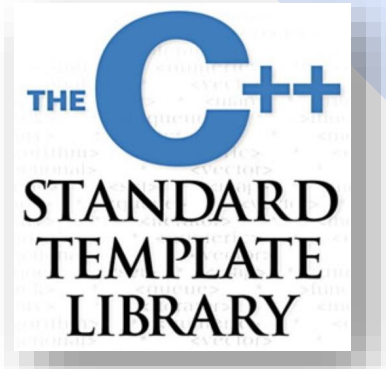
C

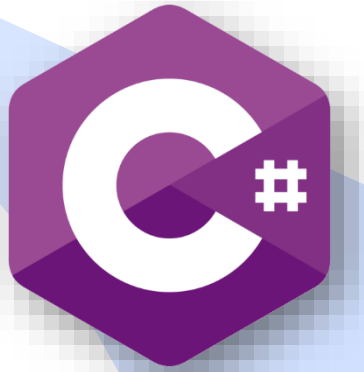
- 侦测快速排序的递归深度，当递归深度达到O(log*n*) 层时，强行停止递归，转而对当前处理的子数组进行堆排序。
- 真正使最坏情况时间复杂度降为O(*n*log*n*)！



Introsort
内省排序

David R. Musser

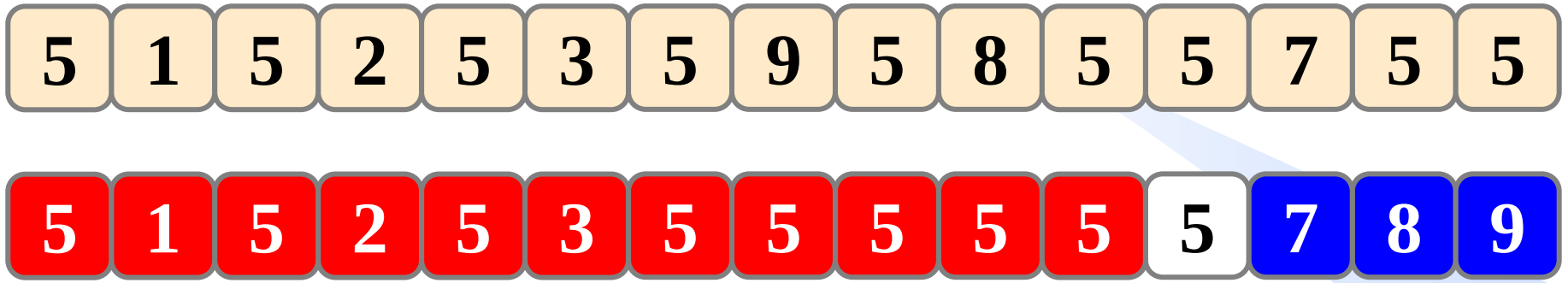




（8）三路分划(3-Way-Partition)

B

- 当重复元素很多时，传统快速排序效率较低。



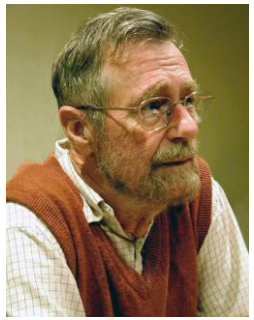
- 修改Partition操作，把当前数组划分为三部分：小于基准元素*K*的元素放在左边，等于*K*的元素放在中间，大于*K*的元素在右边。

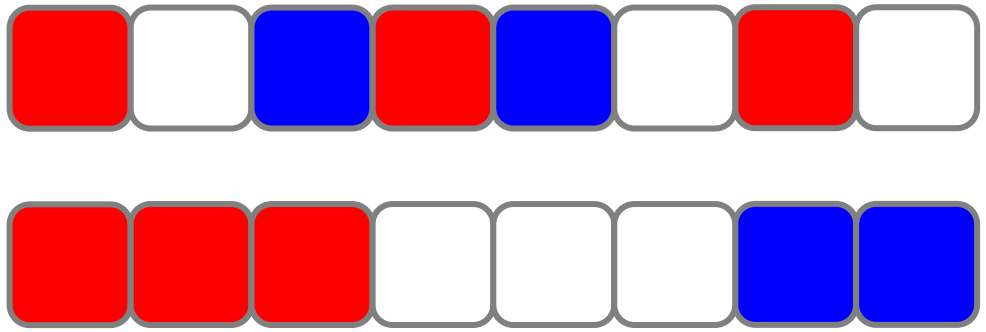
- 进一步递归时，仅对小于*K*的左半部分子数组和大于*K*的右半部分子数组进行递归排序。

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】

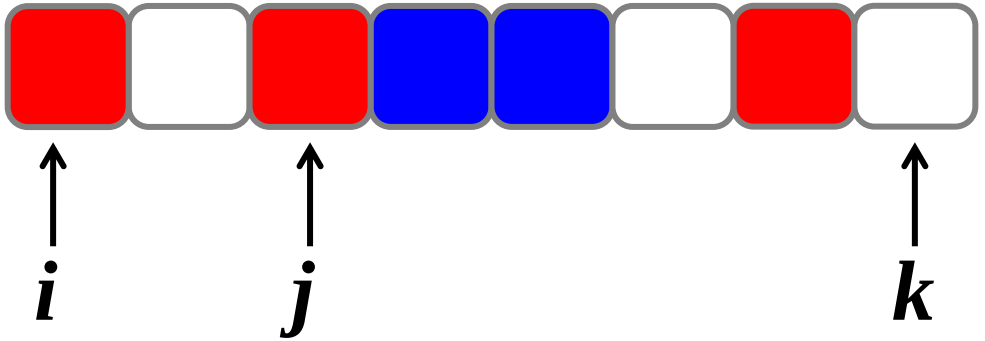




三路分划

B

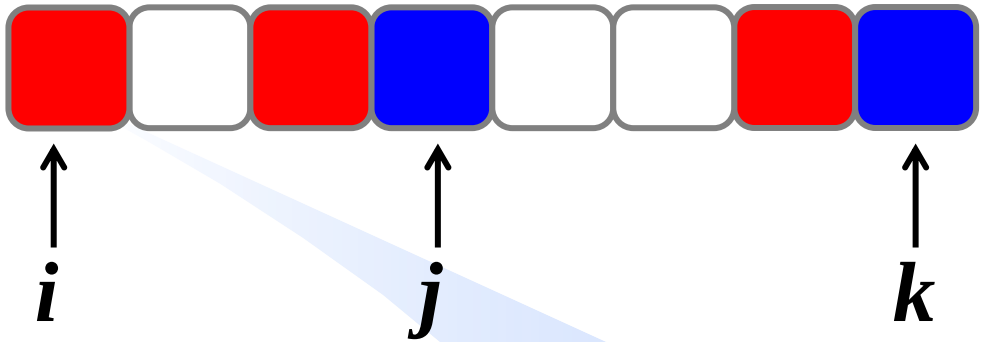
- 设置3个指针，前指针i，中指针j，后指针k；
- 初始时i=1, j=1, k=n；指针j从左往右遍历数组：
- 当R[j]为红，交换R[j]和R[i], i++, j++；
- 当R[j]为蓝，交换R[j]和R[k], k--；
- 通过j的遍历，使红石头换到数组左边，蓝石头换到数组右边。



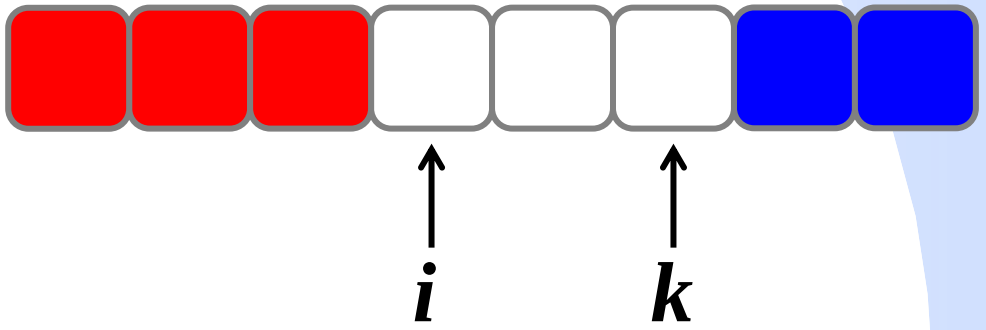
三路分划

B

```
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++;
}
```



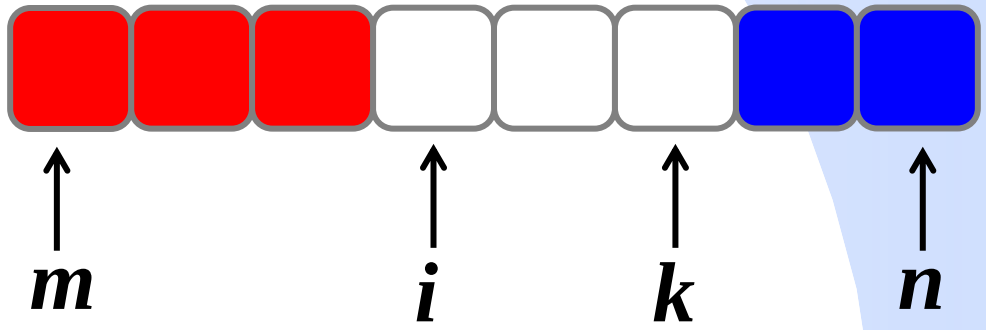
课下思考：
此处为什么不j++？



三路分划

B

```
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++;
   }
}
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

B

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】。

课下思考

B

设有一个整数序列a_1, a_2,…, 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

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$\lfloor 2log_2 n \rfloor$=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*)。

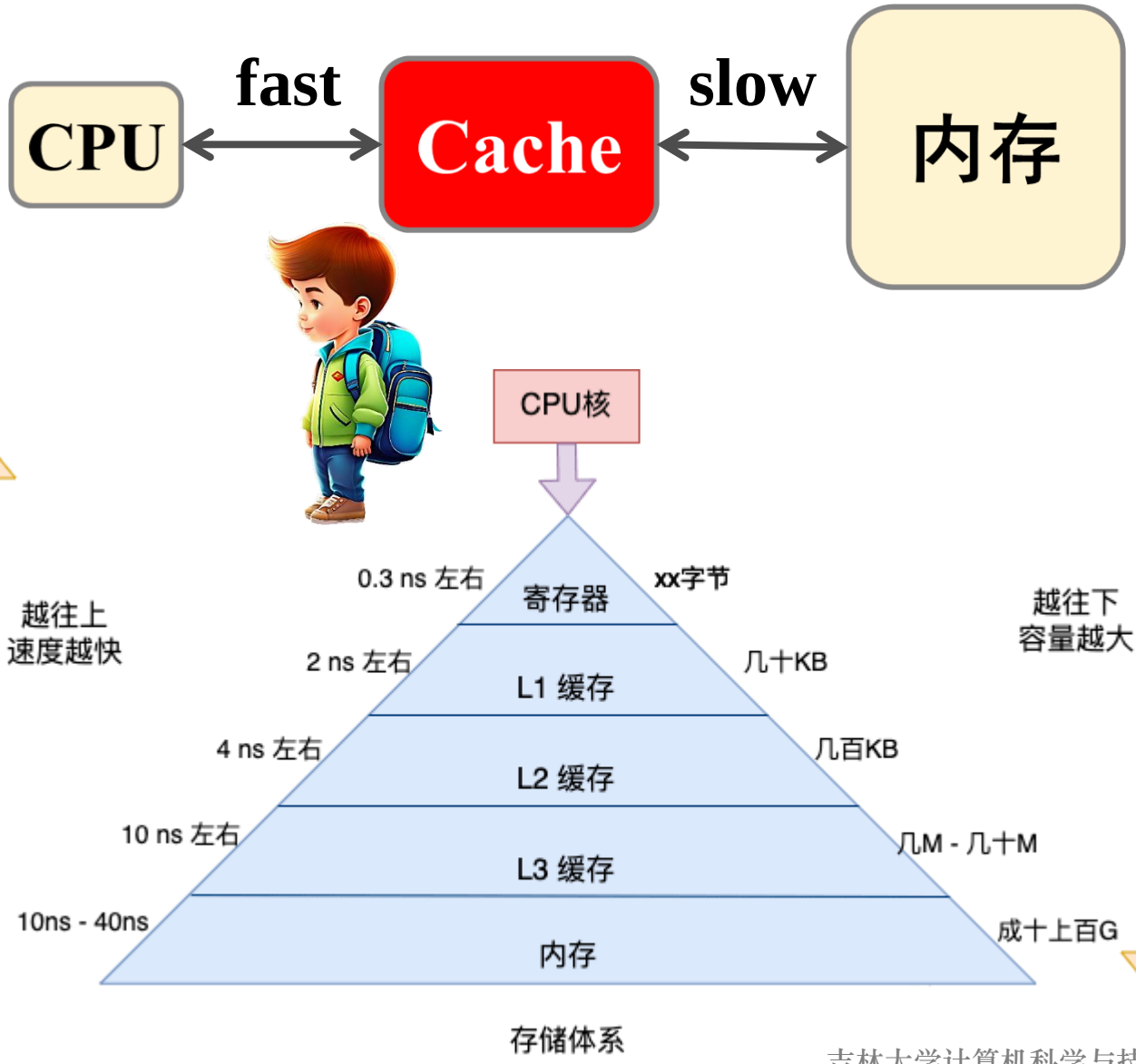
快速排序总结

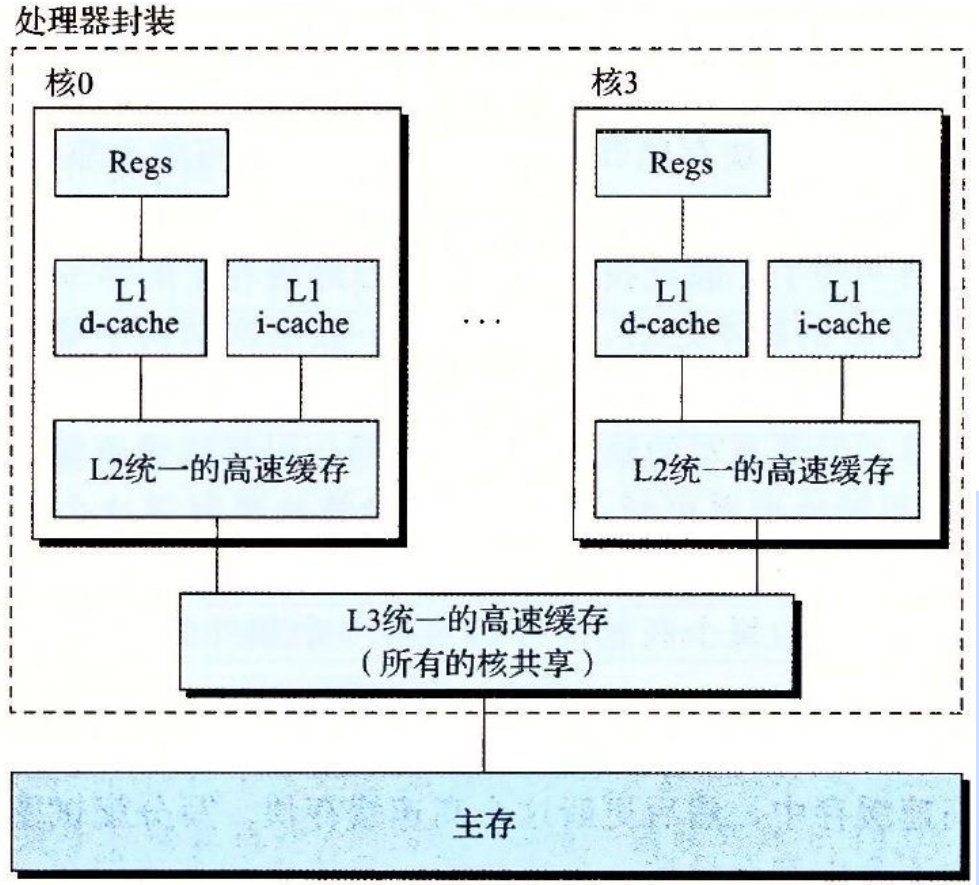
B

- 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
  - ✓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

C



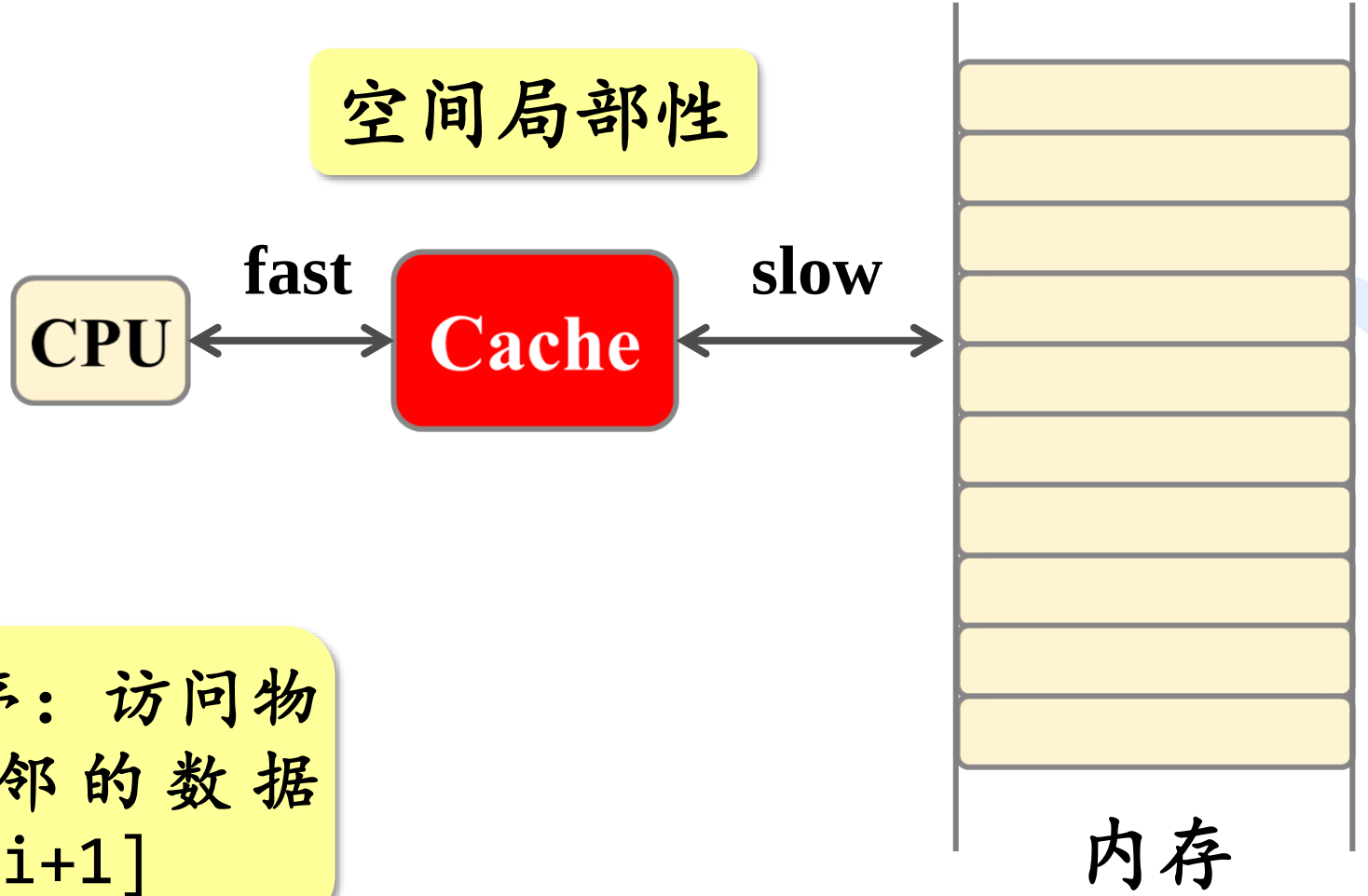


Intel® 酷睿™ i7 处理器的缓存结构

编写缓存友好的代码

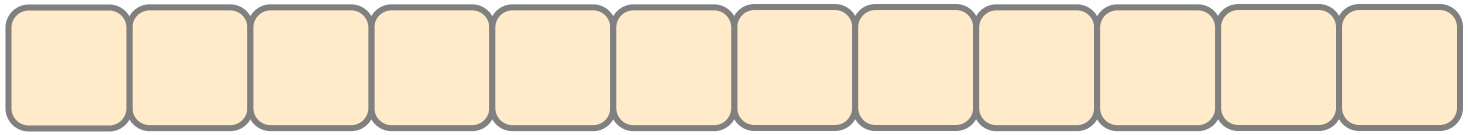
C

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。

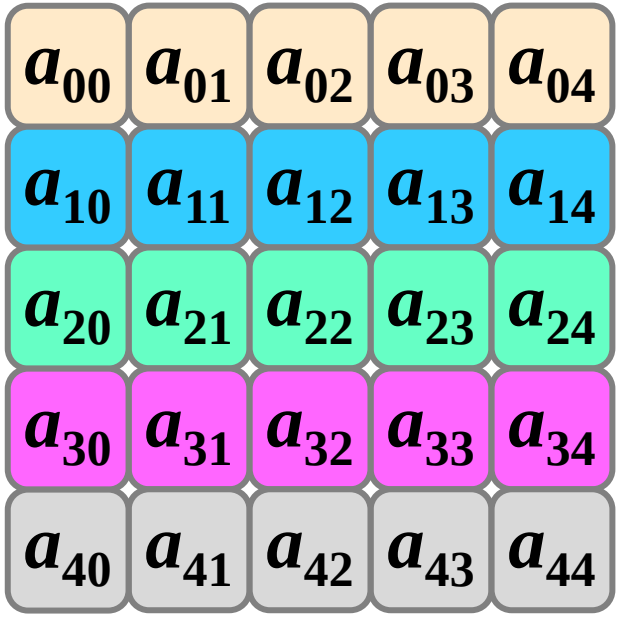
快速排序：访问物理上相邻的数据 R[i]~R[i+1]



堆排序：跳着访问 R[i]~R[2*i]

编写缓存友好的代码

C



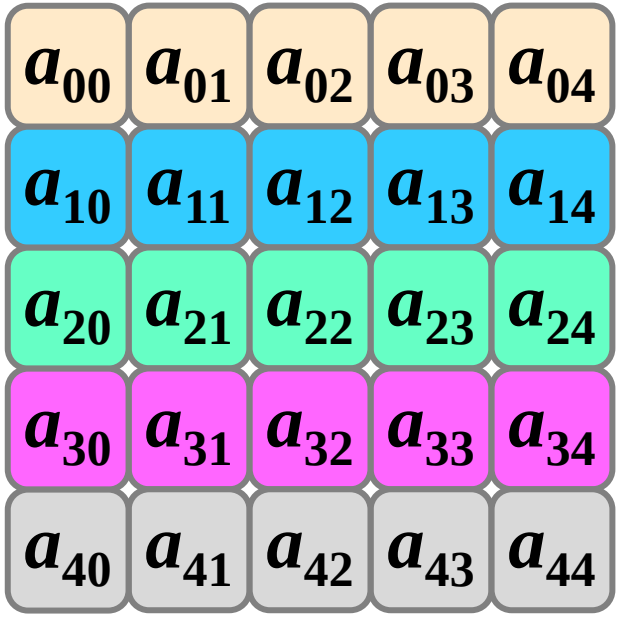

```
int f(int a[N][N]) {
    int sum = 0;
    for(int i=0; i<N; i++)
        for(int j=0; j<N; j++)
            sum += a[i][j];
    return sum;
}
```

有良好的空间局部性，缓存命中率高

内存 a_{00} a_{01} a_{02} a_{03} a_{04} a_{10} a_{11} a_{12} a_{13} a_{14} a_{20} a_{21} a_{22} a_{23} a_{24} a_{30} a_{31} …

编写缓存友好的代码

C

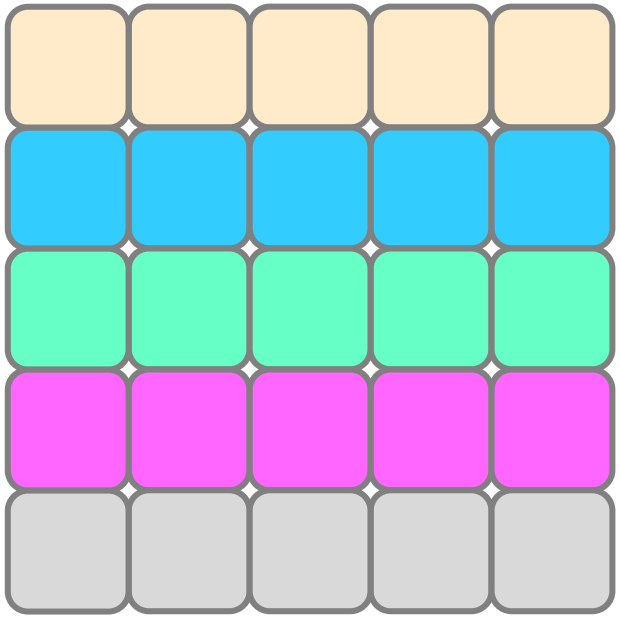



```
int g(int a[N][N]) {
    int sum = 0;
    for(int j=0; j<N; j++)
        for(int i=0; i<N; i++)
            sum += a[i][j];
    return sum;
}
```

有较差的空间局部性，缓存命中率低

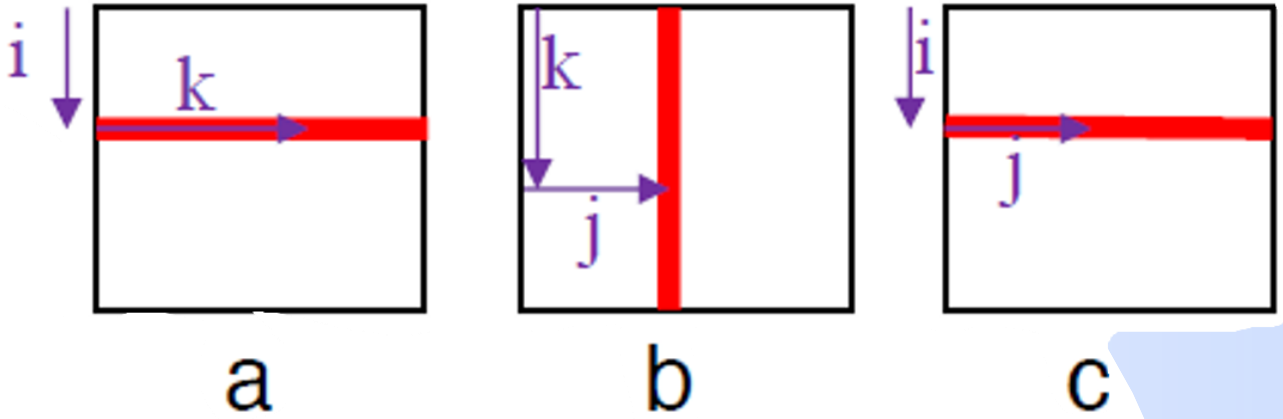
内存 a_{00} a_{01} a_{02} a_{03} a_{04} a_{10} a_{11} a_{12} a_{13} a_{14} a_{20} a_{21} a_{22} a_{23} a_{24} a_{30} a_{31} …

示例:矩阵乘法

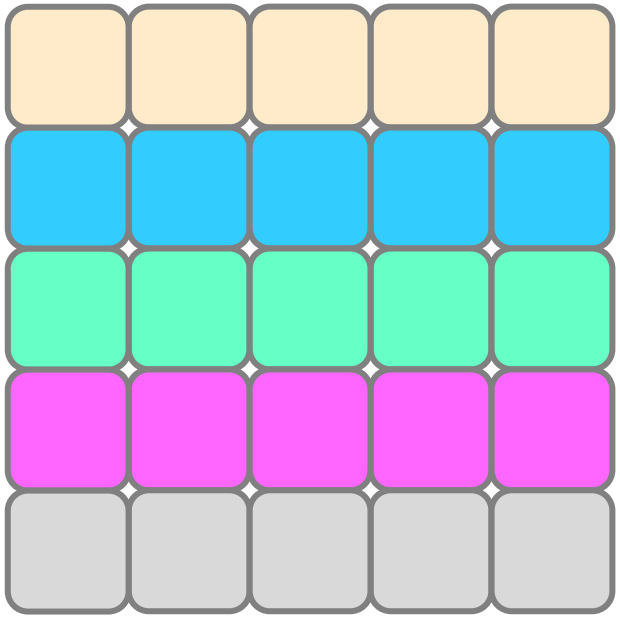


$$C = A \times B$$

$$C_{i,j} = \sum_{k=1}^{n} a_{i,k} \times b_{k,j}$$



示例:矩阵乘法



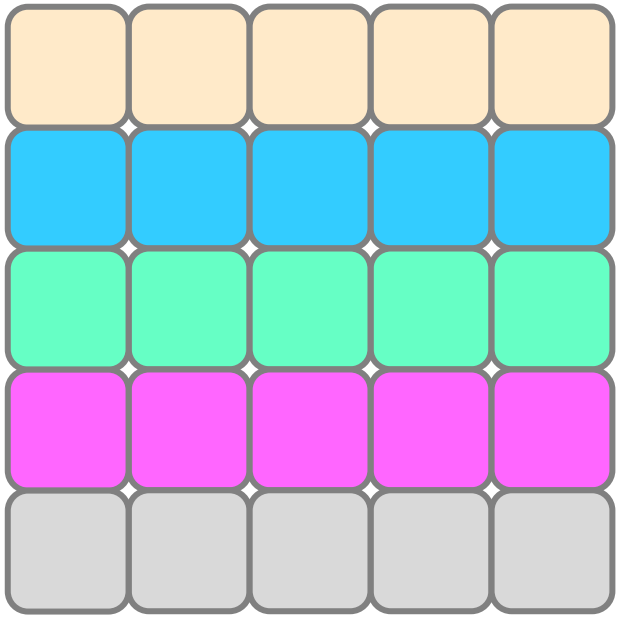
```
for(int i=0; i<n; i++)
    for(int j=0; j<n; j++)
        for(int k=0; k<n; k++)
            c[i][j]+=a[i][k]*b[k][j];
```

对于c：空间局部性好
对于a：空间局部性好
对于b：空间局部性差

内存

示例:矩阵乘法——提高空间局部性

C



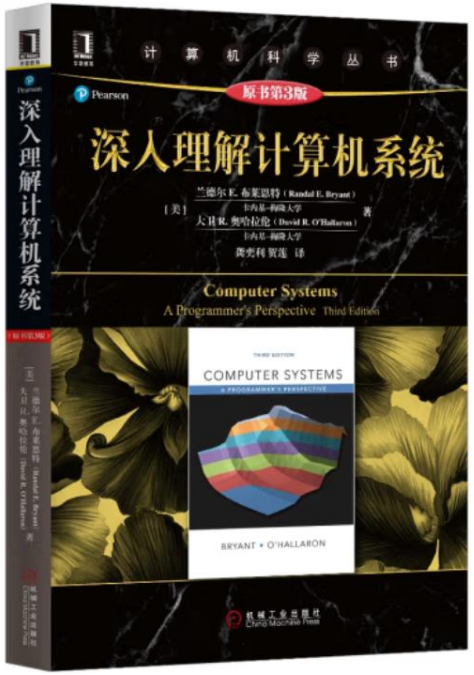
```
for(int i=0; i<n; i++)
    for(int k=0; k<n; k++)
        for(int j=0; j<n; j++)
            c[i][j]+=a[i][k]*b[k][j];
```

对于c：空间局部性好
对于a：空间局部性好
对于b：空间局部性好

内存

示例:矩阵乘法——Intel® 酷睿™ i7 矩阵乘法性能

C



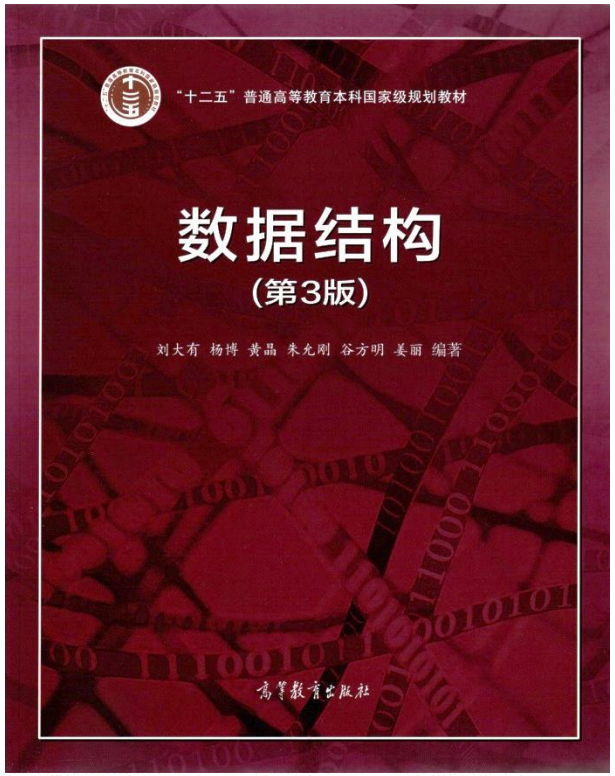
每次内循环迭代所需的CPU周期数

100
10
1

ijk
ikj

50 100 150 200 250 300 350 400 450 500 550 600 650 700

数组大小(n)



快速排序

- 快速排序算法
- 时空复杂度分析
- 优化策略
- **快速选择算法**

数据之法 结构之美 算法之道

zhuyungang@jlu.edu.cn

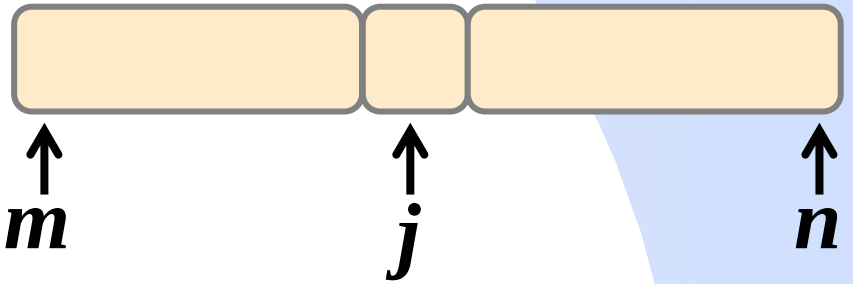
Last updated on 2023.11

快速选择问题

C

给定包含n个元素的整型数组，编写算法找数组中第$k(k<n)$小的数，要求时间复杂度为O(n)。【腾讯、字节跳动、阿里、美团、京东、滴滴、网易、小米、华为、旷视科技、拼多多、苹果、微软、百度、谷歌、快手面试题】

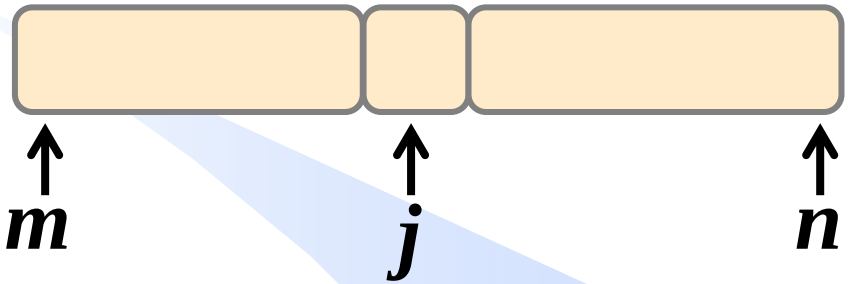
- 基于快速排序的Partition算法，将数组分成两部分。
- 若左区间元素个数等于k-1，则基准元素即为第k小的元素。
- 若左区间元素个数大于k-1，第k小的元素必在左区间，对左区间递归找第k小的元素。
- 若左区间长度小于k-1，第k小的元素必在右区间，若左区间元素个数为n_L，则在右区间递归找第k-n_L-1小的元素。



快速选择问题

C

```
int QuickSelect(int R[],int m, int n, int k){
    int j=Partition(R, m, n);
    int nL=j-m;      //左部分元素个数
    if(nL==k-1) return R[j];
    else if(nL>k-1) //在左部分找第k小数
        return QuickSelect(R,m,j-1,k);
    else                //在右部分找第k-nL-1小数
        return QuickSelect(R, j+1, n, k-nL-1);
}
```



与快速排序不同在于，只需递归处理左右两个子数组中的一个

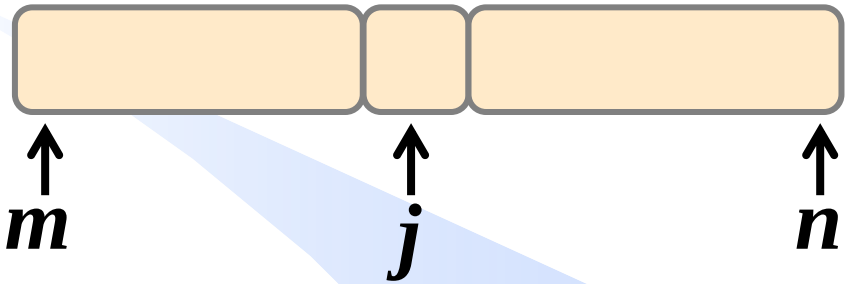
减而治之，减治法

快速选择问题——迭代形式

C

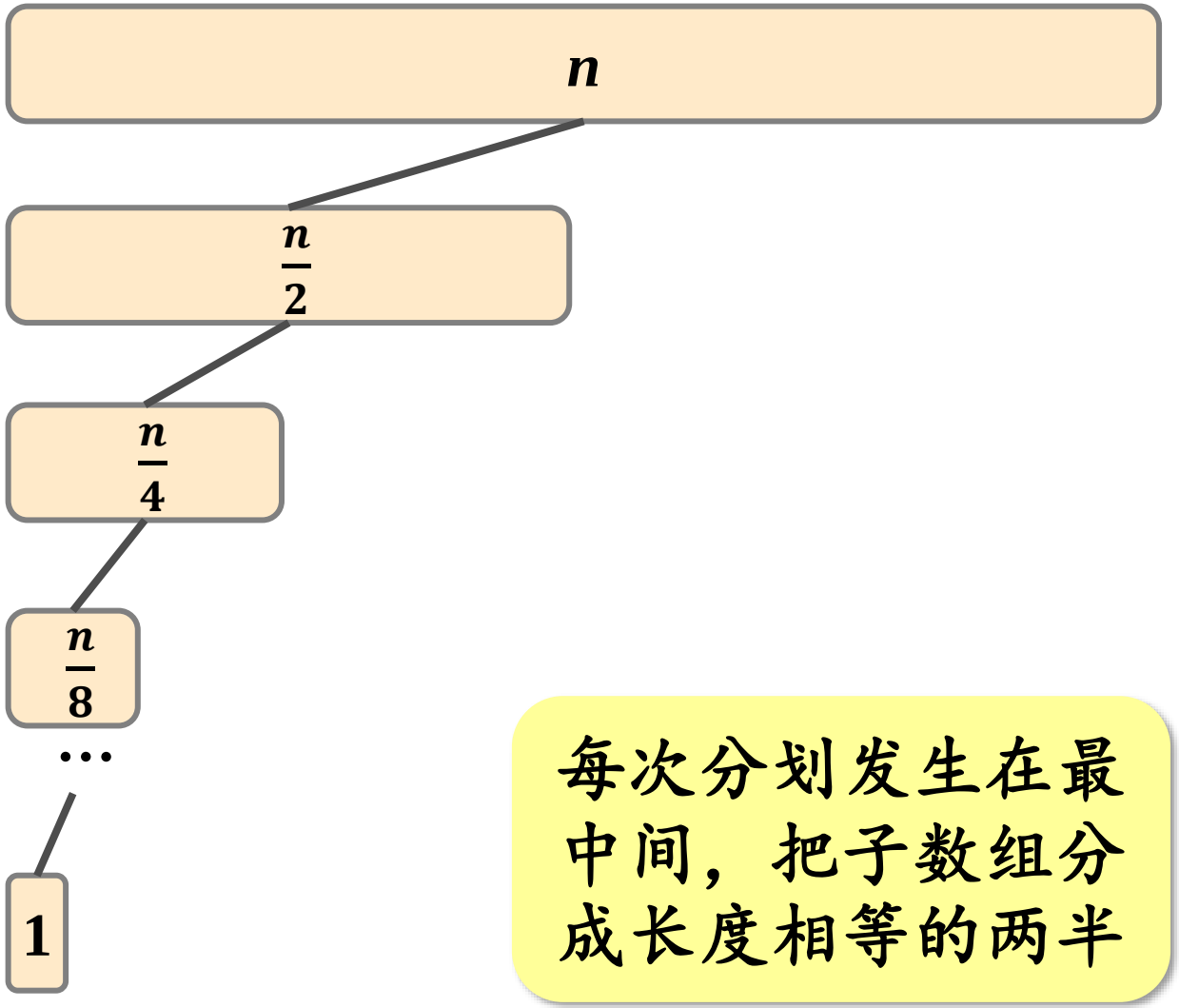
```
int QuickSelect(int R[],int m, int n, int k){
    int j, nL;
    while(true){
        j=Partition(R, m, n);
        nL=j-m;      //左部分元素个数
        if(nL==k-1) return R[j];
        else if(nL>k-1) n=j-1; //在左部找第k小数
        else m=j+1,k=k-nL-1;   //在右部找第k-nL-1小数
    }
}
```

初始调用QuickSelect(R, 1, n, k).



最好/平均情况时间复杂度

C



每次分划发生在最中间，把子数组分成长度相等的两半

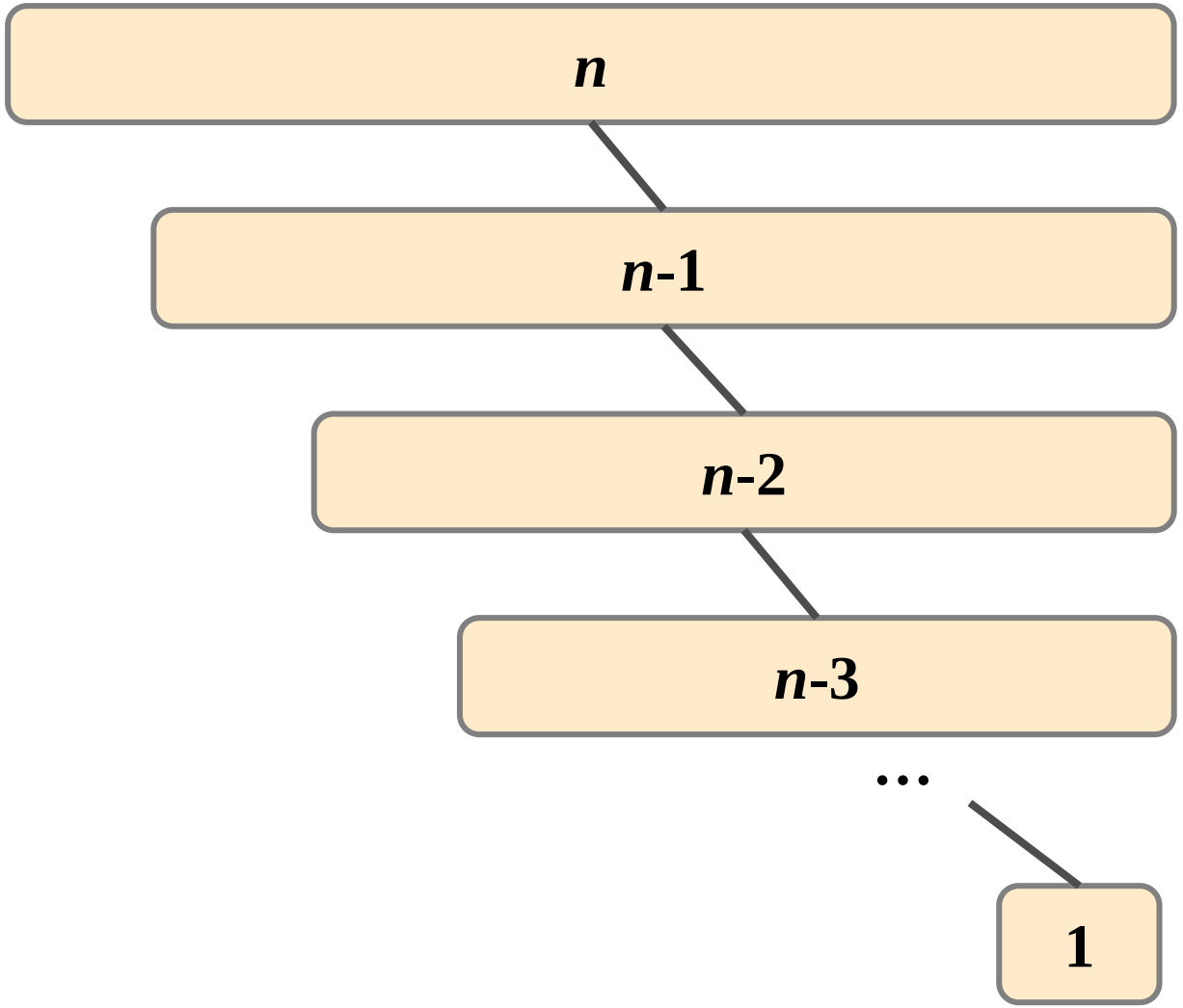
令$n=2^k$

$$
\begin{aligned}
T(n) &= n+\frac{n}{2}+\frac{n}{4}+\frac{n}{8}+\cdots+1 \\
&= n\left(1+\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\cdots+\frac{1}{n}\right) \\
&= n\left(1+\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\cdots+\frac{1}{2^k}\right) \\
&= n\left(2-\frac{1}{2^k}\right) \\
&\le 2n \\
&= O(n)
\end{aligned}
$$

时间复杂度 $O(n)$

最坏情况时间复杂度

C



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 $O(n^2)$

优化策略：随机选取基准元素。
降低最坏情况发生概率，无法杜绝

最坏时间为O(*n*)的选择算法

C

中位数的中位数（Median of Medians）算法，也称BFPRT算法。由如下5位学者提出：

Manuel Blum

图灵奖获得者

美国科学院院士

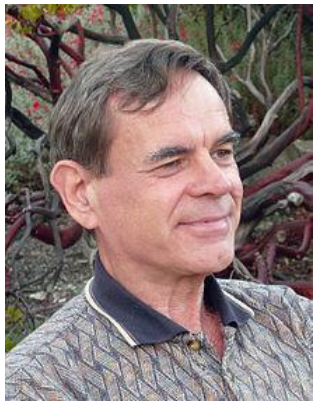
美国工程院院士

卡内基梅隆大学教授

Robert Floyd

图灵奖获得者

斯坦福大学教授



Vaughan Pratt

斯坦福大学教授

Ronald Rivest

图灵奖获得者

美国科学院院士

美国工程院院士

麻省理工学院教授

Robert Tarjan

图灵奖获得者

美国科学院院士

美国工程院院士

普林斯顿大学教授

Median of Medians算法

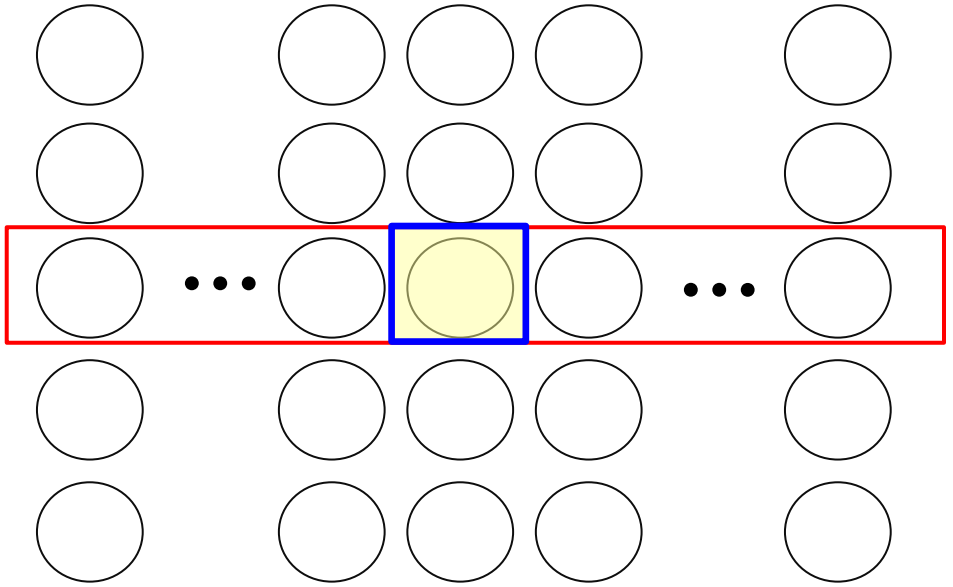
C

算法SELECT

（1）将数组的n个元素划分为$\lceil n/5 \rceil$组，每组5个元素（最后一组可能不足5个）；

（2）找出每一组的中位数：对每组元素进行插入排序，排序后第3小的元素即中位数。

（3）对上步找出的$\lceil n/5 \rceil$个中位数，递归调用SELECT找出其中位数。



合理选择基准元素，使分划不至于太不均衡。

Median of Medians算法

C

算法SELECT

（1）将数组的n个元素划分为$\lceil n/5 \rceil$组，每组5个元素（最后一组可能不足5个）；

（2）找出每一组的中位数：对每组元素进行插入排序，排序后第3小的元素即中位数。

（3）对上步找出的$\lceil n/5 \rceil$个中位数，递归调用SELECT找出其中位数。

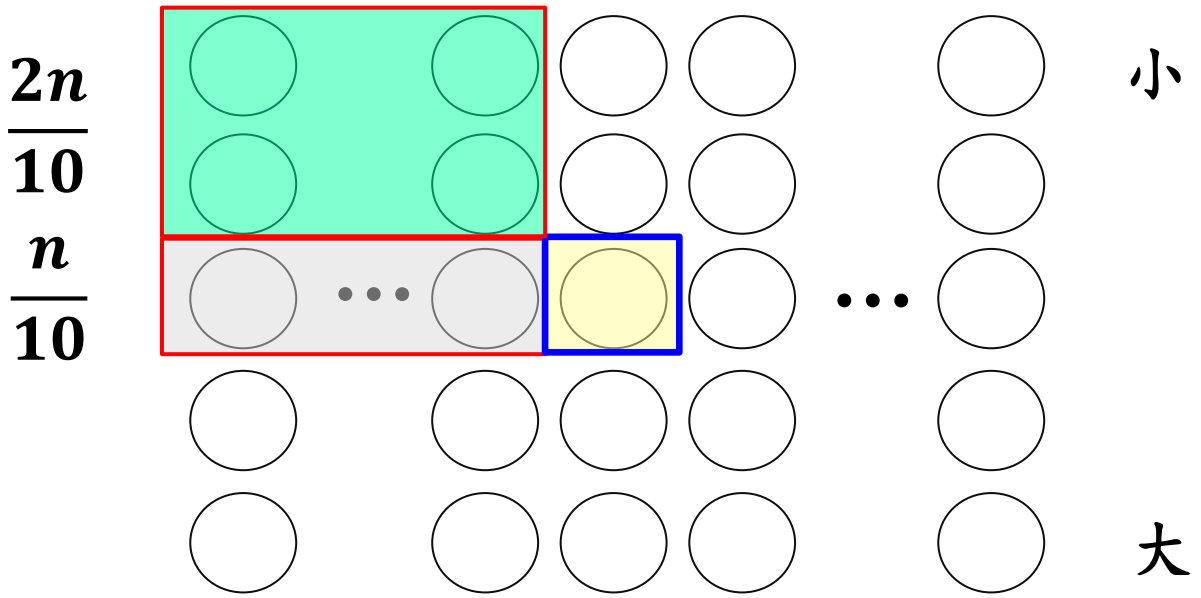
（4）将该元素作为基准元素，进行Partition，将数组划分为左右两部分，左部分元素个数为n_L。

（5）若$n_L=k-1$，算法结束；若n_L大于$k-1$，对左部分递归调用SELECT找第k小元素；若n_L小于$k-1$，对右部分递归调用SELECT找第$k-n_L-1$小元素。

最坏情况时间复杂度分析

C

从理论上保证分划不会过于不平衡，最差也能将数组分为3:7。



最坏情况时间复杂度分析

C

$T(n) \le T(0.2n)+ T(0.7n)+cn$

算法SELECT　　T(*n*)

（1）将数组的*n*个元素划分为$\lceil n/5 \rceil$组，每组5个元素（最后一组可能不足5个）；　　O(*n*)

（2）找出每一组的中位数：对每组元素进行插入排序，排序后第3小的元素即中位数。　　O(*n*)

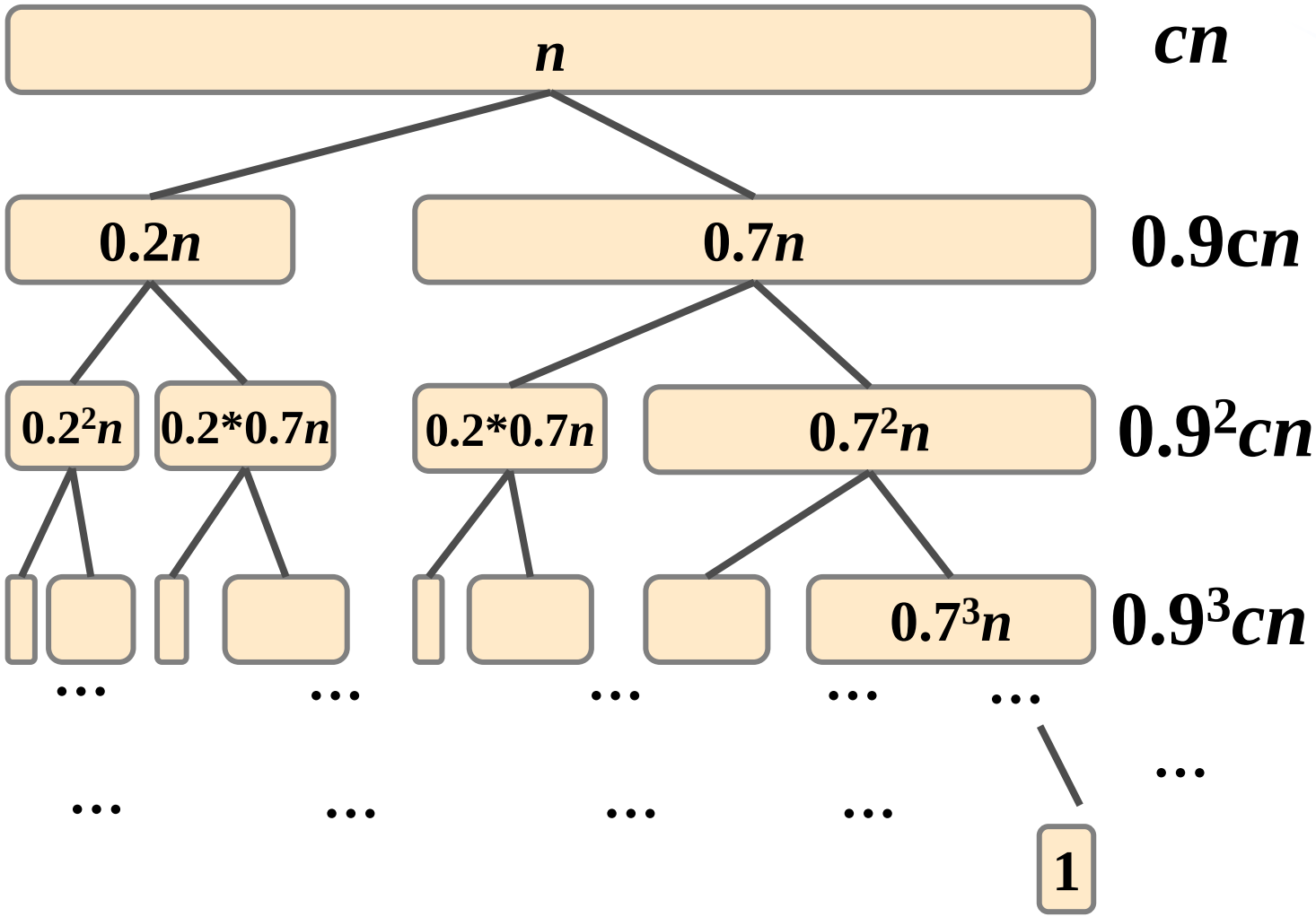
（3）对上步找出的$\lceil n/5 \rceil$个中位数，递归调用SELECT算法，找出其中位数。　　T(*n*/5)=T(0.2*n*)

（4）将该元素作为基准元素，进行Partition，将数组划分为左右两部分，左部分元素个数为n_L。　　O(*n*)

（5）若n_L=*k*-1，算法结束；若n_L大于*k*-1，对左部分递归调用SELECT找第*k*小元素；若n_L小于*k*-1，对右部分递归调用SELECT找第*k*-n_L-1小元素。　　T(7*n*/10) =T(0.7*n*)

最坏情况时间复杂度分析

C



$$T(n) \le T(0.2n) + T(0.7n) + cn$$

$$T(n) \le c(n + 0.9n + 0.9^2n + 0.9^3n + \ldots)$$

$$= \frac{cn(1 - 0.9^k)}{1 - 0.9}$$

$$= 10cn(1 - 0.9^k)$$

$$\le 10cn$$

$$= O(n)$$

快速排序的另一优化策略

C

- 快速排序算法中，可以利用BFPRT算法在最坏O(*n*)时间内，选取数组的中位数作为基准元素。使快速排序最坏情况时间复杂度降为O(*n*log*n*)。
- 但在实际应用中，由于该算法较耗时，复杂度的常数较高，很多情况下速度不如堆排序或随机选取基准元素的快速排序，故实际很少使用该策略。

将数组中k个最小的元素全部输出

C

- 找到第k小的元素B后，再扫描一遍数组，输出数组中小于等于B的元素即可，时间代价O(n)。
- 数组中k个最大的元素，同理。

Top K问题：快速选择 vs 堆

C

- 堆更适合处理增量数据
- 数组R有n个元素，假设已选出其中最大的k个元素，现对数组R再增加1个元素，如何找出当前数组中最大的k个元素？
  - ✓若采用快速选择，需要对长度为n+1的数组重新调用QuickSelect()，时间O(n)。
  - ✓若采用堆：只需将新增的元素适时插入堆中，并做相应调整，时间O(log$k$)。