排序 双指针

计1903 马辰

先复习一下最简单的排序（8min）

- 冒泡排序（2.5min）
- 选择排序（2.5min）
- 插入排序(介绍) （3min）

优 ？
劣 ？

选择排序

- 算法分析：
- 按降序排列
- 若一组整数放在数组a[0],a [1],……,a[n-1]中，
- 第一趟在a[0]～a[n-1]找出最大值，放在a[0];
- 第二趟在a[1]～a[n-1]找出最大值，放在a[1];
- 依此类推，直到从a[n-2]和a[n-1]之中找最大值

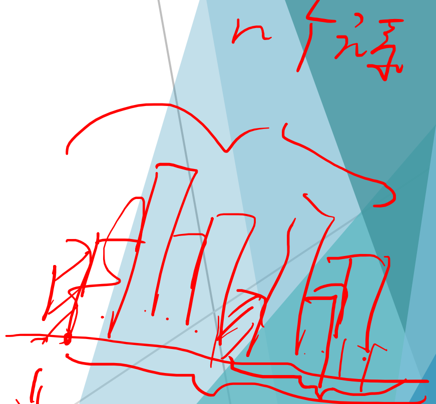
```
for ( int i = 0 ;  i < n-1 ;  i ++ ) {
t = i ;
// i当前元素，t中间变量，存储本轮查找过程最小元素的秩
    for ( j = i + 1 ; j < n ;  j ++ )
        if  ( a[ j ] > a [ t ] ) t = j ;
// 寻找当前元素i至序列末尾中最小者的秩，用t来存储
    swap( a [ t ] ,a [ i ] );
//交换两个元素
  }
```

复杂度?

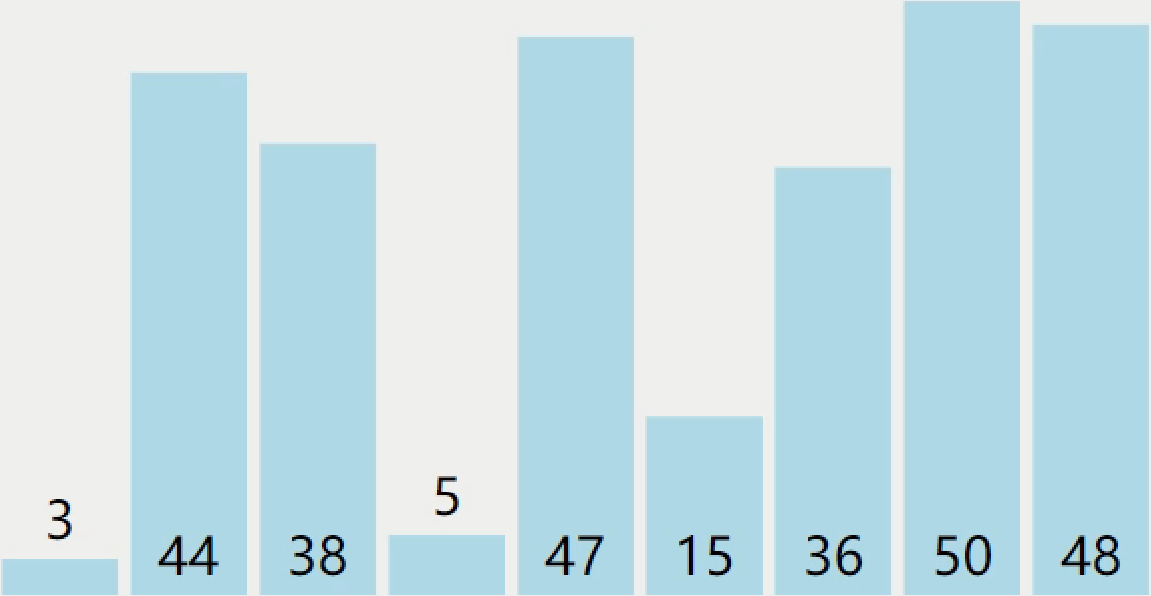
复杂度. $O(n^2)$

⇒ 正比于所执行的基本操作次数.

计算；时间



时间



选择排序

重复（元素个数-1）次

　把第一个没有排序过的元素设置为最小值

　遍历每个没有排序过的元素

　　如果元素 < 现在的最小值

　　　将此元素设置成为新的最小值

　将最小值和第一个没有排序过的位置交换

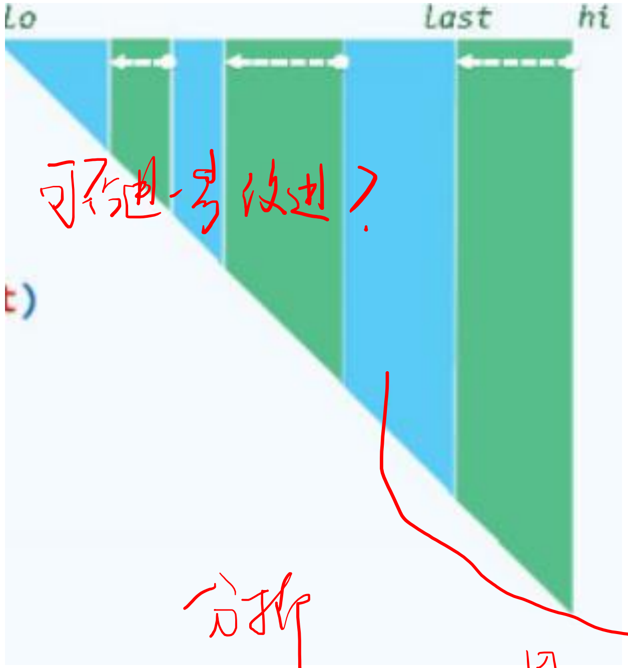
创建

排序

执行

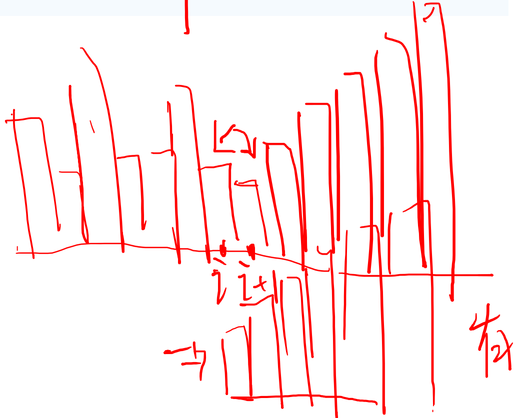
冒泡排序

采用相邻元素比较的方法



可行进一步改进？

分析

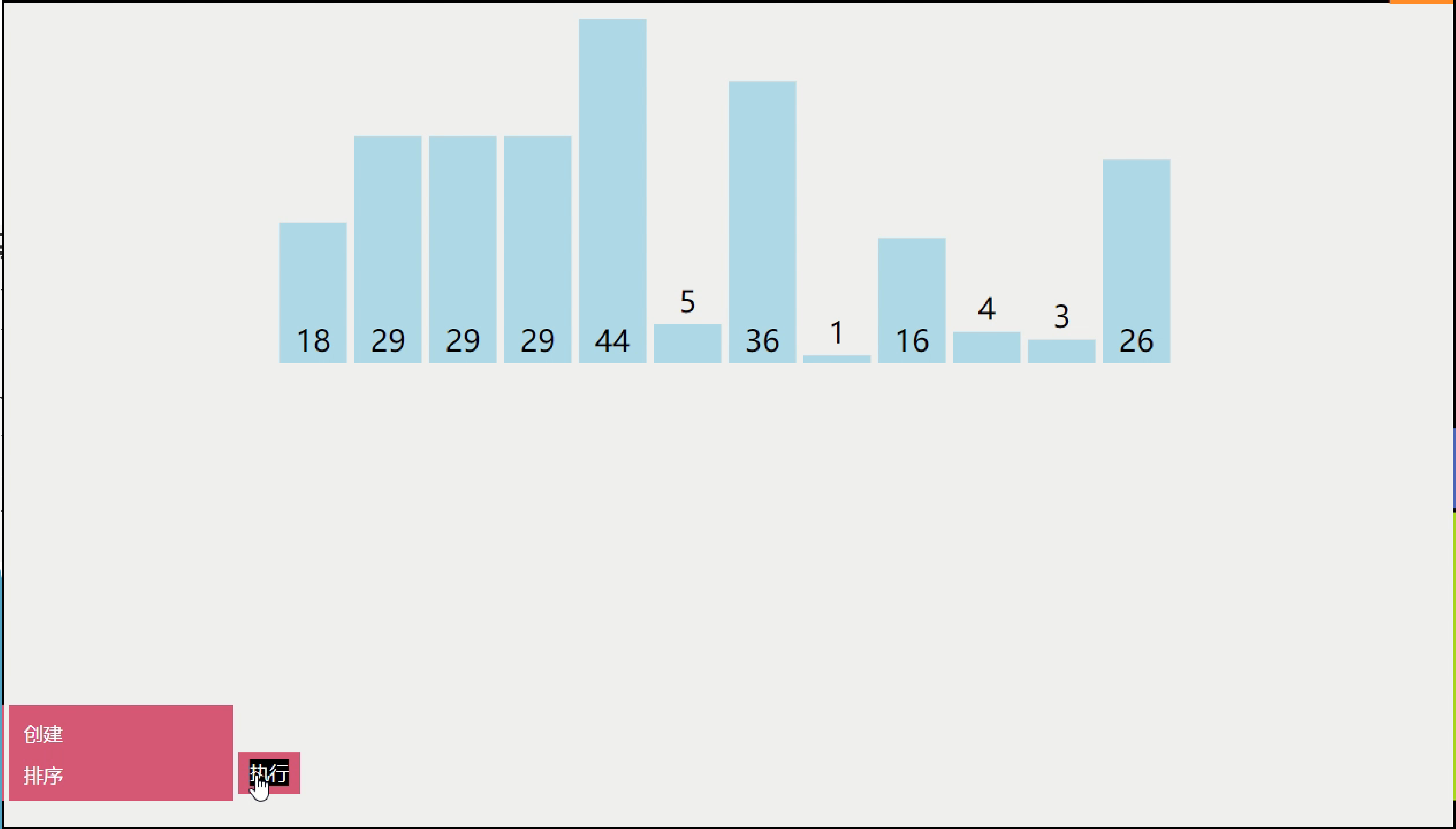


```
void bubble ( int a[] , int size )
{ int i , temp , work ;
   for ( int pass = 1 ;  pass < size ;  pass + + )    //对数组排
     { work = 0 ;//辅助变量 监视实际的排序过程是否还在执行
        for ( i = 0 ;  i < size-pass ;  i + + )
        if ( a [ i ] > a [ i + 1 ] )   temp //相邻元素比较
          {swap( a [ i ] ,a [ i + 1 ])
             work = 1 ;
          }          temp = i ;
        if ( !  work )  break  ;//如果实际未发生交换，则停止
   }
}
```

有待改进. 优化，节省时间.

复杂度仍为 O(n)

最好情况


18
29
29
29
44
5
36
1
16
4
3
26
创建
排序
执行

原理：通过构建有序序列，对未排序的数据，在已排序序列中从后向前扫描，找到相应位置并插入。

一般步骤：

1.从第一个元素开始，该元素**可以认为已被排序；**

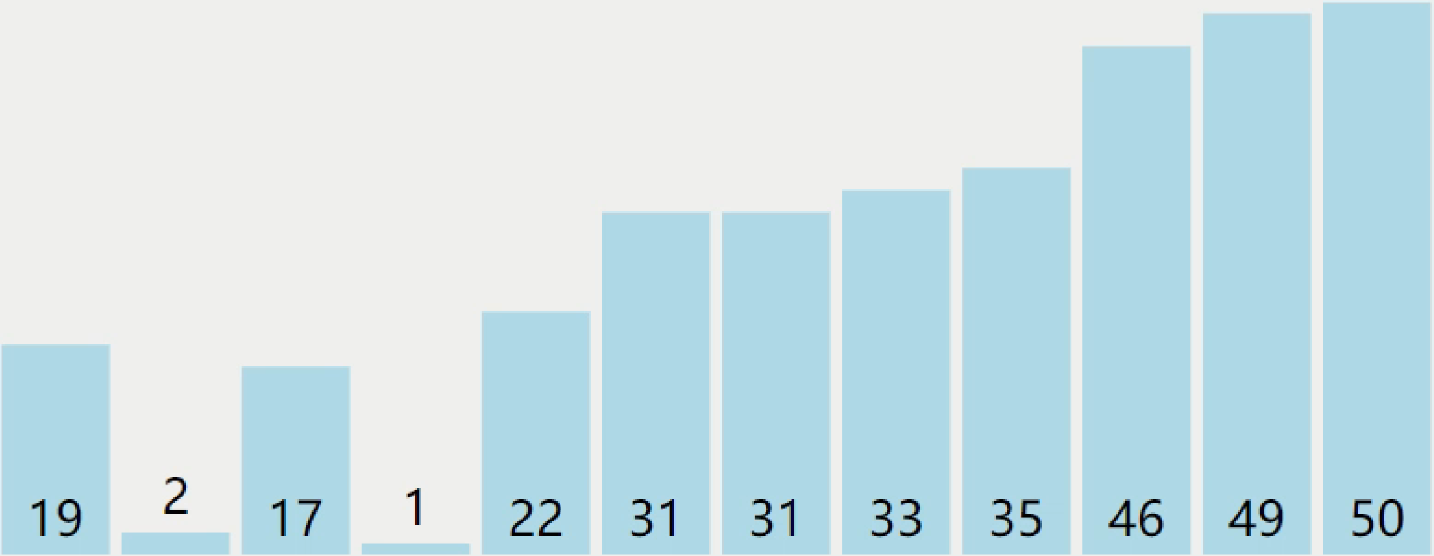
2.取出下一个元素，**在已经排序的元素序列**中**从后向前扫描**；

3.如果该元素（已排序）大于新元素，将该元素移到下一个位置；

4.重复步骤3，直到找到已排序的元素小于或者等于新元素的位置；

5.将新元素插入到该位置后，重复2~5

```
void InsertionSort(int *a, int len)
{
for (int j=1; j<len; j++)
//遍历整个数组的元素
{
int key = a[j];
int i = j-1;
while (i>=0 && a[i]>key)
{//寻找合适位置 并挪动其他元素形成空位
a[i+1] = a[i];
i--;
}
a[i+1] = key;//找到合适的位置插入
}
}
```



插入排序

将第一个元素标记为已排序

遍历每个没有排序过的元素

  “提取” 元素 x

  i = 最后排序过元素的指数 到 0 的遍历

    如果现在排序过的元素 > 提取的元素

      将排序过的元素向右移一格

    否则：插入提取的元素

创建

排序

执行

对比 发现了什么？

输入敏感型：

同等规模，逆序数越多，计算量越大

复杂度：O(n+m)

n为输入规模 m为逆序数量

排的有点**慢**

有没有更**快**的？

在讲更快一点的排序之前　先讲点别的：

双指针（two pointer）（8min）

引例

- 给定一个**递增**的正整数序列和一个正整数M，求序列中两个不同位置的数a和b，使得它们的和恰好为M,输出所有满足条件的方案，例如，给定序列{1，2，3，4，5，6}和正整数M=8,存在2+6==3+5==8成立。

- 直观
- 算法--暴力枚举：二重循环：判断和是否为M，若是则输出

```
for(int i = 0; i < n; i++) {
    for(int j = i + 1; j < n; j++) {
        if(a[i] + a[j] == M) {
            printf("%d %d\n", a[i], a[j]);
        }
    }
}
```

- 复杂度？？

改进

- 给定一个**递增**的正整数序列和一个正整数M，求序列中两个不同位置的数a和b，使得它们的和恰好为M,输出所有满足条件的方案，例如，给定序列{1，2，3，4，5，6}和正整数M=8,存在2+6==3+5==8成立。

- 可否利用数列的单调性，获得更优算法?
- 分析： 对于确定的a[i] 若存在a[j] :a[i]+a[j]>M → a[i]+a[j+l]>M (l>=1
- 可利用此特性减少枚举数量：
  - 易知，符合题意a[i]+a[j]==M的所有解a一定可以构成
  - 形如(i1,j1), (i2,j2) …(in,jn)形式排列，其中**i1<i2<…<in< jn< jn-1<…<j1；a[i1]<a[i2]<…a[in]< b[jn]<b[jn-1]>…>b[j1]**
  - 令**下标i初值为0**（对应**最小元素**），下标初值n-1（对应**最大元素**）接下来根据a[i]+a[j]与M大小 进行以下三种选择的一种：

改进

- 令下标i初值为0（对应最小元素），下标初值n-1（对应最大元素）接下来根据a[i]+a[j]与M的大小 进行以下*三种选择的一种*:

	i					j		
a[0]	…	a[i]	a[i+1]	…	a[j−1]	a[j]	…	a[n−1]

图 4-8 找 M 问题示意图

① 如果满足 a[i] + a[j] == M，说明找到了其中一组方案。由于序列递增，不等式 a[i+1] + a[j] > M 与 a[i] + a[j − 1] < M 均成立，但是 a[i + 1] + a[j − 1]与 M 的大小未知，因此剩余的方案只可能在[i + 1, j − 1]区间内产生，令 i = i + 1、j = j − 1（即令 i 向右移动，j 向左移动）。

② 如果满足 a[i] + a[j] > M，由于序列递增，不等式 a[i + 1] + a[j] > M 成立，但是 a[i] + a[j − 1]与 M 的大小位置，因此剩余的方案只可能在[i, j − 1]区间内产生，令 j = j − 1（即令 j 向左移动）。

③ 如果满足 a[i] + a[j] < M，由于序列递增，不等式 a[i] + a[j − 1] < M 成立，但是 a[i + 1] + a[j]与 M 的大小位置，因此剩余的方案只可能在[i + 1, j]区间内产生，令 i = i + 1（即令 i 向右移动）。

反复执行上面三个判断，直到 i≥j 成立。代码如下：

单调性：i单增
j单减

+ 遍历所有可能符合条件的i值（不符合的被排除）

⇒ 算法的正确性

改进

- 给定一个**递增**的正整数序列和一个正整数M，求序列中两个不同位置的数a和b，使得它们的和恰好为M,输出所有满足条件的方案，例如，给定序列{1，2，3，4，5，6}和正整数M=8,存在2+6==3+5==8成立。

```
while(i < j) {
    if(a[i] + a[j] == m) {
        printf("%d %d\n", i, j);
        i++;
        j--;
    } else if(a[i] + a[j] < m) {
        i++;
    } else {
        j--;
    }
}
```

- 改进后复杂度分析：
- 时间？空间？

例2 Merge

- 给定两个**递增/非降**的序列A,B。将其合并为新的**递增/非降**序列C

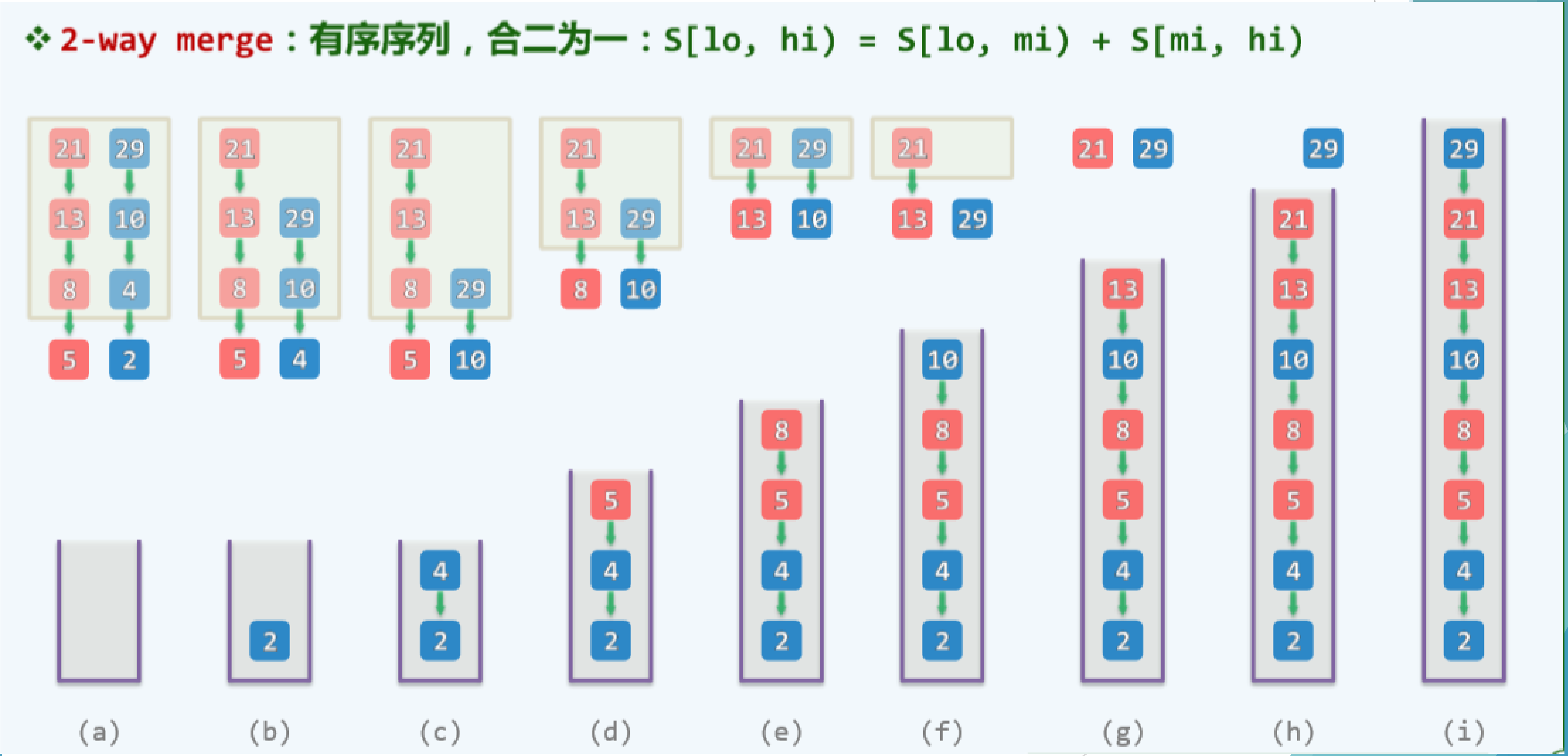
暴力：存到新数组，然后直接排序？ 行不行？ ✓

好不好？ ✗ ⇒ 复杂度：$O(n^2)$ / $O(n\log n)$（时间）

- 可否利用**双指针**进行改进？

例2 Merge

- 给定两个递增/非降的序列A,B。将其合并为新的递增/非降序列C
- 分析



例2 Merge

- 给定两个递增/非降的序列A,B。将其合并为新的递增/非降序列C

- 可否利用双指针进行改进?

双指针 在哪

```
int merge(int A[], int B[], int C[], int n, int m) {
    int i = 0, j = 0, index = 0;      //i 指向 A[0], j 指向 B[0]
    while(i < n && j < m) {
        if(A[i] <= B[j]) {            //如果 A[i]<=B[j]
            C[index++] = A[i++];      //将 A[i]加入序列 C
        } else {                      //如果 A[i]>B[j]
            C[index++] = B[j++];      //将 B[j]加入序列 C
        }
    }
    while(i < n) C[index++] = A[i++]; //将序列 A 的剩余元素加入序列 C
    while(j < m) C[index++] = B[j++]; //将序列 B 的剩余元素加入序列 C
    return index;             //返回序列 C 的长度
}
```

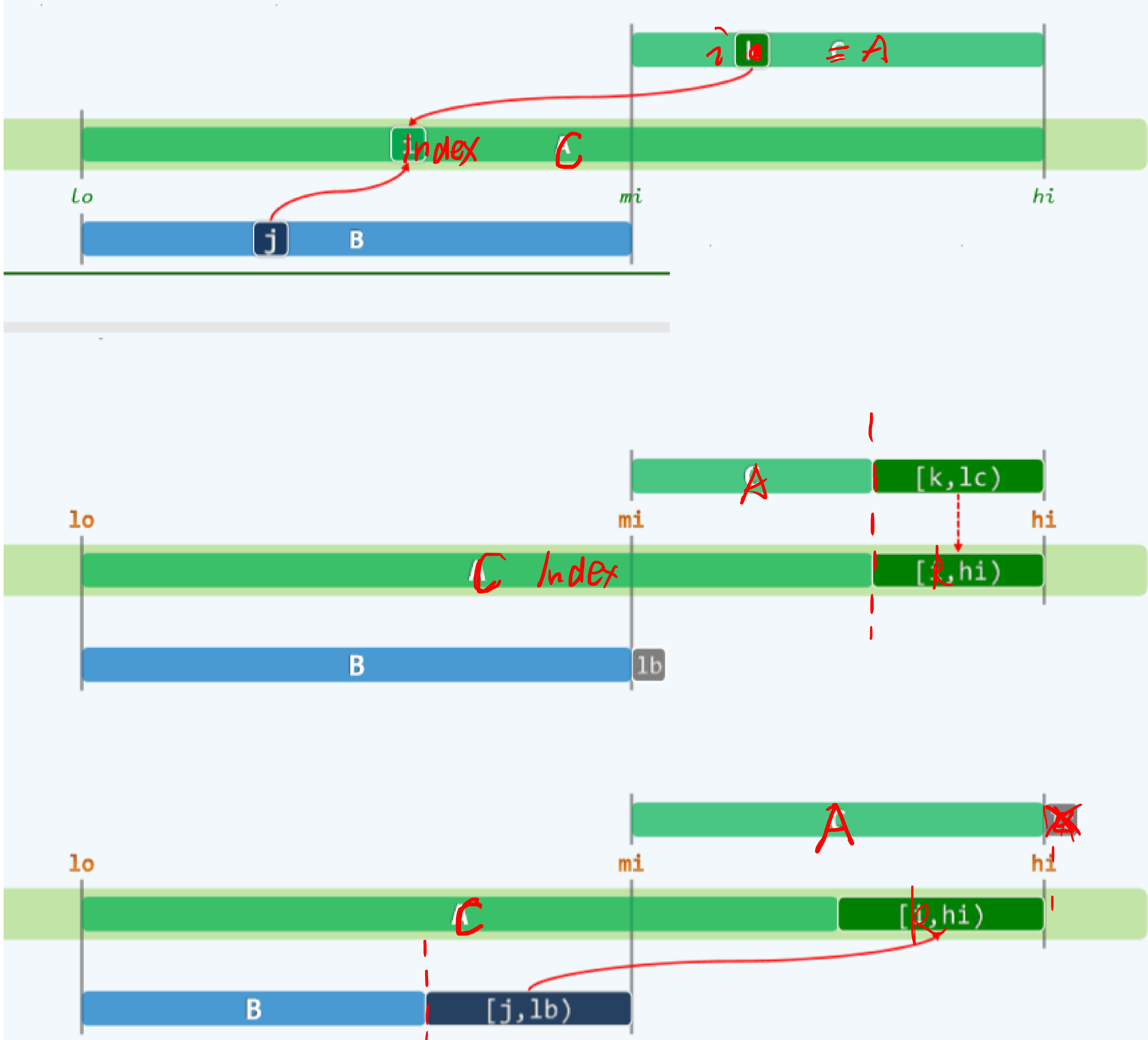
时间复杂度分析。

一层循环

遍历一遍

→ O(n)

正确性分析



i
≤ A
Index
C
lo
mi
hi
j
B
A
[k,lc)
lo
mi
hi
C
Index
[k,hi)
B
lb
A
lo
mi
hi
C
[k,hi)
B
[j,lb)

双指针 总结

- 利用问题本身与序列特性，使用下标i，j对序列进行扫描（可以同向，可以反向）

- 通常以较低的复杂度O(n)解决问题。

扫描一遍 O(n)

常数次扫描 O(n)

在谈归并排序之前：

Divide-and-Conquer

❖ 为求解一个大规模的问题，可以...

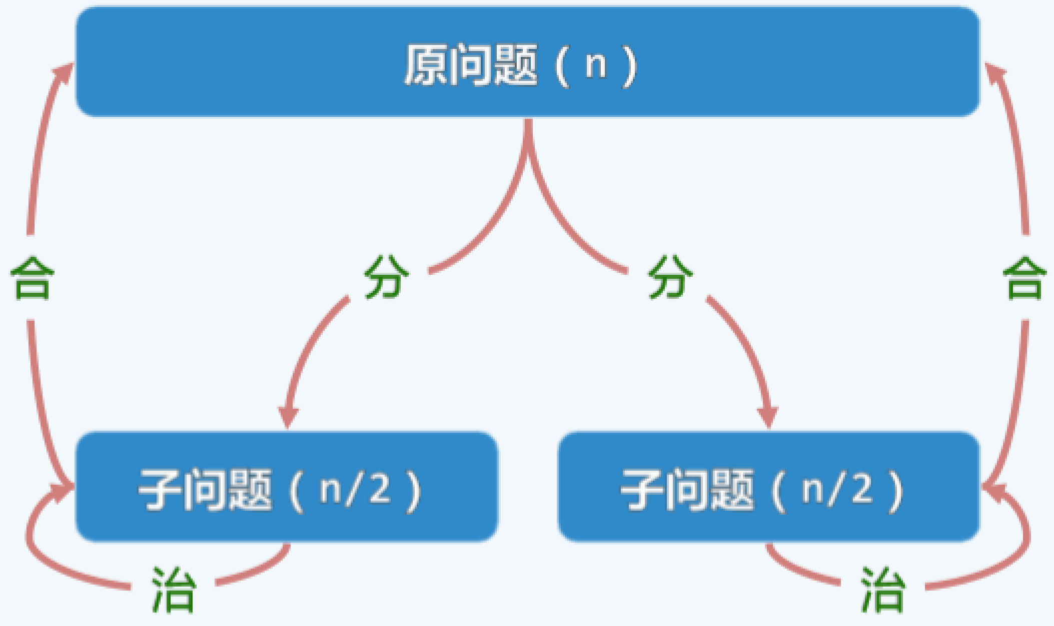
❖ 将其**划分**为若干子问题

（通常两个，且规模**大体相当**）

❖ 分别求解子问题

❖ 由子问题的解

合并得到原问题的解



归并排序（5min）

原理

※复杂度分析方法

❖ //分治策略

//向量与列表通用

//J. von Neumann，1945

序列一分为二 //$O(1)$

子序列递归排序 //2 × T($\boxed{n/2}$)

合并有序子序列 //$O(\boxed{n})$

❖ 若真能如此，整体的运行成本

应是 $\mathcal{O}(n \cdot \log n)$

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

不妨令 $n = 2^k$　$k = \log_2 n$

则

$T(n) = 2^k T(1) + C(k \cdot 2^k)$

⇑

以n为主 nlogn

高度

至多

$\lceil \log_2 n \rceil$层

每层总量加

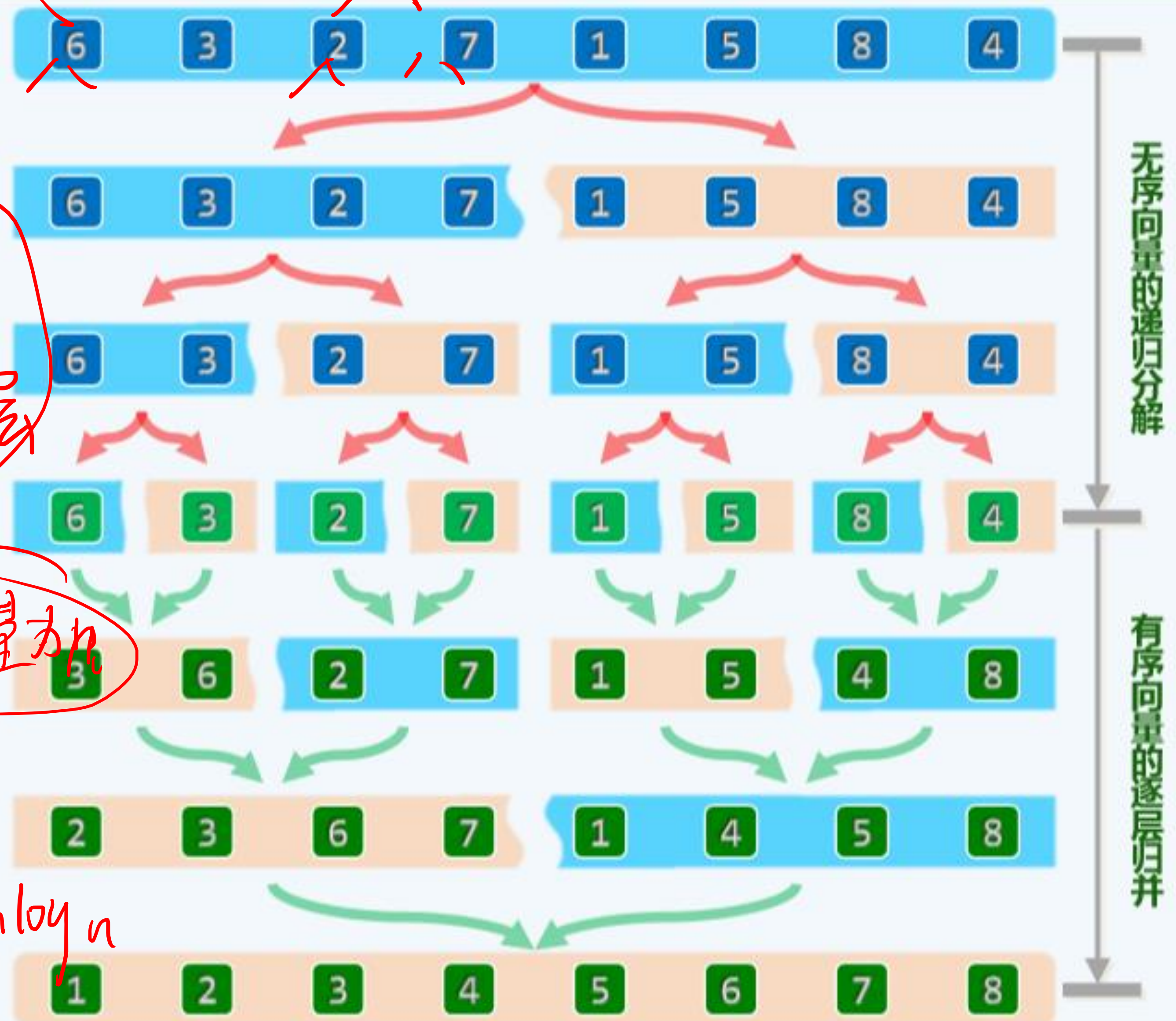
⇓

复杂度 nlogn

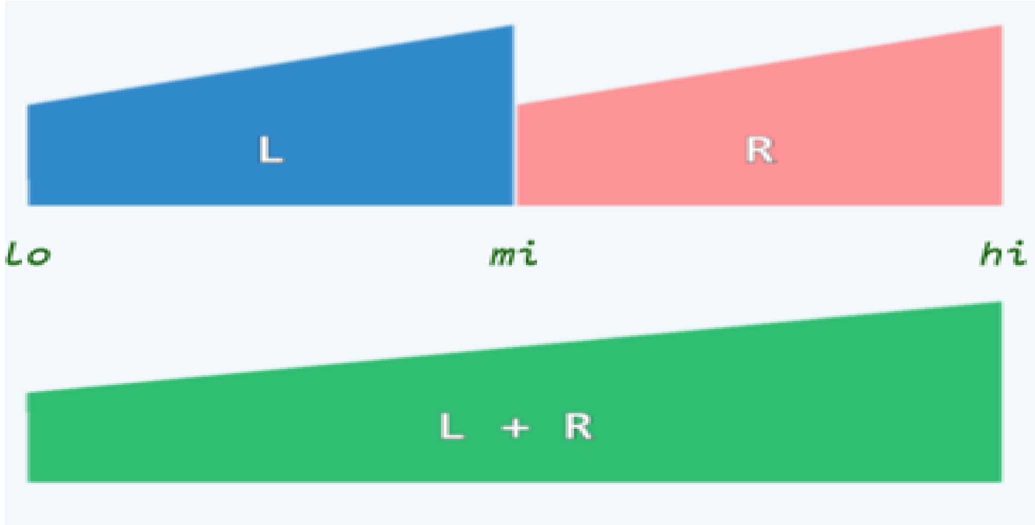
递归树分析

Cn

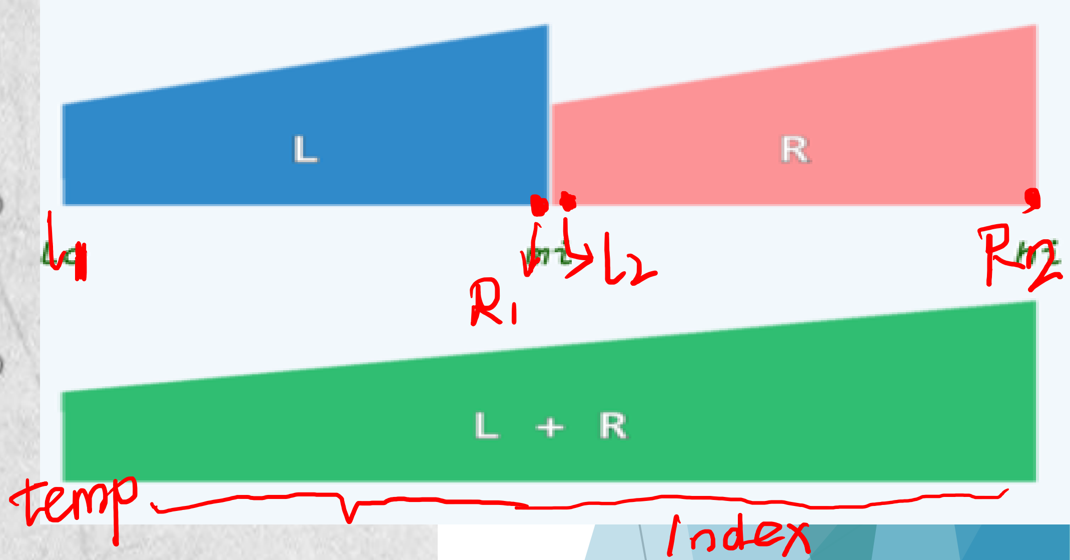
$c \cdot \frac{1}{2}n$　$c \cdot \frac{1}{2}n$

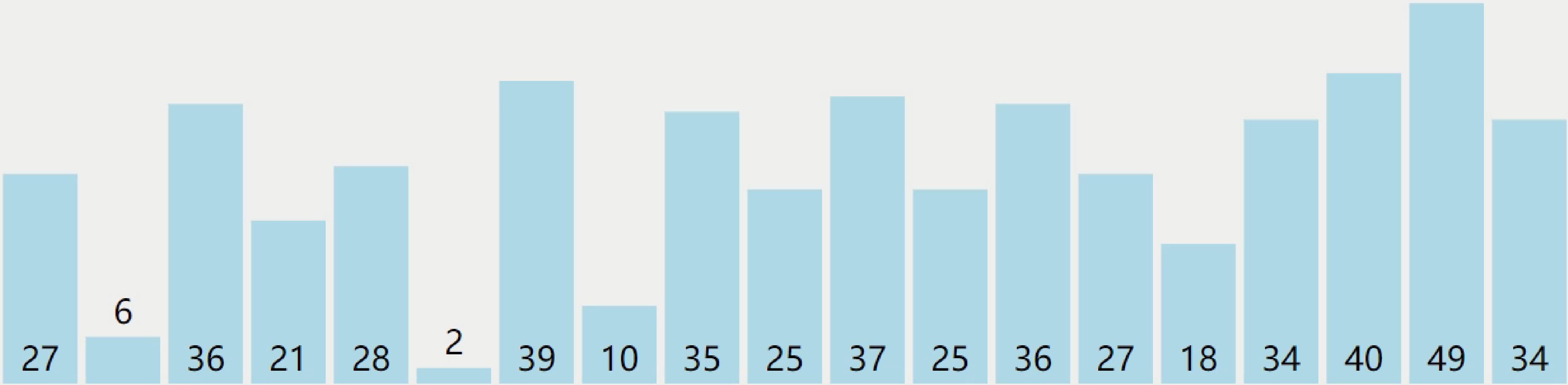


```
//将 array 数组当前区间[left, right]进行归并排序
void mergeSort(int A[], int left, int right) {
    if(left < right) {    //只要 left 小于 right
        int mid = (left + right) / 2;    //取[left, right]的中点
        mergeSort(A, left, mid);    //递归，将左子区间[left,mid]归并排序
        mergeSort(A, mid+1, right);  //递归，将右子区间[mid+1,right]归并排序
        merge(A, left, mid, mid + 1, right);    //将左子区间和右子区间合并
    }
}
```



```
//将数组 A 的[L1, R1]与[L2, R2]区间合并为有序区间（此处 L2 即为 R1 + 1）
void merge(int A[], int L1, int R1, int L2, int R2) {
    int i = L1, j = L2;     //i 指向 A[L1]，j 指向 A[L2]
    int temp[maxn], index = 0;     //temp 临时存放合并后的数组，index 为其下标
    while(i <= R1 && j <= R2) {
        if(A[i] <= A[j]) {             //如果 A[i]<=A[j]
            temp[index++] = A[i++];        //将 A[i]加入序列 temp
        } else {                       //如果 A[i]>A[j]
            temp[index++] = A[j++];        //将 A[j]加入序列 temp
        }
    }
    while(i <= R1) temp[index++] = A[i++]; //将[L1, R1]的剩余元素加入序列 temp
    while(j <= R2) temp[index++] = A[j++]; //将[L2, R2]的剩余元素加入序列 temp
    for(i = 0; i < index; i++) {
        A[L1 + i] = temp[i];     //将合并后的序列赋值回数组 A
    }
}
```





创建

排序

计算倒置指数

执行

快速排序（8-9min）

分而治之

❖ pivot：左侧/右侧的元素，均不比它更大/更小

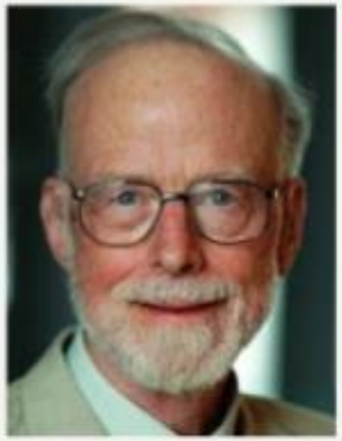
❖ 以轴点为界，自然划分：max([0,mi)) ≤ min((mi, hi))

❖ 前缀、后缀各自（递归）排序之后，原序列自然有序

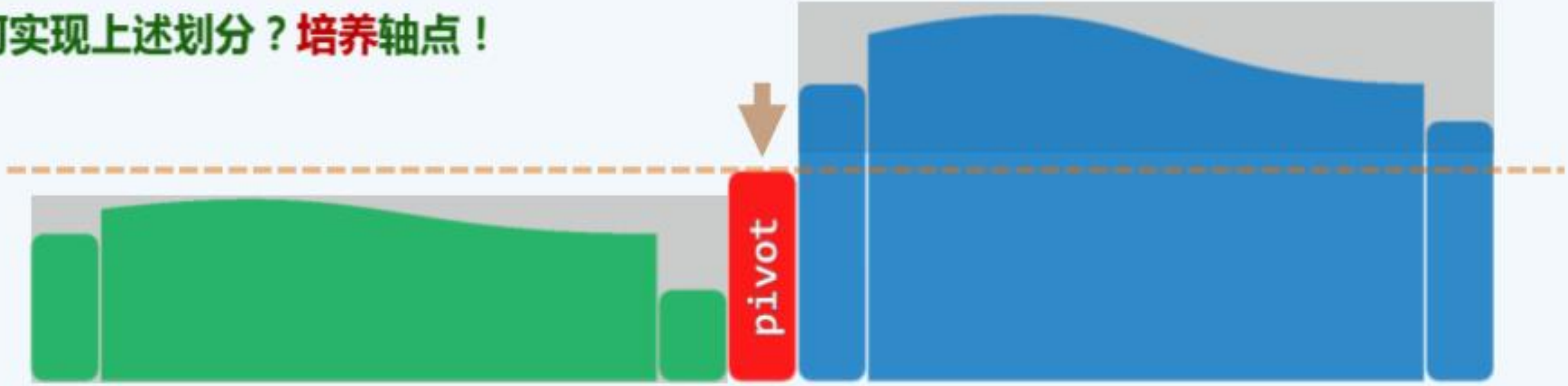
$$\text{sorted}(S) = \text{sorted}(S_L) + \text{sorted}(S_R)$$

❖ mergesort的难点在于合，而quicksort在于分

❖ 如何实现上述划分？培养轴点！

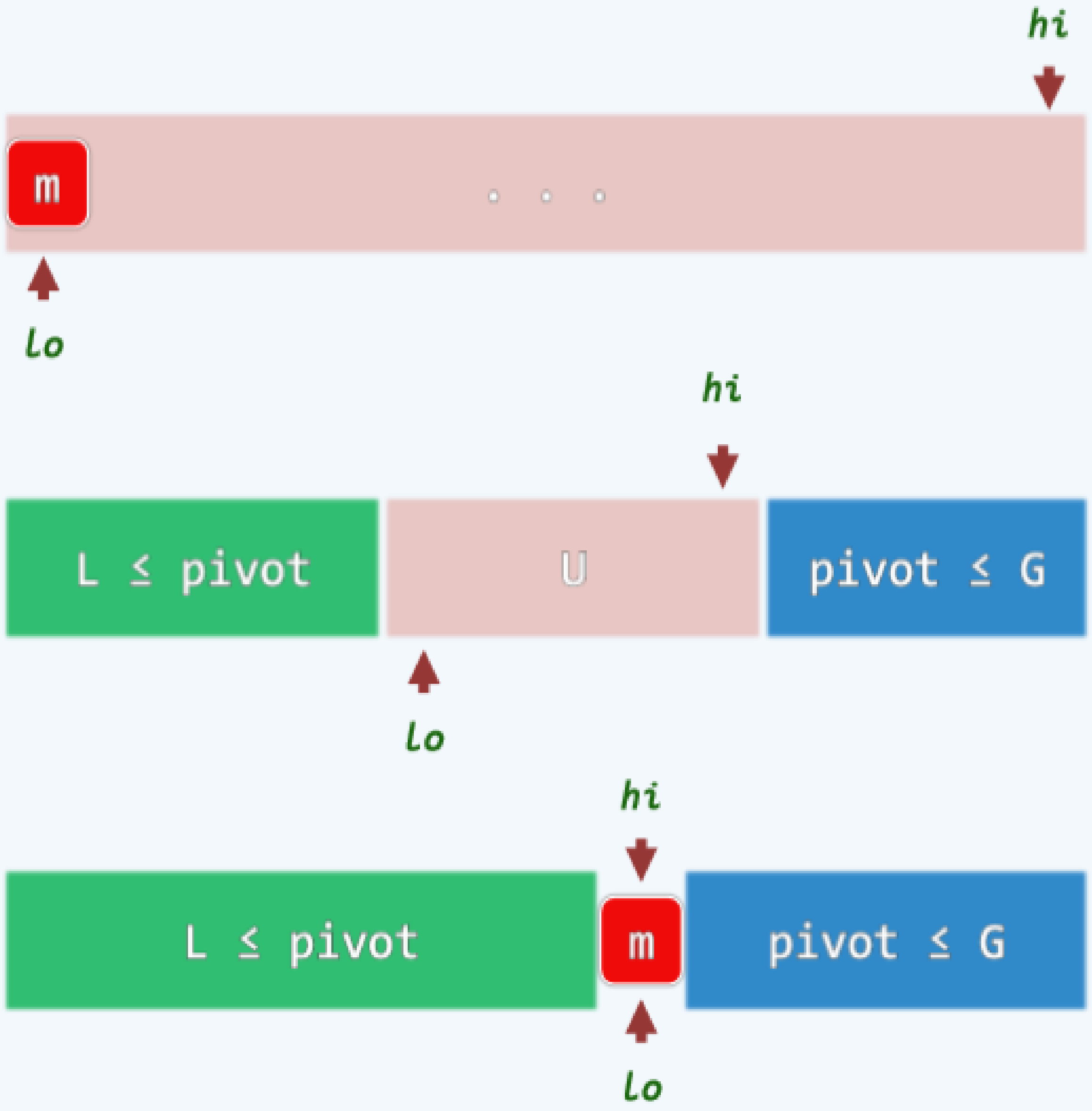


C. A. R. Hoare
(1934 ~)
Turing Award, 1980



减而治之，相向而行

- 任取一个候选者（如[0]）
- 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$\mathcal{O}(n)$时间、$\mathcal{O}(1)$辅助空间



```
//快速排序，left 与 right 初值为序列首尾下标（例如 1 与 n）
void quickSort(int A[], int left, int right) {
    if(left < right) {        //当前区间的长度不超过 1
        //将[left, right]按 A[left]一分为二
        int pos = Partition(A, left, right);
        quickSort(A, left, pos - 1);     //对左子区间递归进行快速排序
        quickSort(A, pos + 1, right);    //对右子区间递归进行快速排序
    }
}
```

→ 快排关键，划分

在整个划分过程中
temp对应元素相当于拿在手上

双指针

~~轴点 主元~~

这样选好吗？

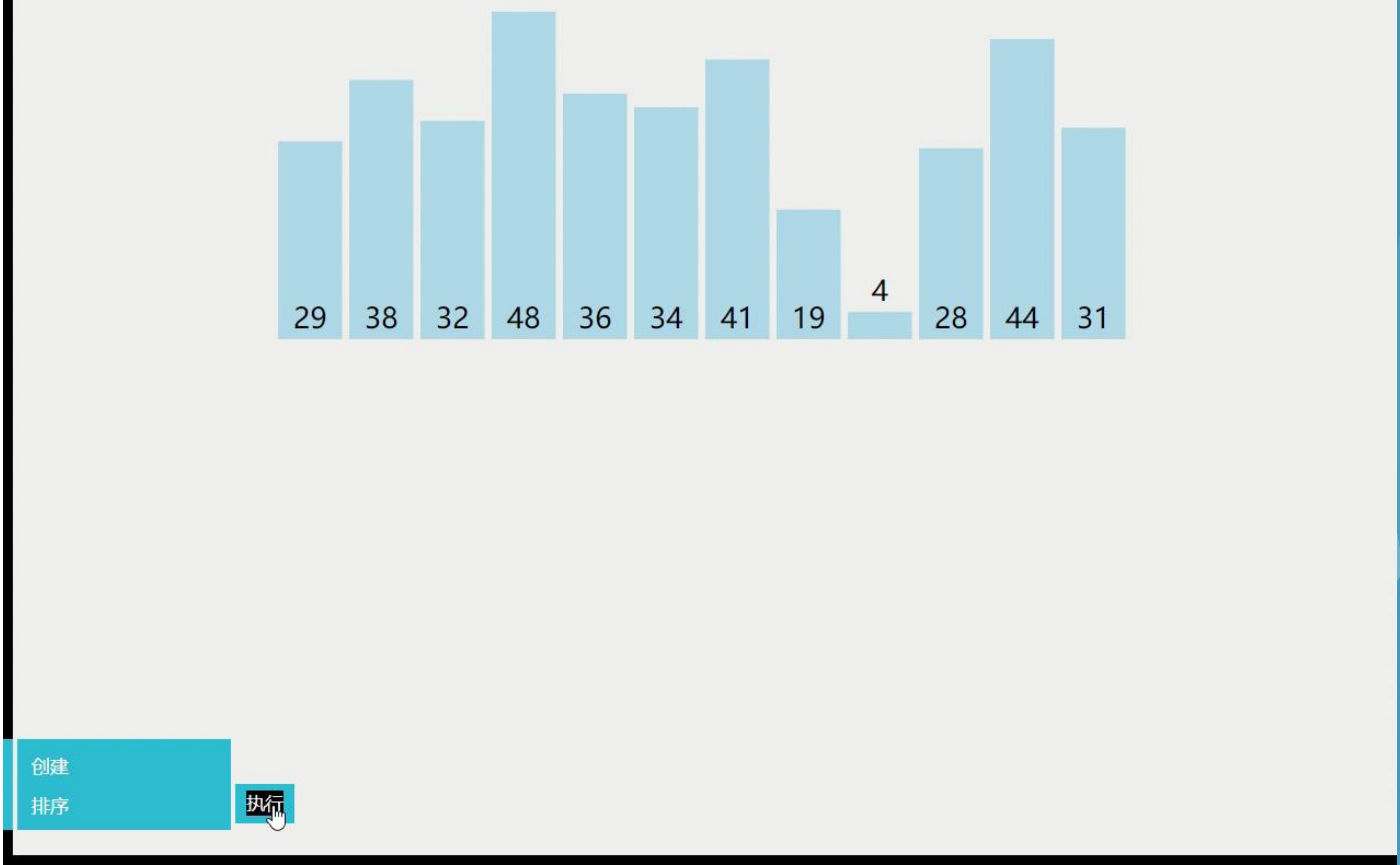
```
//对区间[left,right]进行划分
int Partition(int A[], int left, int right) {
    int temp = A[left];     //将 A[left]存放至临时变量 temp
    while(left < right) {   //只要 left 与 right 不相遇
        while(left < right && A[right] > temp) right--;   //反复左移 right
        A[left] = A[right];      //将 A[right]挪到 A[left]
        while(left < right && A[left] <= temp) left++;    //反复右移 left
        A[right] = A[left];      //将 A[left]挪到 A[right]
    }
    A[left] = temp;     //把 temp 放到 left 与 right 相遇的地方
    return left;        //返回相遇的下标
}
```

原值只起占位作用

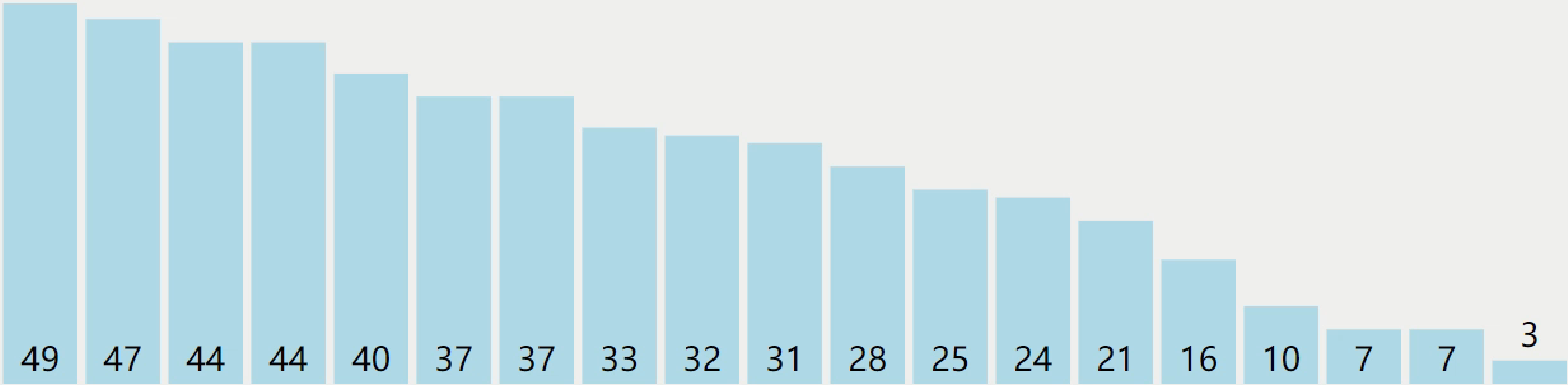
放进去位

划分过程中
每次均有某一元素
具有其"副本"，但只起
占位作用

效果不错，但如果……



啊这！
和冒泡排序差不多慢

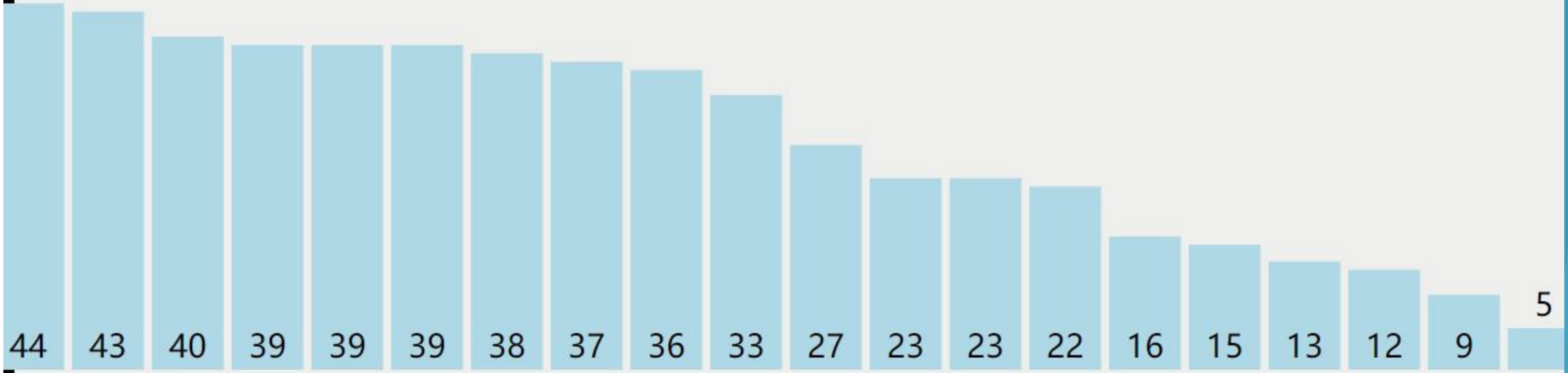


快速排序

每个（未排序）的部分
将第一个元素设为轴心点
　存储指数 = 轴心点指数 +1
　从 i=轴心点指数 +1 到 最右指数 的遍历
　　如果 元素[i] < 元素[轴心点]
　　　交换(i, 存储指数); 存储指数++
　交换(轴心点, 存储指数 - 1)

创建
排序

执行



随机快速排序

```
每个（未排序）的部分
随机选取轴心点，和第一个元素交换
  存储指数 = 轴心点指数 +1
  从 i=轴心点指数 +1 到 最右指数 的遍历
    如果 元素[i] < 元素[轴心点]
      交换(i, 存储指数); 存储指数++
  交换(轴心点, 存储指数 - 1)
```

创建

排序

执行

改进

```
//选取随机主元，对区间[left,right]进行划分
int randPartition(int A[], int left, int right) {
    //生成[left, right]内的随机数p
    int p = (round(1.0*rand()/RAND_MAX*(right - left) + left);
    swap(A[p], A[left]);     //交换A[p]和A[left]
    //以下为原先Partition函数的划分过程，不需要改变任何东西
    int temp = A[left];      //将A[left]存放至临时变量temp
    while(left < right) {    //只要left与right不相遇
        while(left < right && A[right] > temp) right--;     //反复左移right
        A[left] = A[right];      //将A[right]挪到A[left]
        while(left < right && A[left] <= temp) left++;      //反复右移left
        A[right] = A[left];      //将A[left]挪到A[right]
    }
    A[left] = temp;      //把temp放到left与right相遇的地方
    return left;         //返回相遇的下标
}
```

复杂度分析

平均情况O(nlogn)

最坏情况O(n^2)（整体有序或接近有序时）对极端输入比较敏感，是非稳定的

可用随机化尽量避免最坏情况