

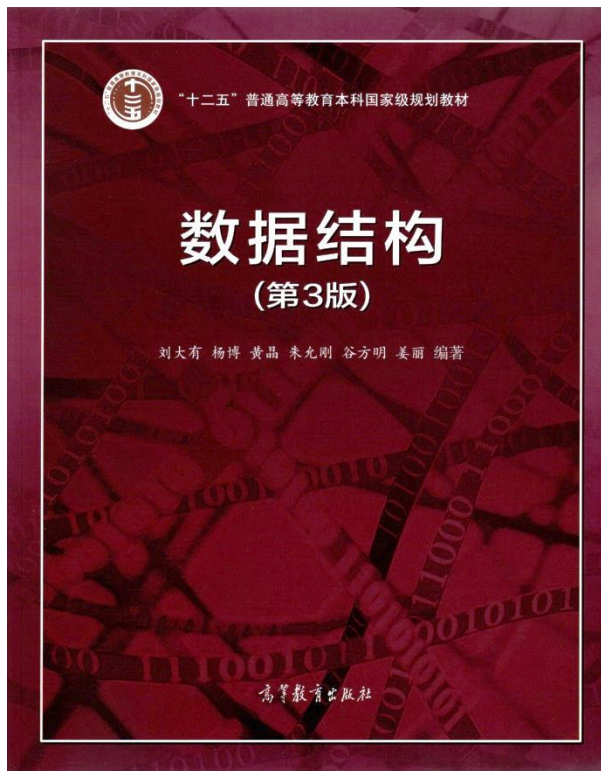


think.create.solve



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

快速排序算法提出者



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

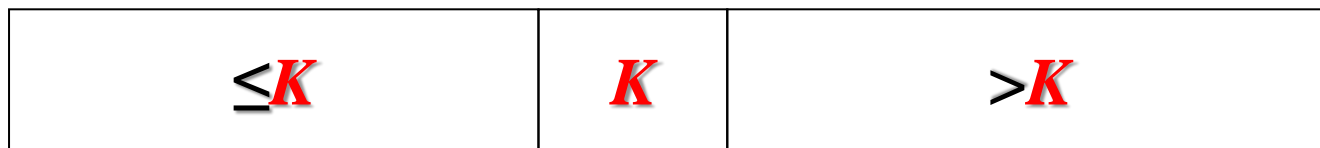
end quicksort

- 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。



快速排序基本思想

- 选取待排序数组中的某个元素 K （例如取第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。
- 分别对两个子数组重复上述方法，直到所有元素都排在相应位置上为止。



快速排序算法简要描述

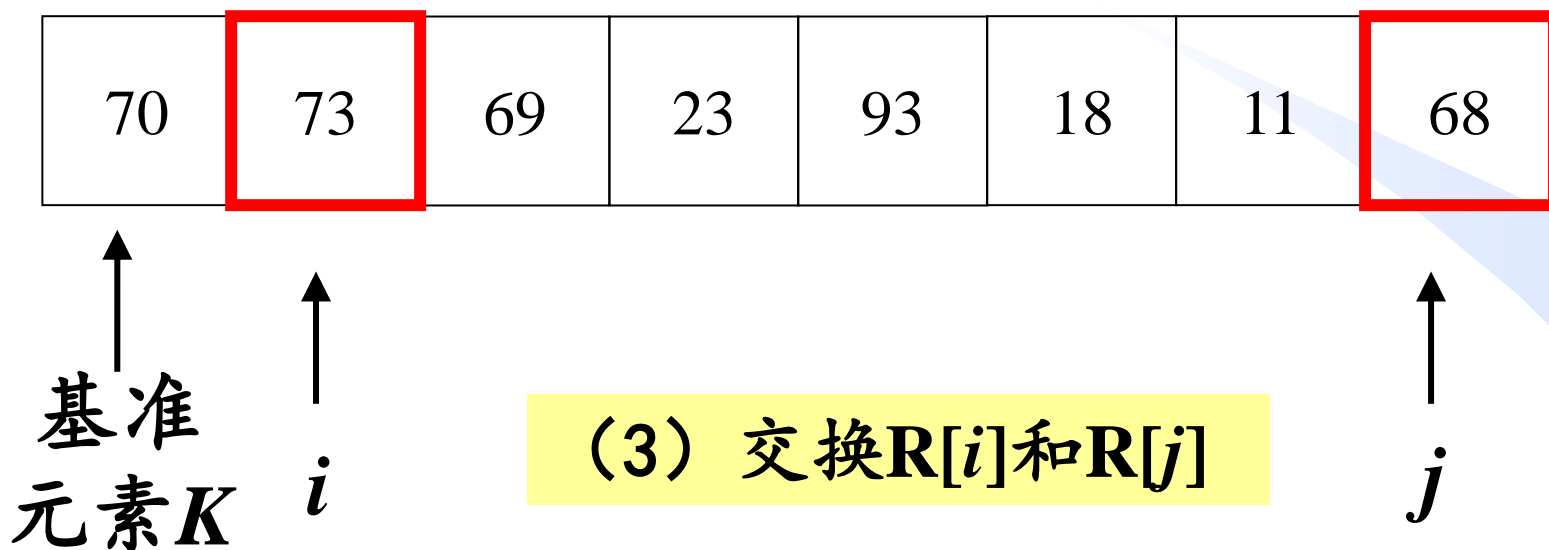
```
void QuickSort (int R[], int n) {  
    if (R的长度大于1) {  
        将R划分为两个子数组LeftR 和 RightR; // ①  
        QuickSort(LeftR); // ②  
        QuickSort(RightR); // ③  
    }  
}
```

操作①: Partition—分划、分割、划分

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。

分划操作算法思想



(1) 指针 i , 从左向右找第一个大于 K 的元素

(2) 指针 j , 从右向左找第一个小于等于 K 的元素

重复(1)-(3), 不断把小于 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 i 和 j 相遇



分划操作算法

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

时间复杂度 $O(n)$

```
void swap(int &m,int &n){
    int temp = m;
    m = n;
    n = temp;
}
```



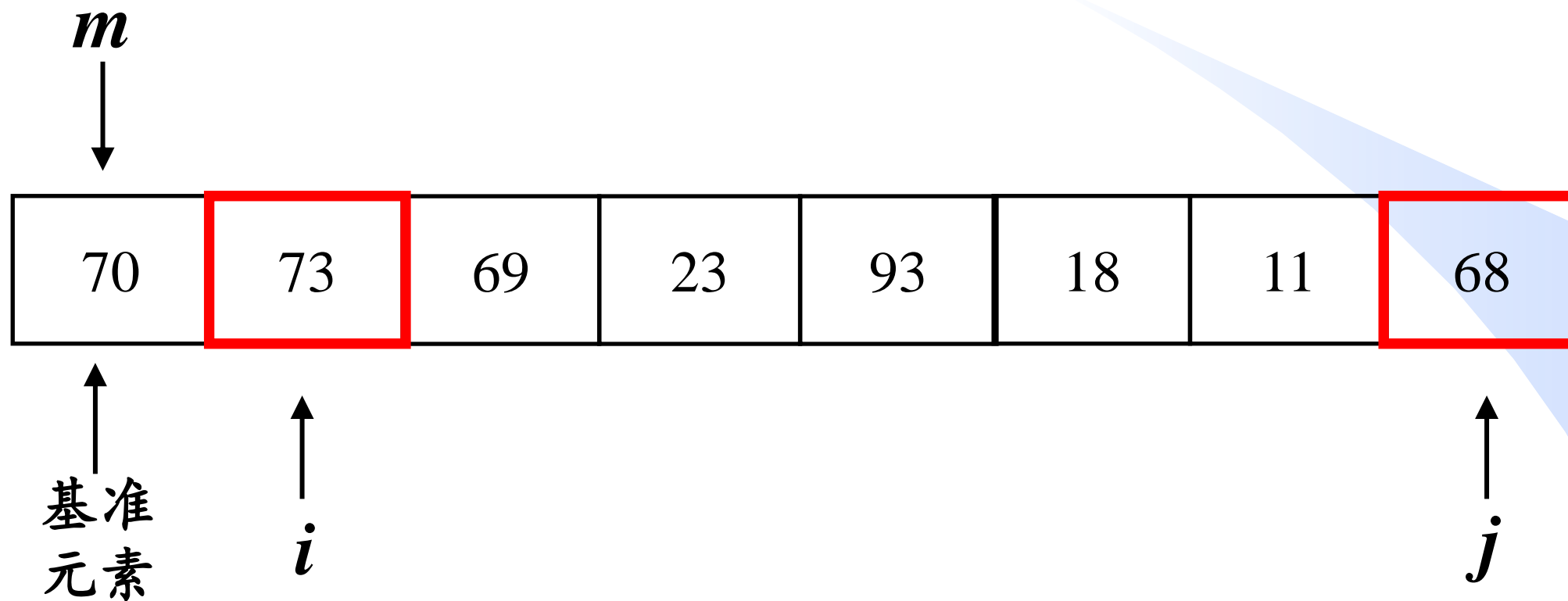
分划操作算法 (C/C++简化写法)

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

时间复杂度 $O(n)$

```
void swap(int &m,int &n){
    int temp = m;
    m = n;
    n = temp;
}
```


分划过程—例子

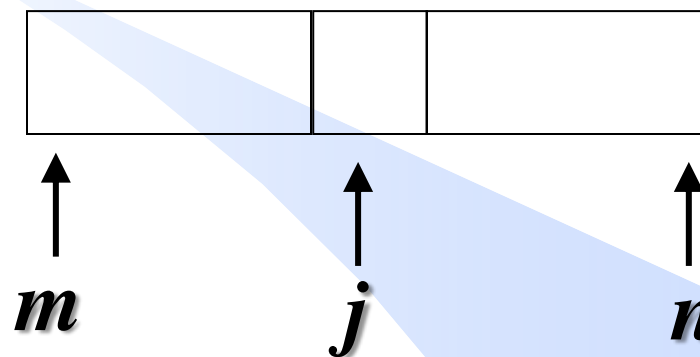


指针 i ，从左向右找第一个大于基准值的元素

指针 j ，从右向左找第一个小于等于基准值的元素

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



初始调用: QuickSort(R, 1, n)

分而治之，分治法

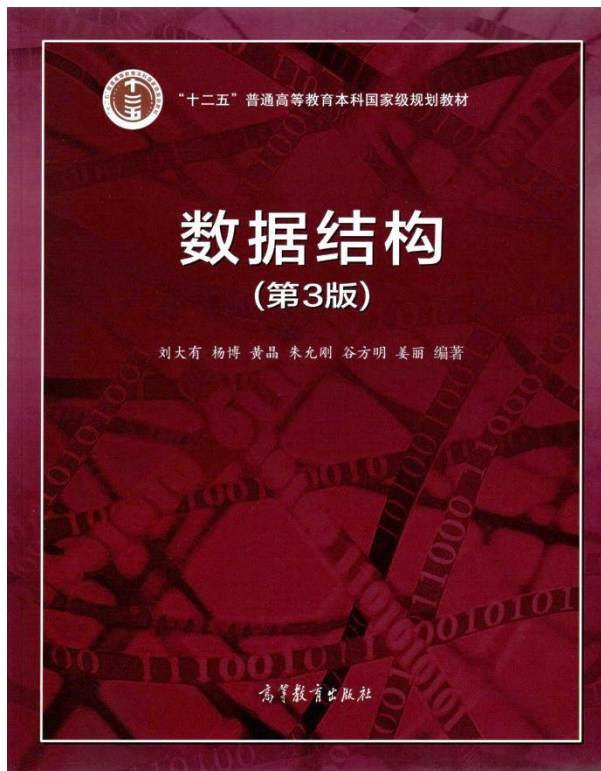


think.create.solve



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

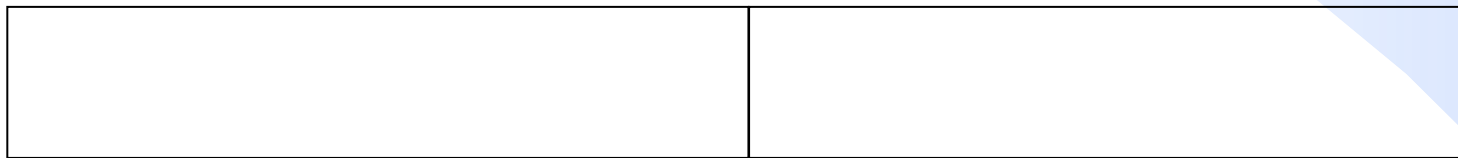


数据之法
结构之美
算法之道



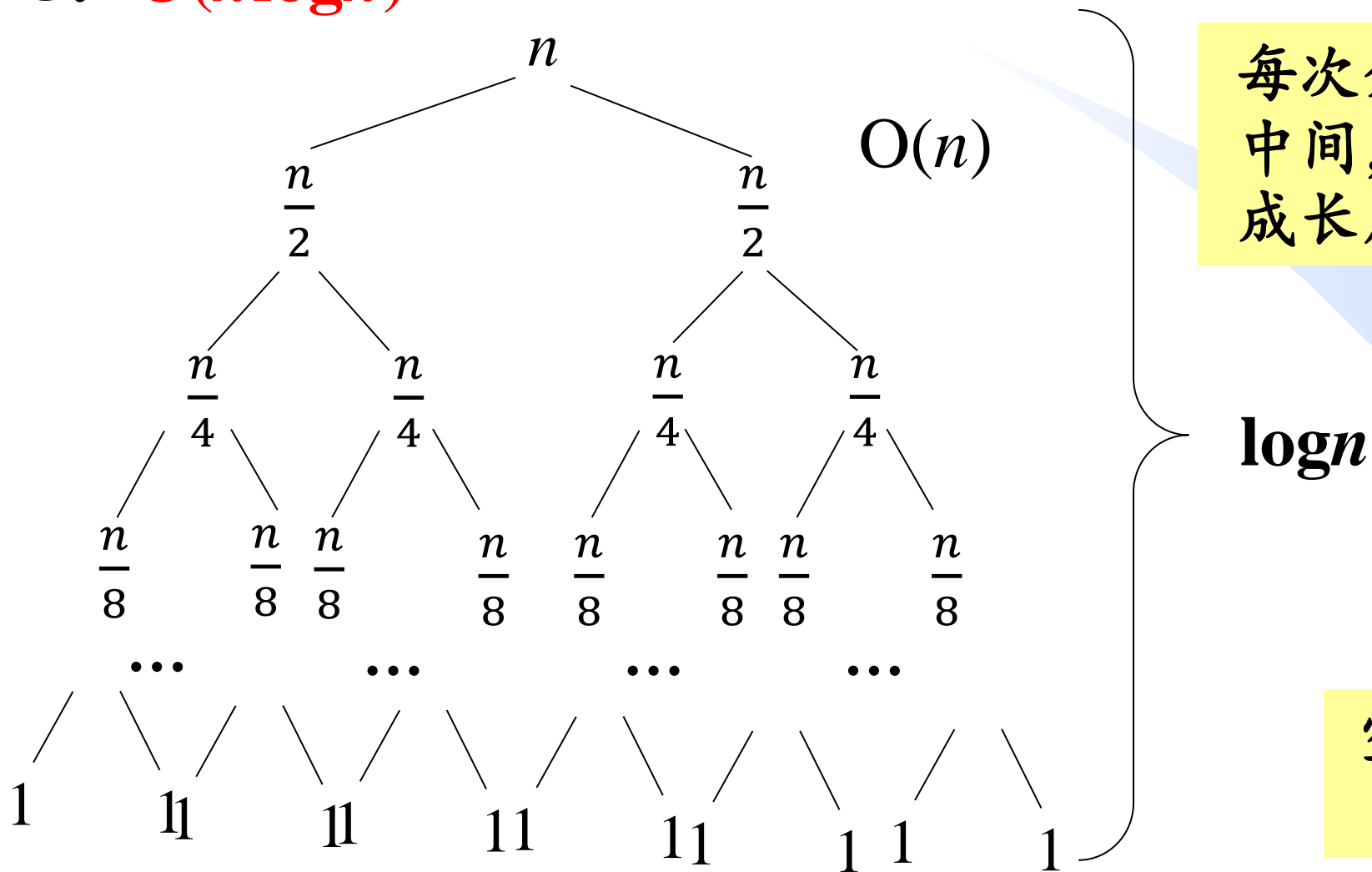
时间复杂度—最好情况

每次分划发生在最中间，把子数组分成长度相等的两半



时间复杂度—最好情况

最好情况: **$O(n \log n)$**

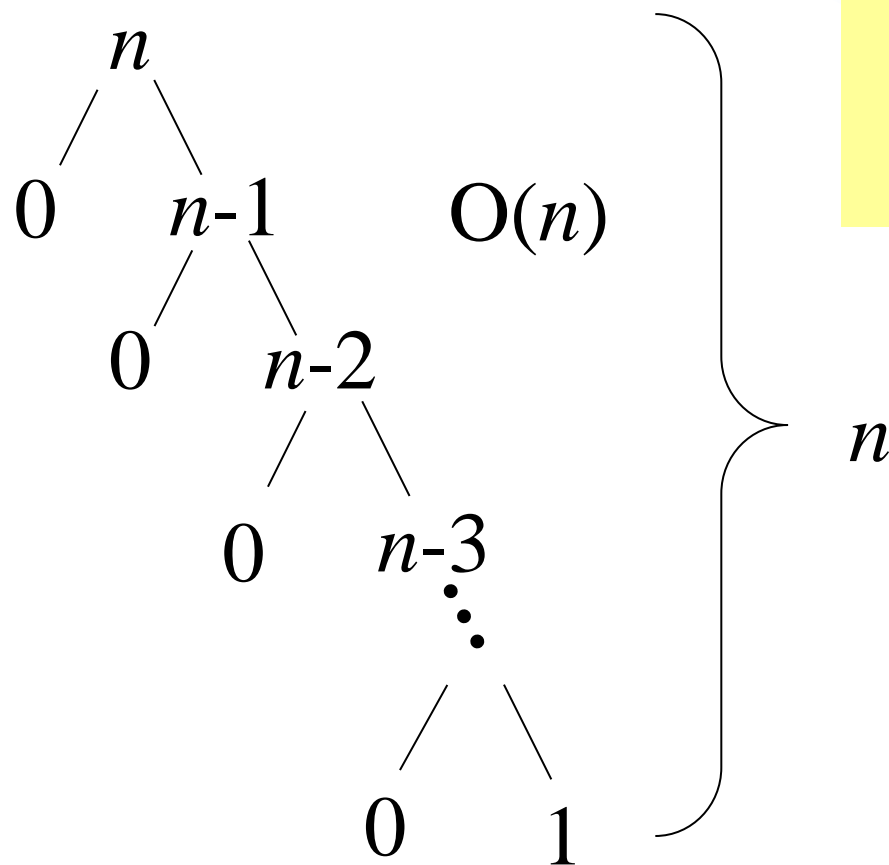


每次分划发生在最中间，把子数组分成长度相等的两半

空间复杂度
 $O(\log n)$

时间复杂度—最坏情况

最坏情况: $O(n^2)$



分划极不平衡，每次分划都发生在子数组最边上

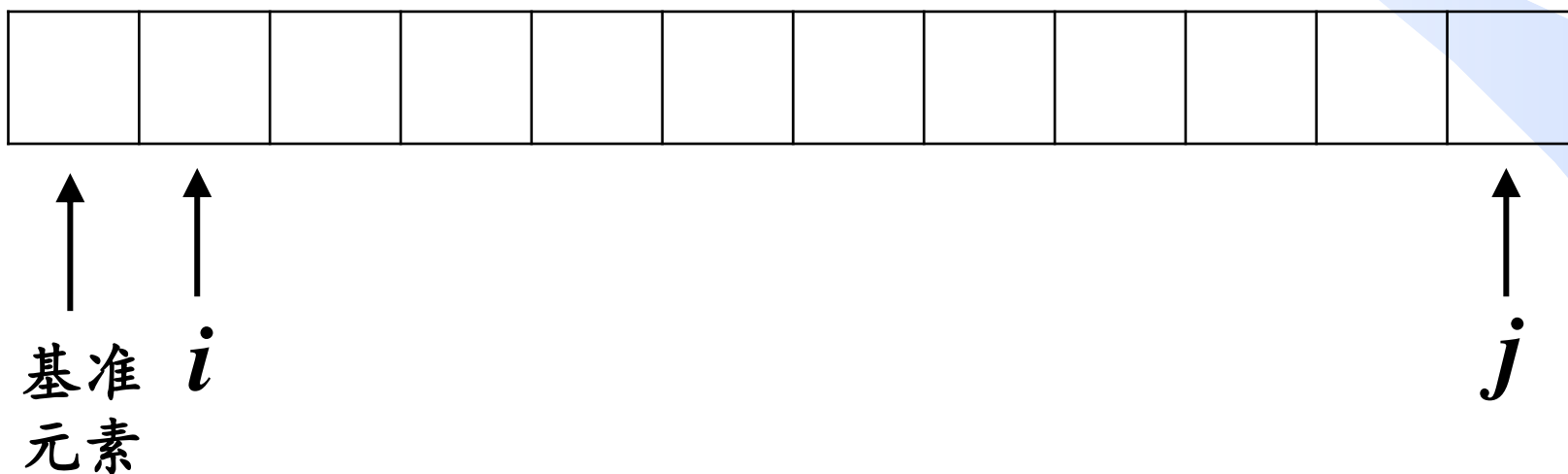
概率很低

空间复杂度
 $O(n)$

时间复杂度—平均情况

Partition操作关键词比较次数

最多 $n+1$ 次



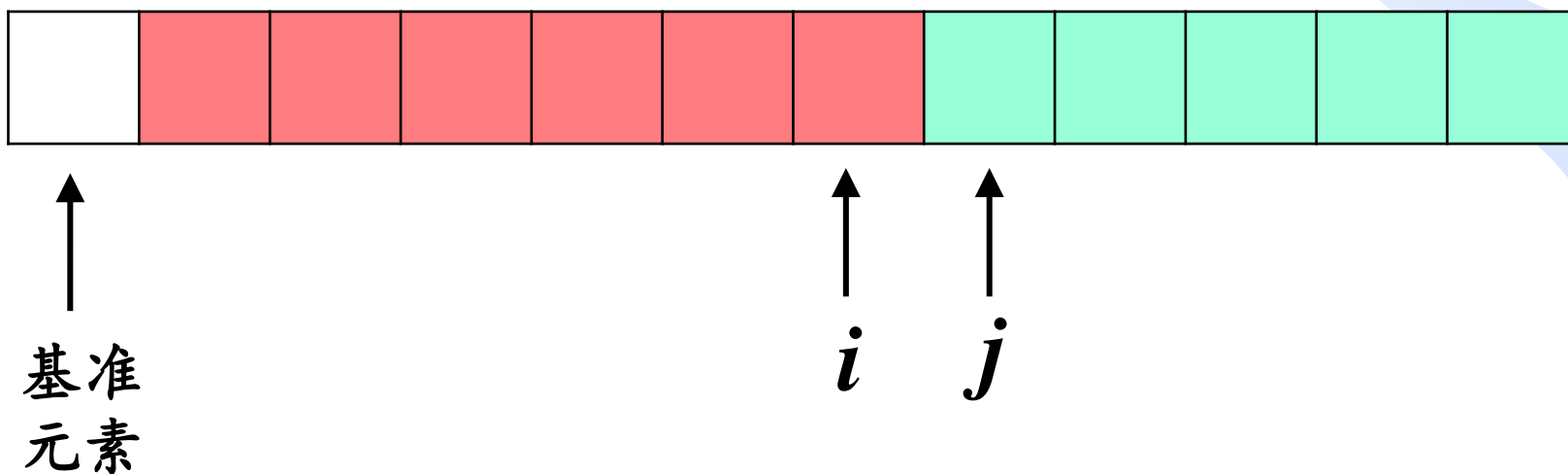
指针 i ，从左向右找第一个大于基准值的元素

指针 j ，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

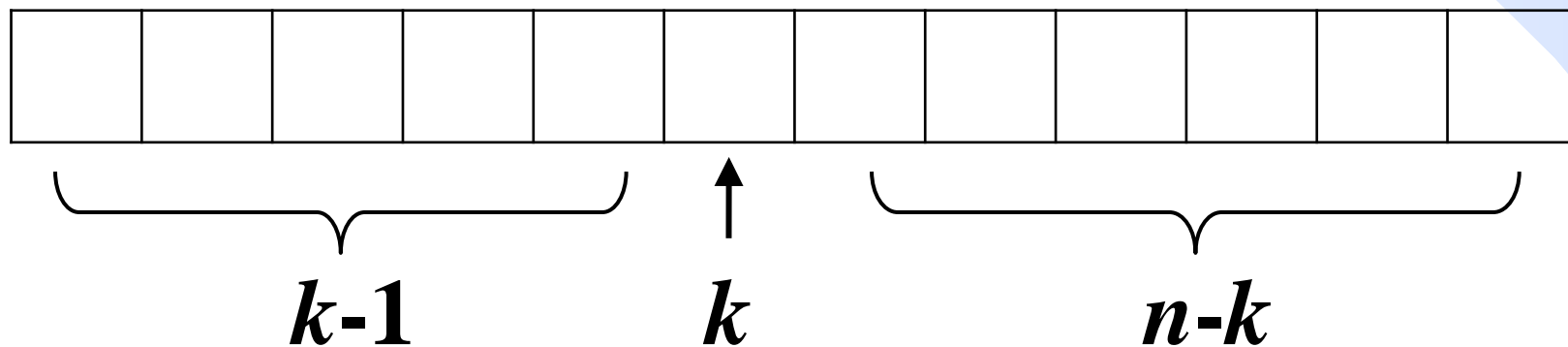


指针 i ，从左向右找第一个大于基准值的元素

指针 j ，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

- 用 $T(n)$ 表示算法 $\text{QuickSort}(R, 1, n)$ 的平均复杂度，并假定与 R 中元素随机分布。
- $T(0) = T(1) = 0$
- 共 n 种可能输入，第 k 种输入使 Partition 操作返回值为 k



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

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 :

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

对上式，用 $n-1$ 替换 n :

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

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

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

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

等式两边除以 $n(n+1)$ ：

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

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

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

$$\begin{aligned} S_n - S_{n-1} &= \frac{2}{n+1} \\ S_{n-1} - S_{n-2} &= \frac{2}{n} \\ S_{n-2} - S_{n-3} &= \frac{2}{n-1} \\ &\vdots \\ S_2 - S_1 &= \frac{2}{3} \end{aligned}$$



时间复杂度—平均情况

$$\begin{aligned}T(n) &= O(2n \ln n) \\&= O(2n \ln 2 \log_2 n) \\&= O(1.386n \log_2 n) \\&= O(n \log n)\end{aligned}$$



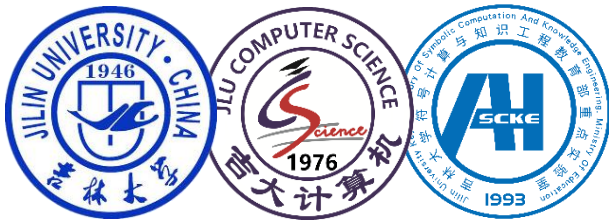
空间复杂度

- 取决于递归所需的系统栈空间（递归深度）。
- 最好和平均情况： $O(\log n)$ 。
- 最坏情况： $O(n)$ 。



稳定性

- 快速排序是不稳定的排序算法。
- 课下验证：3, 3*, 2 → 2, 3*, 3。

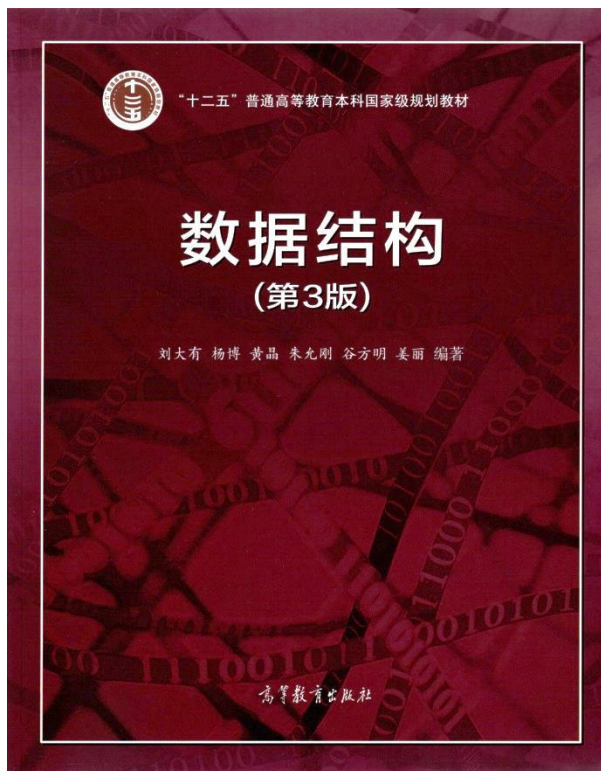


think.create.solve



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



(1) 小数据量采用插入排序

- 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。
- 递归过程中，当前处理的子数组长度小于等于某个阈值（一般取16左右）时：
 - ✓ 策略1：可直接对当前子数组插入排序。
 - ✓ 策略2：对当前子数组什么也不做，直接退出本层递归。
待最后快速排序所有递归都退出后，再对整个大数组统一进行一次插入排序（插入排序在面对“接近有序”的序列时拥有良好的性能）。

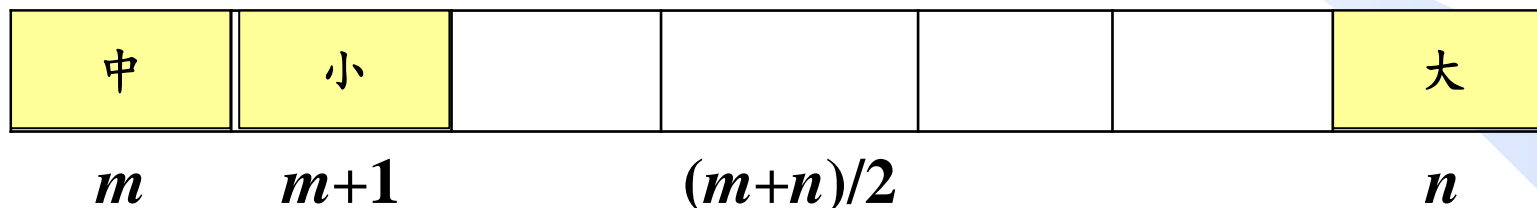


(2) 随机选取基准元素

- 快速排序达最坏情况的一种原因：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：
 $k \leftarrow \text{rand}(m, n)$. // $\text{rand}(m, n)$ 返回 m 到 n 间的随机数。
 $R_m \leftrightarrow R_k$.
- 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，该策略可获得很好的性能。

(3) 三数取中(Median of Three)选基准元素

- 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
```

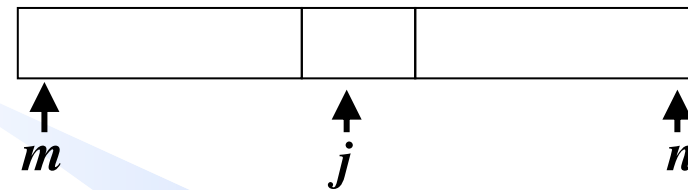
```
if(R[m+1] > R[m]) swap(R[m+1], R[m]); //  $R_{m+1}$  和  $R_m$  的较大者移到  $R_m$  处
```

- 降低最坏情况发生的概率，但无法杜绝。

(4) 尾递归优化I

将尾递归转为循环，减小递归深度，降低空间复杂度

```
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);
    }
}
```



左侧区间递归处理
右侧区间循环处理

该形式意义不大
最坏情况空间仍为 $O(n)$

每次分划都分到最右边
使左侧区间长度 $n, n-1, n-2...$

```
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;
    }
}
```

(4) 尾递归优化II

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){
```

短区间递归处理
长区间循环处理

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

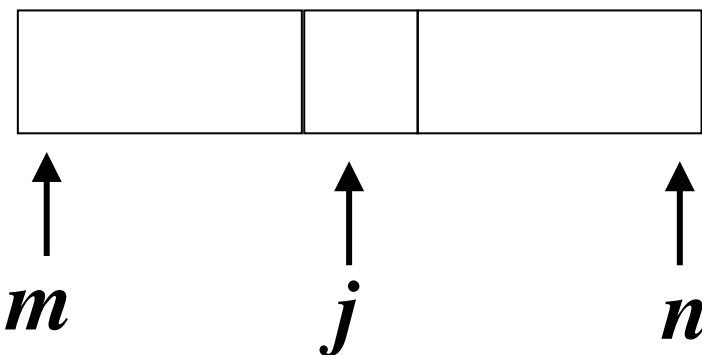
```
        int j=Partition(R,m,n);
```

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

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

```
    }
```

```
}
```



每次递归处理的子数组
长度至少缩减一半

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？

(5) 利用栈消除所有递归

栈内元素：二元组，标识当前处理的区间的左右边界

```
while(m < n){ //m和n为当前处理子数组的起止下标
```

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

```
    int leftLen=j-m, rightLen=n-j;
```

短区间迭代处理，长区间压栈

```
    if(leftLen<M && rightLen>=M) m = j+1;
```

//处理右区间

```
    else if(leftLen>=M && rightLen<M) n = j-1; //处理左区间
```

```
    else if(leftLen>=M && rightLen>=M){ //先处理短区间，长区间压栈
```

```
        if(leftLen>rightLen) { S.PUSH(m,j-1); m = j+1; }
```

```
        else { S.PUSH(j+1, n); n = j-1; }
```

```
    }
```

```
    else if(leftLen<M && rightLen<M){
```

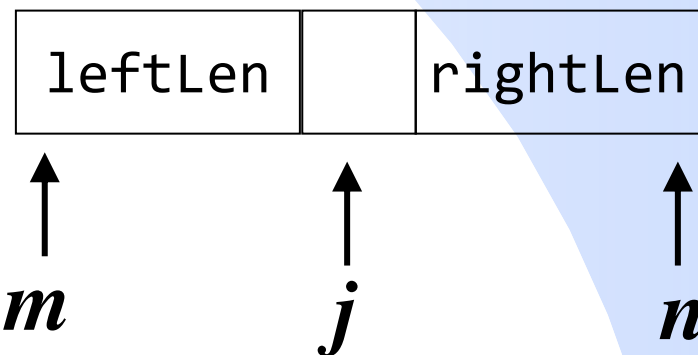
```
        if(!S.Empty()) S.POP(m,n);
```

```
        else break;
```

```
    }
```

```
}
```

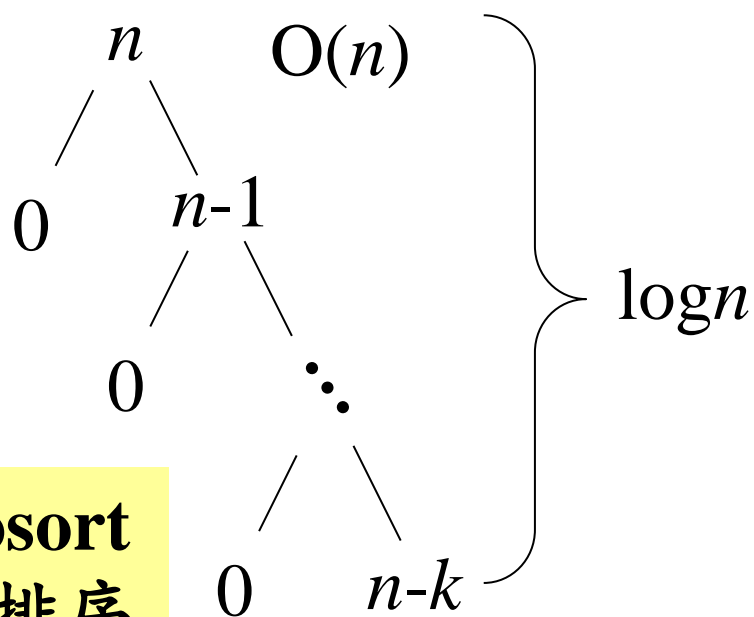
```
InsertSort(R,m,n);
```



当区间长度小于阈值M时，不处理该区间，待最后对整个数组做一次插入排序

(6) 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 $O(n \log n)$!



Introsort
内省排序



David R. Musser

THE C++
STANDARD
TEMPLATE
LIBRARY

GO



Microsoft®
.NET
Framework

(7) 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
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

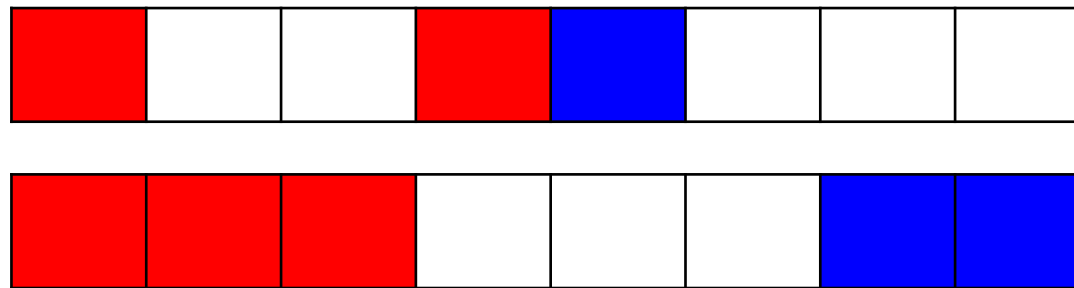
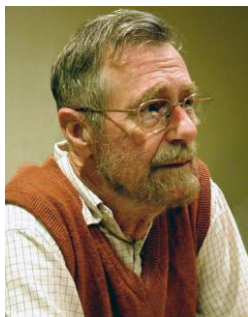
- 修改Partition操作，将当前数组划分为三部分：小于基准元素 K 的元素放在左边，等于 K 的元素放在中间，大于 K 的元素在右边。
- 进一步递归时，仅对小于 K 的左半部分子数组和大于 K 的右半部分子数组进行递归排序。

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

荷兰国旗问题 (Dijkstra提出)



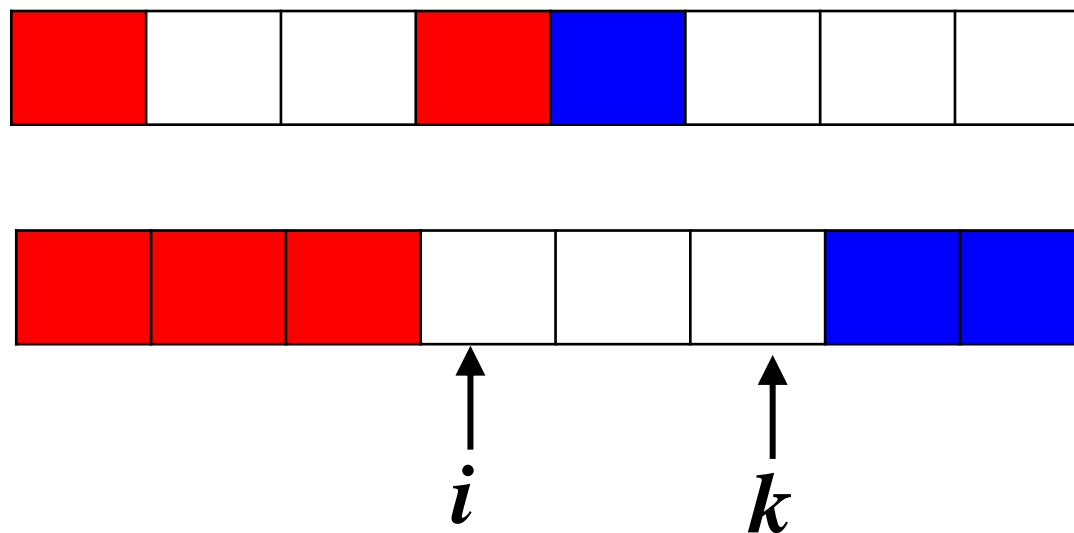
给定 n 粒石头，每粒石头的颜色是红，白，蓝之一。要求重新排列这些石头，使得所有**红色石头在前**，所有白色石头居中，所有**蓝色石头居后**。【腾讯、华为、小米、字节跳动、苹果、微软、谷歌、滴滴、快手面试题，浙大考研题】

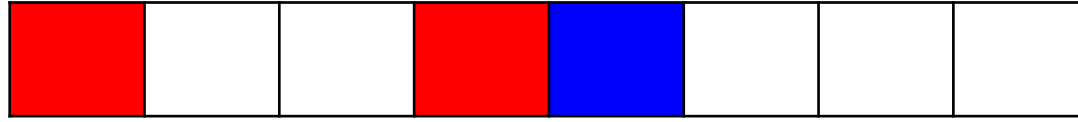


- 设置3个指针，前指针 i ，中指针 j ，后指针 k ；
- 初始时 $i=1, j=1, k=n$ ；
- 通过指针 j 遍历数组。

三路分划

- 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**交换到数组左部，**蓝石头**交换到数组右部。





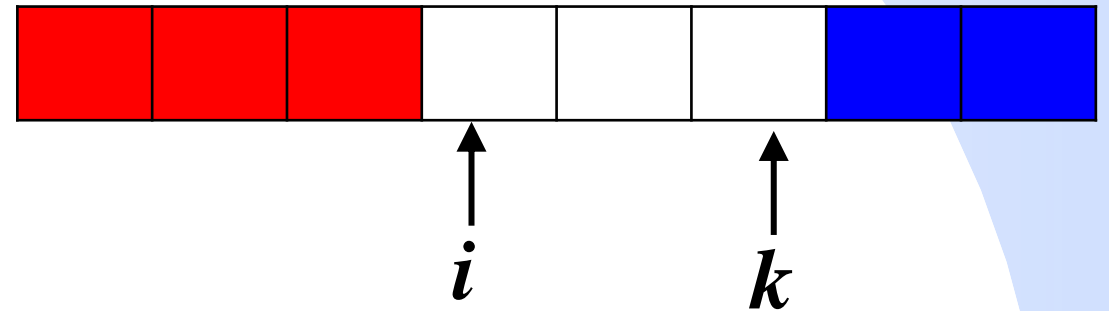
```
while (j<=k){  
    if (R[j]=='红'){  
        swap(R[j], R[i]);  
        j++; i++;  
    }  
    else if (R[j]=='蓝'){  
        swap(R[j], R[k]);  
        k--;  
    }  
    else // R[j]=='白'  
        j++;  
}
```

此处为什么
不j++？





```
void QuickSort3way(int R[], int m, int n){  
    if(m>=n) return;  
    int 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 // R[j]==pivot  
            j++;  
    }  
    QuickSort3way(R, m, i-1);  
    QuickSort3way(R, k+1, n);  
}
```





课下练习

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】。



课下练习

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；
- 三路分划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：

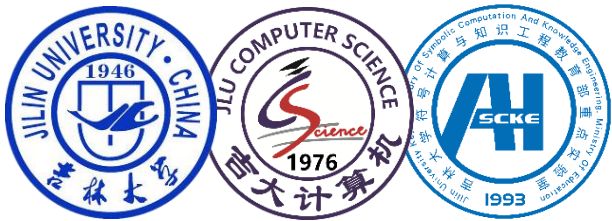
- 子数组长度小于等于16时，直接退出本层递归。待最后快速排序所有递归都退出后，再对整个大数组进行**一次**插入排序；
- 当递归深度达到 $\lfloor 2\log_2 n \rfloor = O(\log n)$ 时，强行停止递归，转而对当前子数组进行堆排序；
- 三数取中选基准元素；
- 尾递归优化，先处理短区间；
- 最坏情况时间复杂度 $O(n\log n)$ 。





快速排序总结

- 最好/平均时间复杂度： $O(n\log n)$
- 最坏时间复杂度： $O(n^2)$
- 平均空间复杂度： $O(\log n)$
- 稳定性：快速排序是不稳定的排序方法。
- 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
 - ✓ CPU Cache

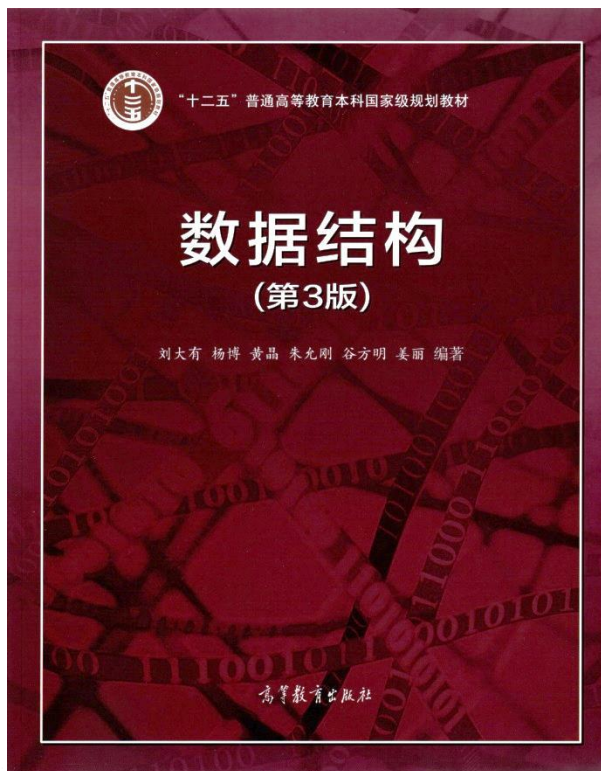


think.create.solve



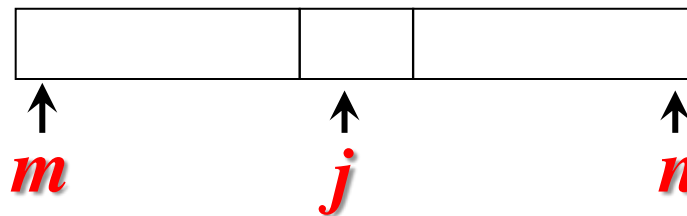
快速排序

- 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

快速选择问题

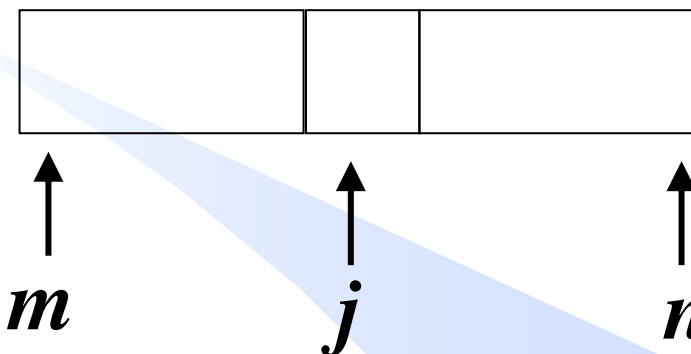


给定包含 n 个元素的整型数组，编写时间复杂度为 $O(n)$ 的算法，求得数组中第 k ($k < n$)小的数。【腾讯、字节跳动、阿里、美团、京东、滴滴、网易、小米、华为、旷视科技、拼多多、苹果、微软、百度、谷歌、快手面试题】

- 基于快速排序的Partition算法，将数组分成两部分。
- 若左部分元素个数等于 $k-1$ ，则基准元素即为第 k 小的元素。
- 若左部分元素个数大于 $k-1$ ，第 k 小的元素必在左部分，对左部分递归找第 k 小的元素。
- 若左部分元素个数小于 $k-1$ ，第 k 小的元素必在右部分，若左部分包含元素个数 n_L ，则对右部分递归找第 $k - n_L - 1$ 小的元素。

快速选择问题

```
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);  
}
```

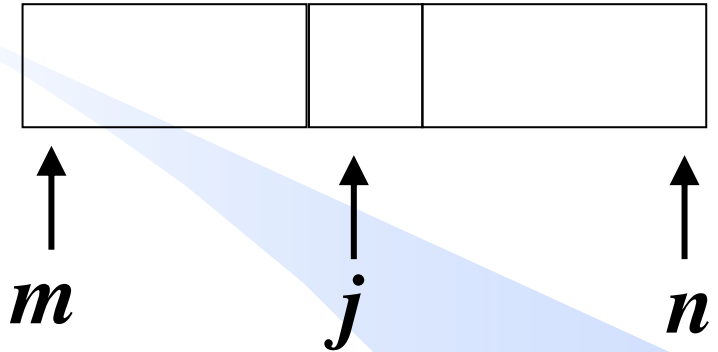


与快速排序不同在于，只需递归处理左右两个子数组中的一个

减而治之，减治法

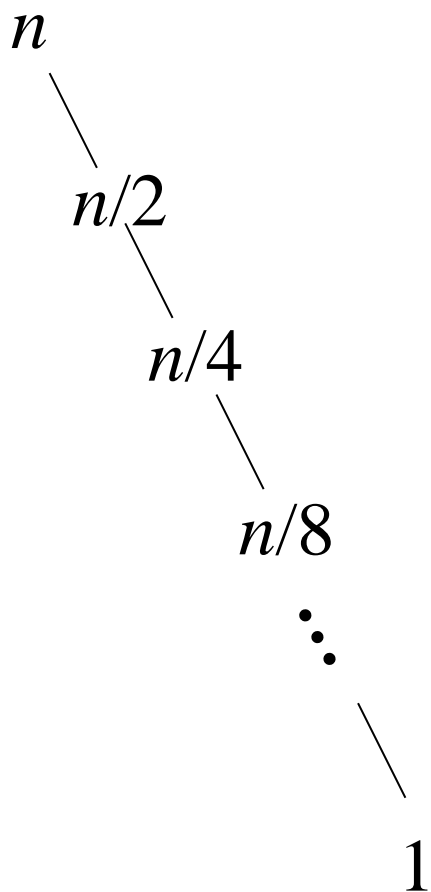
快速选择问题——迭代形式

```
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).

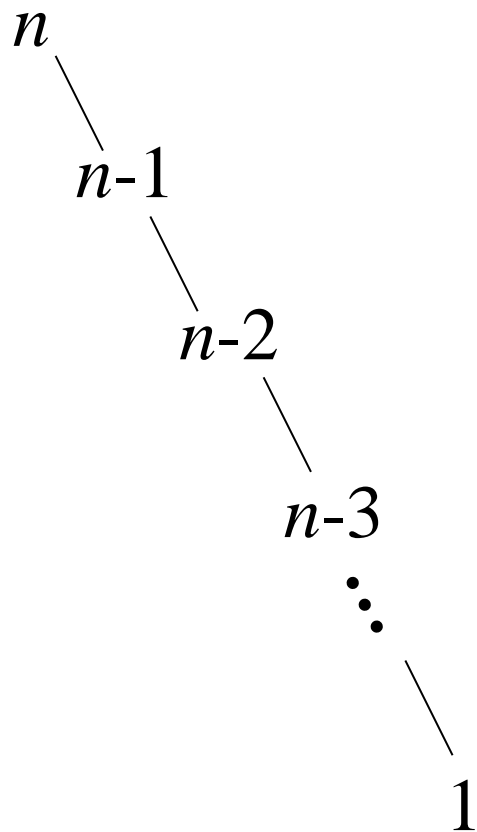
最好/平均情况时间复杂度



令 $n=2^k$

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

最坏情况时间复杂度



每次分划都分在最边上，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

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

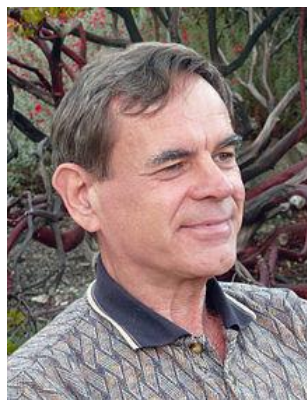
卡内基梅隆大学教授



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

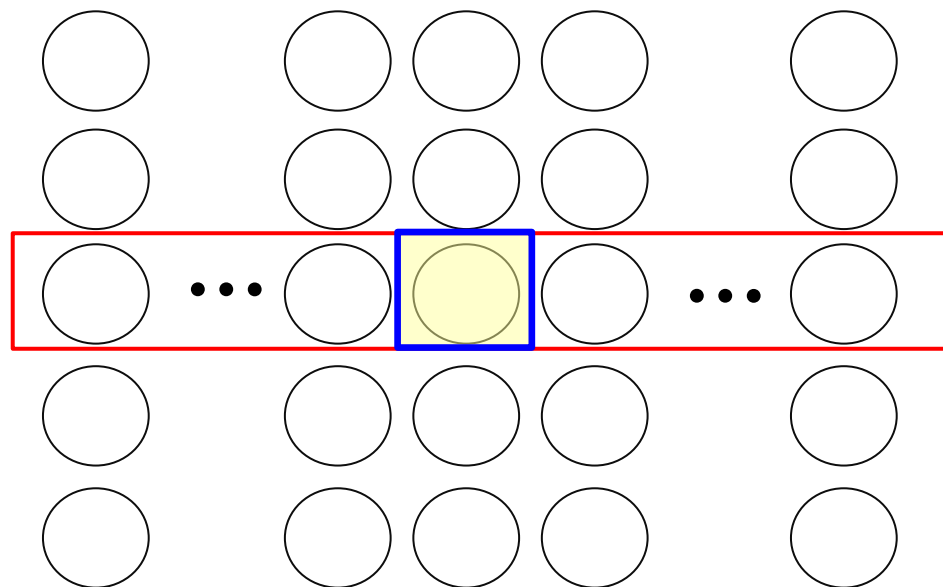
美国工程院院士

普林斯顿大学教授

Median of Medians 算法

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。



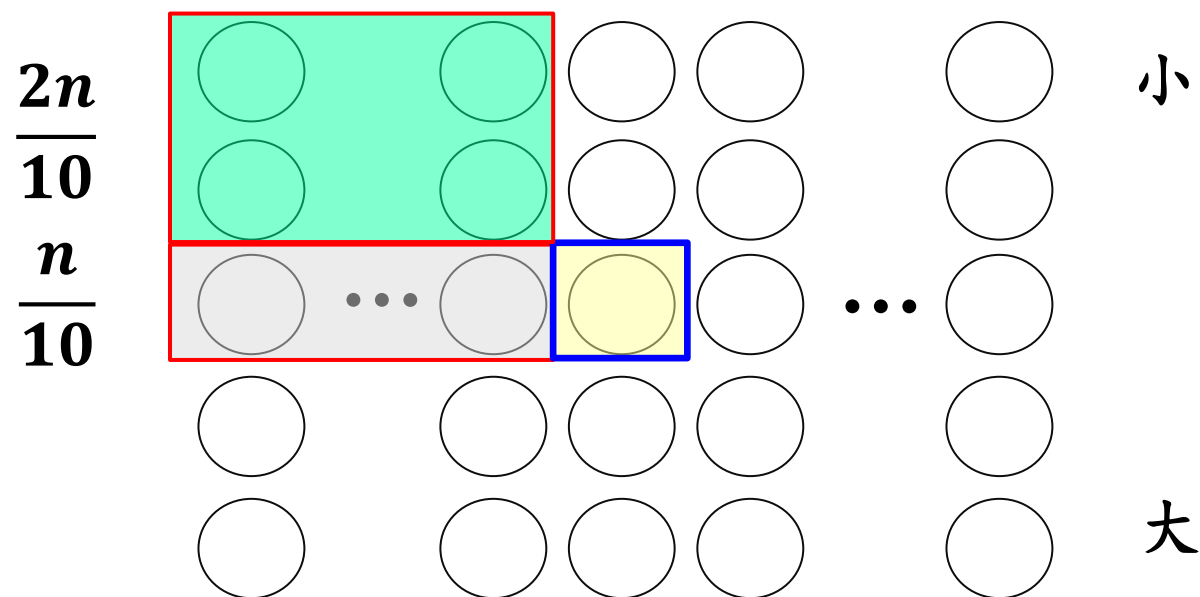
Median of Medians 算法

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

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

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。
 $T(n/5)$

(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。
 $T(7n/10)$



最坏情况时间复杂度分析

$$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + cn$$

$$T(n) \leq 10cn = O(n)$$



快速排序的另一优化策略

- 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

- 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：快速选择与堆方法的比较

数组R有 n 个元素，挑选其中最大的 k 个元素。

堆的优势在于适合处理动态**增量数据**，例如：对数组R再增加1个元素后，找出当前数组中最大的 k 个元素。

- ✓快速选择：需要重新调用QuickSelect(), 时间 $O(n)$ 。

- ✓包含 k 个元素的最小堆：新增的元素若大于堆顶元素，则替换堆顶元素并调整堆，时间只需 $O(\log k)$ 。



练习

- 编写一个时间复杂度为 $O(n)$ 的算法，对给定的整数数组进行排列，使得所有正整数均出现在负整数和零之前。
- 例如：原数组为-1、4、-3、0、2、1、-9、7，则排列后为4、2、1、7、0、-1、-3、-9。

```
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++;    //从左往右找小于等于0的数
        while(j>=1 && a[j]<=0) j--;    //从右往右找大于0的数
        if(i<j) swap(a[i],a[j]);
    }
}
```




练习

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为 $O(n)$ 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】

练习

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】

- 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 $A[j]$



```
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;  
        }  
    }  
}
```