

数据结构之排序

2019-1-19

Youtube频道: hwdong

B站: hw-dong

排序



hwdong

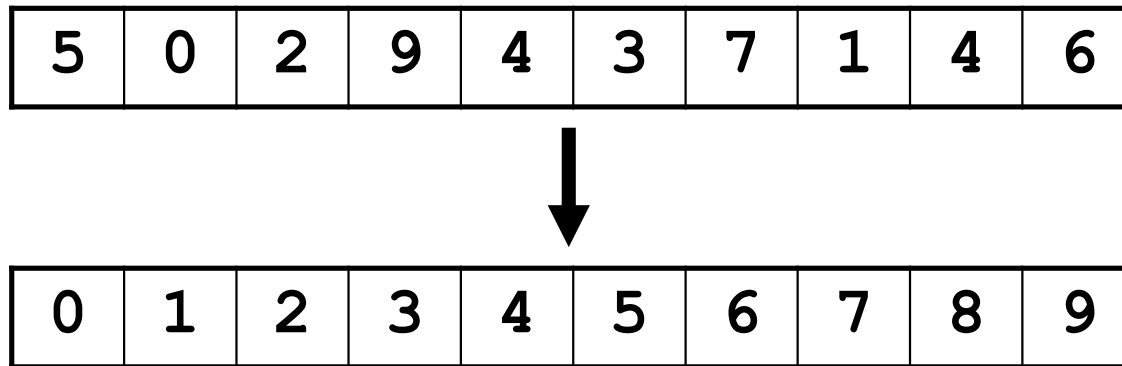
hw-dong

主要内容

- 什么是排序
- 内部排序
 - 插入式排序：直接插入排序法、希尔排序法
 - 交换式排序：气泡法、快速排序法
 - 选择式排序：直接选择排序法、锦标赛排序法、堆排序
 - 归并排序
 - 基数排序
- 各种内部排序方法的比较

什么是排序

- **排序**：按照一定的规则，对一系列数据进行排列



什么是排序

- **排序**：按照一定的规则，对一系列数据进行排列

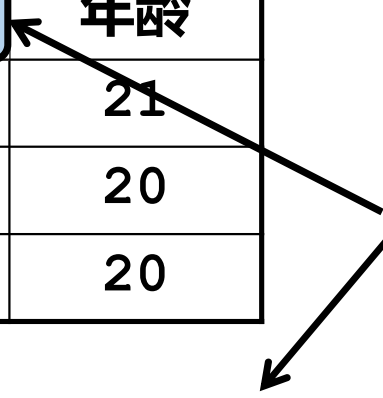
按学号排序：

姓名	学号	年龄
张三	1	21
李四	2	20
王五	3	20

按年龄排序：

姓名	学号	年龄
王五	3	20
李四	2	20
张三	1	21

排序码
关键字



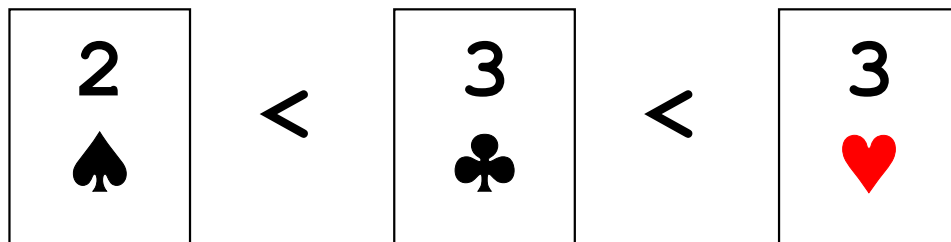
排序码（关键字）

- 排序码可以是数据元素本身或其1个或多个数据项

$13 < 38 < 49 < 56 < 76 < 97$

姓名	学号	年龄
张三	1	21
李四	2	20
王五	3	20

- 按分值再按花色



为什么排序?

帮助我们评估、选择

综合 **销量** 上市时间 价格从高到低 **价格从低到高** ¥ - ¥ < 1/100 >

<p>小屏1.5"</p>  <p>华为 C2823 超长待机, 学生备用机</p> <p>约 ¥34 293 人付款 点评 178 条</p> <p>共有4商家在售</p>	<p>小屏1.5"</p>  <p>降价</p> <p>华为 C2906 超长待机, 耐用老人机</p> <p>约 ¥38 245 人付款 点评 1488 条</p> <p>共有8商家在售</p>	<p>小屏1.5"</p>  <p>降价</p> <p>华为 C2800 超长待机, 老人学生手机, 自带QQ</p> <p>约 ¥39 2415 人付款 点评 55443 条</p> <p>共有26商家在售</p>	<p>小屏1.5"</p>  <p>华为 C2829 大音量大字体, 超长待机, 老人机</p> <p>约 ¥55 683 人付款 点评 9949 条</p> <p>共有31商家在售</p>
<p>小屏1.8"</p>  <p>关爱通 A111 高亮手电筒, 验钞灯</p>	<p>小屏1.5"</p>  <p>降价</p> <p>华为 C2857 QQ后台, 磨砂大字大音量, 学生机</p>	<p>小屏1.8"</p>  <p>橄榄树(数码) X600 大电池长续航, 一键SOS</p>	<p>小屏1.8"</p>  <p>科宝 KB1288 大字体大按键, 双卡双待</p>

为什么排序?

Adobe	2016/9/10 ...	文件夹
Application Verifier	2016/2/5 4:...	文件夹
CMAK	2016/7/30 ...	文件夹
Common Files	2017/9/29 ...	文件夹
DVD Maker	2016/5/13 ...	文件夹
GIMP 2	2017/9/5 1...	文件夹
Intel	2016/5/13 ...	文件夹
Internet Explorer	2017/9/18 ...	文件夹
MATLAB	2016/2/5 3:...	文件夹
Microsoft Analysis Services	2017/9/29 ...	文件夹
Microsoft Help Viewer	2016/2/5 4:...	文件夹
Microsoft Kinect Drivers	2016/5/13 ...	文件夹
Microsoft SDKs	2016/5/13 ...	文件夹
Microsoft SQL Server	2016/2/5 5:...	文件夹
Microsoft SQL Server Co...	2016/2/5 4:...	文件夹
Microsoft Synchronization...	2016/2/5 4:...	文件夹
Microsoft Visual Studio 1...	2016/2/5 4:...	文件夹
Microsoft Visual Studio 1...	2016/2/5 4:...	文件夹
Microsoft VS Code	2017/10/27...	文件夹
Mozilla Firefox	2017/11/5 ...	文件夹
MSBuild	2016/5/13 ...	文件夹
NVIDIA Corporation	2016/5/13 ...	文件夹
Realtek	2016/5/13 ...	文件夹
Reference Assemblies	2016/5/13 ...	文件夹
rempl	2017/11/3 ...	文件夹
SharePoint Client Compon...	2016/2/5 4:...	文件夹
Sublime Text 3	2017/10/28...	文件夹
Synaptics	2016/5/13 ...	文件夹
TrueCrypt	2017/9/5 9:...	文件夹
Uninstall Information	2016/2/13 ...	文件夹
UNP	2017/8/31 ...	文件夹
Windows Defender	2017/9/18 ...	文件夹
Windows Mail	2017/9/18 ...	文件夹

提高可读性、加快搜索

线性搜索： $O(n)$

二分搜索： $O(\log n)$

排序算法的稳定性

- 如果某排序算法,不会改变相同排序码的数据元素的前后次序。该算法称为**稳定的**排序算法。否则该算法就是**不稳定的**。如

Li	21
Wang	21
Zhang	19

排序后:

Zhang	19
Wang	21
Li	21

内部排序和外部排序

- 内部排序
 - 数据元素全部存放在内存中进行的排序
- 外部排序
 - 数据元素个数太多，不能同时存放在内存中，需要借助于外部存储器，排序过程中，取一部分到内存中来排序，然后再存回外部存储器

排序算法优劣的衡量

- 时间复杂度

平均情况

最好情况和最差情况：有一些算法，其复杂度受最初数据的排列情况影响较大

- 空间复杂度

排序时所需的额外的存储空间

额外：指除了存放数据以外还需要的

插入式排序

插入式排序

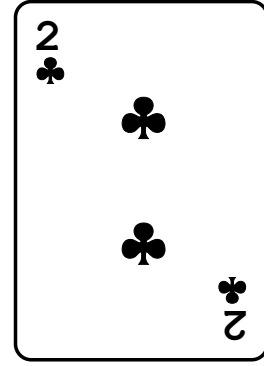
- 基本思想:

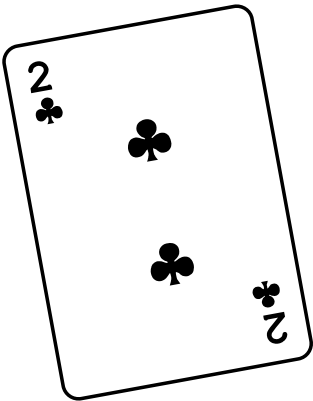
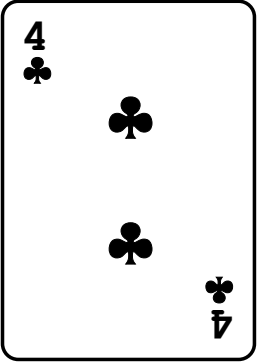
- 每一步将一个待排序的对象，按其排序码大小，插入到前面已经排好序的一组对象的适当位置上，直到全部插入为止

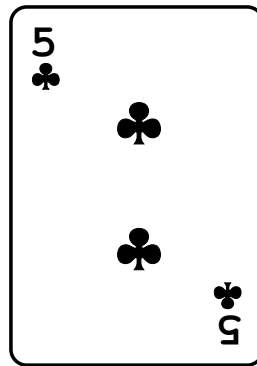
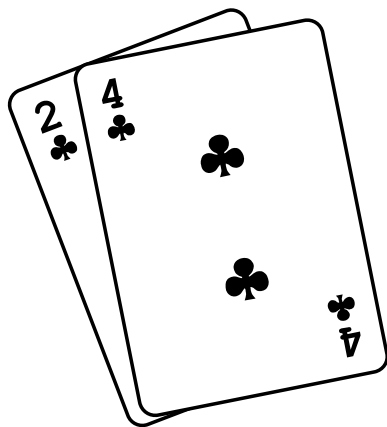
2 8 11 3 ...

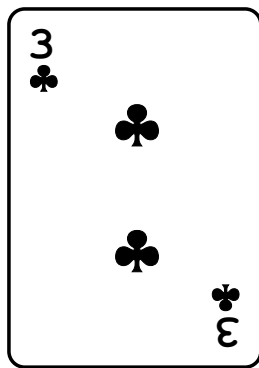
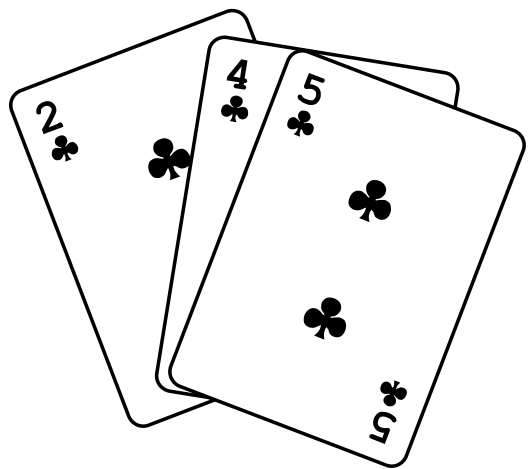
- 类比：扑克牌抓牌







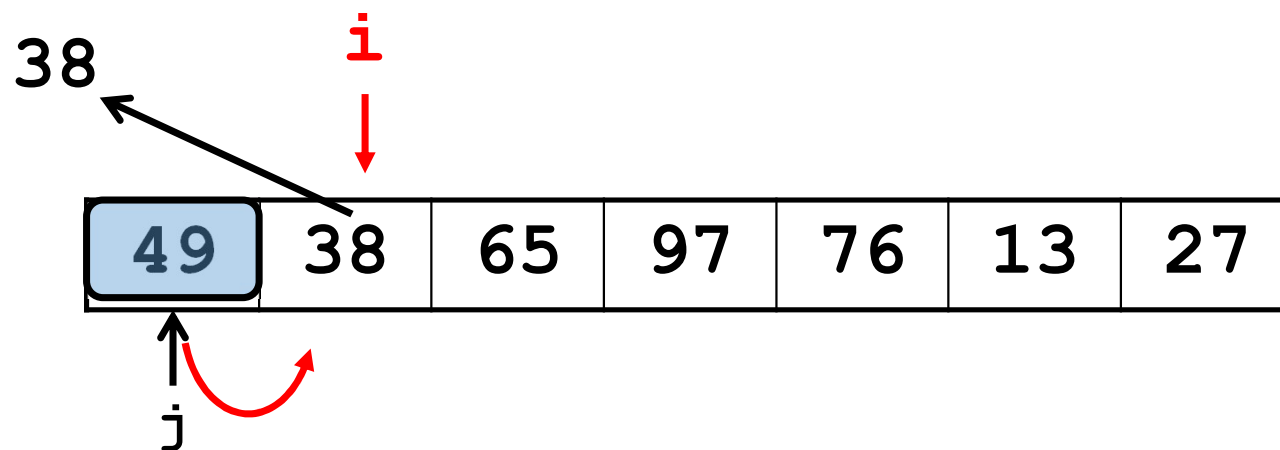




直接插入排序：基本思想

• 例

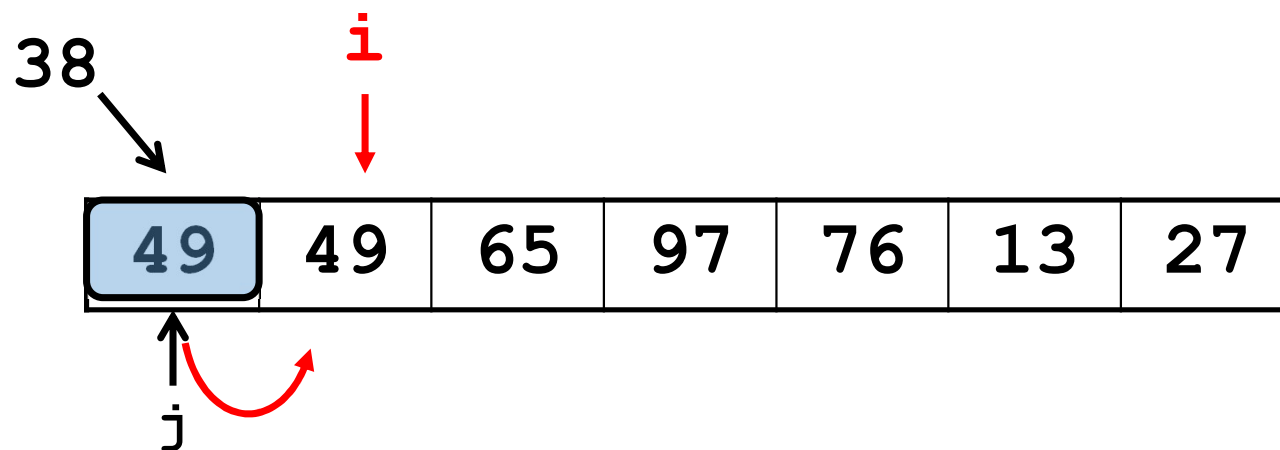
从 $i=1$ 开始



直接插入排序：基本思想

• 例

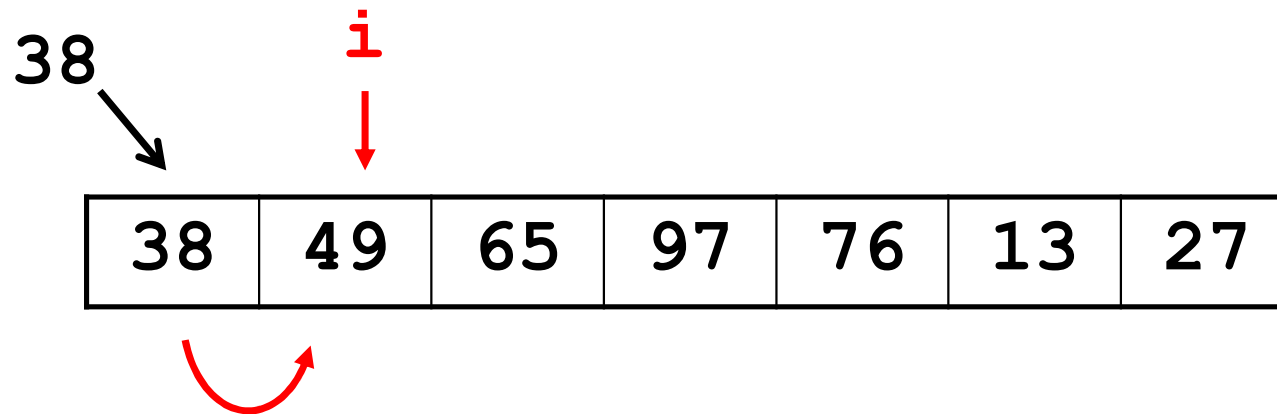
从 $i=1$ 开始



直接插入排序：基本思想

• 例

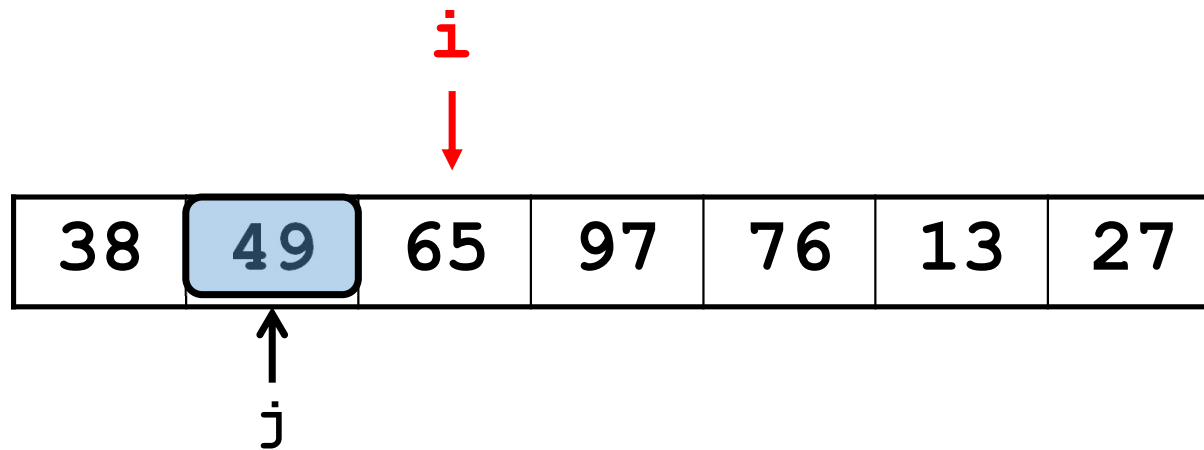
从 $i=1$ 开始



直接插入排序：基本思想

• 例

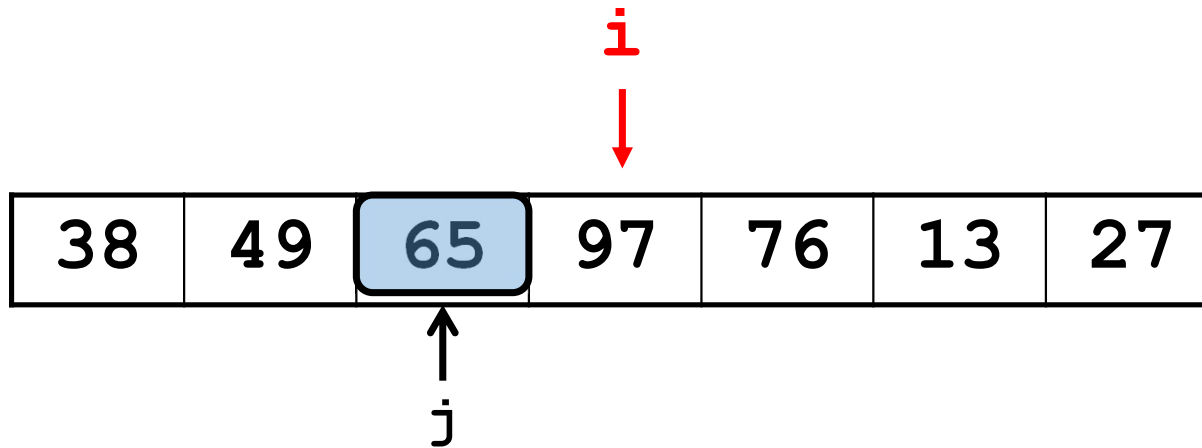
$a[i] < a[i-1] ?$



直接插入排序：基本思想

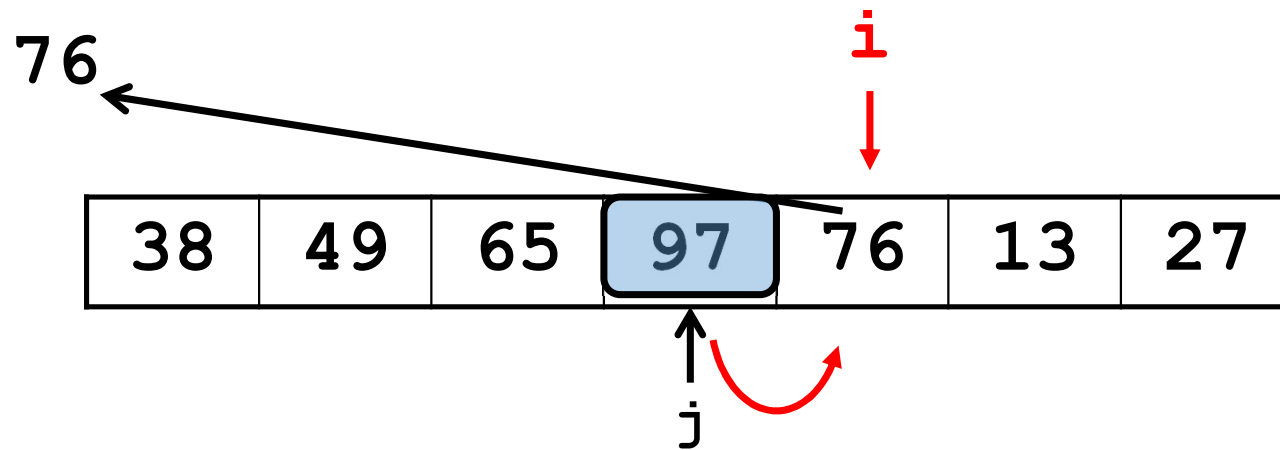
• 例

$a[i] < a[i-1] ?$



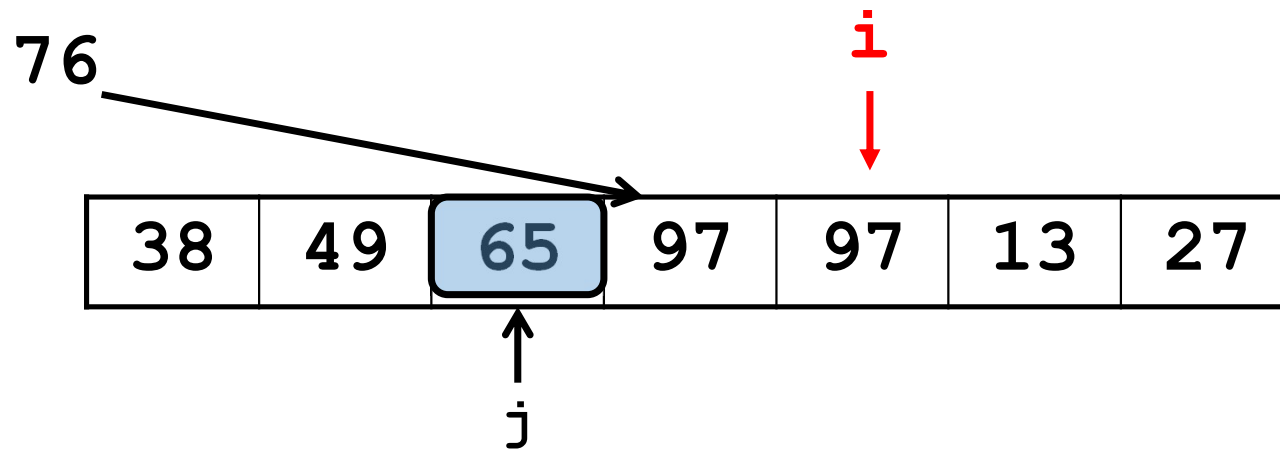
直接插入排序：基本思想

• 例



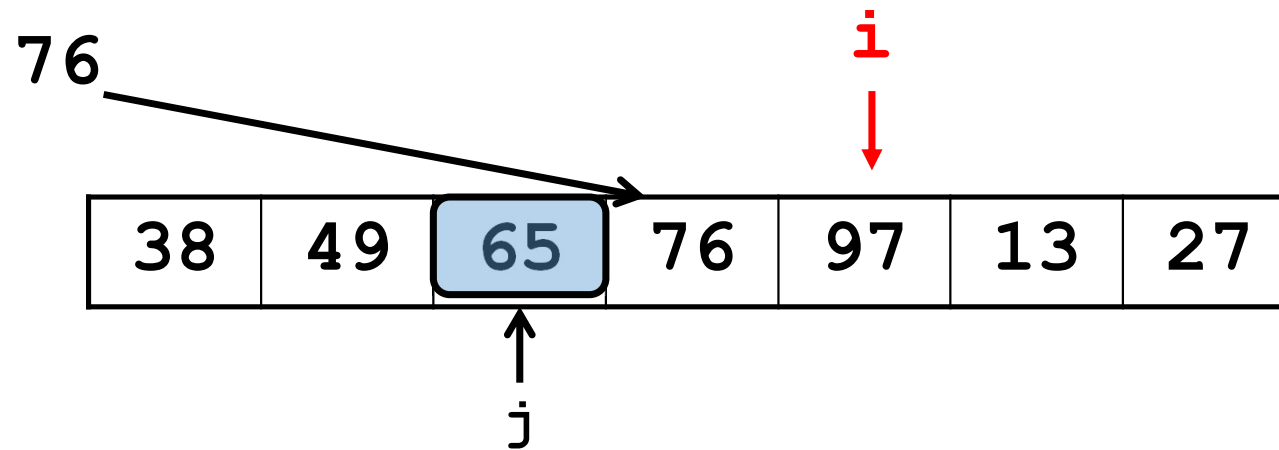
直接插入排序：基本思想

• 例



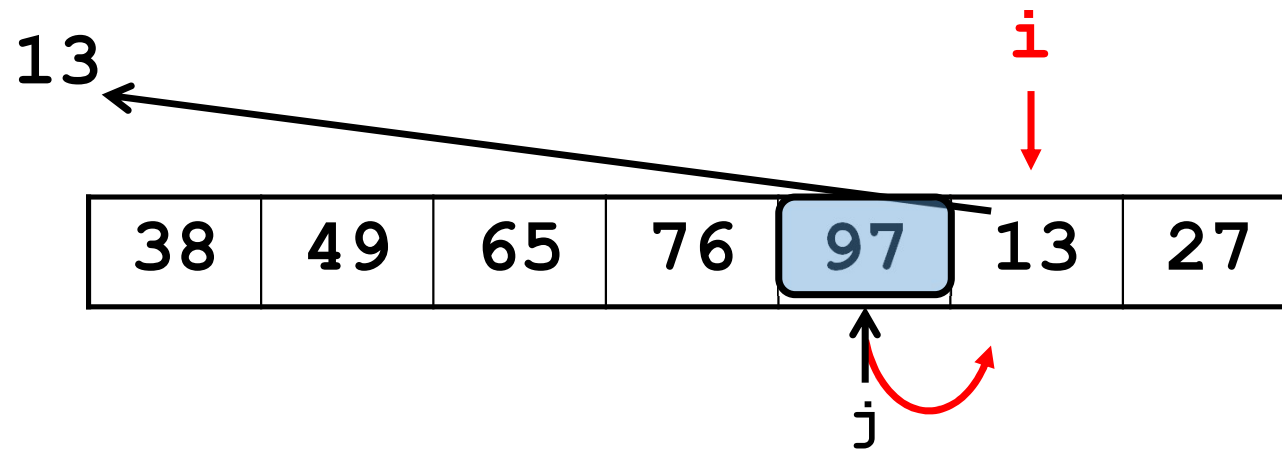
直接插入排序：基本思想

• 例



直接插入排序：基本思想

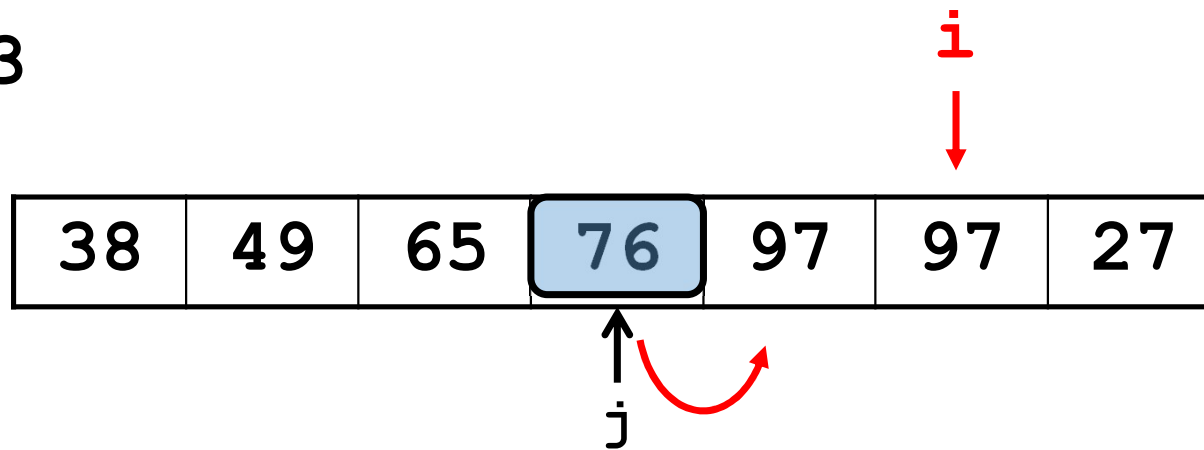
• 例



直接插入排序：基本思想

• 例

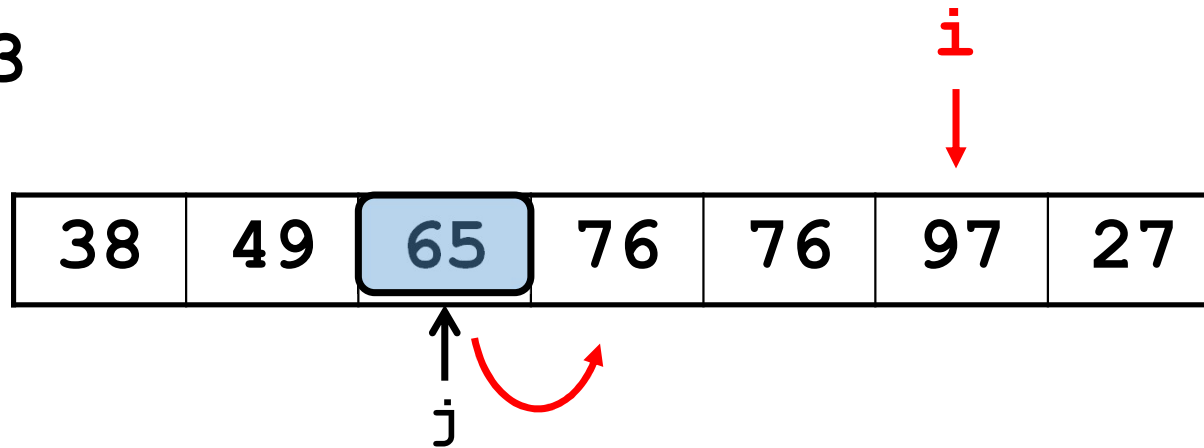
13



直接插入排序：基本思想

• 例

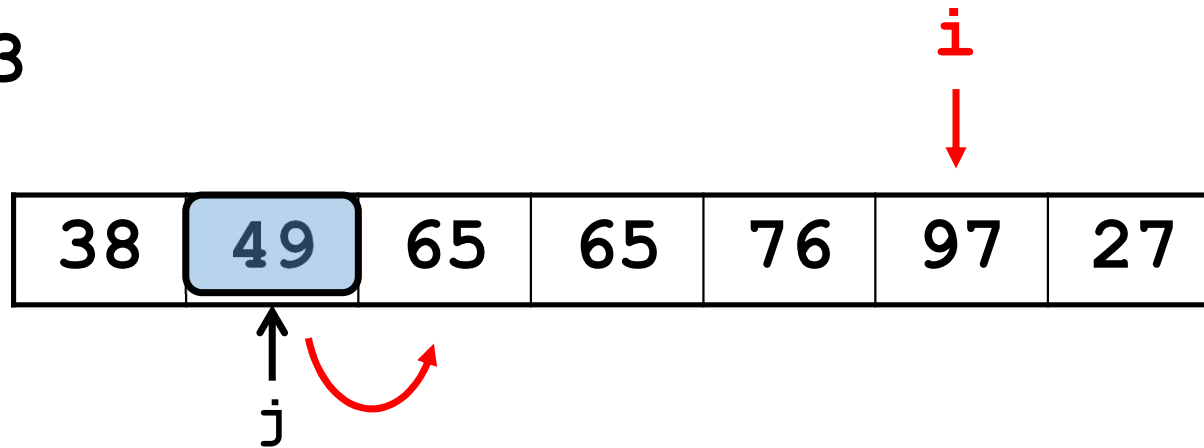
13



直接插入排序：基本思想

• 例

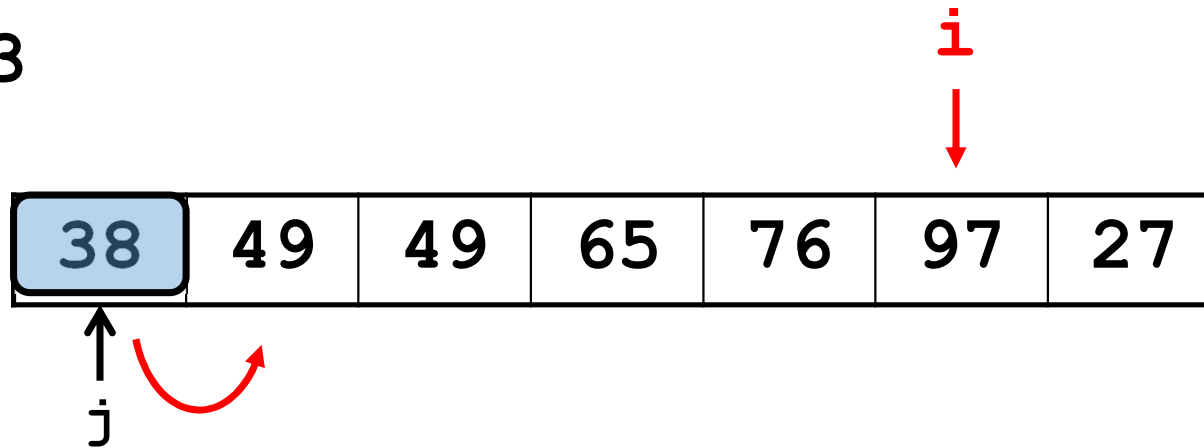
13



直接插入排序：基本思想

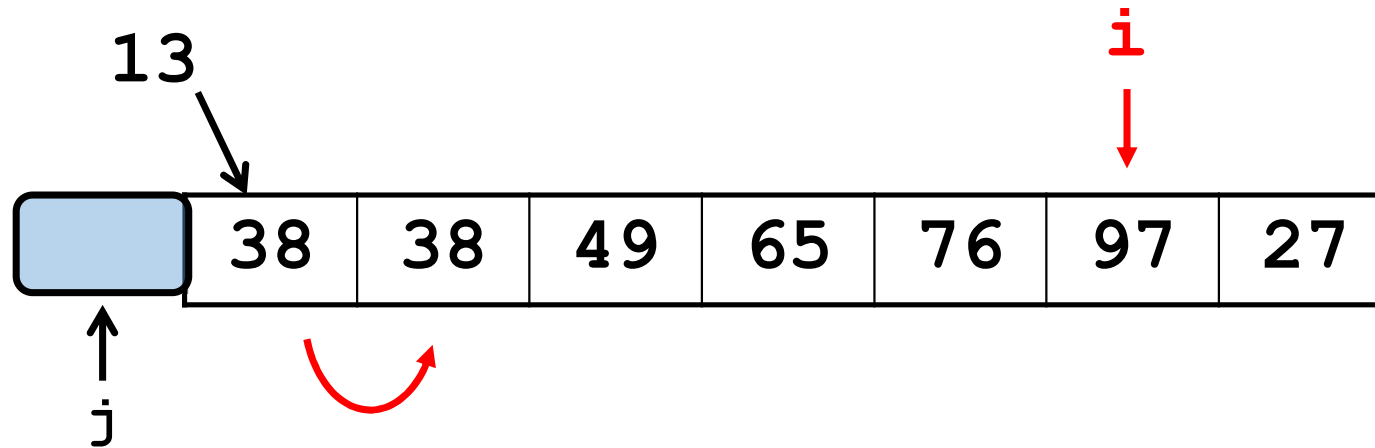
• 例

13



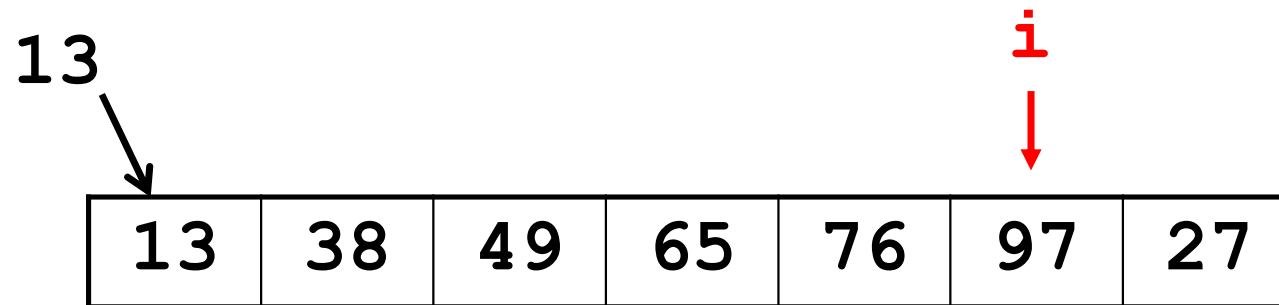
直接插入排序：基本思想

• 例



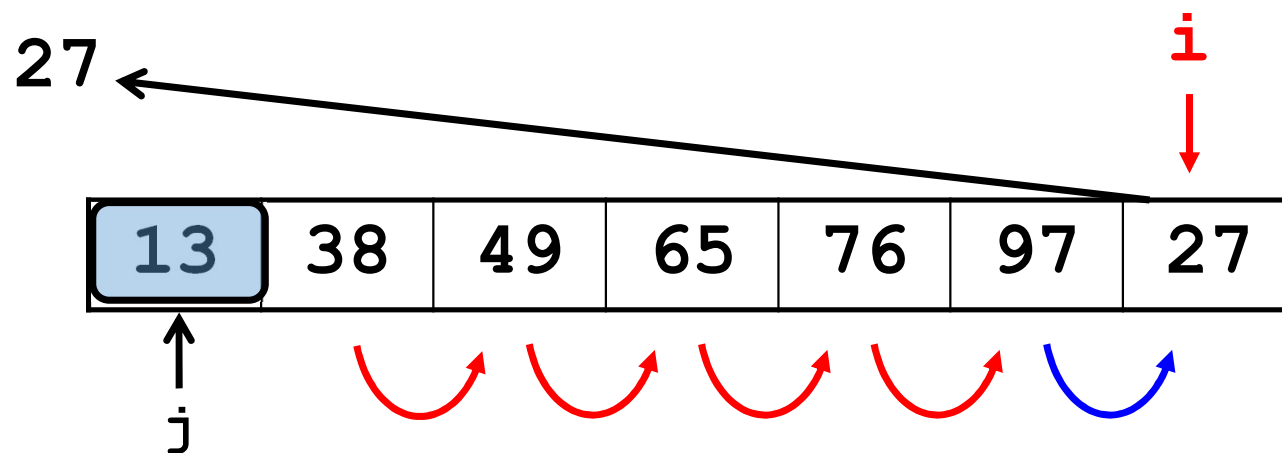
直接插入排序：基本思想

• 例



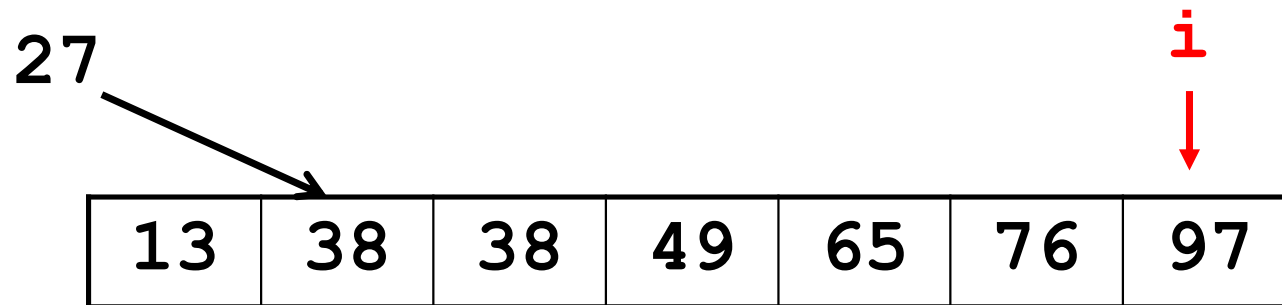
直接插入排序：基本思想

• 例



直接插入排序：基本思想

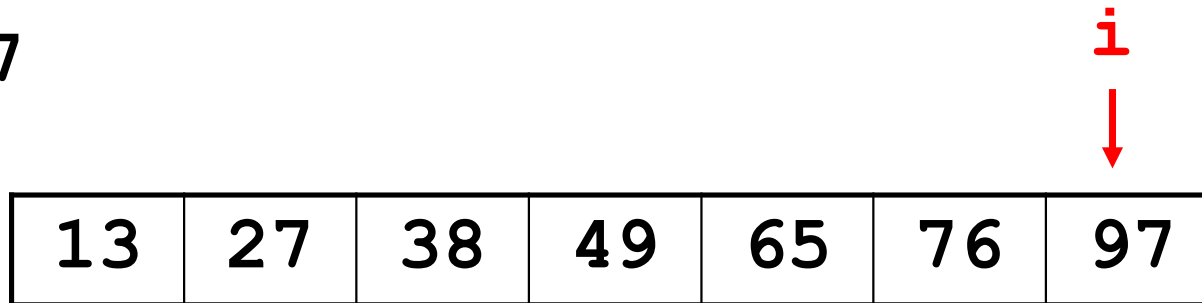
• 例

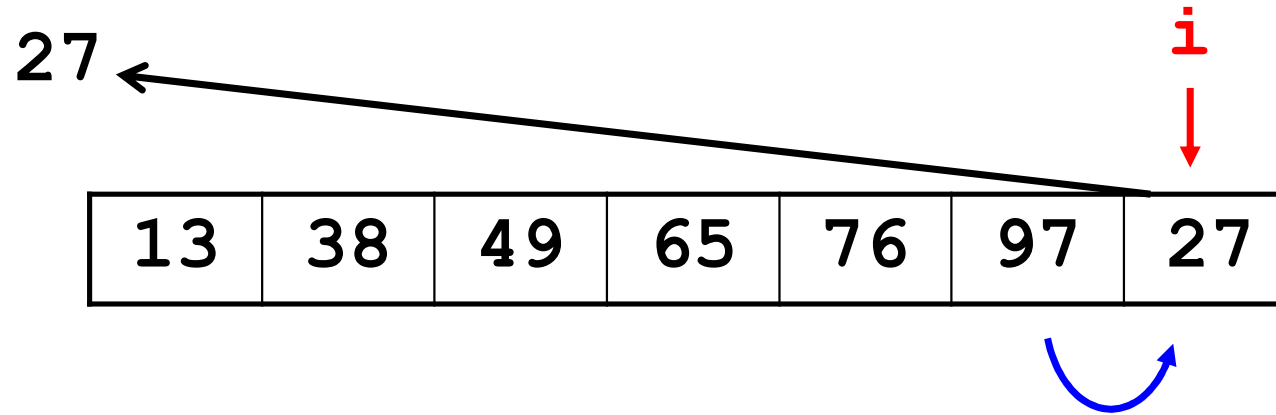


直接插入排序：基本思想

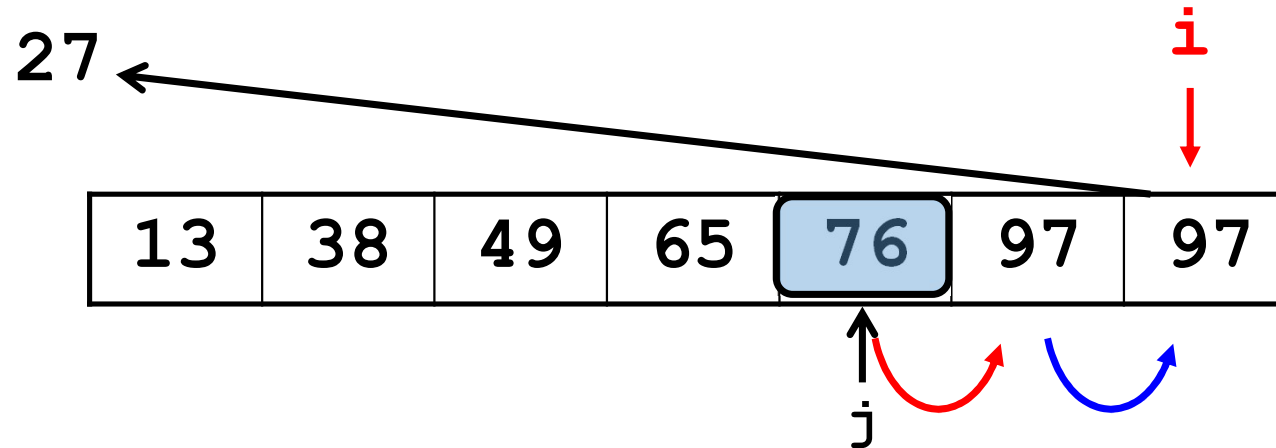
• 例

27

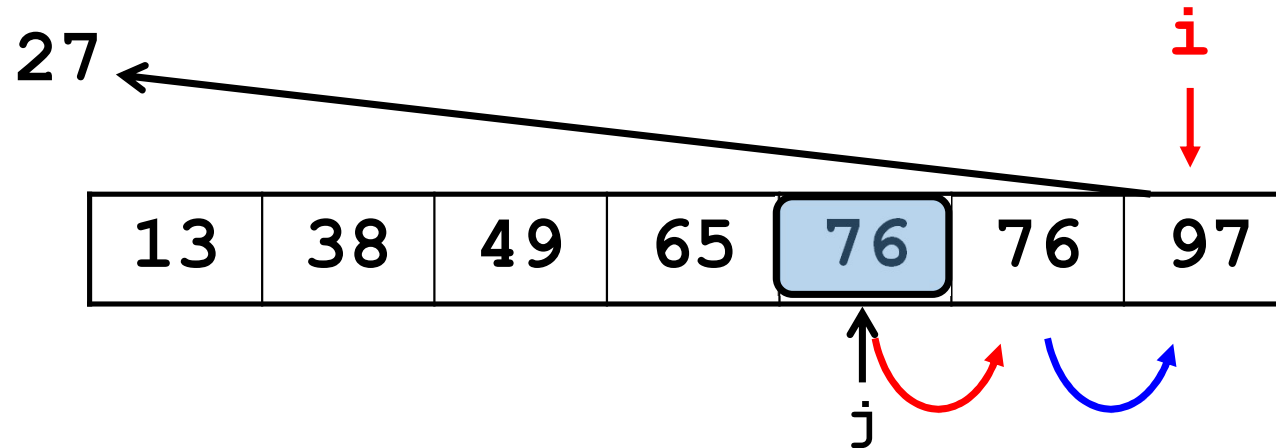




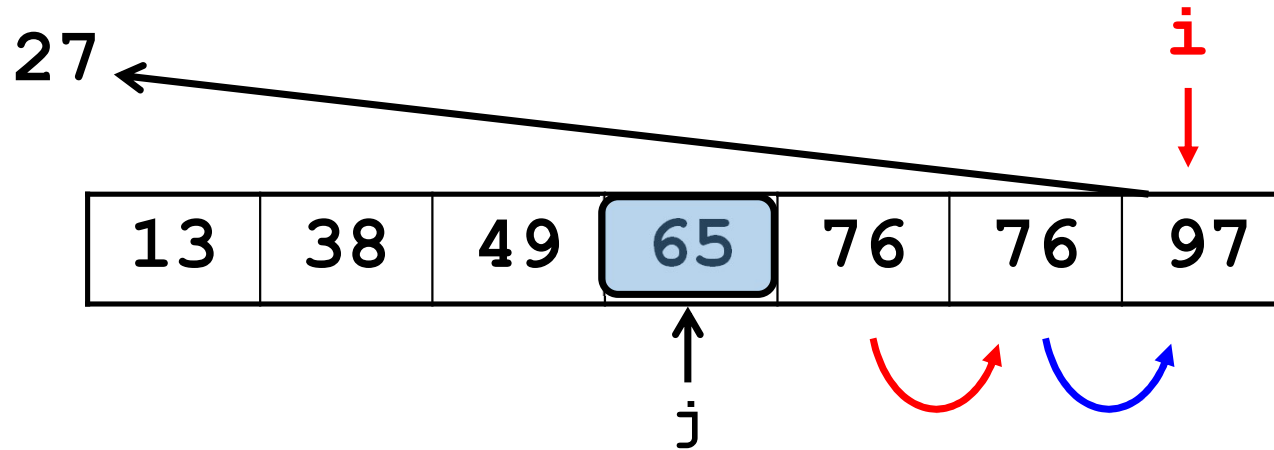
```
if(a[i]<a[i-1]){  
    T t  = a[i];  
    a[i] = a[i-1]; //后移  
  
}
```



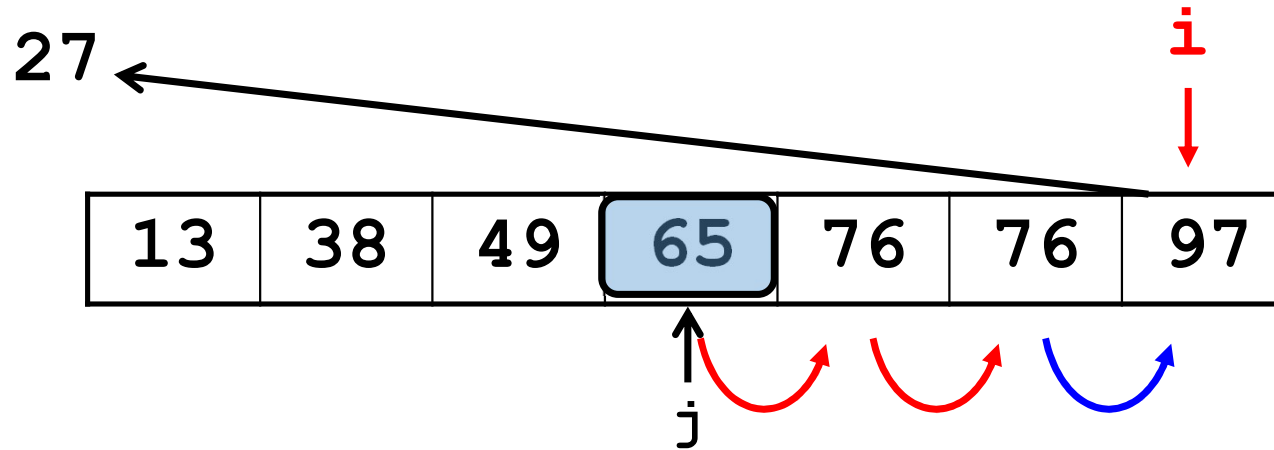
```
if(a[i]<a[i-1]){  
    T t  = a[i];  
    a[i] = a[i-1]; //后移  
    for(j=i-2; j>=0&& t<a[j];    )  
        a[j+1] = a[j]; //a[j]后移  
}
```



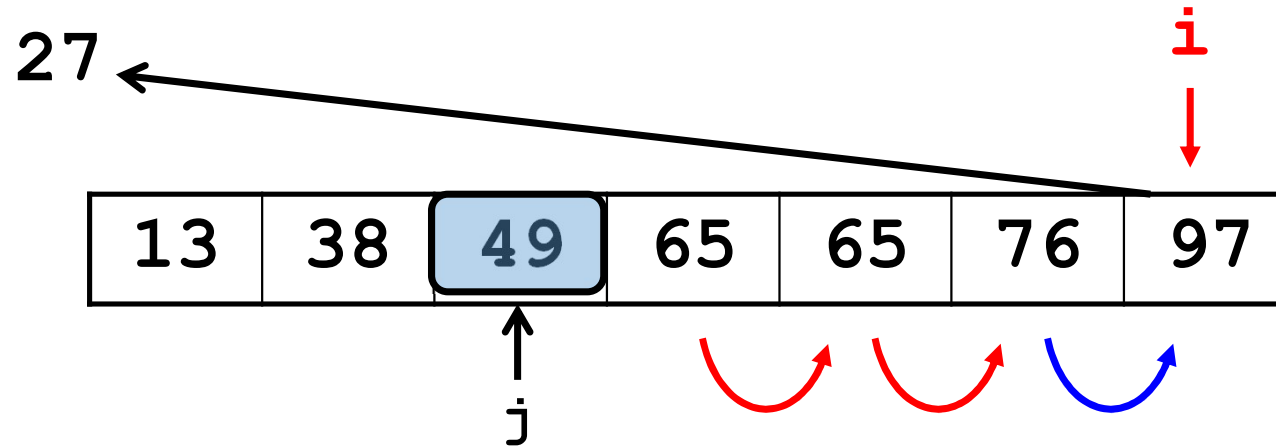
```
if(a[i]<a[i-1]){  
    T t  = a[i];  
    a[i] = a[i-1]; //后移  
    for(j=i-2; j>=0&& t<a[j]; j--)  
        a[j+1] = a[j]; //a[j]后移  
}
```



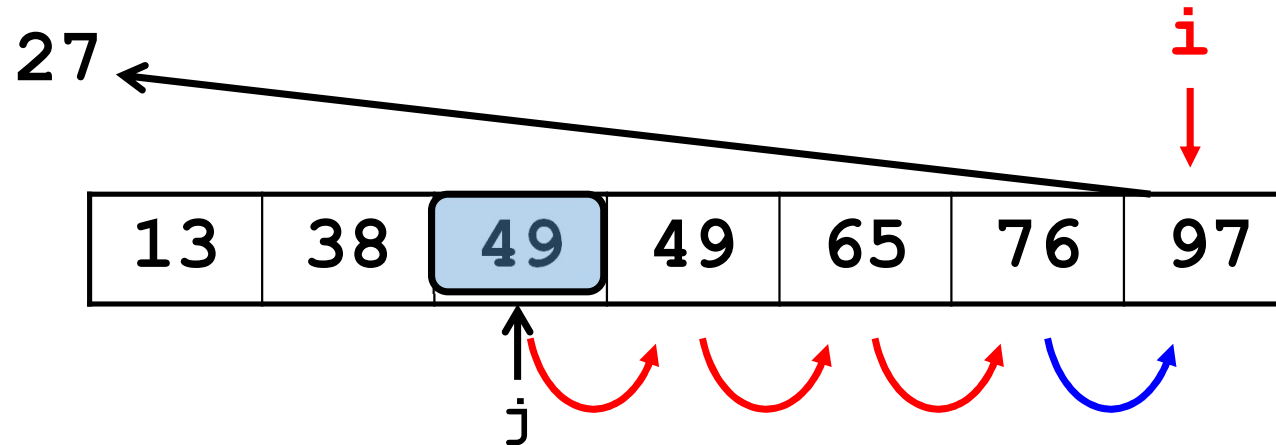
```
if(a[i]<a[i-1]){  
    T t  = a[i];  
    a[i] = a[i-1]; //后移  
    for(j=i-2; j>=0&& t<a[j]; j--)  
        a[j+1] = a[j]; //a[j]后移  
}
```



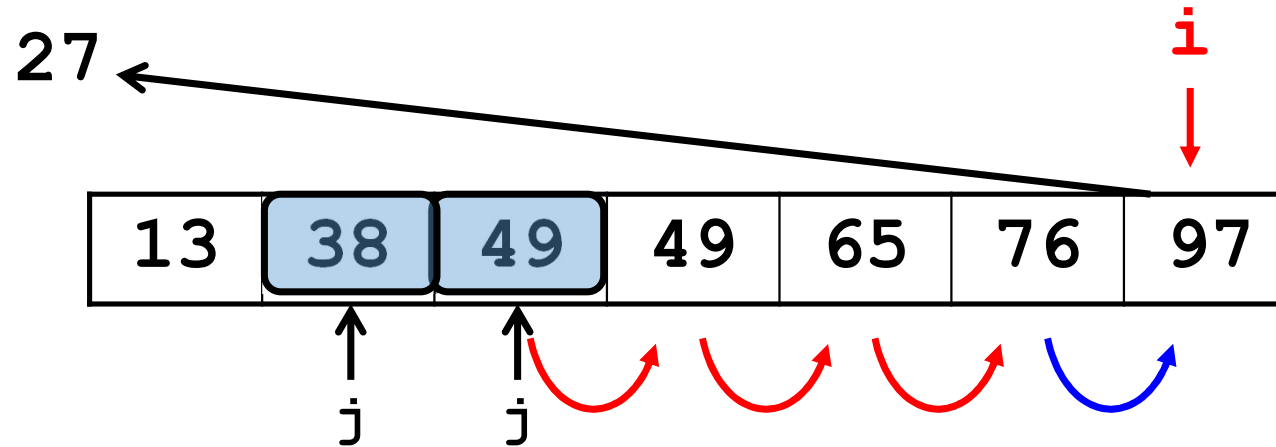
```
if(a[i]<a[i-1]){  
    T t  = a[i];  
    a[i] = a[i-1]; //后移  
    for(j=i-2; j>=0&& t<a[j]; j--)  
        a[j+1] = a[j]; //a[j]后移  
}
```

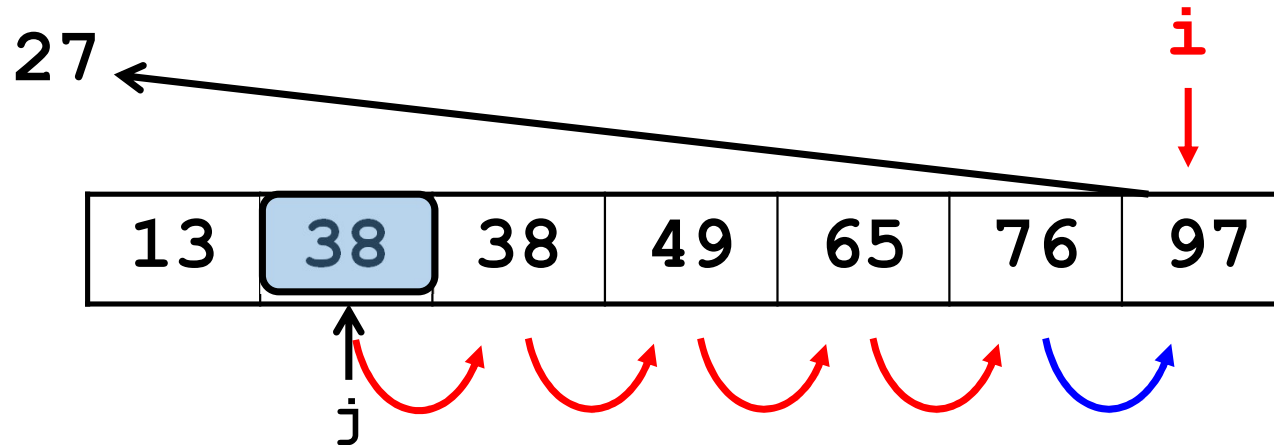
```
if(a[i]<a[i-1]){  
    T t  = a[i];  
    a[i] = a[i-1]; //后移  
    for(j=i-2; j>=0&&t<a[j]; j--)  
        a[j+1] = a[j]; //a[j]后移  
}
```



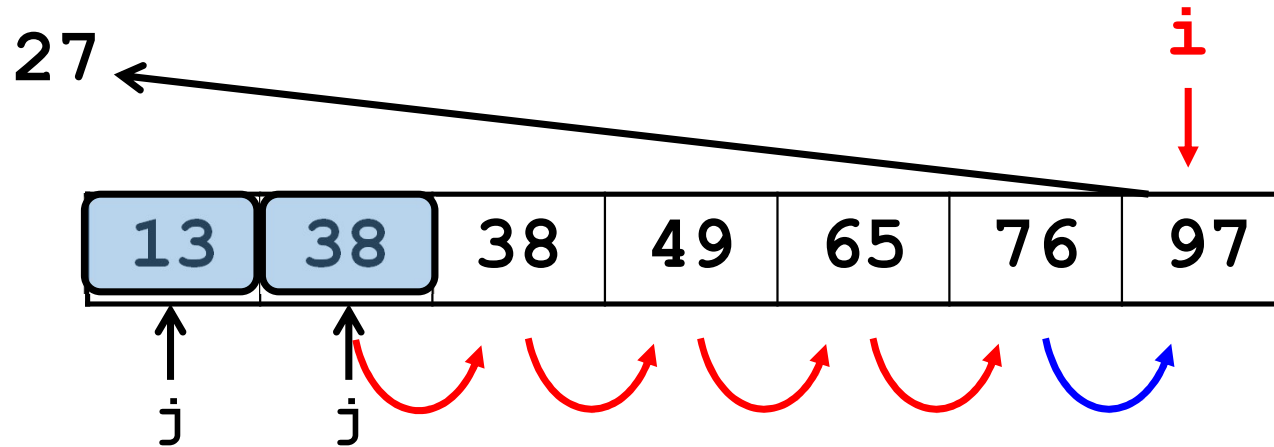
```
if(a[i]<a[i-1]){  
    T t  = a[i];  
    a[i] = a[i-1]; //后移  
    for(j=i-2; j>=0&& t<a[j];  j--)  
        a[j+1] = a[j]; //a[j]后移  
}
```



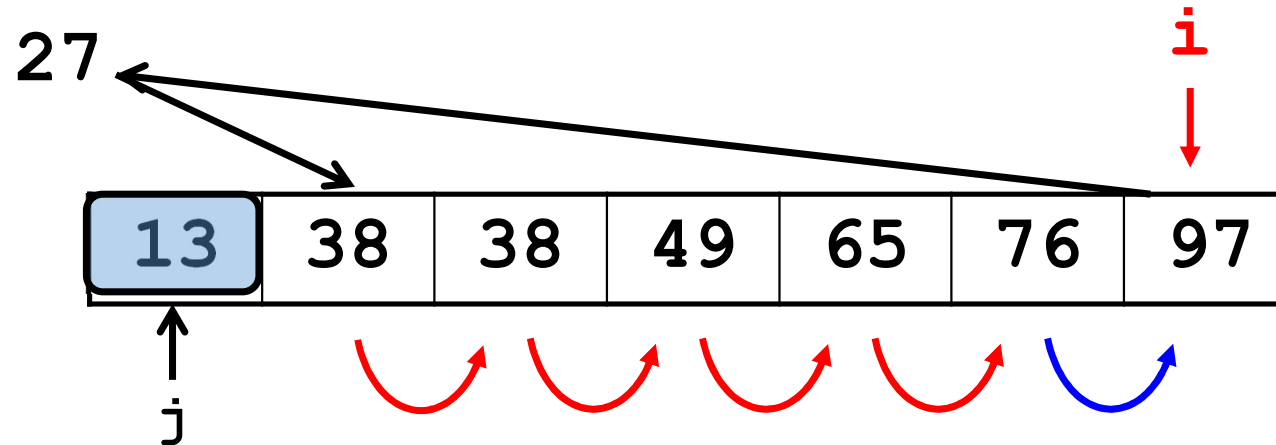
```
if(a[i]<a[i-1]){  
    T t  = a[i];  
    a[i] = a[i-1]; //后移  
    for(j=i-2; j>=0&& t<a[j];  j--)  
        a[j+1] = a[j];  //a[j]后移  
}
```



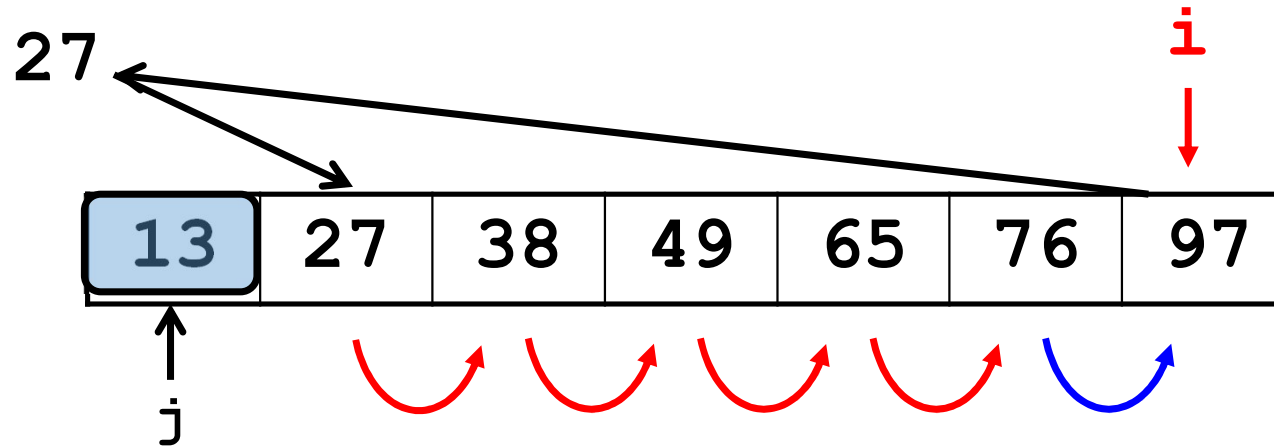
```
if(a[i]<a[i-1]){  
    T t  = a[i];  
    a[i] = a[i-1]; //后移  
    for(j=i-2; j>=0&& t<a[j]; j--)  
        a[j+1] = a[j]; //a[j]后移  
}
```



```
if(a[i]<a[i-1]){  
    T t = a[i];  
    a[i] = a[i-1]; //后移  
    for(j=i-2; j>=0&&t<a[j]; j--)  
        a[j+1] = a[j]; //a[j]后移  
}
```



```
if(a[i]<a[i-1]){  
    T t  = a[i];  
    a[i] = a[i-1]; //后移  
    for(j=i-2; j>=0&& t<a[j];  j--)  
        a[j+1] = a[j];  //a[j]后移  
    a[j+1] = t;  
}
```



```
if (a[i]<a[i-1]){
```

```
    T t = a[i];
```

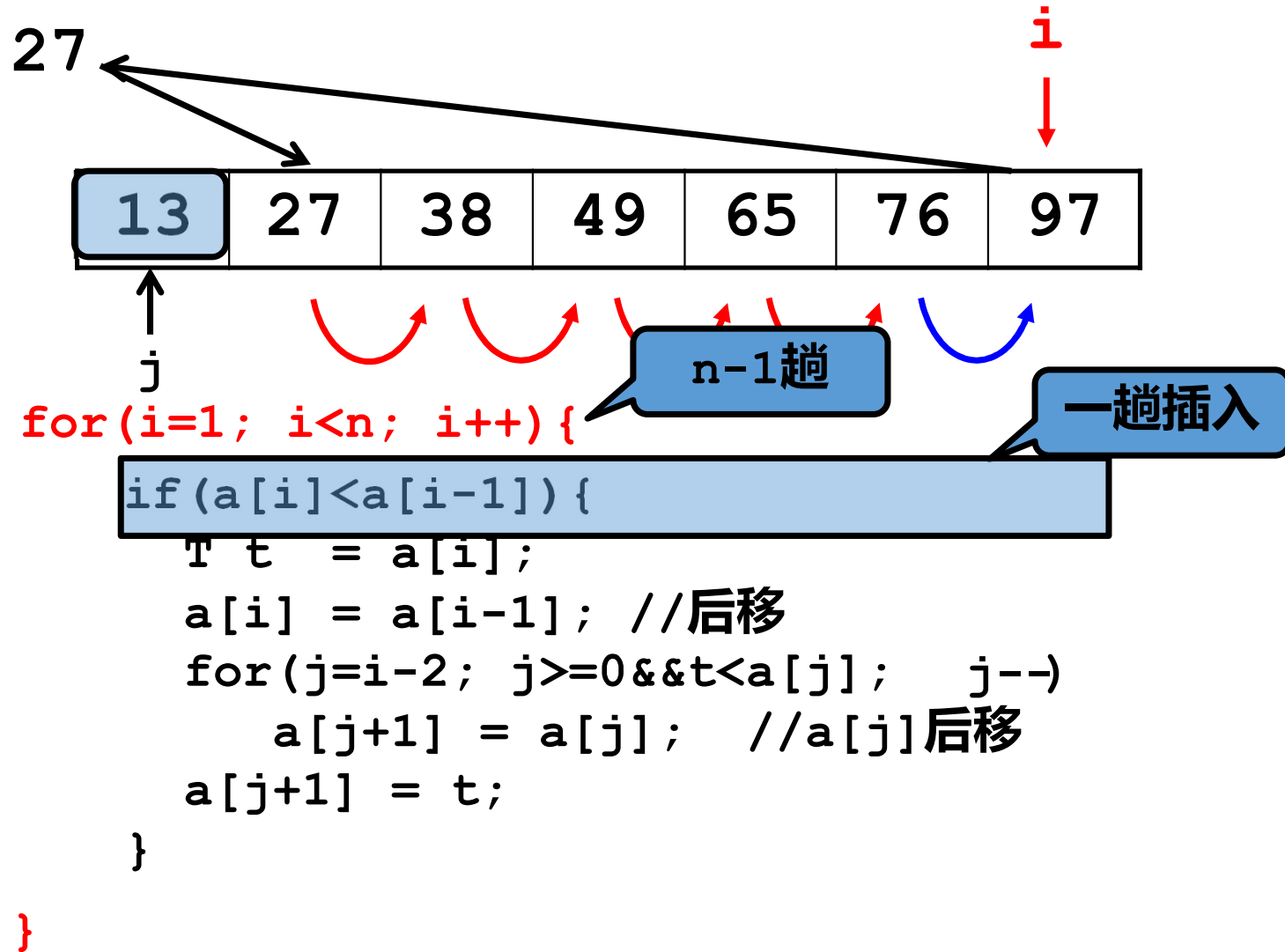
```
    a[i] = a[i-1]; //后移
```

```
    for(j=i-2; j>=0&& t<a[j]; j--)
```

```
        a[j+1] = a[j]; //a[j]后移
```

```
    a[j+1] = t;
```

```
}
```




```
void insert_sort(T a[], int n){
```

```
    for(i=1; i<n; i++){
```

n-1趟

一趟插入

```
        if(a[i]<a[i-1]){
```

```
            T t = a[i];
```

```
            a[i] = a[i-1]; //后移
```

```
            for(j=i-2; j>=0&& t<a[j]; j--)
```

```
                a[j+1] = a[j]; //a[j]后移
```

```
            a[j+1] = t;
```

```
        }
```

```
    }
```

```
}
```

时间复杂度

$$3+4+\dots+(n+1) = (n+4)(n-1)/2: O(n^2)$$

```
void insert_sort(T a[], int n){
```

```
    for(i=1; i<n; i++){
```

n-1趟

一趟插入

```
        if(a[i]<a[i-1]){
```

i+2次
移动

```
            T t = a[i];
```

```
            a[i] = a[i-1]; //后移
```

```
            for(j=i-2; j>=0&& t<a[j]; j--)
```

i-1次
比较

```
                a[j+1] = a[j]; //a[j]后移
```

```
            a[j+1] = t;
```

```
        }
```

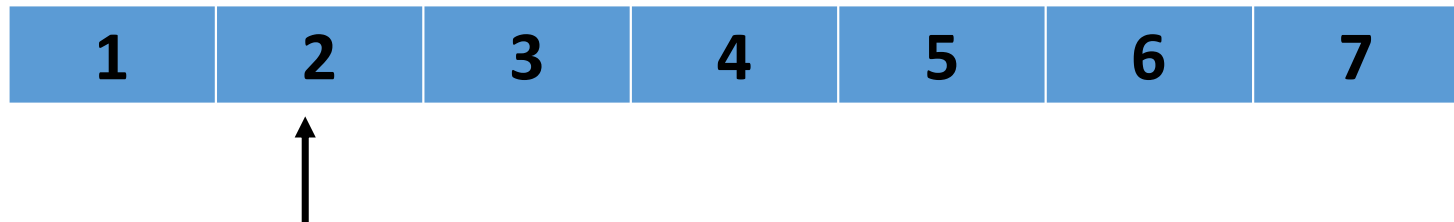
```
    }
```

```
}
```

时间复杂度

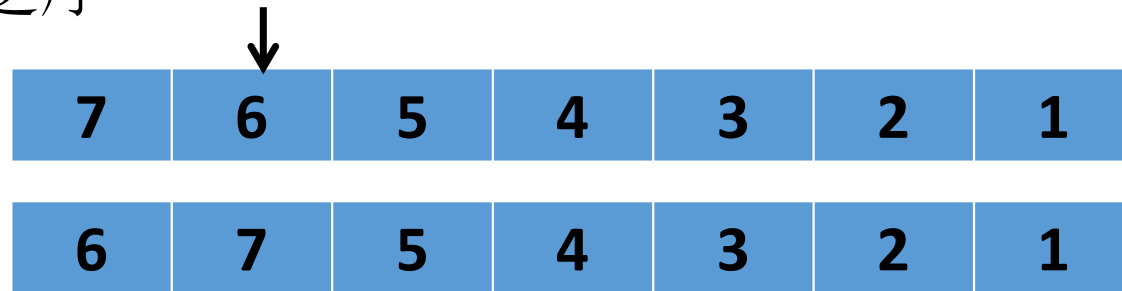
最好的情况： $O(n)$

数列已经排好序，只需要比较 $n-1$ 次



时间复杂度

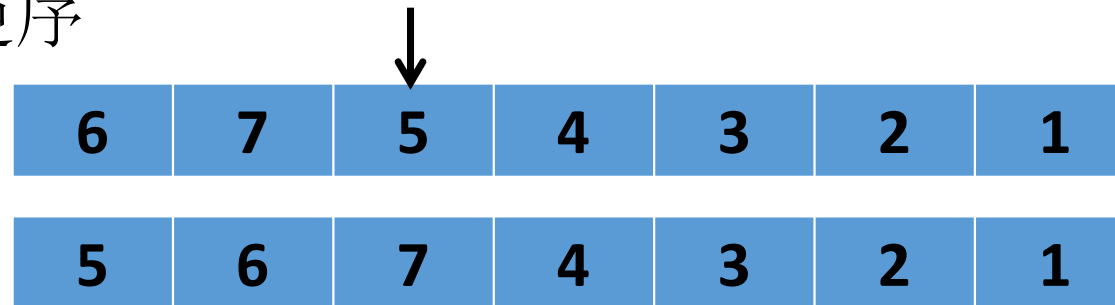
- 最差的情况: $O(n^2)$
排序之前是逆序



时间复杂度

最差的情况： $O(n^2)$

排序之前是逆序



$n-1$ 次插入，每次都应插入到最左

而每次插入从最右开始扫描直到最左，需要比较 i 次

所以总的比较次数 =
$$\sum_{i=2}^n i = (n+2)(n-1) / 2$$

时间复杂度

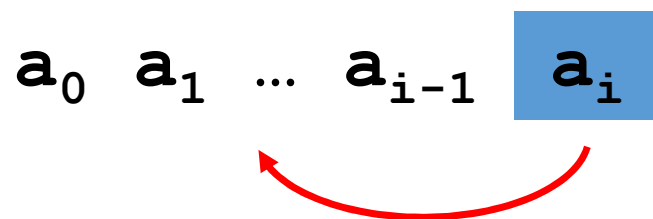
- 平均情况： $O(n^2)$
 $n-1$ 次插入，每次插入从最右开始扫描直到遇见一个比当前元素小的，平均需要比较 $i/2$ 次
- 空间复杂度
 $O(1)$
 一个数据元素的辅助空间
- 稳定性：
 稳定

折半插入排序

是对直接插入排序的改进

在查找插入位置时，直接插入排序是采用
顺序查找法

折半插入排序采用折半查找法（即二分法）



l ↓		m ↓		h ↓	i ↓	
0	1	2	3	4	5	6
<hr/>						
13	38	65	76	97	27	49

l ↓		m ↓		h ↓	i ↓	
0	1	2	3	4	5	6
<hr/>						
13	38	65	76	97	27	49

lm ↓ ↓	h ↓				i ↓	
0	1	2	3	4	5	6
<hr/>						
13	38	65	76	97	27	49

l ↓		m ↓		h ↓	i ↓	
0	1	2	3	4	5	6
13	38	65	76	97	27	49

lm ↓ ↓	h ↓				i ↓	
0	1	2	3	4	5	6
13	38	65	76	97	27	49

	lmh ↓ ↓ ↓				i ↓	
0	1	2	3	4	5	6
13	38	65	76	97	27	49

l ↓ 0	1	m ↓ 2	3	h ↓ 4	i ↓ 5	6
13	38	65	76	97	27	49

l ↓ 0	m ↓ 1	h ↓ 2	3	4	i ↓ 5	6
13	38	65	76	97	27	49

0	l ↓ 1	m ↓ 2	h ↓ 3	4	i ↓ 5	6
13	38	65	76	97	27	49

h ↓ 0	l ↓ 1	m ↓ 2	3	4	i ↓ 5	6
13	38	65	76	97	27	49

$h+1$ 就是插入位置



$\log_2 n$

$h+1$ 就是插入位置

l ↓		m ↓			h ↓	i ↓
0	1	2	3	4	5	6
<hr/>						
13	27	38	65	76	97	49
			l ↓	m ↓	h ↓	i ↓
0	1	2	3	4	5	6
<hr/>						
13	27	38	65	76	97	49
			m ↓	h ↓		i ↓
0	1	2	3	4	5	6
<hr/>						
13	27	38	65	76	97	49
		h ↓	l ↓			i ↓
0	1	2	3	4	5	6
<hr/>						
13	27	38	65	76	97	49

```

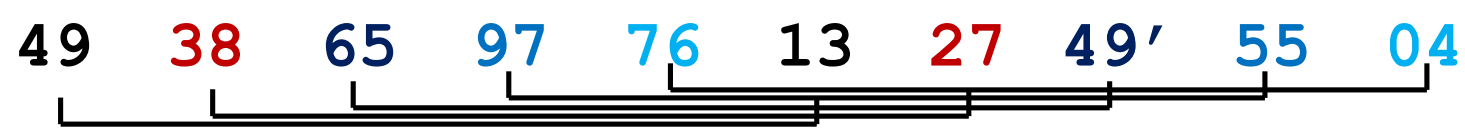
void binary_insert_sort(T a[],int n) {
    int i,j,low,high,m; T t;
    for( i=1; i<n; ++i ) {
        t = a[i]; /* 将a[i]暂存到a[0] */
        l = 0; h = i-1;
        while(l<=h) {
            /*在a[l...h]中折半查找插入的位置*/
            m = (l+h)/2; /* 折半 */
            if ( t<a[m] )
                h = m-1; /* 插入点在低半区*/
            else l = m+1; /* 插入点在高半区*/
        }
        for( j=i-1; j>=h+1; --j )
            a[j+1] = a[j]; /* 记录后移 */
        a[h+1] = t; /* 插入*/
    }
}

```

希尔排序

希尔排序

- 希尔排序：又称缩小增量排序
- 基本思想：
 - 设待排序列有 n 个元素，取一整数 gap ($gap < n$) 作为间隔，将全部元素分为 gap 个子序列，所有距离为 gap 的元素放在同一个子序列中
 - 对每一个子序列分别（直接插入）排序
 - 然后缩小间隔 gap ，例如取 $gap = gap/2$ ，重复上述的子序列划分和排序工作
 - 直到最后取 $gap = 1$ ，将所有元素放在同一个序列中排序为止




49 38 65 97 76 13 27 49' 55 04 gap=5

A horizontal sequence of numbers: 49, 38, 65, 97, 76, 13, 27, 49', 55, 04, followed by the text gap=5. A black bracket is positioned below the first five numbers (49, 38, 65, 97, 76), spanning from the start of '49' to the end of '76'.

13 38 65 97 76 49 27 49' 55 04 gap=5

A horizontal sequence of numbers: 13, 38, 65, 97, 76, 49, 27, 49', 55, 04, followed by the text gap=5. A black bracket is positioned below the first five numbers (13, 38, 65, 97, 76), spanning from the start of '13' to the end of '76'.

13 38 65 97 76 49 27 49' 55 04 gap=5



13 27 65 97 76 49 38 49' 55 04 gap=5



13 38 65 97 76 49 27 49' 55 04 gap=5



The diagram shows a sequence of numbers: 13, 38, 65, 97, 76, 49, 27, 49', 55, 04. A horizontal bracket is positioned below the last five numbers (65, 97, 76, 49, 27). To the right of the sequence, the text 'gap=5' is displayed. The numbers are color-coded: 13 is black, 38 is red, 65 is dark blue, 97 is blue, 76 is light blue, 49 is black, 27 is red, 49' is dark blue, 55 is blue, and 04 is light blue.

13 38 49' 97 76 49 27 65 55 04 gap=5

A horizontal sequence of numbers: 13, 38, 49', 97, 76, 49, 27, 65, 55, 04. A black bracket is positioned below the last five numbers (49', 97, 76, 49, 27). To the right of the sequence, the text 'gap=5' is displayed.

13 38 49' 97 76 49 27 65 55 04 gap=5



The diagram shows a sequence of numbers: 13, 38, 49', 97, 76, 49, 27, 65, 55, 04. A horizontal bracket is positioned below the numbers 97, 76, 49, 27, and 65. To the right of the bracket, the text 'gap=5' is displayed. The numbers are color-coded: 13 is black, 38 is red, 49' is dark blue, 97 is light blue, 76 is light blue, 49 is black, 27 is red, 65 is dark blue, 55 is light blue, and 04 is light blue.

13 38 49' 55 76 49 27 65 97 04 gap=5



The diagram shows a sequence of numbers: 13, 38, 49', 55, 76, 49, 27, 65, 97, 04. A horizontal bracket is positioned below the numbers 55, 76, 49, 27, and 65. To the right of the bracket, the text 'gap=5' is displayed.

Index	Value
1	13
2	38
3	49'
4	55
5	76
6	49
7	27
8	65
9	97
10	04

13 38 49' 55 76 49 27 65 97 04 gap=5

A horizontal sequence of numbers: 13, 38, 49', 55, 76, 49, 27, 65, 97, 04. The numbers 38, 49', 55, 76, 97, and 04 are colored blue, while 13, 49, and 27 are black. A black bracket is positioned below the numbers 76, 49, 27, 65, and 97, spanning from the start of 76 to the end of 97.

13 38 49' 55 04 49 27 65 97 76 gap=5



The diagram shows a sequence of numbers: 13, 38, 49', 55, 04, 49, 27, 65, 97, 76. A horizontal bracket is positioned below the last six numbers (04, 49, 27, 65, 97, 76). To the right of the bracket, the text 'gap=5' is displayed. The numbers are color-coded: 13 is black, 38 is red, 49' is dark blue, 55 is blue, 04 is light blue, 49 is black, 27 is red, 65 is dark blue, 97 is blue, and 76 is light blue.

13 38 49' 55 04 49 27 65 97 76



gap=3

13 38 49' 27 04 49 55 65 97 76



gap=3

13 38 49' 27 04 49 55 65 97 76 gap=3



A horizontal sequence of numbers: 13, 38, 49', 27, 04, 49, 55, 65, 97, 76, followed by the text 'gap=3'. A horizontal bracket is drawn below the numbers, starting at the vertical position of '38' and ending at the vertical position of '65'. There is a small vertical tick mark on the bracket line at the vertical position of '04'.

13 04 49' 27 38 49 55 65 97 76 gap=3



The diagram shows a sequence of numbers: 13, 04, 49', 27, 38, 49, 55, 65, 97, 76. A bracket is drawn below the sequence, starting at 04 and ending at 65. A vertical tick mark is placed at 38. The gap between 65 and 97 is labeled as 3.

13 04 49' 27 38 49 55 65 97 76 gap=3



A horizontal line with vertical end caps at the positions of '49'' and '97'. A vertical tick mark is also present at the position of '49'.

13 04 49' 27 38 49 55 65 97 76 gap=1

04 13 27 38 49' 49 55 65 76 97 gap=1

回顾：直接插入排序

49 38 65 97 76 13 27 49' 55 04

↑

i=2

```
for(int i = 2; i<n;i++){  
    //将i插入到a[0,...,i-1]的有序序列中  
    if(a[i]<a[i-1]){  
        }  
    }  
}
```

回顾：直接插入排序

49 38 65 97 76 13 27 49' 55 04

↑
i=3

```
for(int i = 2; i<n;i++){  
    //将i插入到a[0,...,i-1]的有序序列中  
    if(a[i]<a[i-1]){  
        }  
    }  
}
```

49 38 65 97 76 13 27 49' 55 04 gap=5

↑ ↑

i-d i=d

```
for(int i = d; i<n; i++){
    //将i插入到a[,...,i-d]的有序序列中
    if(a[i]<a[i-d]){
        }
    }
}
```

49 38 65 97 76 13 27 49' 55 04 gap=5

 ↑ ↑

 i-d i

```
for(int i = d; i<n; i++){  
    //将i插入到a[,...,i-d]的有序序列中  
    if(a[i]<a[i-d]){  
        }  
    }  
}
```

49 38 65 97 76 13 27 49' 55 04 gap=5

 ↑ ↑

 i-d i

```
for(int i = d; i<n; i++){  
    //将i插入到a[,...,i-d]的有序序列中  
    if(a[i]<a[i-d]){  
        }  
    }  
}
```

.... a_{i-2} a_{i-1} a_i

直接插入

.... a_{i-2d} a_{i-d} a_i

希尔的直接插入

13 27 49' 55 04 49 38 65 97 76



```
void insert_sort( T a[], int n) {  
    int i,j; T t;  
    for(i = 1; i < n; i++ )  
        if( a[i]< a[i-1]){          //i比i-1小  
            t = a[i];              //用t先保存a[i]的值  
            a[i] = a[i-1];         //a[i-1]后移一个单元  
  
            //从i-2开始, 往左扫描, 直到找到一个<=t的  
            for(j=i-2; j>=0&&(t<a[j]); j--)  
                a[j+1] = a[j];     //每个元素后移  
  
            a[j+1] = t;            //最后把t写入j+1位置  
        }  
}
```




```

void shell_pass( T a[], int n,int d) {
    int i,j; T t;
    for(i = d; i < n; i++ )
        if( a[i] < a[i-d])){          //i比i-d小
            t = a[i];                //用t先保存a[i]的值
            a[i] = a[i-d];           //a[i-d]后移一个单元

            //从i-2*d开始, 往左扫描, 直到找到一个<=t的
            for(j=i-2*d; j>=0&&(t<a[j]); j-=d)
                a[j+d] = a[j];        //每个元素后移

            a[j+d] = t;                //最后把t写入j+1位置
        }
}

```



```

void shell_pass( T a[], int n, int d) {
    int i, j; T t;
    for(i = d; i < n; i++) {
        if( a[i] < a[i-d]) {           //i比i-d小
            t = a[i];                 //用t先保存a[i]的值
            a[i] = a[i-d];           //a[i-d]后移一个单元

            //从i-2*d开始, 往左扫描, 直到找到一个<=t的
            for(j=i-2*d; j>=0&&(t<a[j]); j-=d)
                a[j+d] = a[j];       //每个元素后移

            a[j+d] = t;               //最后把t写入j+1位置
        }
    }
}

```

13 38 65 97 76 39 27 49' 55 04

```

void shell_pass( T a[], int n,int d) {
    int i,j; T t;
    for(i = d; i < n; i++) {
        if( a[i] < a[i-d]) {           //i比i-d小
            t = a[i];                 //用t先保存a[i]的值
            a[i] = a[i-d];            //a[i-d]后移一个单元

            //从i-2*d开始, 往左扫描, 直到找到一个<=t的
            for(j=i-2*d; j>=0&&(t<a[j]); j-=d)
                a[j+d] = a[j];        //每个元素后移

            a[j+d] = t;                //最后把t写入j+1位置
        }
    }
}

```

13 27 65 97 76 39 38 49' 55 04

```
void shell_sort(T a[],int n){  
    int d[] = {7,5,3,1};  
    for(int i = 0; i<4;i++)  
        shell_pass(a, n, d[i]);  
}
```

希尔排序

- 回顾直接插入排序的特点：
 - 数据大致有序时最快， $O(n)$
- 希尔排序的原理
 - 开始时gap的值较大，子序列中的元素较少，排序速度较快
 - 随着排序进展，gap值逐渐变小，子序列中元素个数变多，但是由于前面工作的基础，大多数元素已基本有序，所以排序速度仍然很快

希尔排序

- Gap的取法
 - 最初Shell提出取 $gap = n/2$, $gap = gap/2$, ..., 直到 $gap = 1$
 - Knuth提出取 $gap = \lfloor gap/3 \rfloor + 1$
 - 还有人提出都取奇数为好
 - 也有人提出各gap互质为好
- Knuth利用大量实验统计资料得出：
 - 当n很大时, 排序码平均比较次数和对象平均移动次数大约在 $n^{1.25}$ 到 $1.6n^{1.25}$ 的范围内
- 稳定性: 不稳定

交换式排序



hwdong

hw-dong

交换式排序

$a_i \dots a_j$ 如果逆序, 则交换之

9 ... 3

3 ... 9

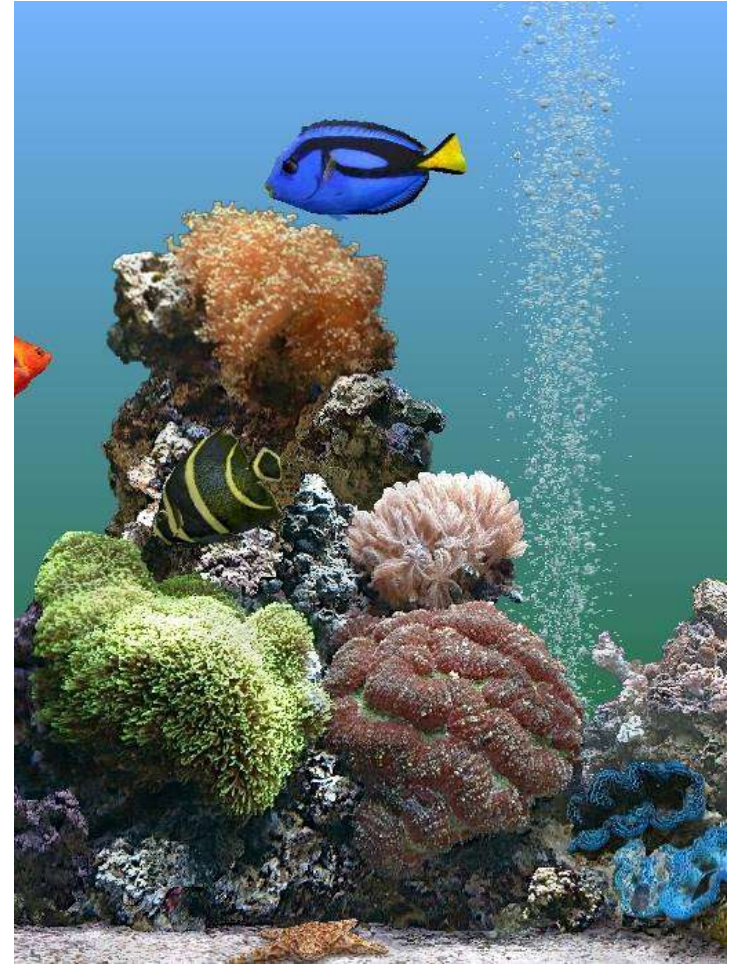
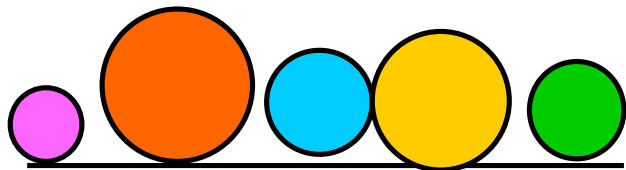
直到所有对象都排好序为止

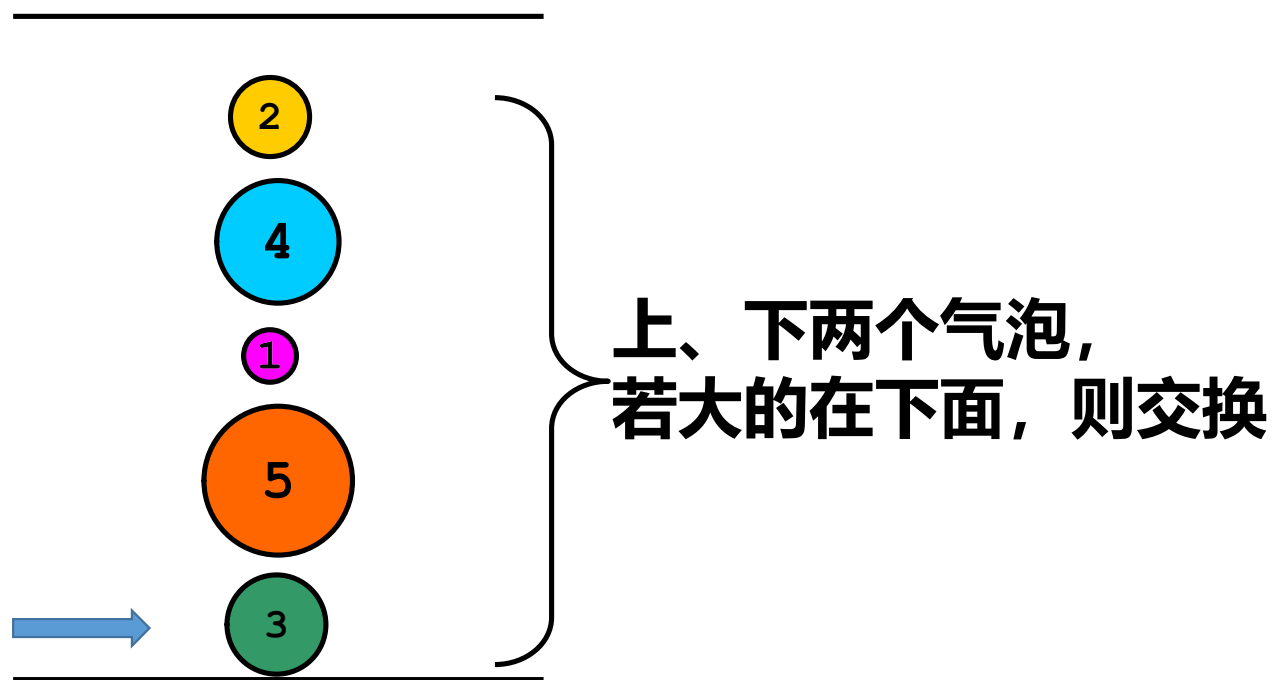
冒泡排序

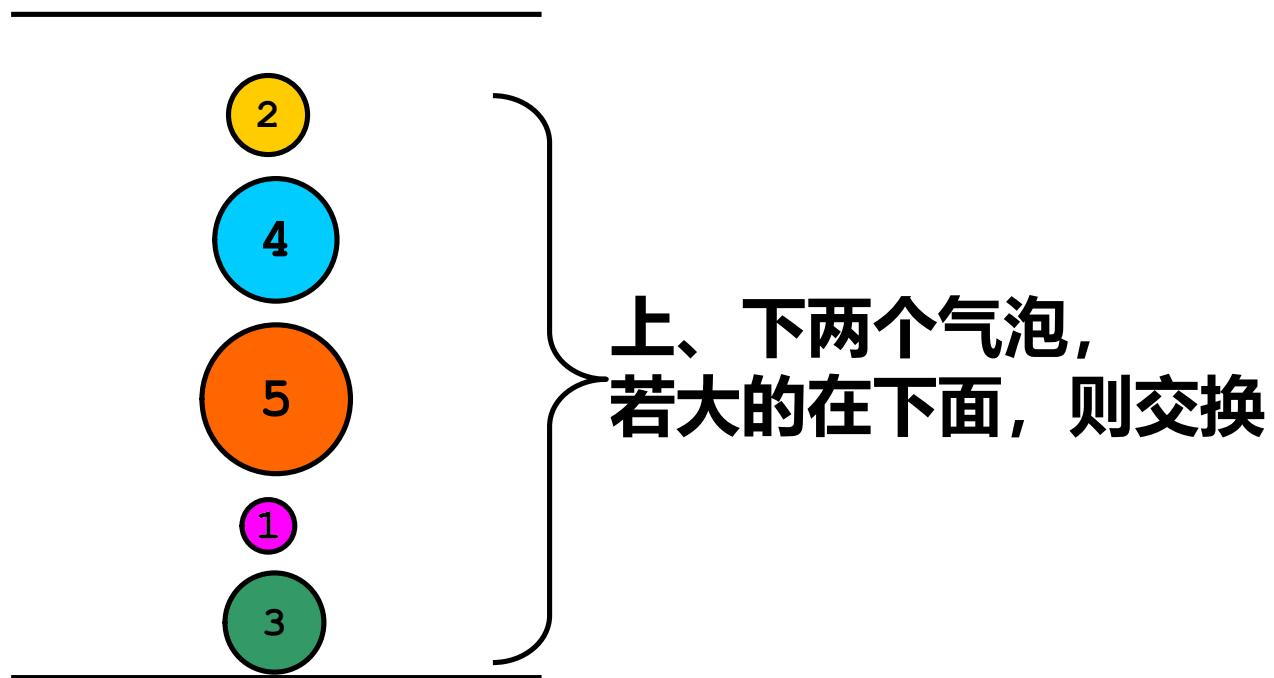
快速排序

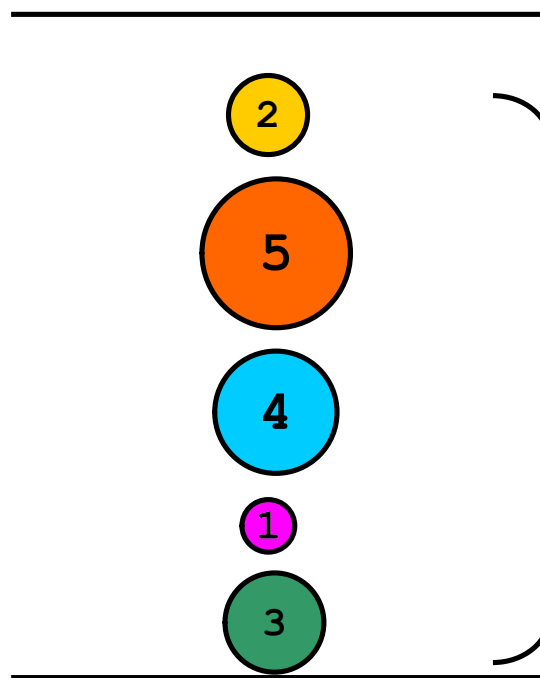
气泡排序法

- 基本思想
 - 水中气泡，大的还是小的上浮更快？

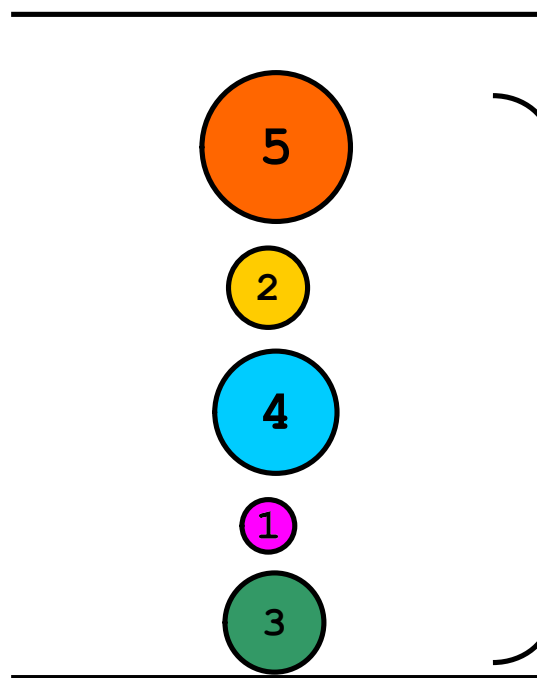




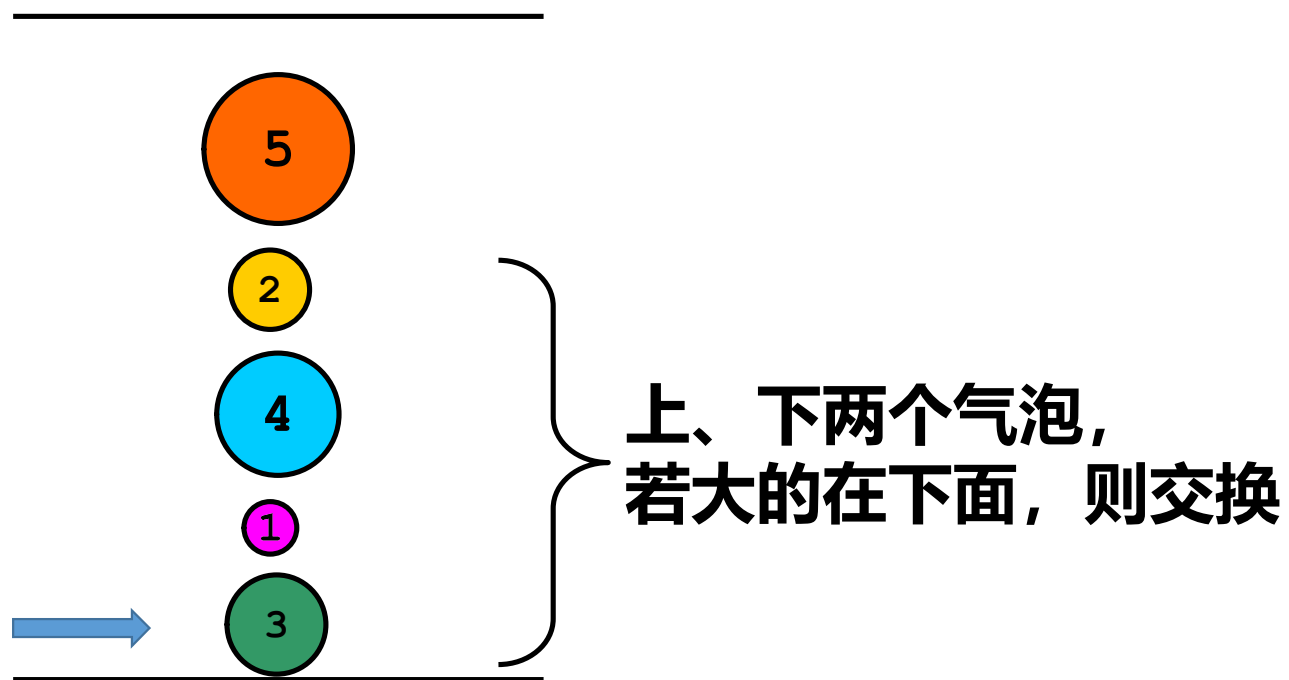


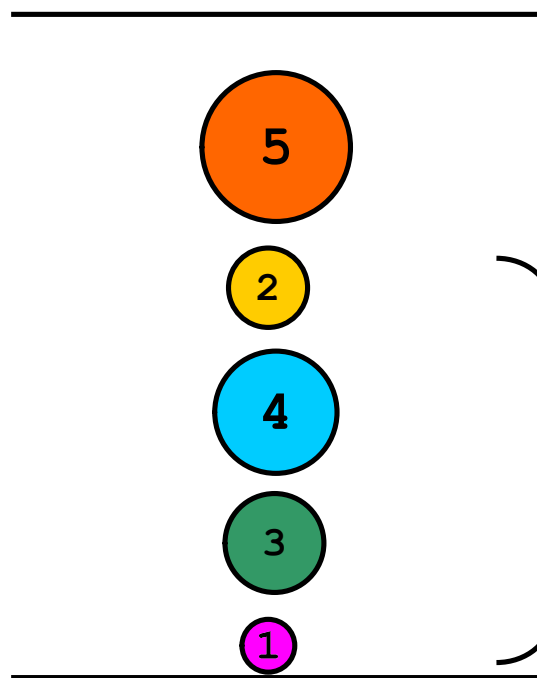


**上、下两个气泡，
若大的在下面，则交换**

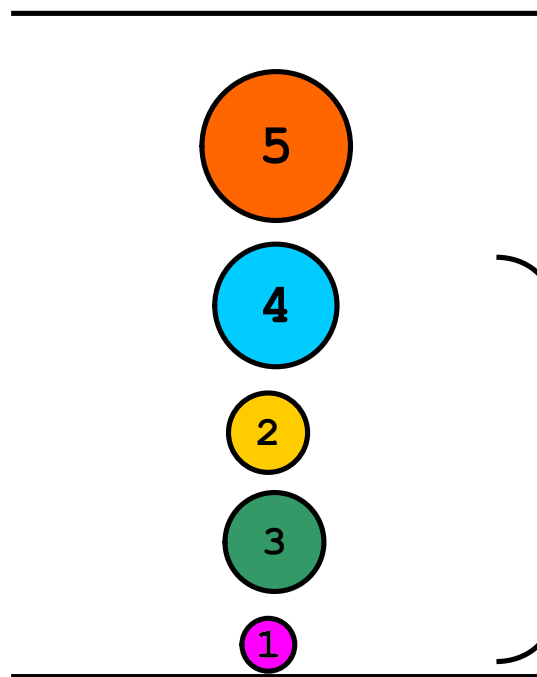


**上、下两个气泡，
若大的在下面，则交换**

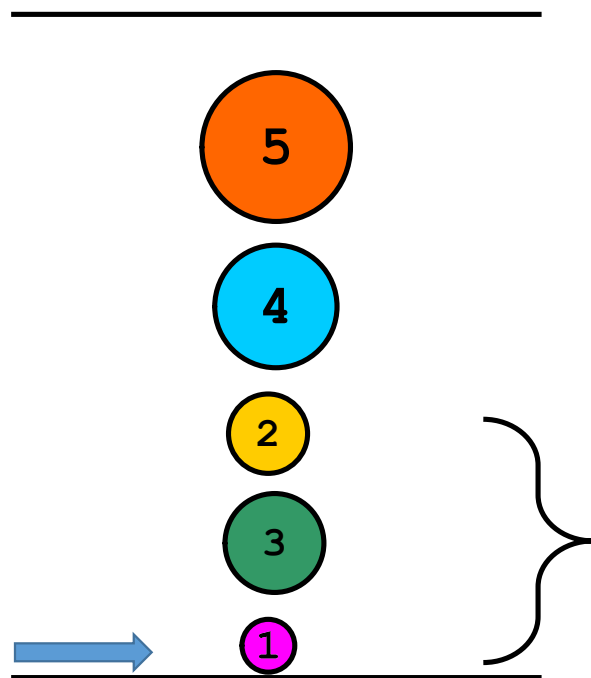




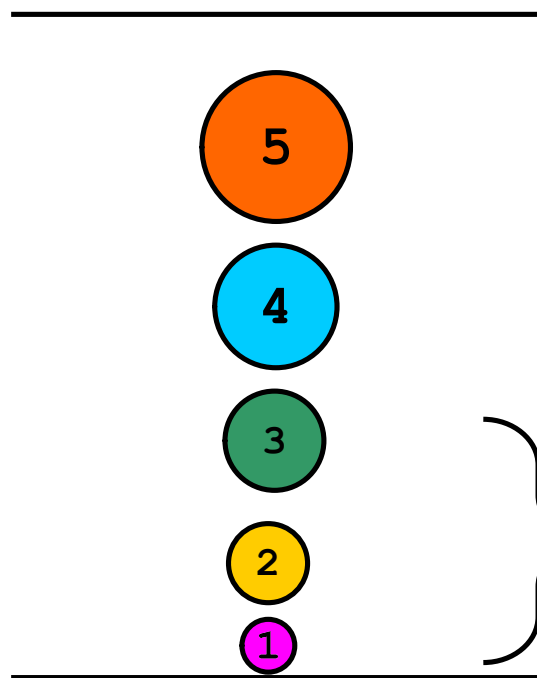
上、下两个气泡，
若大的在下面，则交换



**上、下两个气泡，
若大的在下面，则交换**



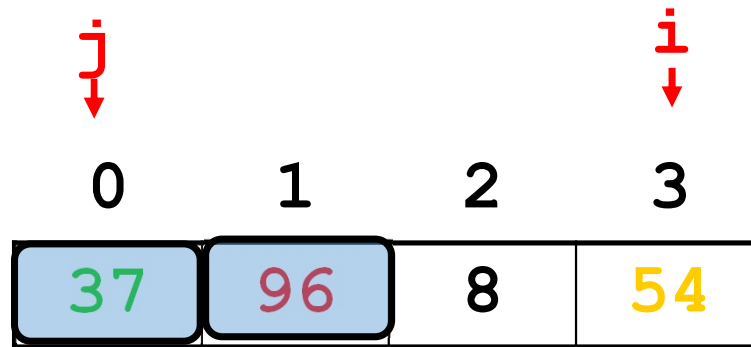
**上、下两个气泡，
若大的在下面，则交换**



**上、下两个气泡，
若大的在下面，则交换**

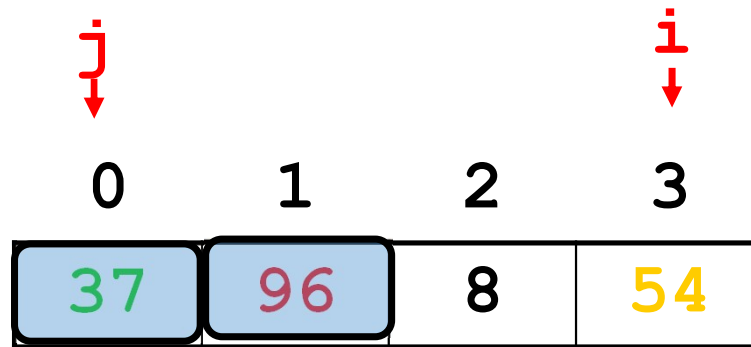
一趟冒泡

- $i=3$ 固定
 - $j=0$
 - $\text{if}(\text{data}[1] < \text{data}[0])$
交换



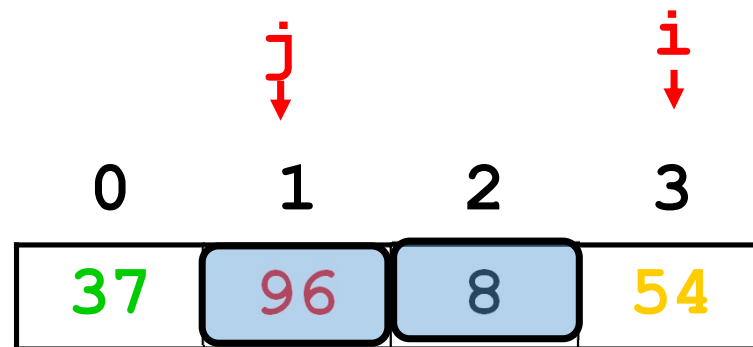
一趟冒泡

- $i=3$ 固定
 - $j=0$
 - $\text{if}(\text{data}[1] < \text{data}[0])$
交换



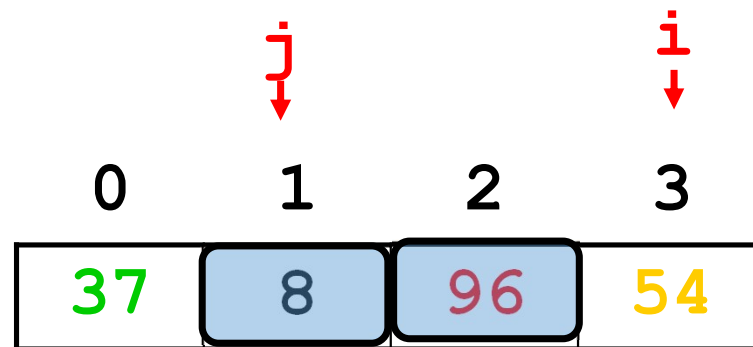
一趟冒泡

- $i=3$ 固定
 - $j=1$
 - $\text{if}(\text{data}[2] < \text{data}[1])$
交换



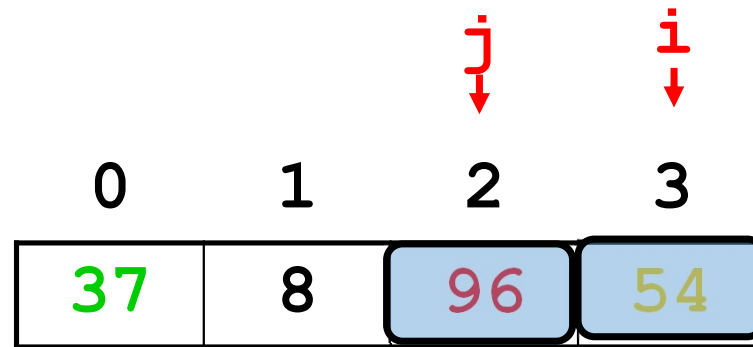
一趟冒泡

- $i=3$ 固定
 - $j=1$
 - $\text{if}(\text{data}[2] < \text{data}[1])$
交换



一趟冒泡

- $i=3$ 固定
 - $j=2$
 - $\text{if}(\text{data}[3] < \text{data}[2])$
交换



一趟冒泡

- $i=3$ 固定

- $j=2$

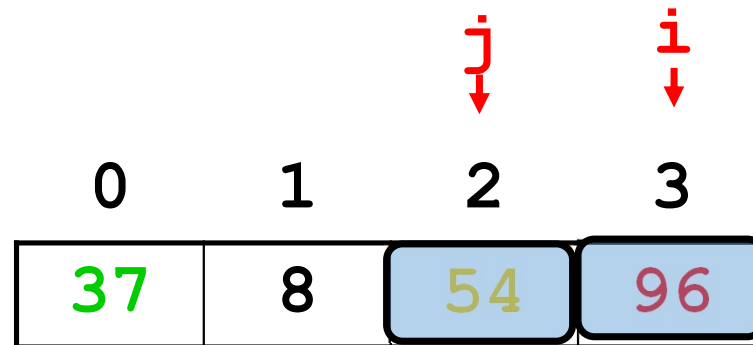
- $\text{if}(\text{data}[3] < \text{data}[2])$

交换

```
for(int j = 0; j < i; j++)
```

```
if (a[j+1] < a[j])
```

```
swap(a[j], a[j+1]);
```



一趟冒泡

- $i=2$ 固定

- $j=0$

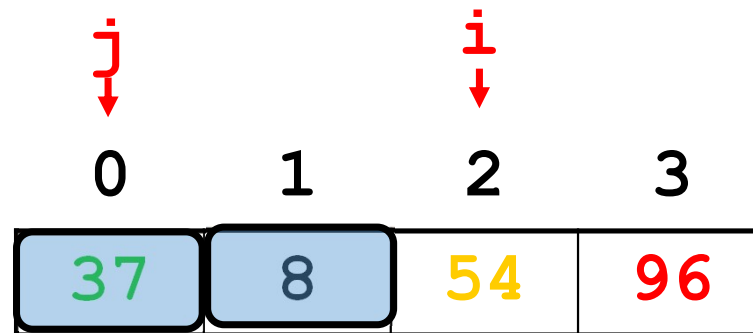
- $\text{if}(\text{data}[1] < \text{data}[0])$

交换

```
for(int j = 0; j < i; j++)
```

```
if (a[j+1] < a[j])
```

```
swap(a[j], a[j+1]);
```



一趟冒泡

- $i=2$ 固定

- $j=0$

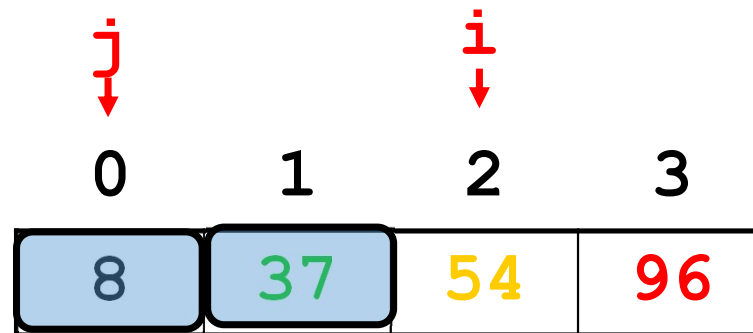
- $\text{if}(\text{data}[1] < \text{data}[0])$

交换

```
for(int j = 0; j < i; j++)
```

```
if (a[j+1] < a[j])
```

```
swap(a[j], a[j+1]);
```



一趟冒泡

- $i=2$ 固定

- $j=1$

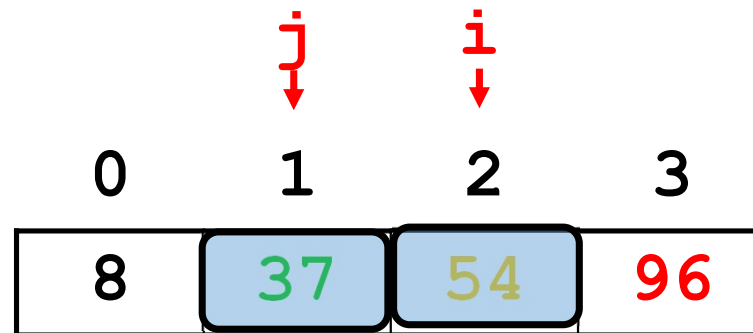
- $\text{if}(\text{data}[1] < \text{data}[0])$

交换

```
for(int j = 0; j < i; j++)
```

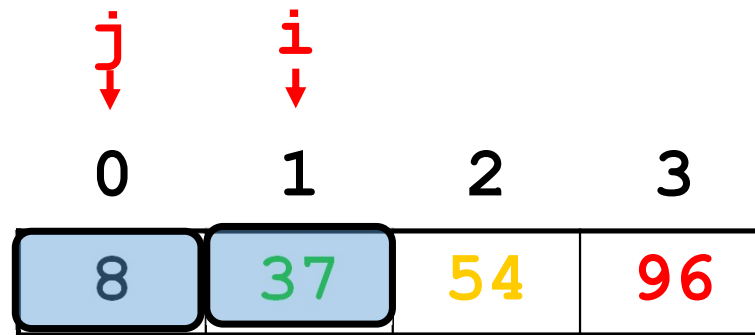
```
if (a[j+1] < a[j])
```

```
swap(a[j], a[j+1]);
```



一趟冒泡

- $i=1$ 固定
 - $j=0$
 - $\text{if}(\text{data}[1] < \text{data}[0])$
交换



冒泡排序

j ↓ 0	1	2	i ↓ 3
2	8	13	26

```
for(int i = n-1; i>0; )
```

```
    for(int j = 0; j < i; j++)
```

```
        if (a[j+1] < a[j])
```

```
            swap(a[j], a[j+1]);
```

冒泡排序

```
void bubble_sort(T a[], int n) {
```

	0	1	2	3
	j		i	
	↓		↓	
	2	8	13	26

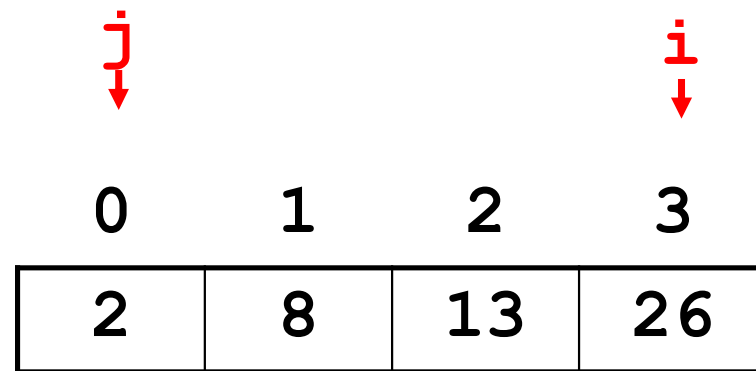
```
    for(int i = n-1; i>0; i--)
```

```
        for(int j = 0; j < i; j++)
```

```
            if (a[j+1] < a[j])
```

```
                swap(&a[j], &a[j+1]);
```

```
    }
```



A diagram illustrating an array structure. The array is represented as a horizontal row of four cells, each containing a number. Above the first cell is the index '0', above the second is '1', above the third is '2', and above the fourth is '3'. A red arrow labeled 'j' points down to the first cell (index 0). Another red arrow labeled 'i' points down to the fourth cell (index 3).

0	1	2	3
2	8	13	26

冒泡排序

```
void bubble_sort(T a[], int n) {  
    for(int i = n-1; i>0; i--){  
        int swaped = 0;  
        for(int j = 0; j < i; j++){  
            if (a[j+1] < a[j]){  
                swap(&a[j], &a[j+1]);  
                swaped = 1;  
            }  
        }  
        if(swaped==0) break;  
    }  
}
```


时间复杂度

- 最差情况 (原始数据是倒序):

i 从 $n-1$ 到 1

交换 i 次

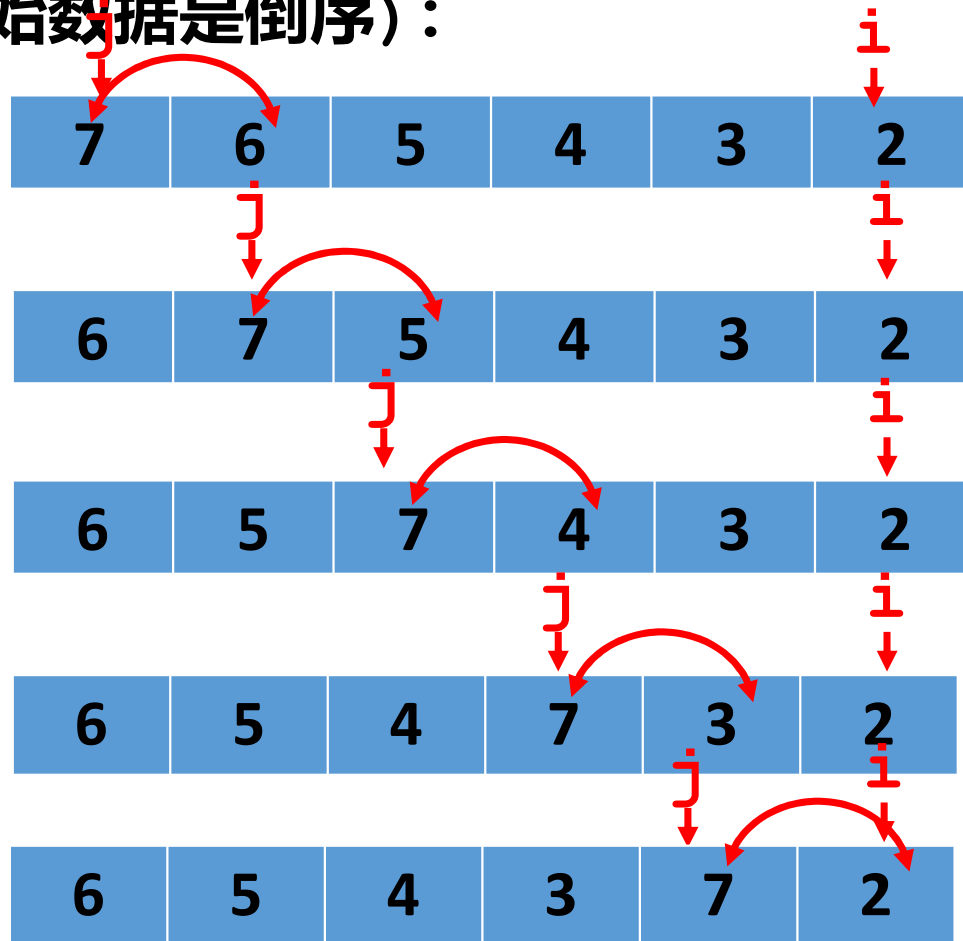
$n-1$

$n-2$

...

1

$$= n(n-1)/2$$



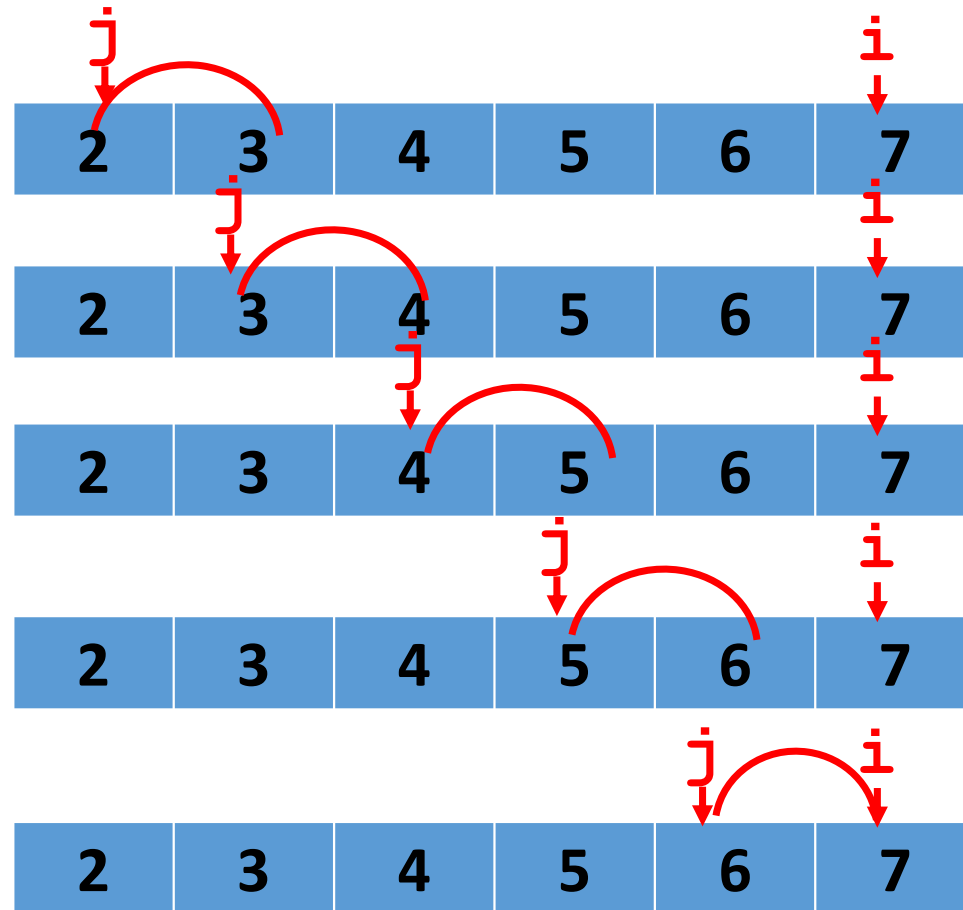
时间复杂度

- 最好情况 (本来就排序好了):

$i=n$ 时, 未发生
任何交换, 说明
数据已经排序好!

其他的 i 不需要
再进行!

比较 $n-1$ 次,
交换0次



时间复杂度

- 最好: $O(n)$
- 最差: $O(n^2)$
- 平均: $O(n^2)$
- 空间复杂度: $O(1)$. 交换时需要一个临时变量
- 稳定性: 稳定

选择式排序



hwdong

hw-dong

选择式排序

- 直接选择排序
- 树形选择排序
- 堆排序

直接选择排序

```
for (i = 0; i < n-1; i++)
```

64 25 12 22 11

- 从1到n个数据中选出最小的放在1号位置 $n-1$ 次比较

(11) 64 25 12 22

- 从2到n个数据中选出最小的放在2号位置 $n-2$ 次比较

11 (12) 64 25 22

- 从3到n个数据中选出最小的放在i号位置

11 12 (22) 64 25

• ...

$$O(n(n-1)/2) \\ = O(n^2)$$

直接选择排序

```
void SelectSort( T data[], int n) {  
    for(int i = 0; i < n-1; i ++) {  
        // 找a[i..n-1]最小值位置IndMin  
        IndMin = i;  
        for(int j = i+1; j < n; j ++)  
            if( data[j] < data[IndMin])  
                IndMin = j;  
        // 把IndMin和i作交换  
        if(IndMin != i)  
            Swap(&data[IndMin], &data[i]);  
    }  
}
```

```

for(i = 0; i < n-1; i++ ) {
    IndMin = i; // 找到i~n中最小的
    for(j = i+1; j < n; j ++ )
        if( data[j] < data[IndMin] )
            IndMin = j;
    if(IndMin != i) //交换
        Swap(&data[IndMin], &data[i]);
}

```

<i>i</i> ↓				
0	1	2	3	4
28	41	36	7	16
	↑ <i>j</i>			

IndMin = 0


```

for(i = 0; i < n-1; i++ ) {
    IndMin = i; // 找到i~n中最小的
    for(j = i+1; j < n; j ++ )
        if( data[j] < data[IndMin] )
            IndMin = j;
    if(IndMin != i) //交换
        Swap(&data[IndMin], &data[i]);
}

```

<i>i</i> ↓				
0	1	2	3	4
28	41	36	7	16
		↑ <i>j</i>		

IndMin = 0

```

for(i = 0; i < n-1; i++ ) {
    IndMin = i; // 找到i~n中最小的
    for(j = i+1; j < n; j ++ )
        if( data[j] < data[IndMin] )
            IndMin = j;
    if(IndMin != i) //交换
        Swap(&data[IndMin], &data[i]);
}

```

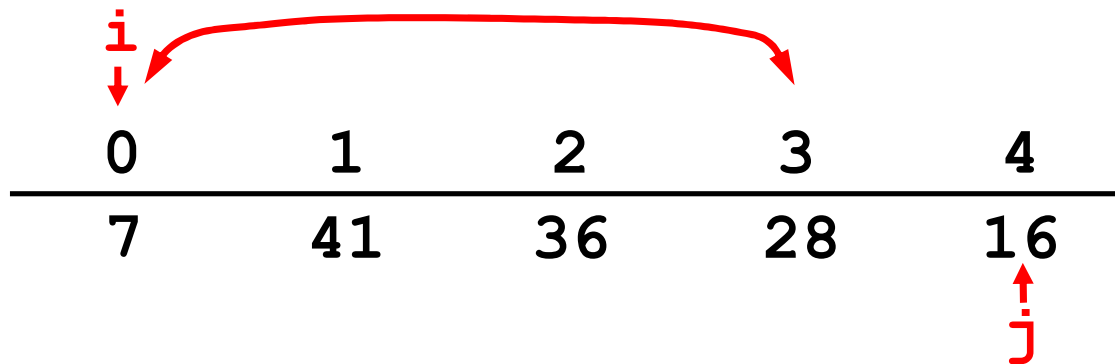
<i>i</i> ↓				
0	1	2	3	4
28	41	36	7	16
			↑ <i>j</i>	

IndMin = 3

```

for(i = 0; i < n-1; i++ ) {
    IndMin = i; // 找到i~n中最小的
    for(j = i+1; j < n; j ++ )
        if( data[j] < data[IndMin] )
            IndMin = j;
    if(IndMin != i) //交换
        Swap(&data[IndMin], &data[i]);
}

```



IndMin = 3

```

for(i = 0; i < n-1; i++) {
    IndMin = i; // 找到i~n中最小的
    for(j = i+1; j < n; j++)
        if( data[j] < data[IndMin] )
            IndMin = j;
    if(IndMin != i) //交换
        Swap(&data[IndMin], &data[i]);
}

```

0	1	2	3	4
7	41	36	28	16

i
↓
↑
j

IndMin = 1

```

for(i = 0; i < n-1; i++) {
    IndMin = i; // 找到i~n中最小的
    for(j = i+1; j < n; j++)
        if( data[j] < data[IndMin] )
            IndMin = j;
    if(IndMin != i) //交换
        Swap(&data[IndMin], &data[i]);
}

```

0	1	2	3	4
7	41	36	28	16

↓ ↑
i j

IndMin = 2

```

for(i = 0; i < n-1; i++ ) {
    IndMin = i; // 找到i~n中最小的
    for(j = i+1; j < n; j ++ )
        if( data[j] < data[IndMin] )
            IndMin = j;
    if(IndMin != i) //交换
        Swap(&data[IndMin], &data[i]);
}

```

0	1	2	3	4
7	41	36	28	16

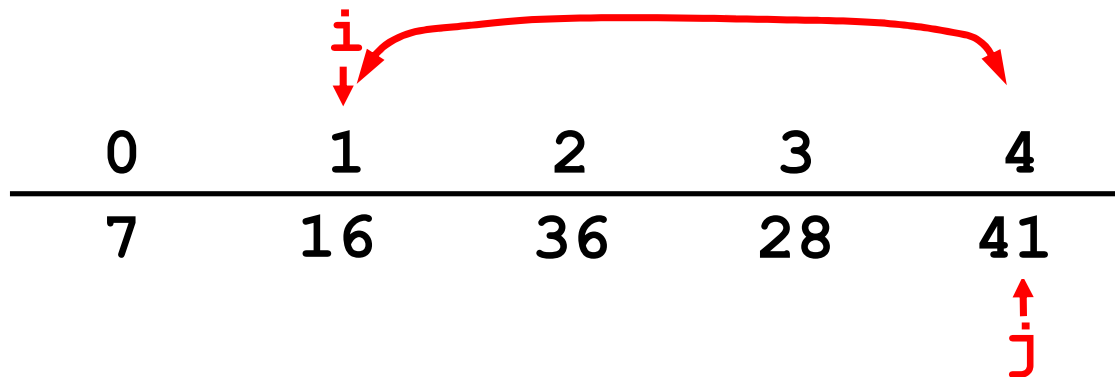
↓ ↑
i j

IndMin = 3

```

for(i = 0; i < n-1; i++ ) {
    IndMin = i; // 找到i~n中最小的
    for(j = i+1; j < n; j ++ )
        if( data[j] < data[IndMin] )
            IndMin = j;
    if(IndMin != i) //交换
        Swap(&data[IndMin], &data[i]);
}

```



0	1	2	3	4
7	16	36	28	41

IndMin = 4

时间复杂度

```
for (i = 0; i < n-1; i++) {  
    IndMin = i; // 找到i~n中最小的  
    for (j = i+1; j < n; j++)  
        if ( data[j] < data[IndMin] )  
            IndMin = j;  
    if (IndMin != i) // 交换  
        Swap (&data [IndMin] , &data [i] ) ;  
}
```

i=0: n-1次比较

i=1: n-2次比较

...

i=n-1: 1次比较

数据比较次数: $n(n-1)/2$

时间复杂度

n-1趟, 每趟最多交换1次

3(n-1) 移动

```
for(i = 0; i < n-1; i++) {  
    IndMin = i; // 找到i~n中最小的  
    for(j = i+1; j < n; j++)  
        if( data[j] < data[IndMin] )  
            IndMin = j;  
    if(IndMin != i) // 交换  
        Swap(&data[IndMin], &data[i]);  
}
```

i=0: n-1次比较

i=1: n-2次比较

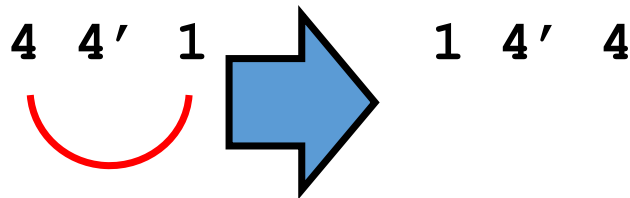
...

i=n-1: 1次比较

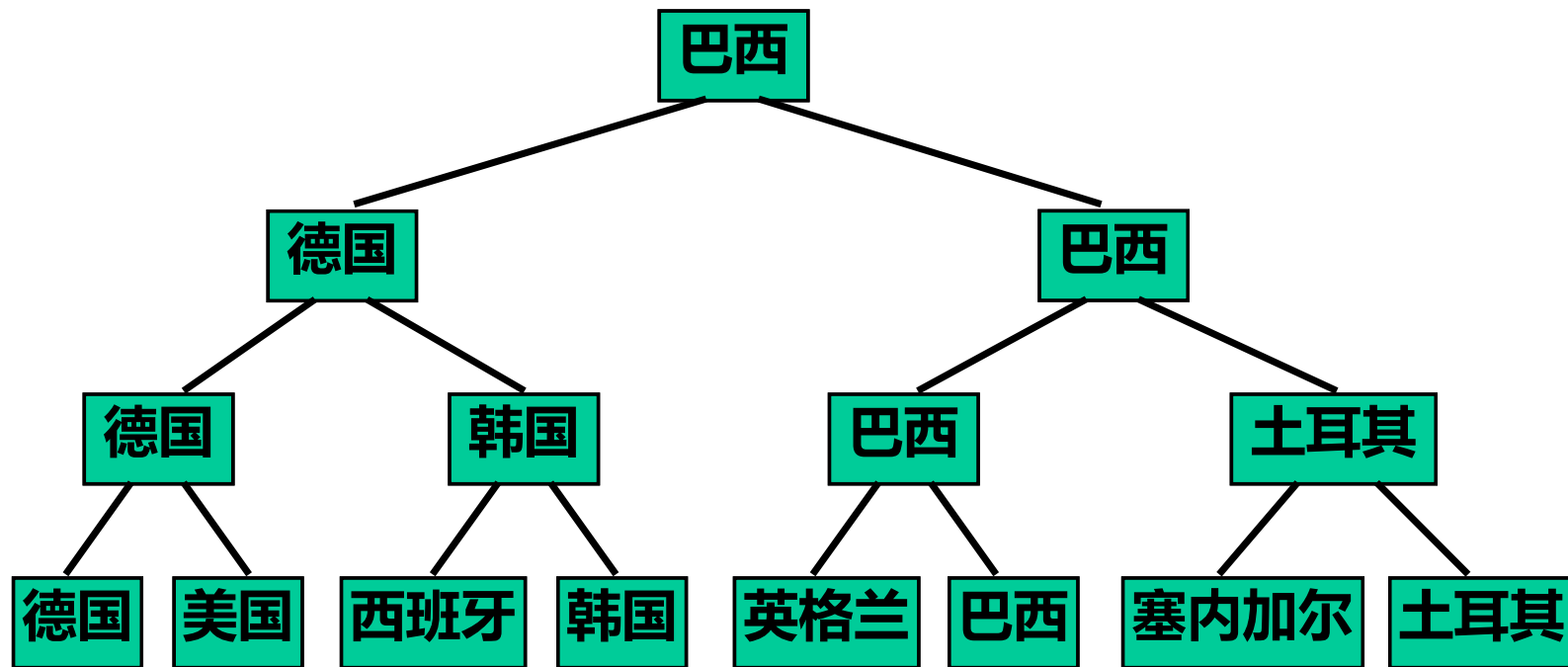
数据比较次数: $n(n-1)/2$

时间复杂度

- 数据移动的次数：2 5 8 13 25 37
 - 当元素已经排好序时，不需要移动数据
 - 当元素是逆序时，需要移动 $3(n-1)$ 次
- 数据比较次数： $n(n-1)/2$
- 所以不论什么情况，时间复杂度都是 $O(n^2)$
- 空间复杂度： $O(1)$
- 稳定性：不稳定



2002年日韩世界杯淘汰赛8强

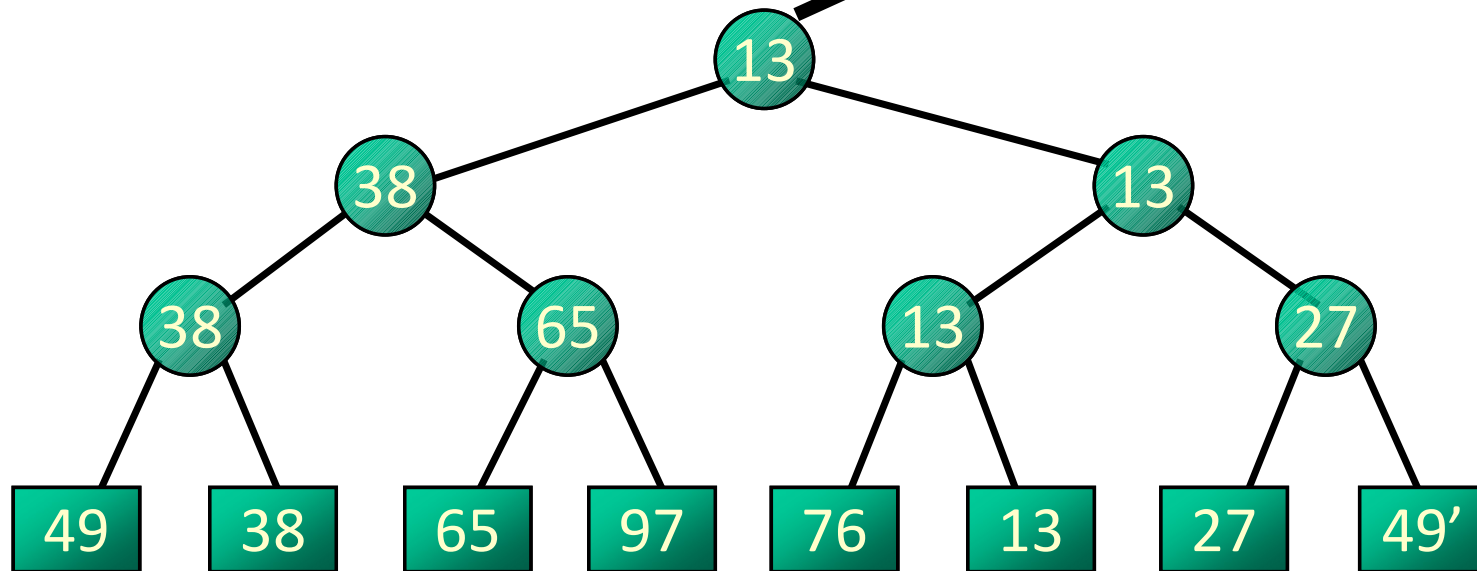


锦标赛排序

- 基本思想
 - 类似体育比赛时的淘汰赛
 - n 个对象，两两比较，得到 $\lfloor n/2 \rfloor$ 个比较的优胜者
 - 对这 $\lfloor n/2 \rfloor$ 个对象再两两比较，...
 - 重复，直到选出一个排序码最小的对象为止

锦标赛排序

冠军



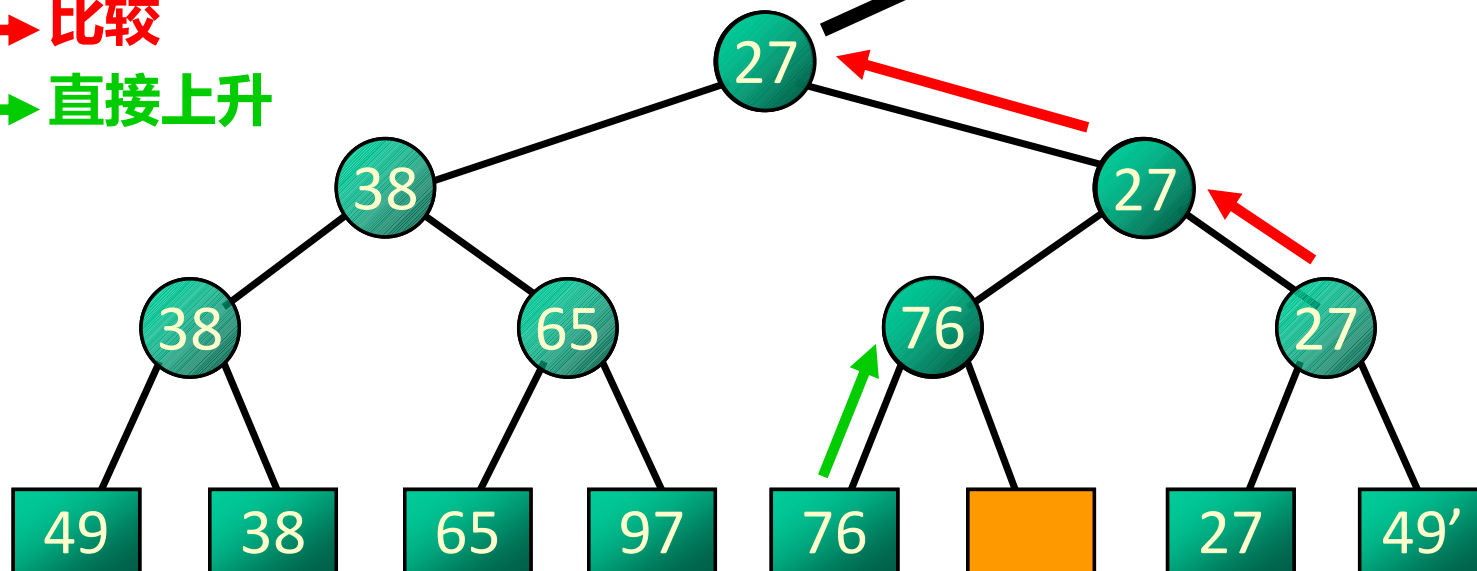
- 比较次数 = 7
- 输出冠军

锦标赛排序

亚军

→ 比较

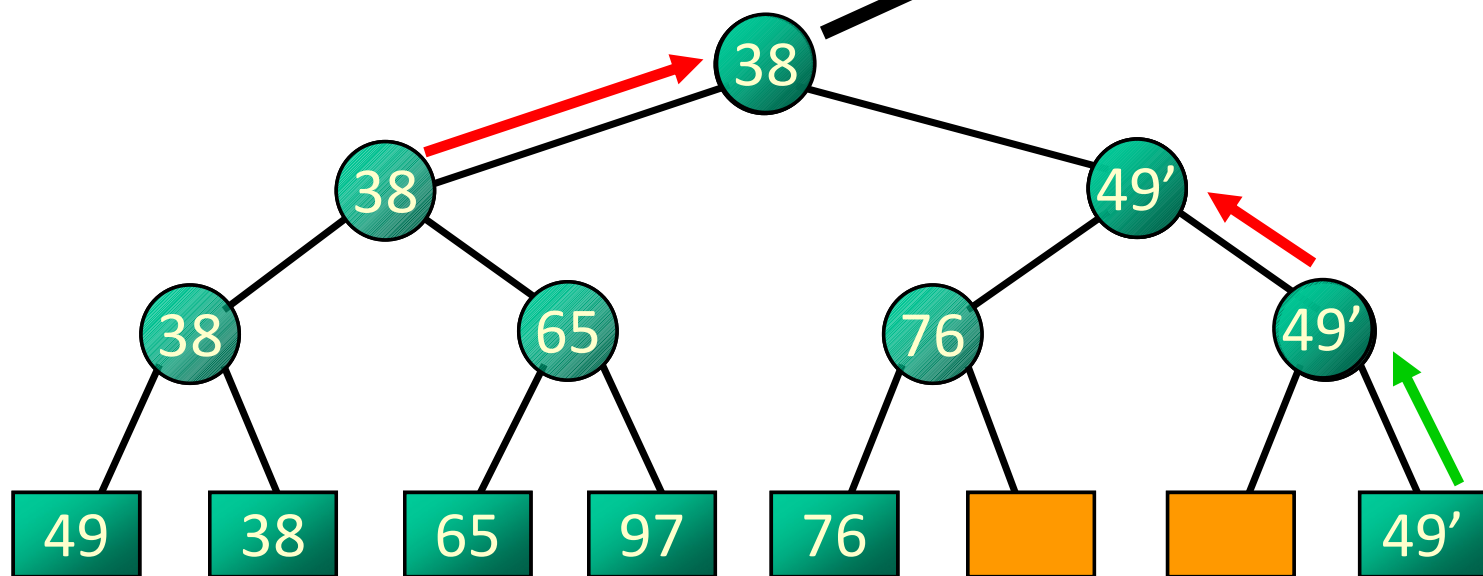
→ 直接上升



- 比较次数 = 2
- 输出亚军

锦标赛排序

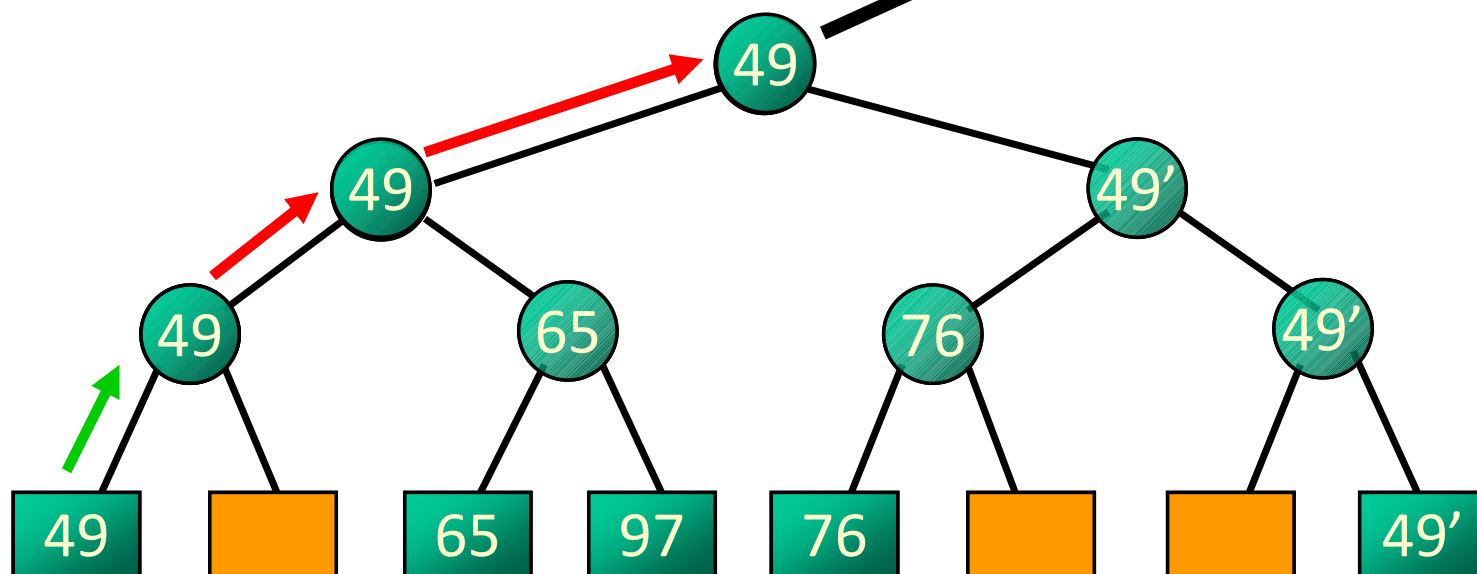
季军



- 比较次数 = 2
- 输出季军

锦标赛排序

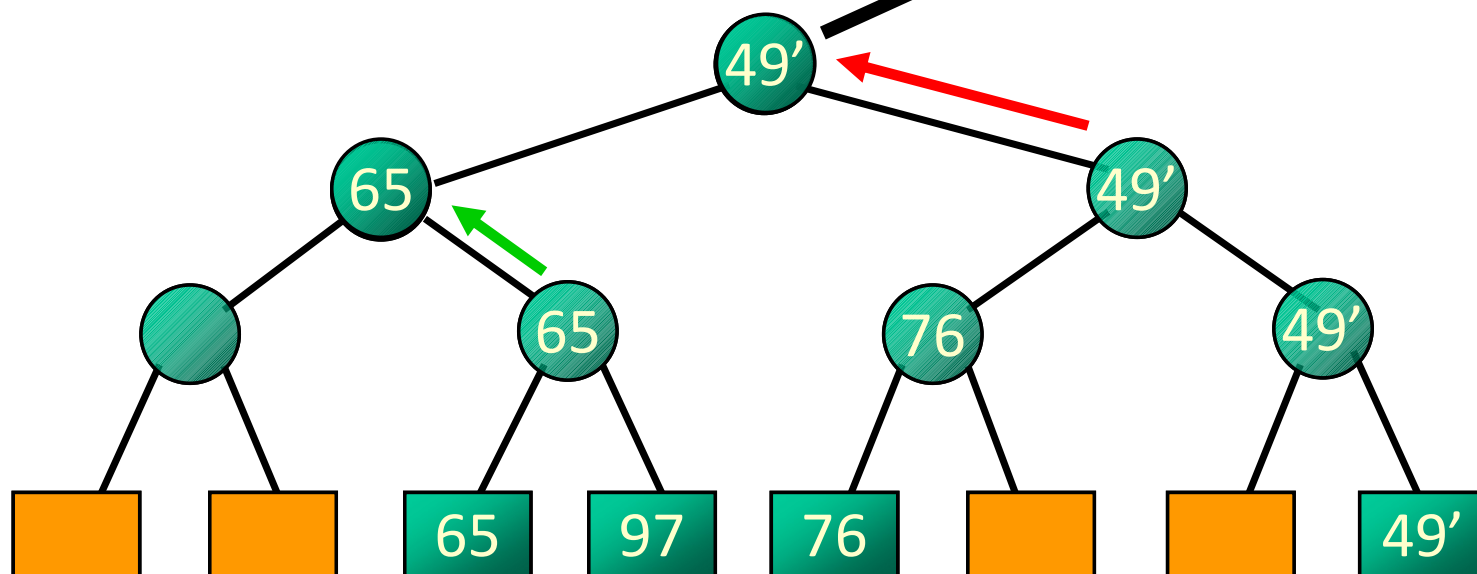
第四名



- 比较次数 = 2
- 输出第四名

锦标赛排序

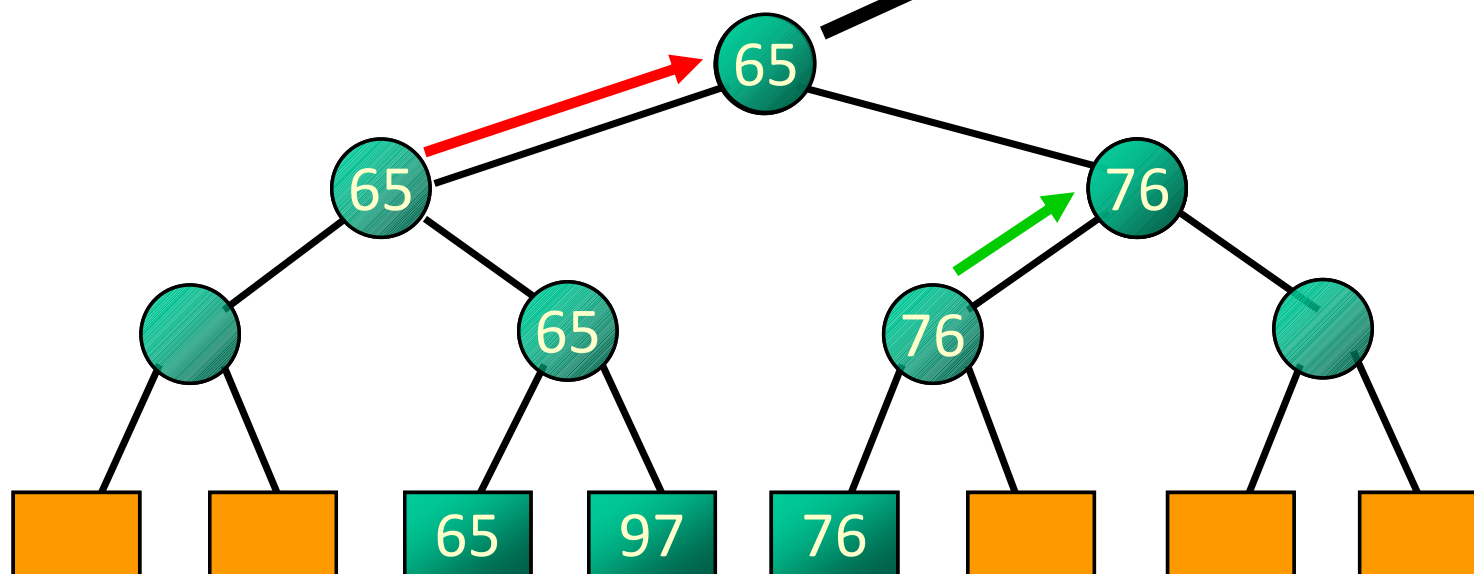
第五名



- 比较次数 = 1
- 输出第五名

锦标赛排序

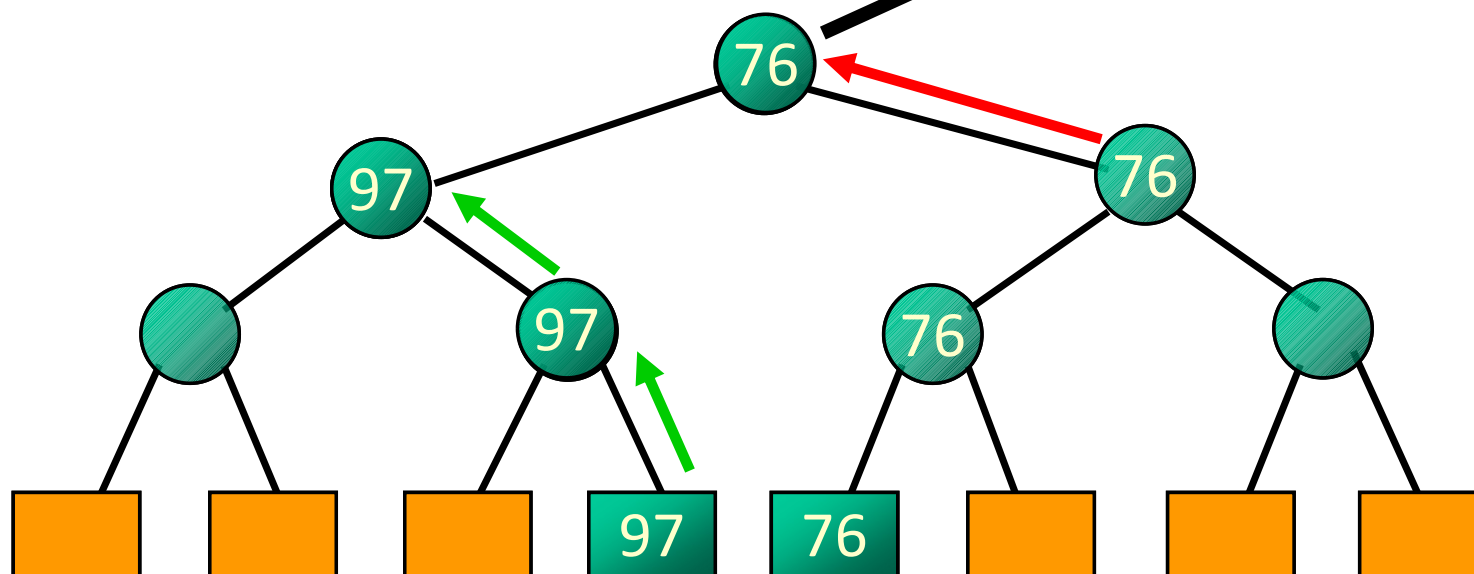
第六名



- 比较次数 = 1
- 输出第六名

锦标赛排序

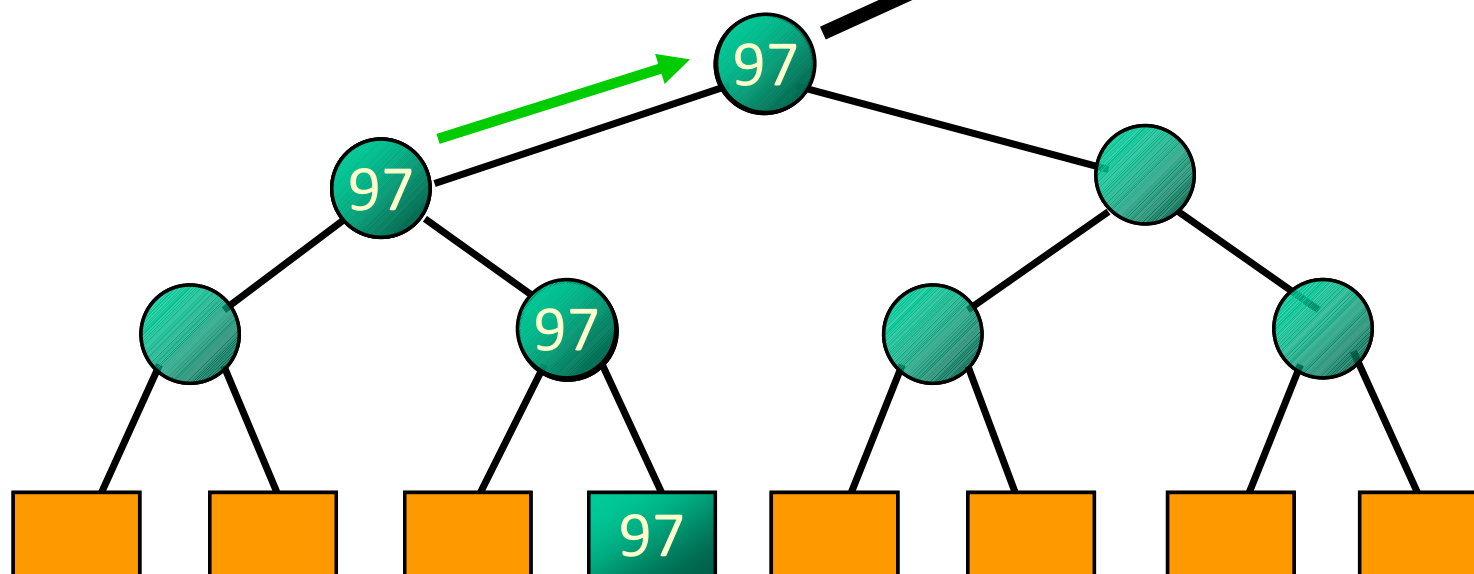
第七名



- 比较次数 = 1
- 输出第七名

锦标赛排序

第八名



锦标赛排序

- 时间复杂度

- 锦标赛排序构成的选择树是满二叉树（如果元素不够补充空节点），其深度为 $\lceil \log_2 n \rceil$
- 除第一次选择时需要进行 $n-1$ 次比较外，选择其它元素每次只需比较 $O(\log_2 n)$ 次，所以总的比较次数为 $O(n \log_2 n)$
- 对象的移动次数不超过排序码的比较次数，所以锦标赛排序总时间复杂度为 $O(n \log_2 n)$

锦标赛排序

- 空间复杂度
 - 锦标赛排序法虽然减少了许多排序时间，但是使用了较多的附加存储
 - 如果有 n 个对象，必须使用至少 $2n-1$ 个结点来存放选择树
- 稳定性：
 - 稳定

归并排序



hwdong

hw-dong

归并排序

(迭代, 自地向上) 归并排序

(递归, 自顶向下) 归并排序

(2路) 归并

- 归并

- 将两个或两个以上的有序表合并成一个新的有序表

08	21	25	25'	49	62	72	93	16	37	54
----	----	----	-----	----	----	----	----	----	----	----

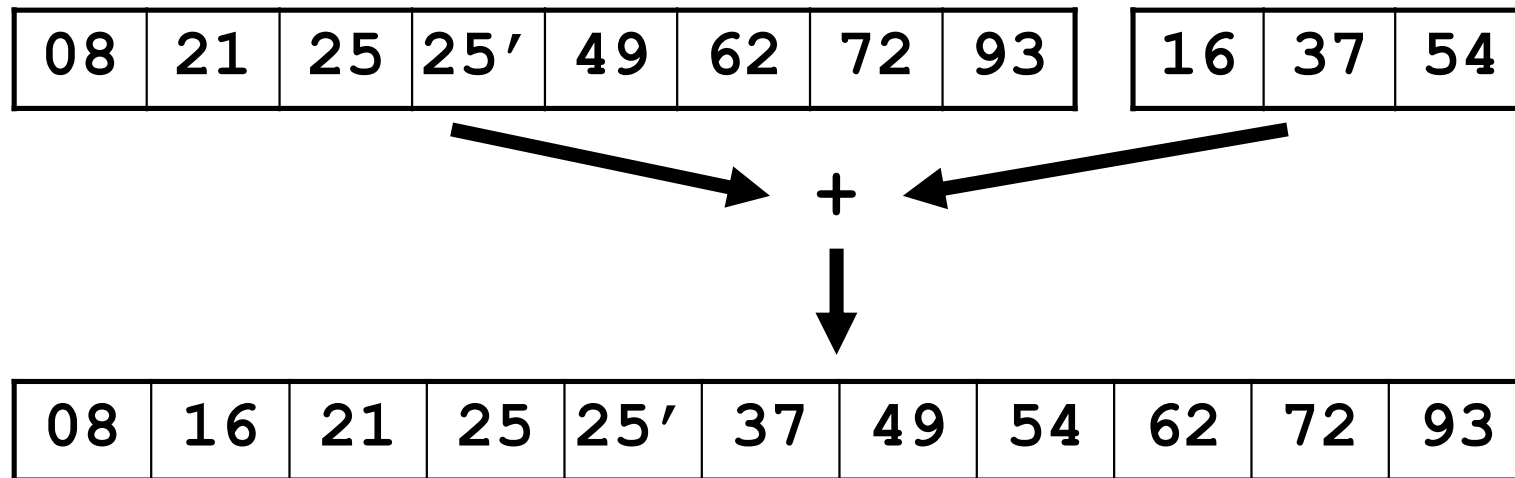


08	16	21	25	25'	37	49	54	62	72	93
----	----	----	----	-----	----	----	----	----	----	----

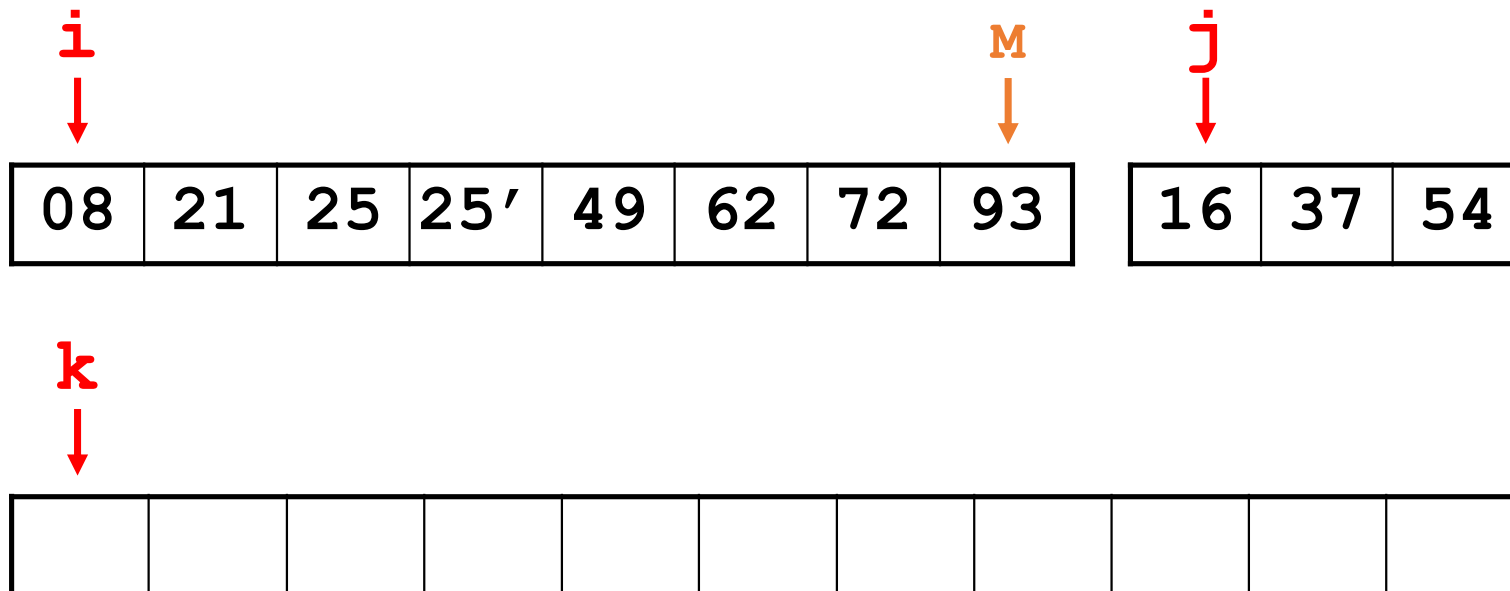
(2路) 归并

- 归并

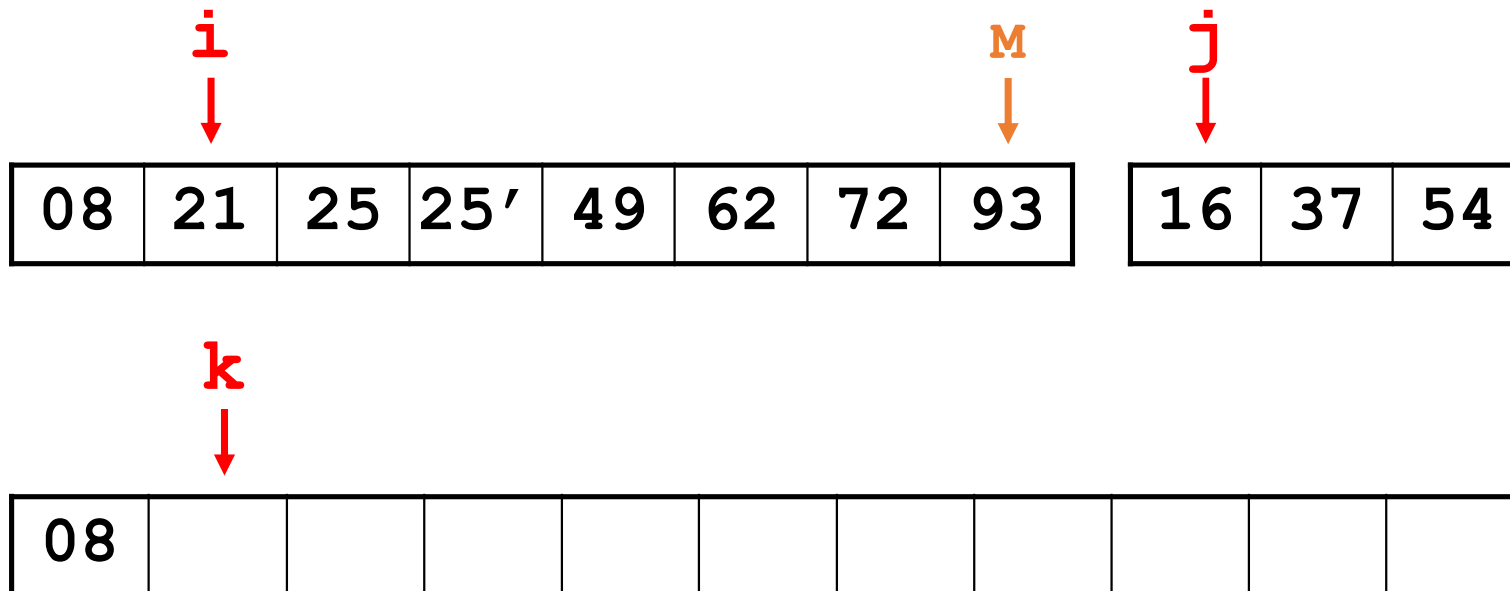
- 将两个或两个以上的有序表合并成一个新的有序表



```
void merge(T data[], T out[], int s, int M, int N) {  
    int i = s, j = M+1;  
    while(i <= M && j <= N)  
        if(data[i] < data[j])  
            out[k++] = data[i++];  
        else  
            out[k++] = data[j++];  
}
```



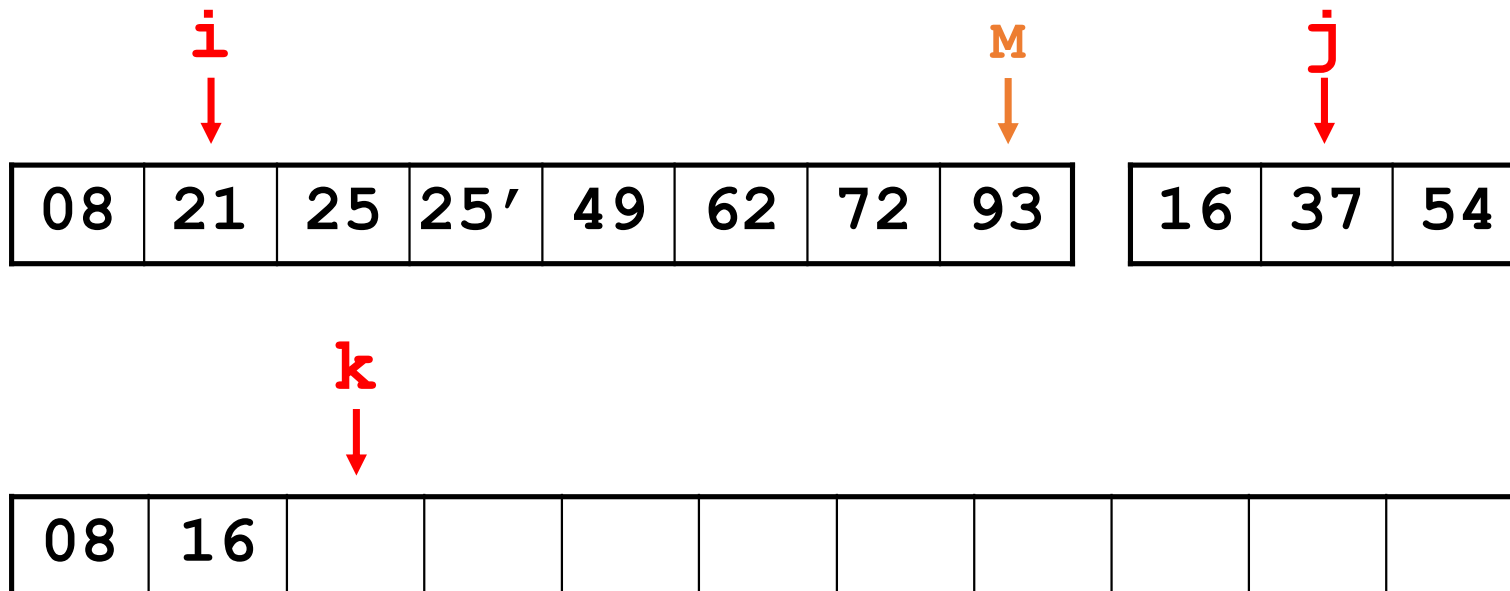
```
void merge(T data[], T out[], int s, int M, int N) {  
    int i = s, j = M+1, k = s;  
    while(i <= M && j <= N)  
        if(data[i] < data[j])  
            out[k++] = data[i++];  
        else  
            out[k++] = data[j++];  
}
```



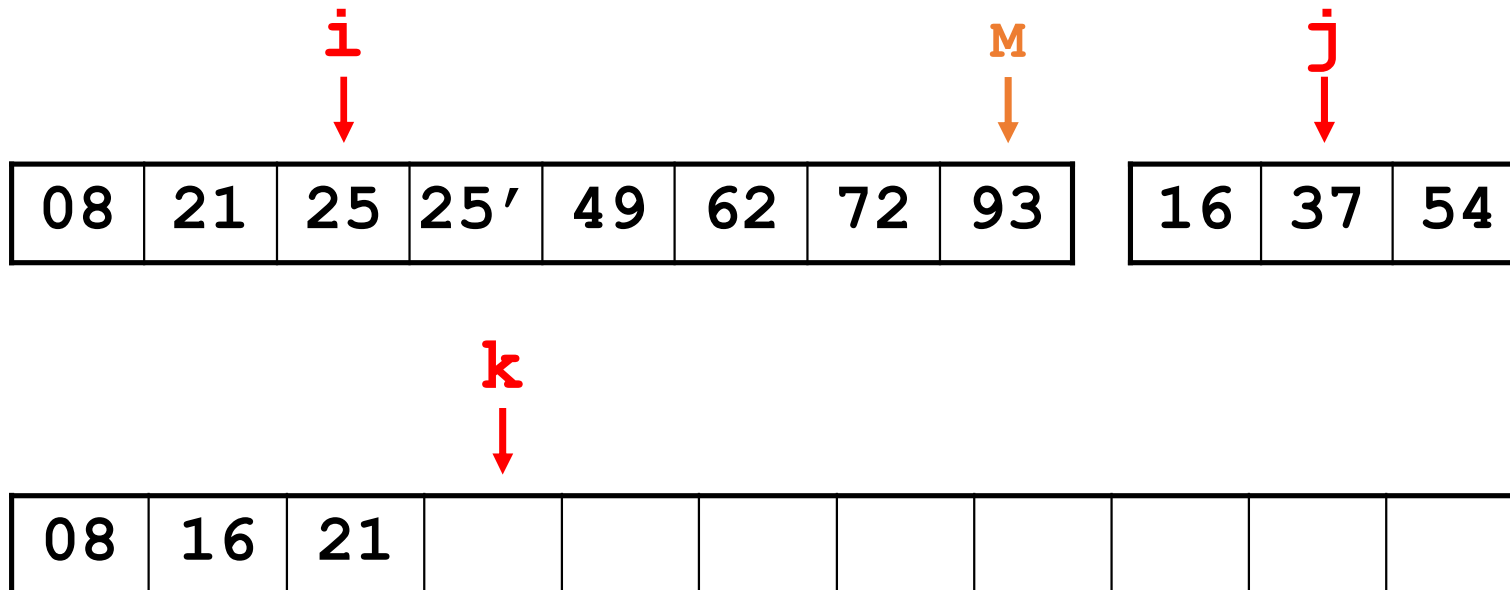
```

void merge(T data[], T out[], int s, int M, int N) {
    int i = s, j = M+1;
    while(i <= M && j <= N)
        if(data[i] < data[j])
            out[k++] = data[i++];
        else
            out[k++] = data[j++];
}

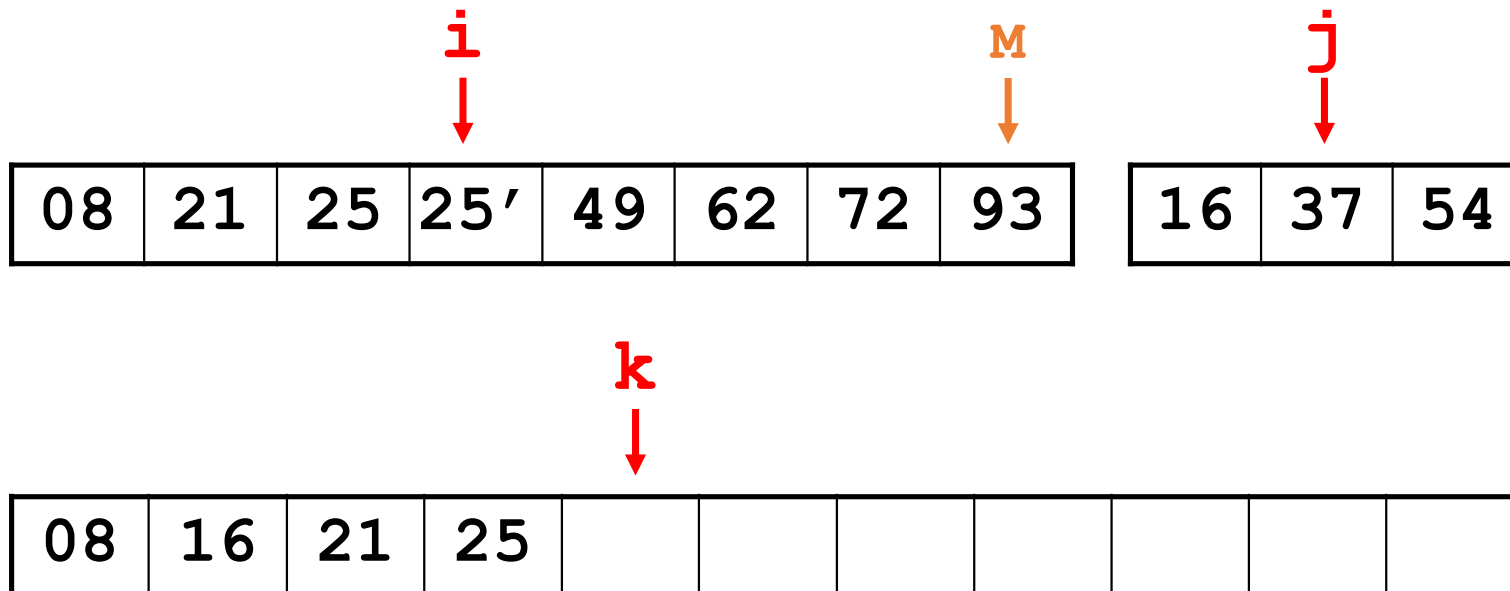
```



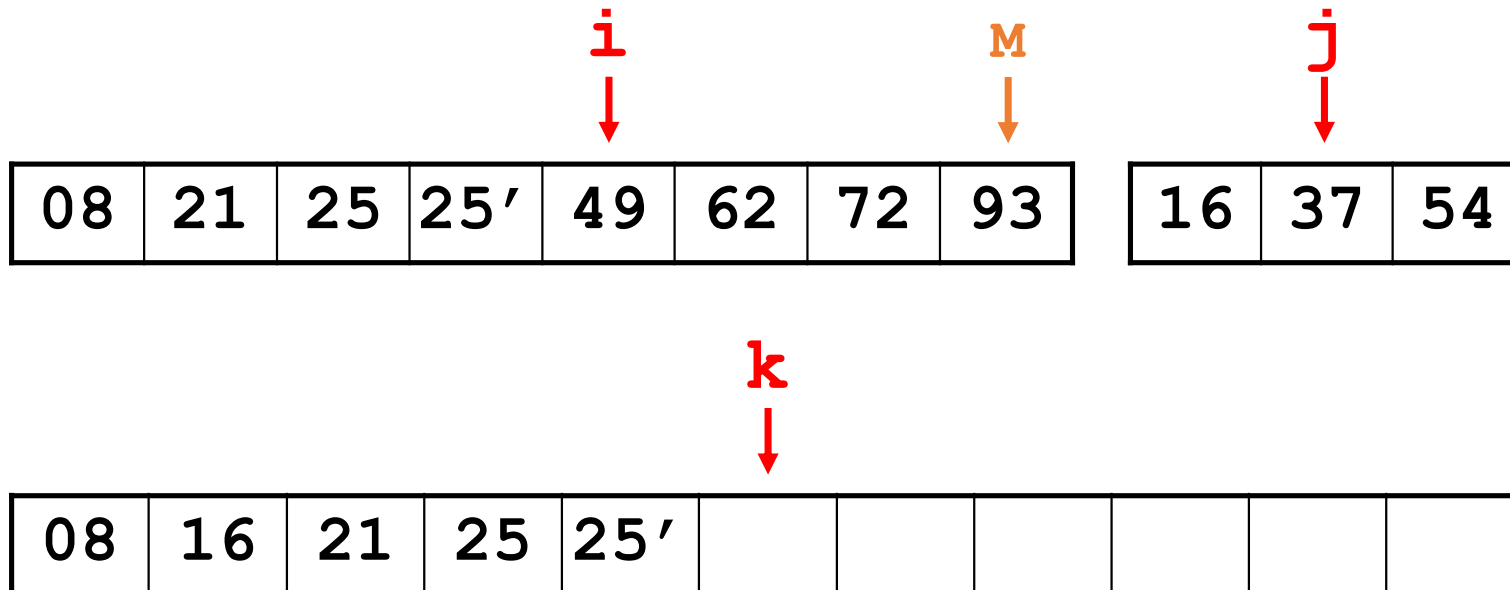
```
void merge(T data[], T out[], int s, int M, int N) {  
    int i = s, j = M+1, k = s;  
    while(i <= M && j <= N)  
        if(data[i] < data[j])  
            out[k++] = data[i++];  
        else  
            out[k++] = data[j++];  
}
```



```
void merge(T data[], T out[], int s, int M, int N) {  
    int i = s, j = M+1, k = s;  
    while(i <= M && j <= N)  
        if(data[i] < data[j])  
            out[k++] = data[i++];  
        else  
            out[k++] = data[j++];  
}
```



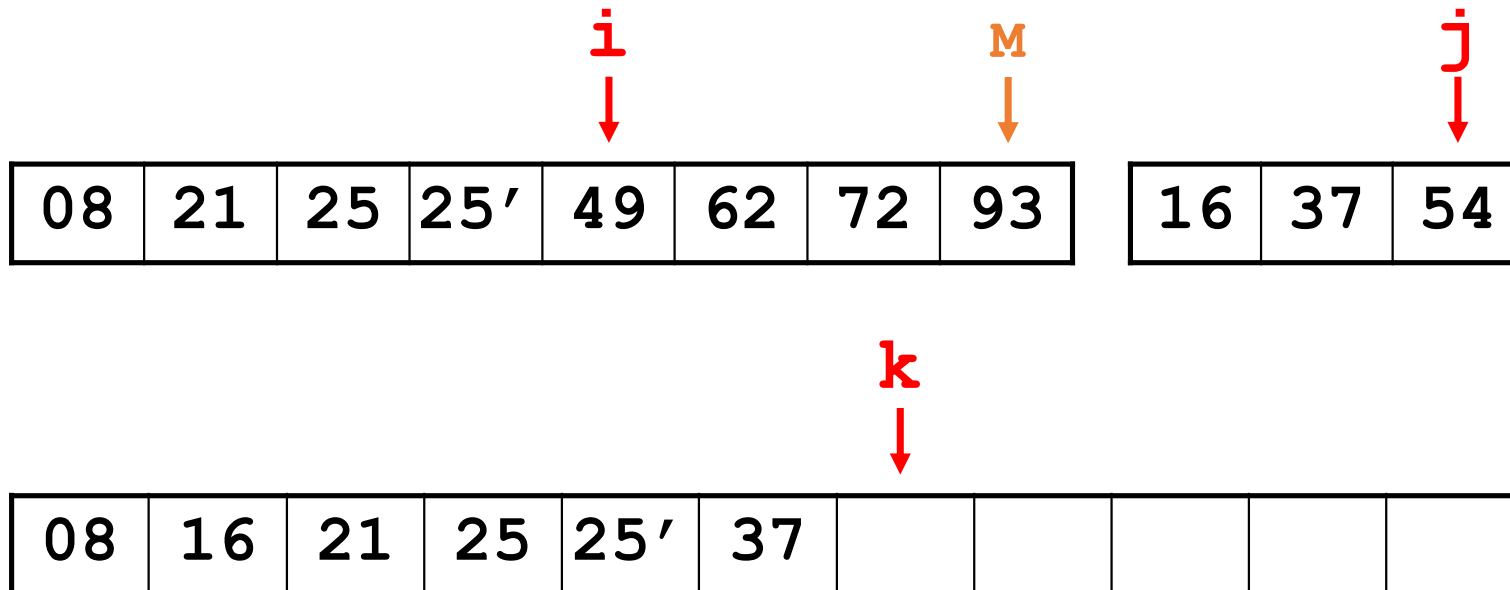
```
void merge(T data[], T out[], int s, int M, int N) {  
    int i = s, j = M+1, k = s;  
    while(i <= M && j <= N)  
        if(data[i] < data[j])  
            out[k++] = data[i++];  
        else  
            out[k++] = data[j++];  
}
```




```

void merge(T data[], T out[], int s, int M, int N) {
    int i = s, j = M+1, k = s;
    while(i <= M && j <= N)
        if(data[i] < data[j])
            out[k++] = data[i++];
        else
            out[k++] = data[j++];
}

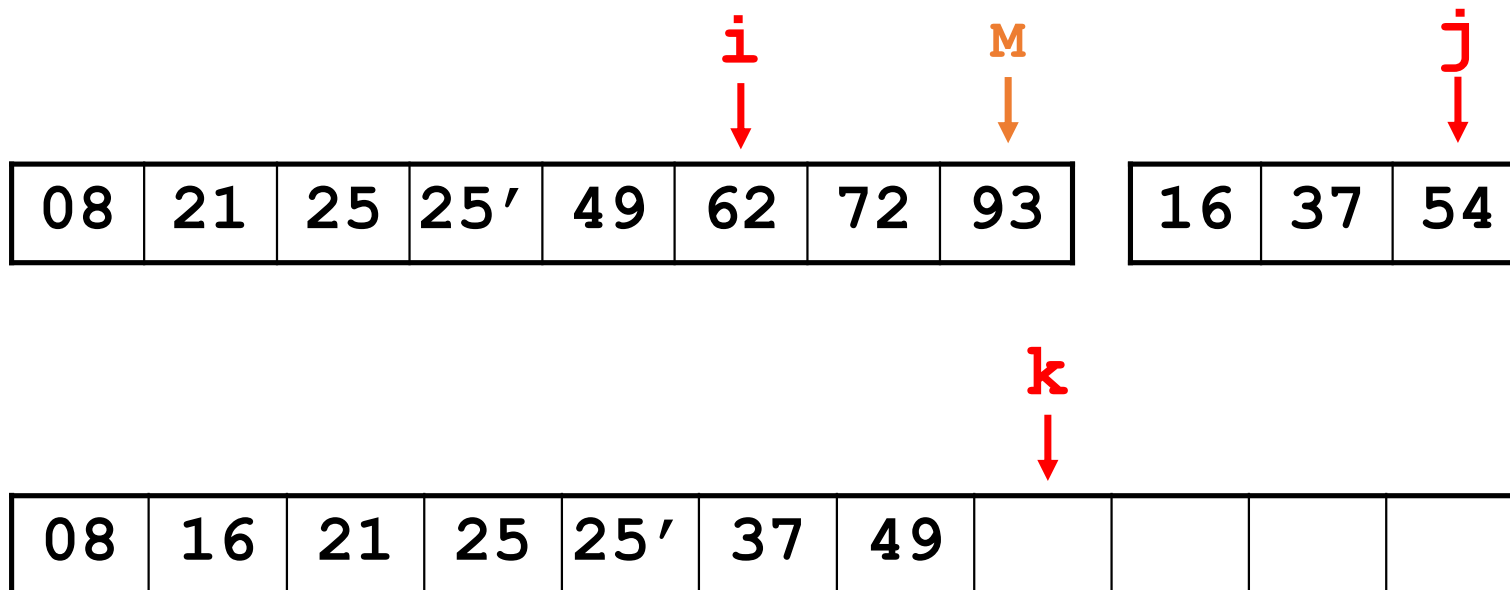
```



```

void merge(T data[], T out[], int s, int M, int N) {
    int i = s, j = M+1, k = s;
    while(i <= M && j <= N)
        if(data[i] < data[j])
            out[k++] = data[i++];
        else
            out[k++] = data[j++];
}

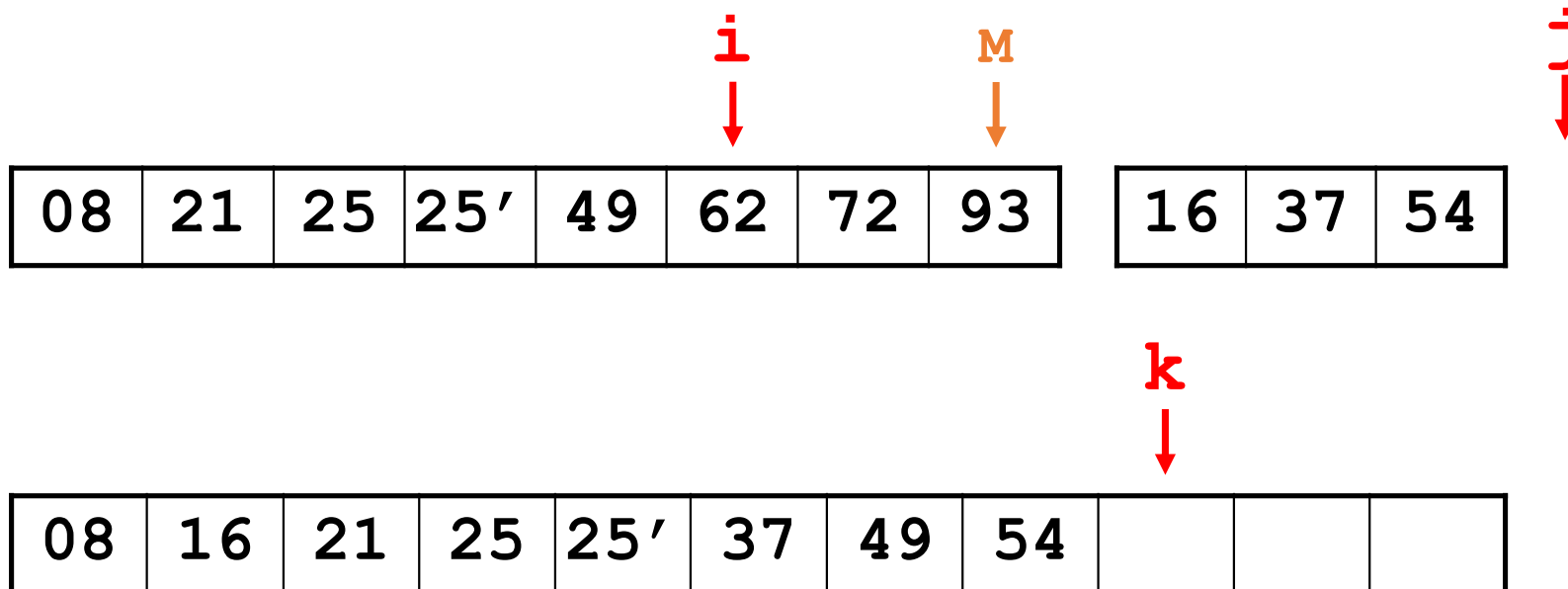
```



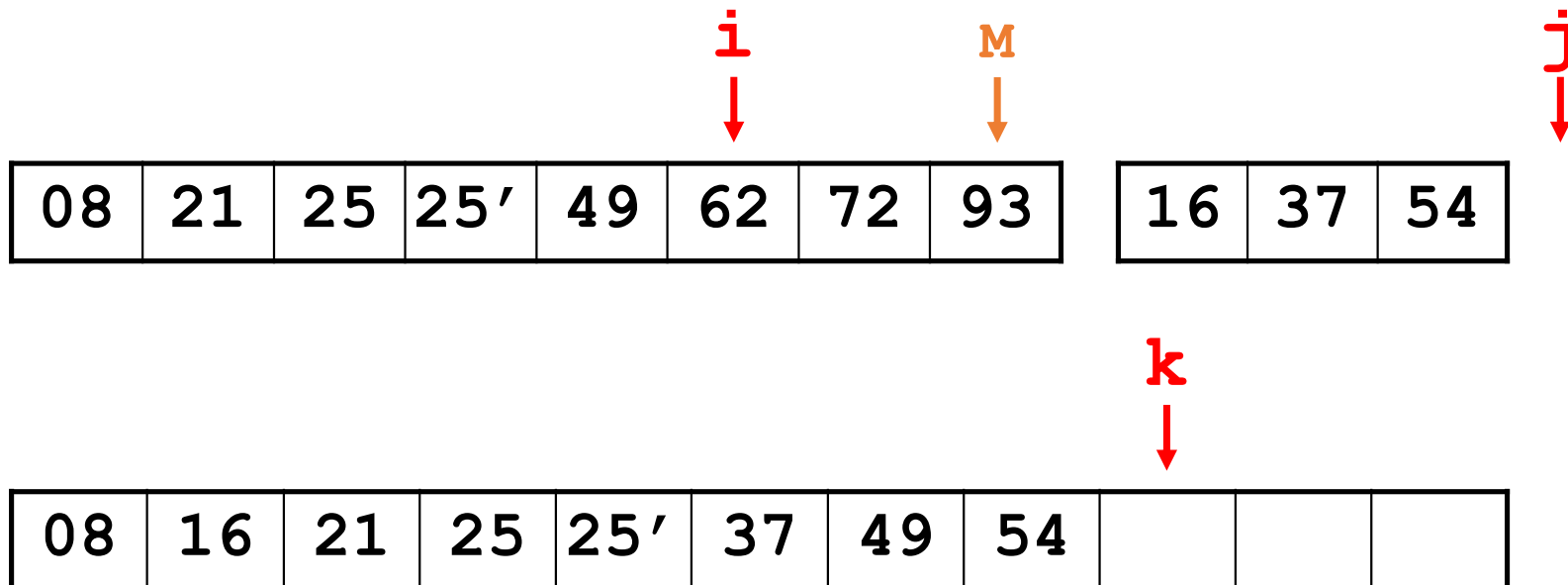
```

void merge(T data[], T out[], int s, int M, int N) {
    int i = s, j = M+1, k = s;
    while(i <= M && j <= N)
        if(data[i] < data[j])
            out[k++] = data[i++];
        else
            out[k++] = data[j++];
}

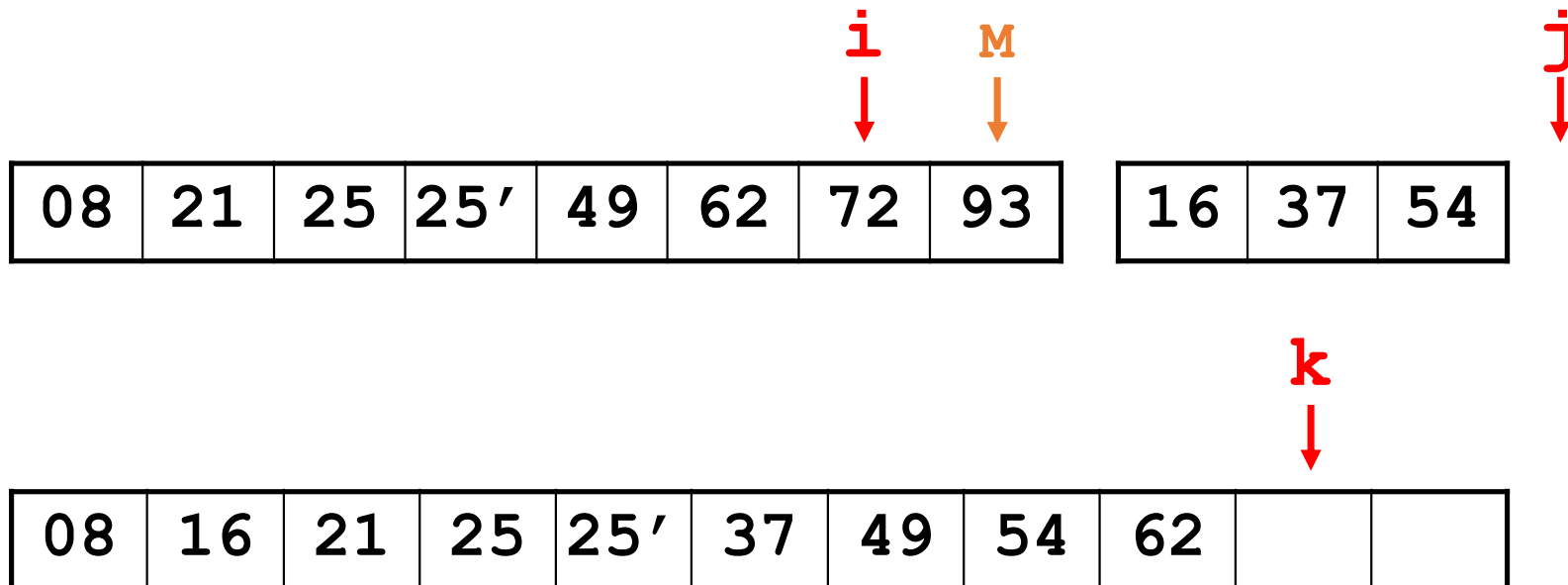
```



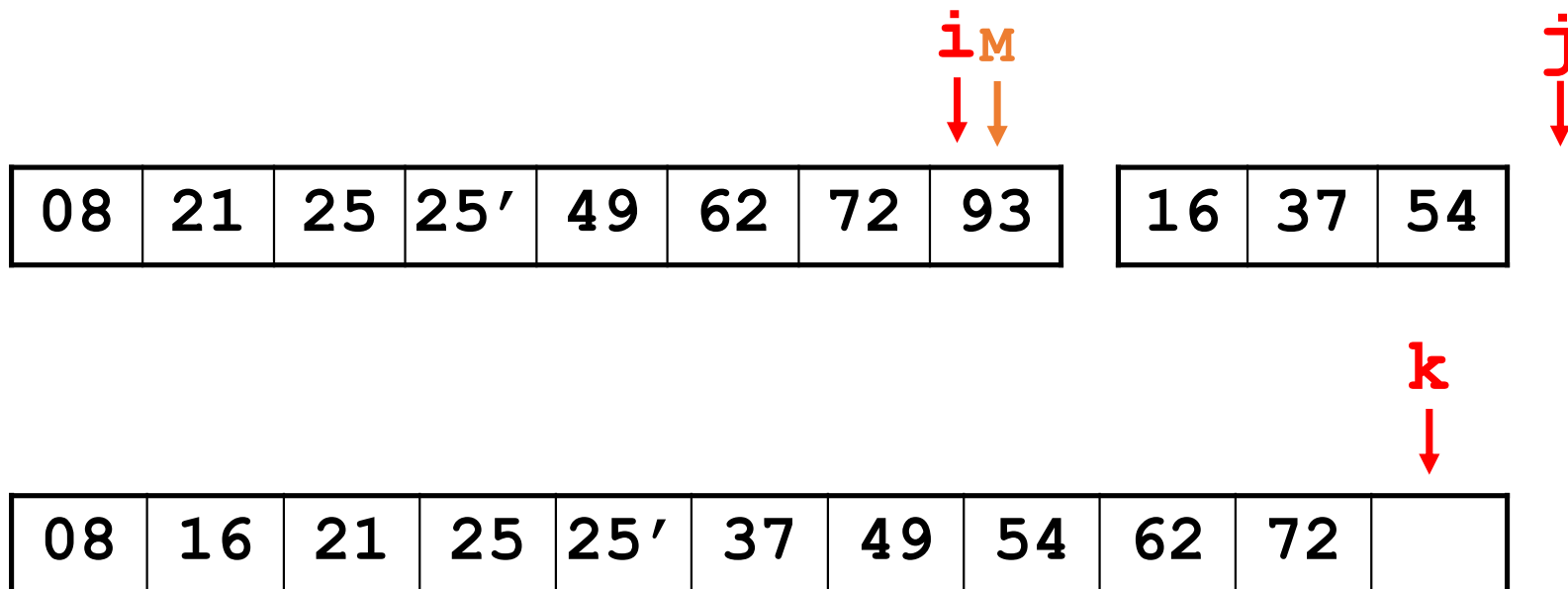
```
while (i <= M)
    out[k++] = data[i++];
while (j <= N)
    out[k++] = data[j++];
```



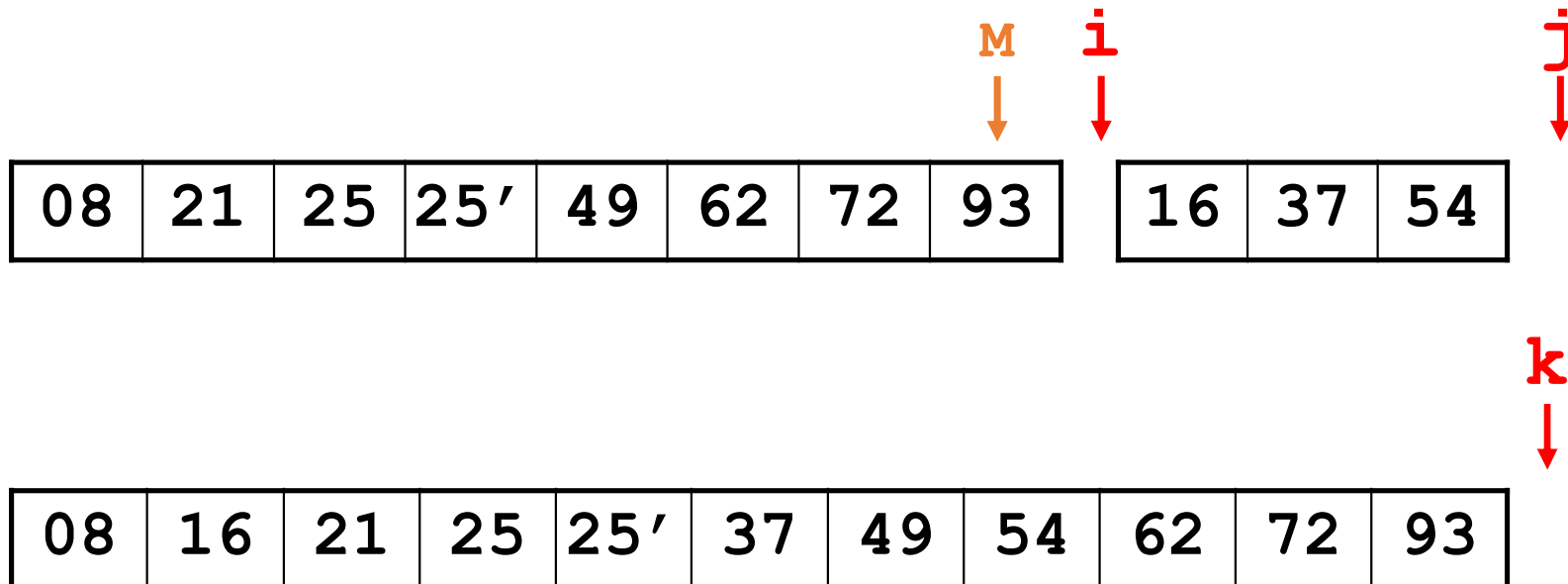
```
while (i <= M)
    out[k++] = data[i++];
while (j <= N)
    out[k++] = data[j++];
```



```
while (i <= M)
    out[k++] = data[i++];
while (j <= N)
    out[k++] = data[j++];
```



```
while (i <= M)
    out[k++] = data[i++];
while (j <= N)
    out[k++] = data[j++];
```



```
void merge(T data[], T out[], int s, int M, int N) {  
    int i = s, j = M+1, k = s;  
    //1. 两个序列都还有数据  
    while(i <= M && j <= N)  
        if(data[i] < data[j])  
            out[k++] = data[i++];  
        else  
            out[k++] = data[j++];  
  
    //2. 序列剩余数据依次写入目标序列  
    while(i <= M)  
        out[k++] = data[i++];  
    while(j <= N)  
        out[k++] = data[j++];  
}
```

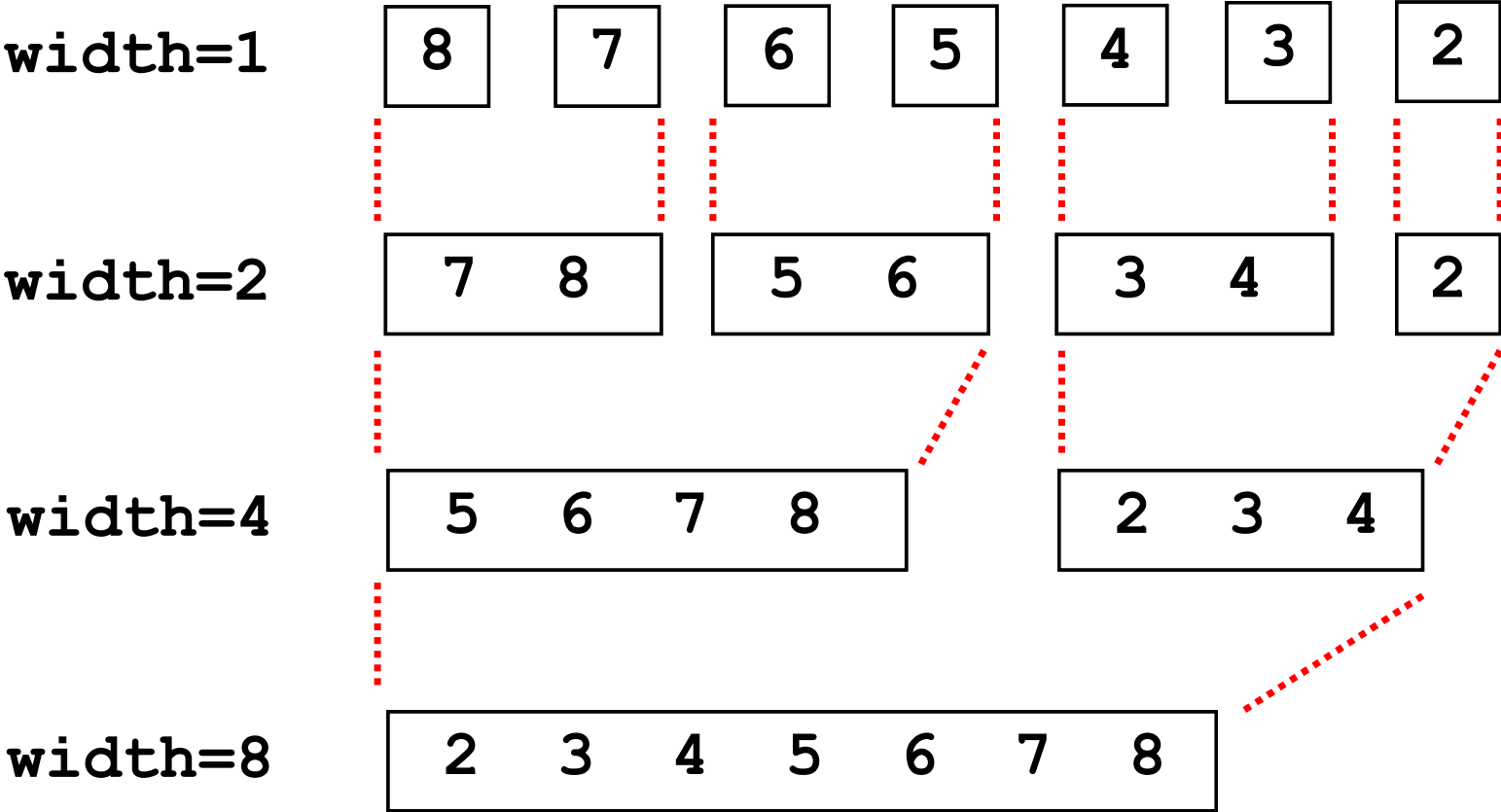

(迭代)归并排序

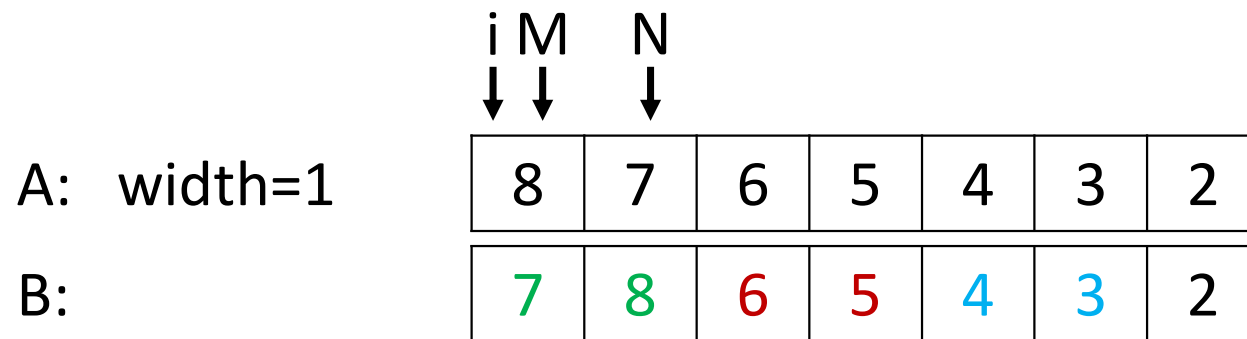
假设每个元素就是一个有序序列

width=1



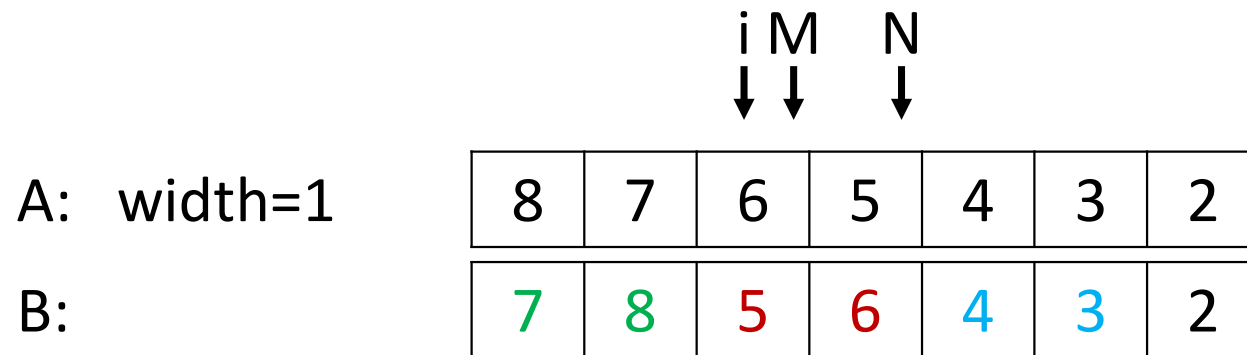
相邻有序序列“两两合并”





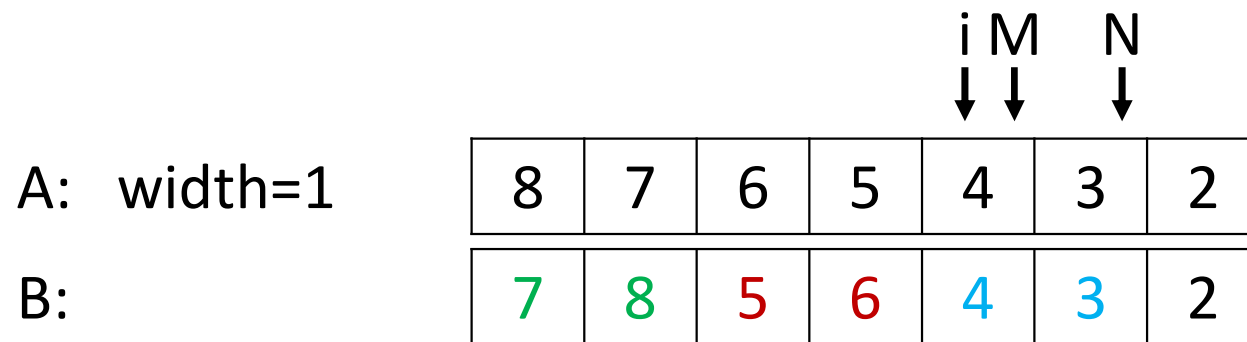
```
for( int i = 0; i < n; ? )
    merge(A, B, i, M, N);
```

$M = i + \text{width} - 1$
 $N = i + 2 * \text{width} - 1$



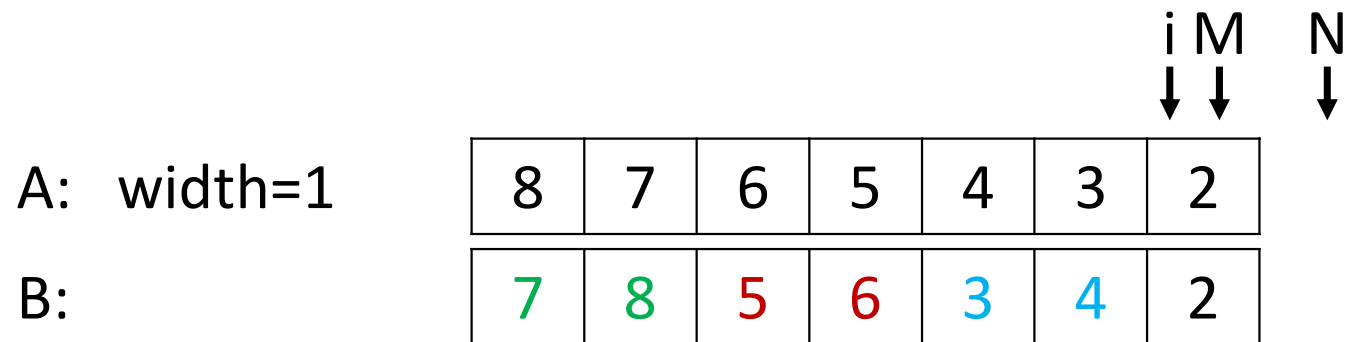
```
for( int i = 0; i < n; i = i + 2 * width )
    merge(A, B, i, M, N);
```

```
M = i + width - 1
N = i + 2 * width - 1
```



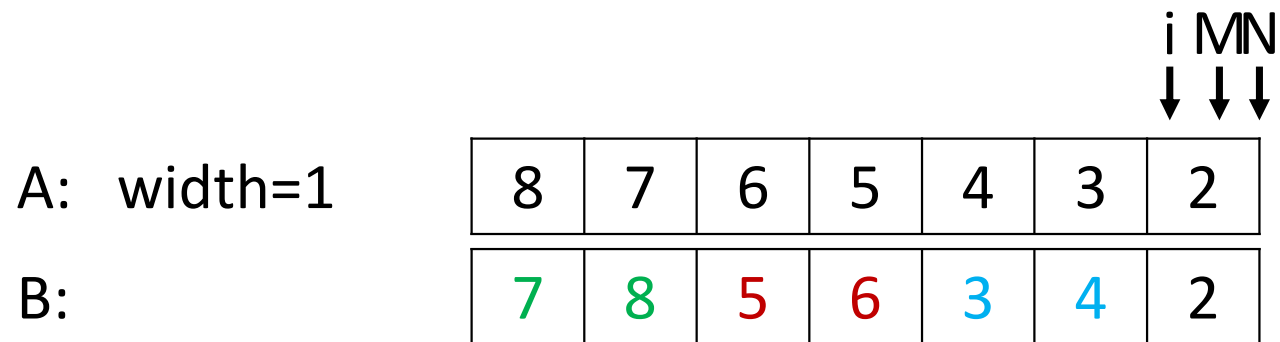
```
for( int i = 0; i < n; i = i+2*width )
    merge(A, B, i, M, N);
```

```
M = i+width-1
N = i+2*width-1
```



```
for( int i = 0; i < n; i = i + 2 * width )  
    merge(A, B, i, M, N);
```

$M = i + \text{width} - 1$
 $N = i + 2 * \text{width} - 1$



```
for( int i = 0; i < n; i = i+2*width )
    merge(A, B, i, M, N);
```

$M = i + \text{width} - 1$

$N = \min(i + 2 * \text{width} - 1, n - 1)$

							i	M	N
							↓	↓	↓
A: width=1	8	7	6	5	4	3	2		
B:	7	8	5	6	3	4	2		

```

for(int width = 1; width < n; width *=2){
    for( int i = 0; i< n; i= i+2*width )
        merge(A, B, i, M, N);
    copy(B,A,n);
}
M = i+width-1
N = min(i+2*width-1,n-1)

```

	i	M	N			
	↓	↓	↓			
A: width=2	7	8	5	6	3	4
B:	5	6	7	8	3	4

```

for(int width = 1; width < n; width *=2){
    for( int i = 0; i< n; i= i+2*width )
        merge(A, B, i, M, N);
    copy(B,A,n);
}
M = i+width-1
N = min(i+2*width-1,n-1)

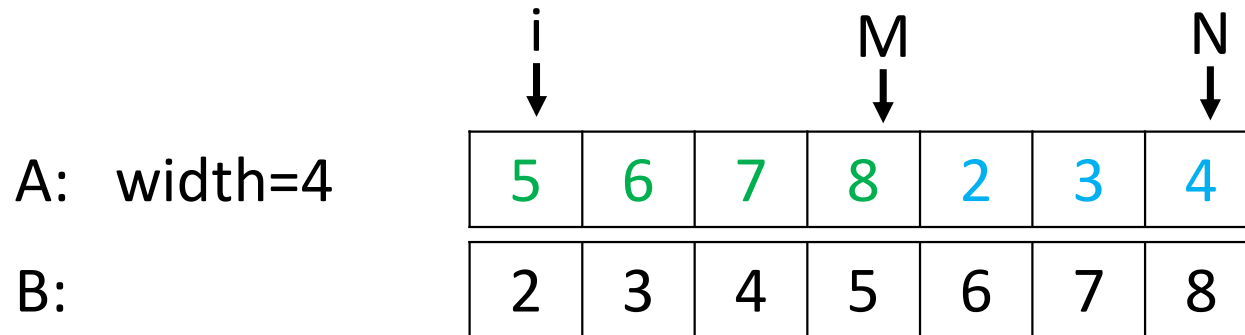
```

A: width=2

B:

				i	M	N
				↓	↓	↓
7	8	5	6	3	4	2
5	6	7	8	2	3	4

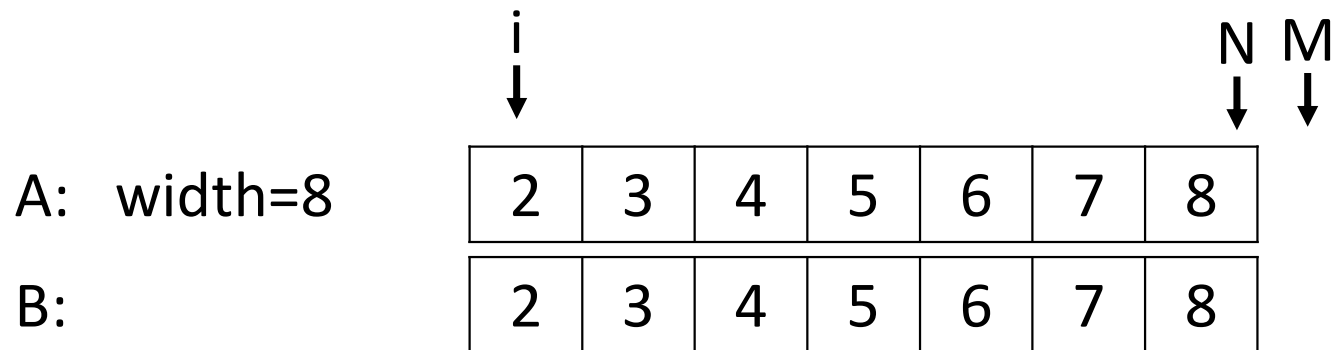
```
for(int width = 1; width < n; width *=2){  
    for( int i = 0; i< n; i= i+2*width )  
        merge(A, B, i, M, N);  
    copy(B,A,n);  
    }  
    M = i+width-1  
    N = min(i+2*width-1,n-1)
```



```

for(int width = 1; width < n; width *=2){
    for( int i = 0; i< n; i= i+2*width )
        merge(A, B, i, M, N);
    copy(B,A,n);
}
M = i+width-1
N = min(i+2*width-1,n-1)

```



```
for(int width = 1; width < n; width *=2){  
    for( int i = 0; i< n; i= i+2*width )  
        merge(A, B, i, M, N);  
    copy(B,A,n);  
}  
    M = i+width-1  
    N = min(i+2*width-1,n-1)
```

	i ↓						MN ↓ ↓
A: width=8	7	8	5	6	3	4	2
B:	5	6	7	8	2	3	4

```

void merge_sort_iter( T A[], T B[], int n){
    for(int width = 1; width < n; width *=2){
        for( int i = 0; i < n; i= i+2*width )
            merge(A, B, i, i+width-1, min(i+2*width-1,n-1));
        copy(B,A,n);
    }
}

```

一趟归并: $O(n)$

A: width=8

B:

<div><div>i</div><div>↓</div></div>							<div><div>MN</div><div>↓ ↓</div></div>	
7	8	5	6	3	4	2		
5	6	7	8	2	3	4		

时间复杂度
 $O(n \log_2 n)$

```
void merge_sort_iter( T A[], T B[], int n){
```

```
    for(int width = 1; width < n; width *= 2){
```

```
        for( int i = 0; i < n; i = i + 2 * width )
```

```
            merge(A, B, i, i + width - 1, min(i + 2 * width - 1, n - 1));
```

```
        copy(B, A, n);
```

```
    }
```

```
}
```

k趟, $2^k \geq n$ 停止,
 $k = \log_2 n$

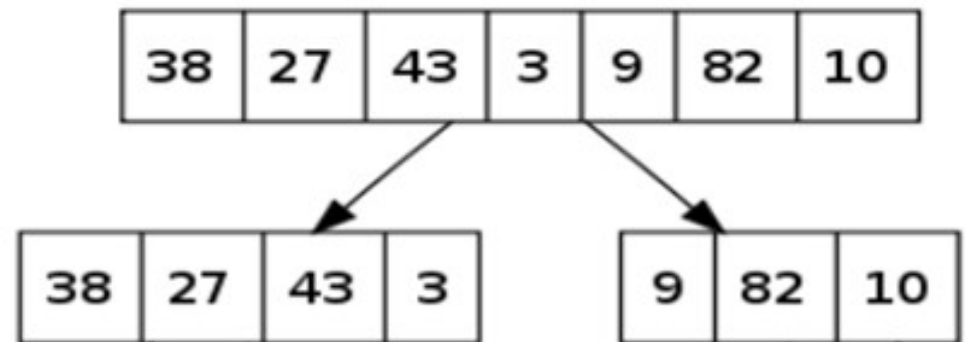
一趟归并: $O(n)$

(递归) 归并排序

递归的归并排序

区间在 $[s, t]$ 的序列的归并排序被分解为区间为 $[s, (s+t)/2]$ 和 $[(s+t)/2+1, t]$ 的归并排序子问题。

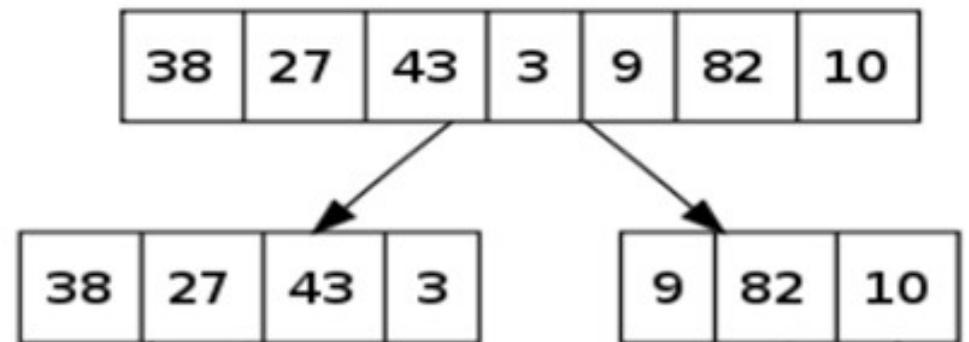
```
merge_sort(T A[], int s, int t) {  
    if (t == s) {return;}  
    merge_sort(A, s, (s+t)/2);  
    merge_sort(A, (s+t)/2+1, t);  
    merge(A, B, s, (s+t)/2, t);  
    copy(B, A, s, t);  
}
```

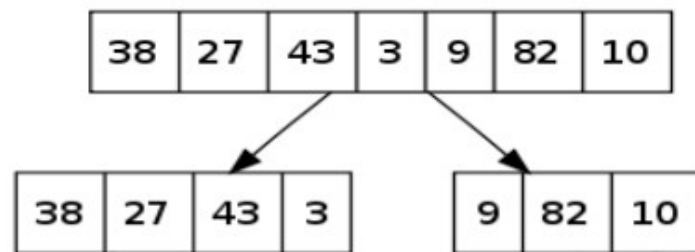


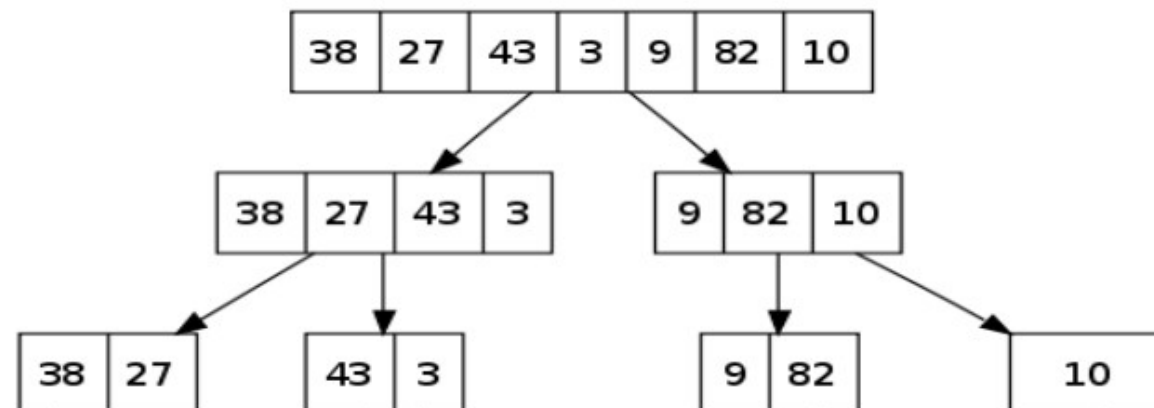
递归的归并排序

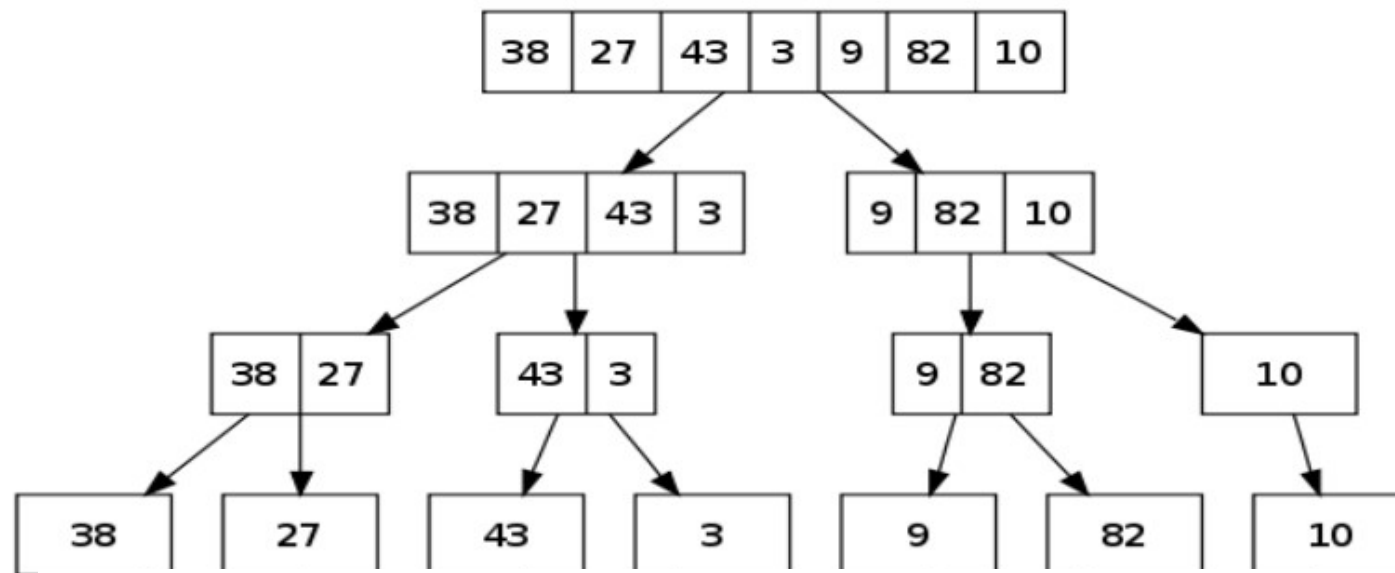
区间在 $[s, t]$ 的序列的归并排序被分解为区间为 $[s, (s+t)/2]$ 和 $[(s+t)/2+1, t]$ 的归并排序子问题。

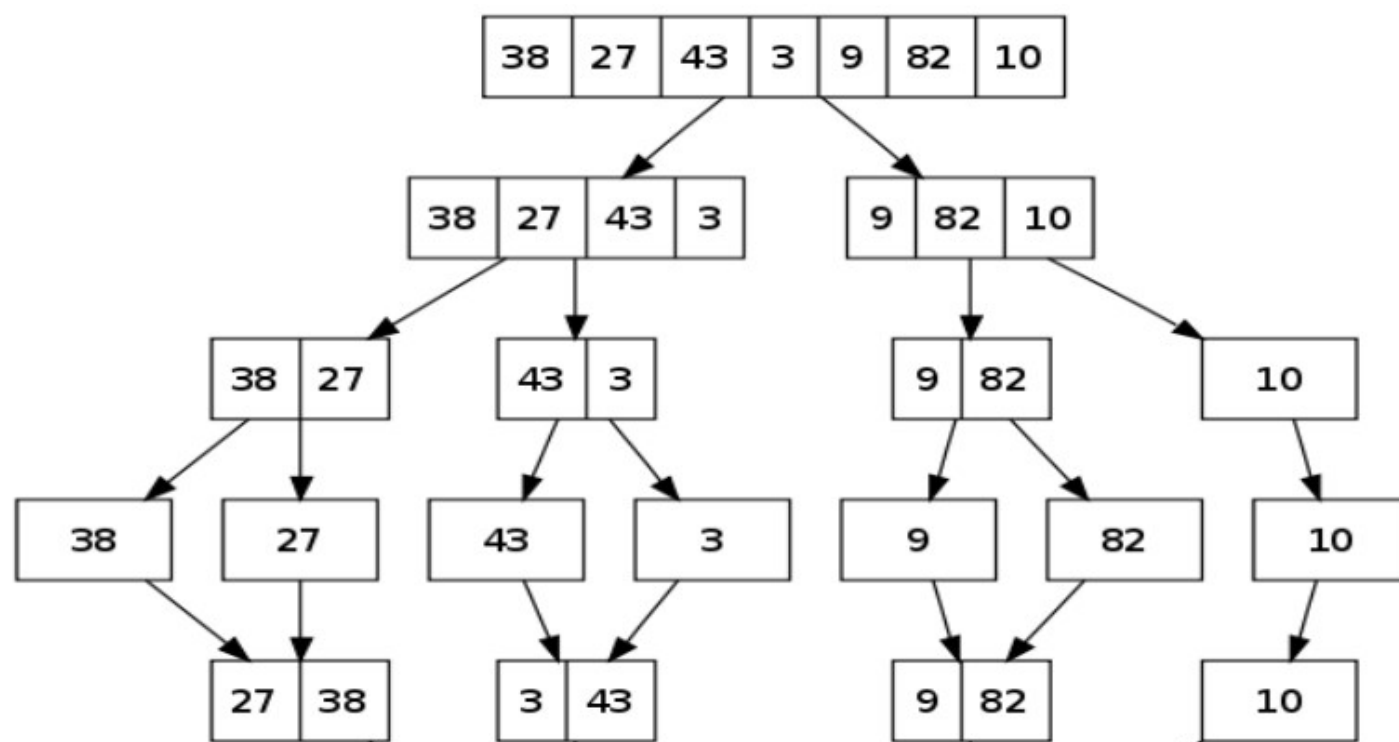
```
merge_sort(T A[], int s, int t) {  
    if (t == s) {return;}  
    merge_sort(A, s, (s+t)/2);  
    merge_sort(A, (s+t)/2+1, t);  
    merge(A, B, s, (s+t)/2, t);  
    copy(B, A, s, t);  
}
```

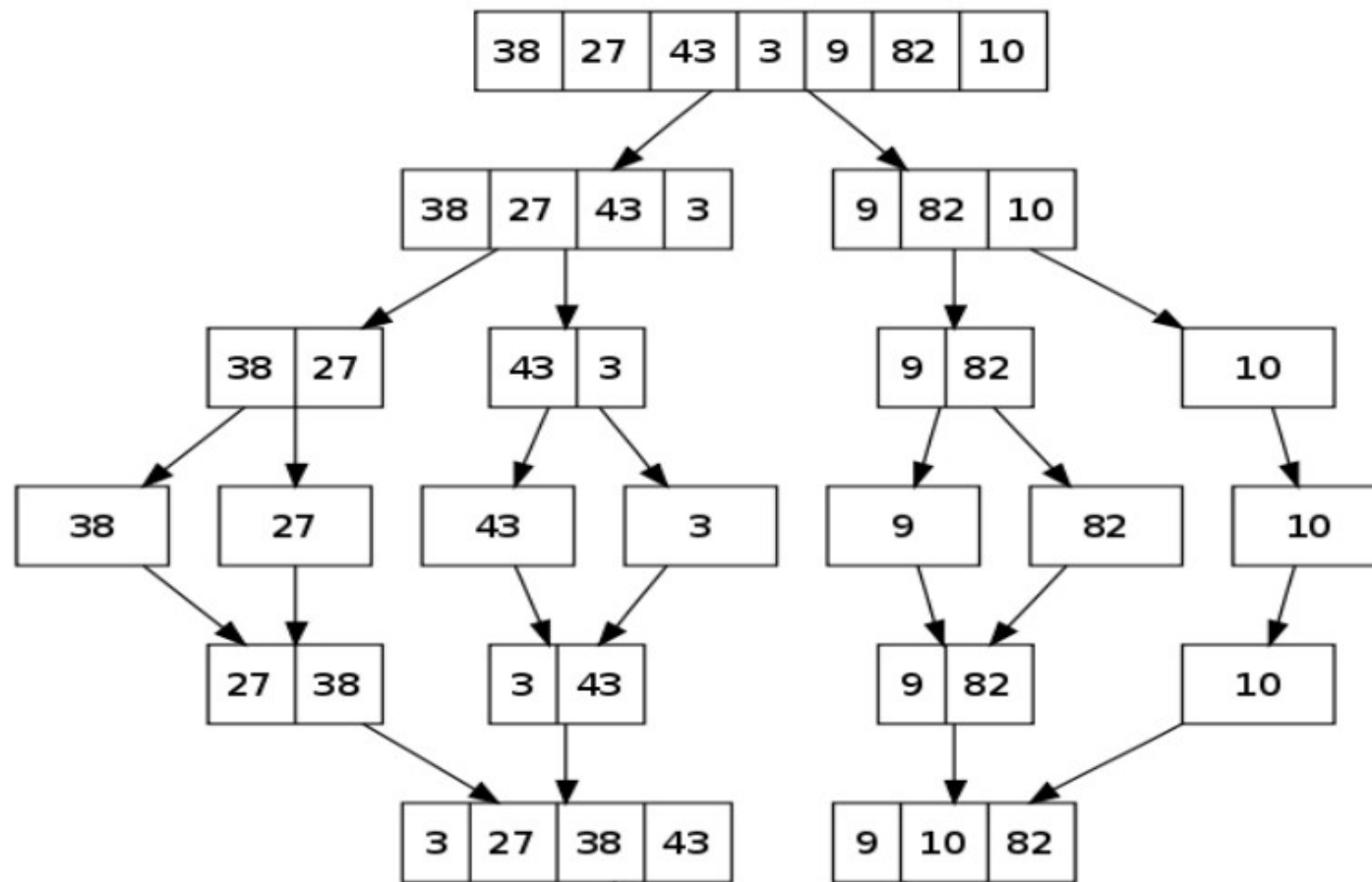


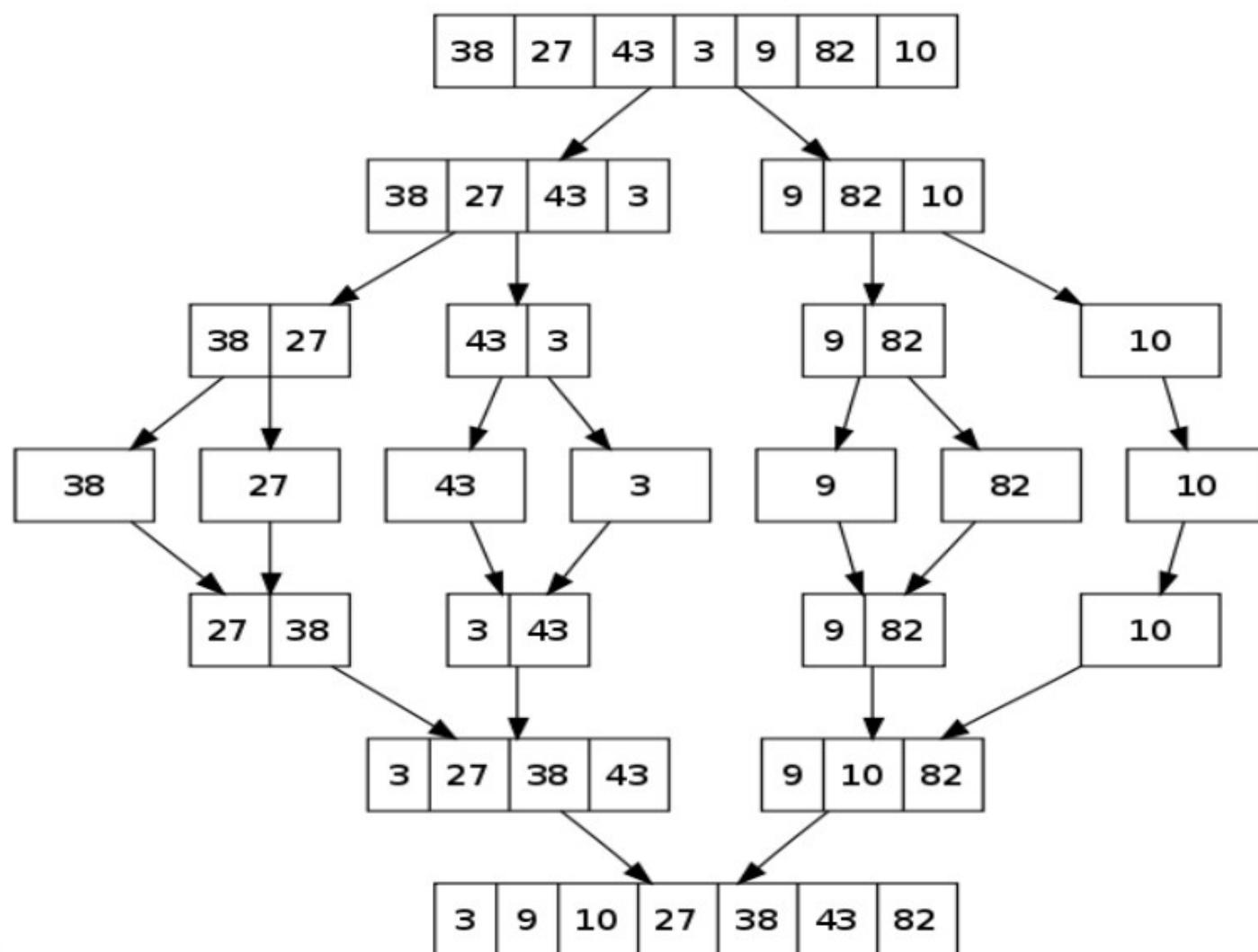












归并排序法

- 时间复杂度
 - $O(n\log_2 n)$
- 空间复杂度
 - 需要另外一个与原待排序序列同样大小的辅助空间
 - 这是这个算法的缺点
- 稳定性
 - 稳定

快速排序

快速排序

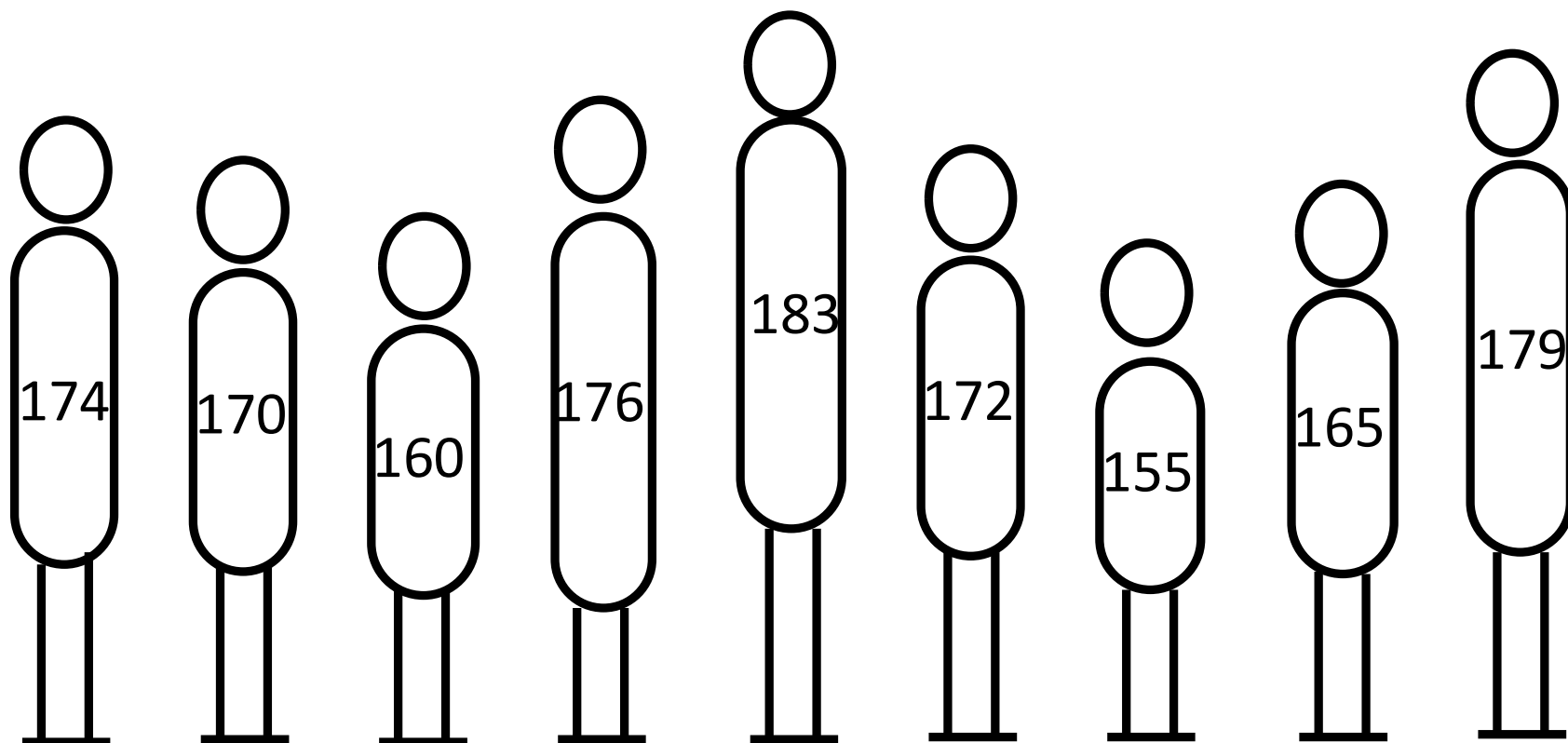
- 递归：一个问题可以分解为规模更小的子问题，子问题的解决方法与原问题一样。
- solve(原问题): solve(子问题1)、 solve(子问题2)

$$n! = n * (n-1)!$$

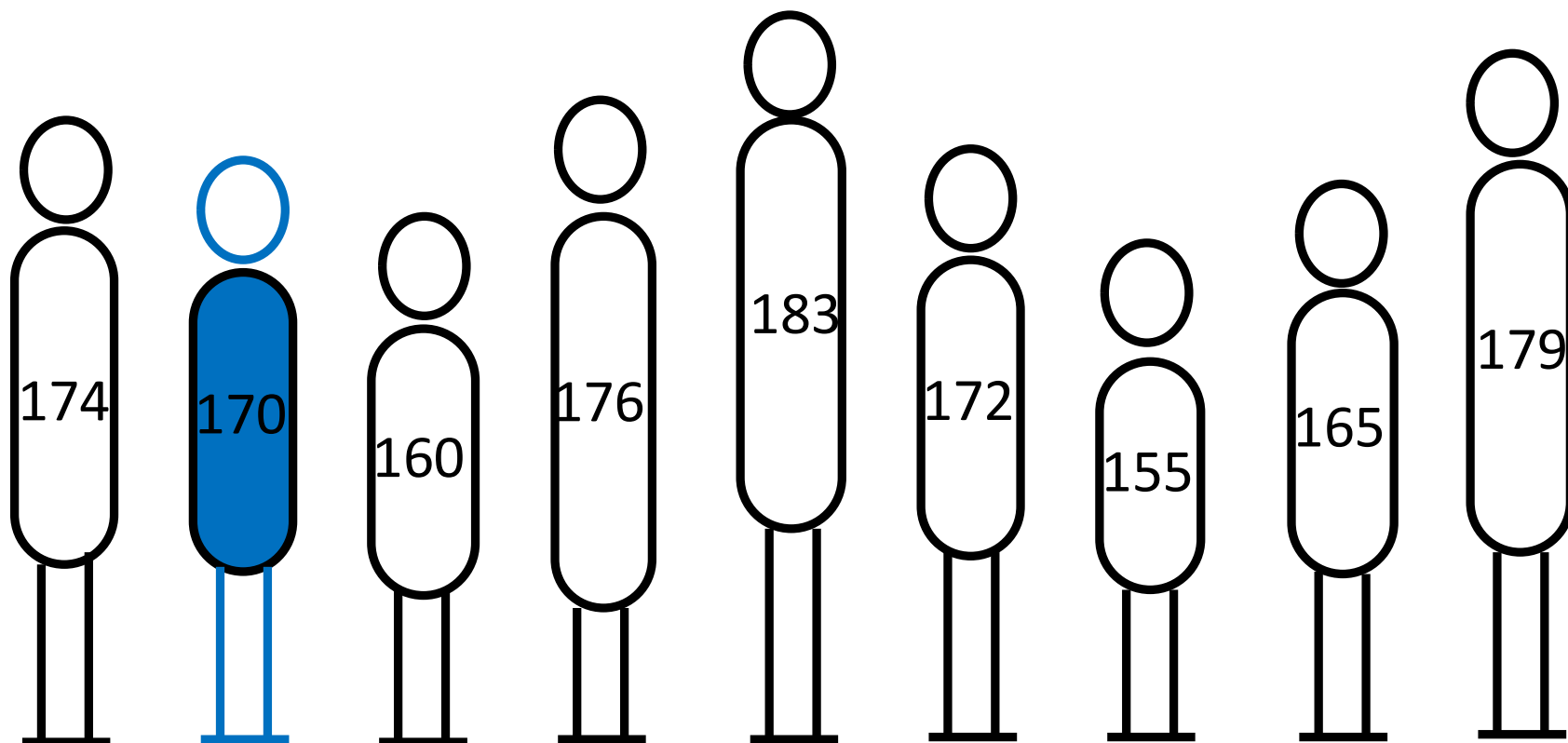
```
int fact(int n){  
    if(n<=1) return 1;  
    return = n * fact(n-1);  
}
```

$$\begin{aligned} \text{fact}(4) &= 4 * \text{fact}(3) \\ &= 4 * 3 * \text{fact}(2) \\ &= 4 * 3 * 2 * \text{fact}(1) \\ &= 4 * 3 * 2 * 1 \\ &= 4 * 3 * 2 \\ &= 4 * 6 \\ &= 24 \end{aligned}$$

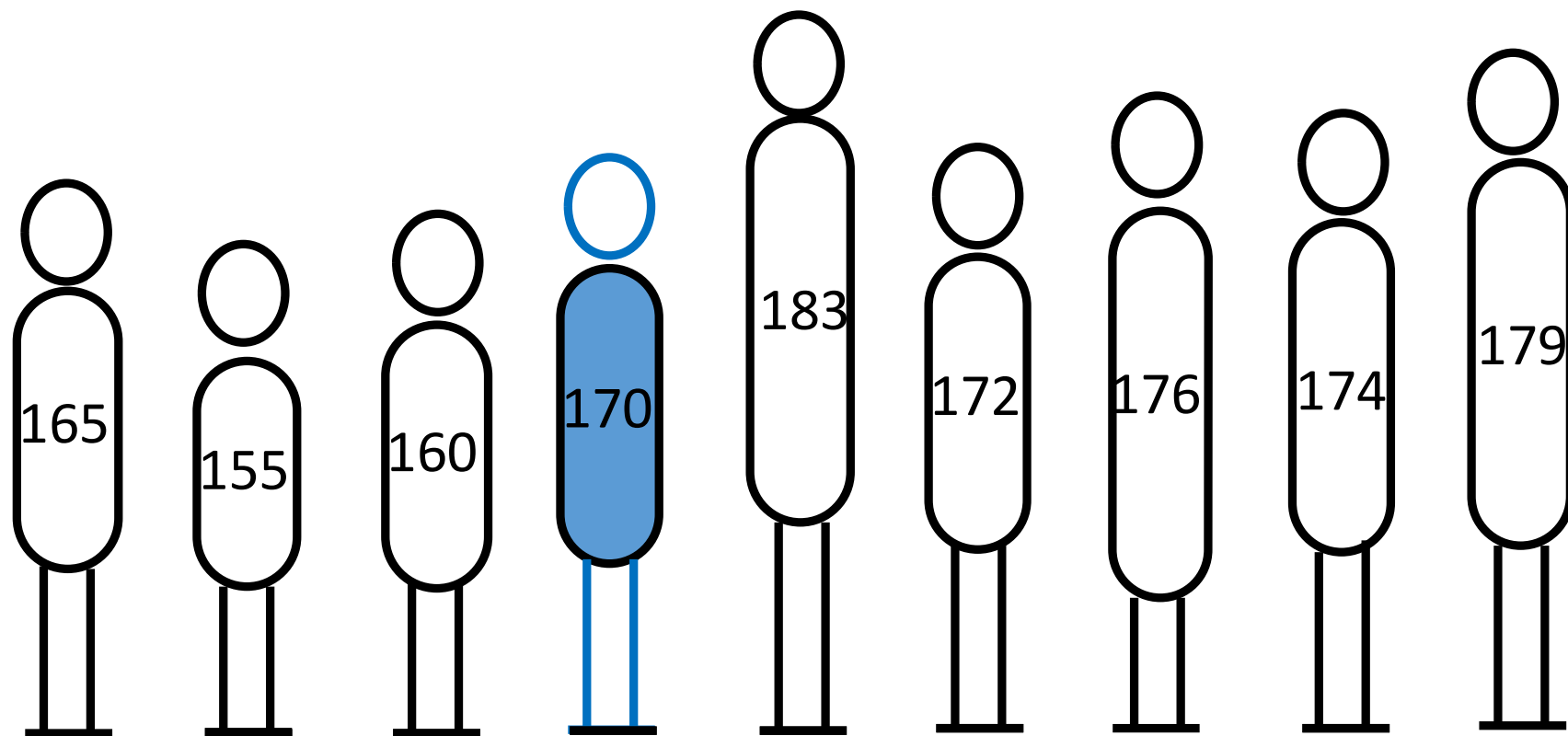
快速排序



一次划分

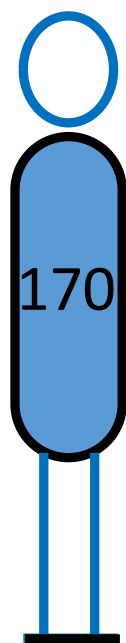
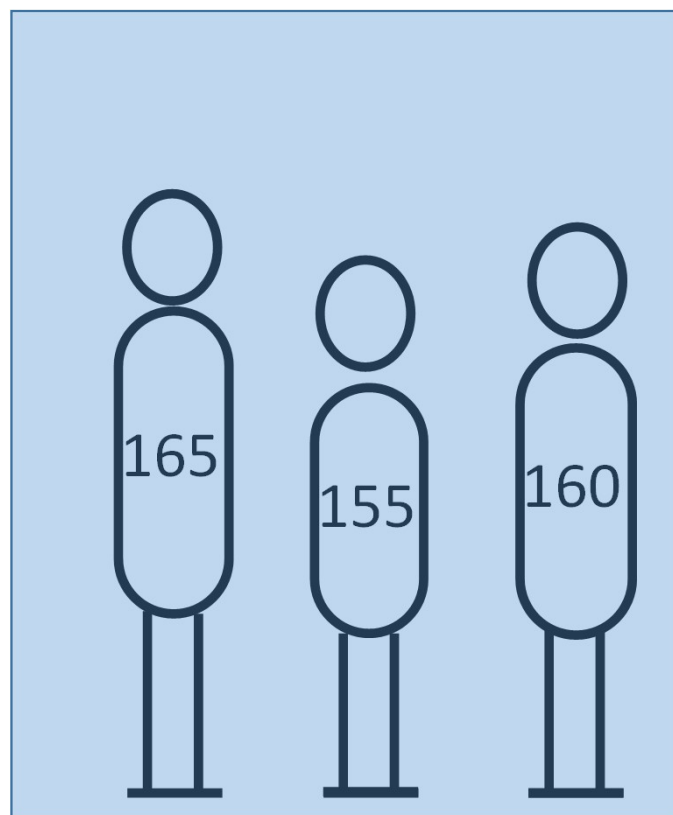


一次划分



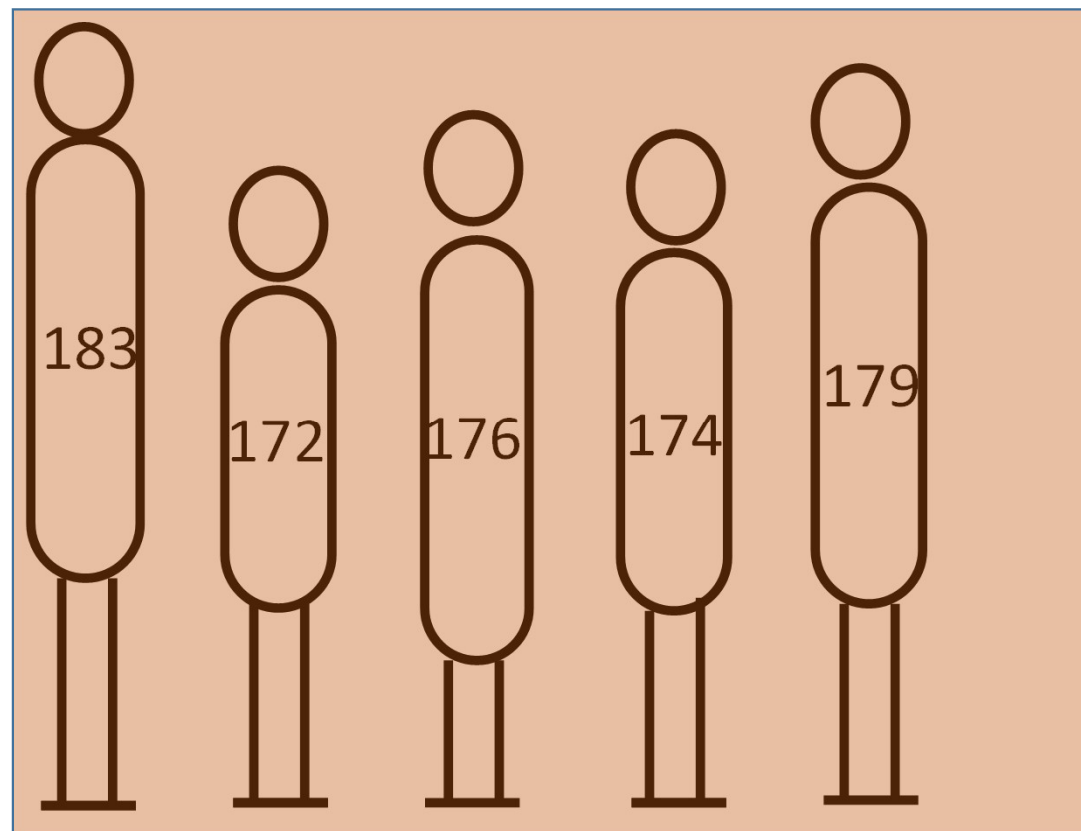
基准元素位置

快速排序

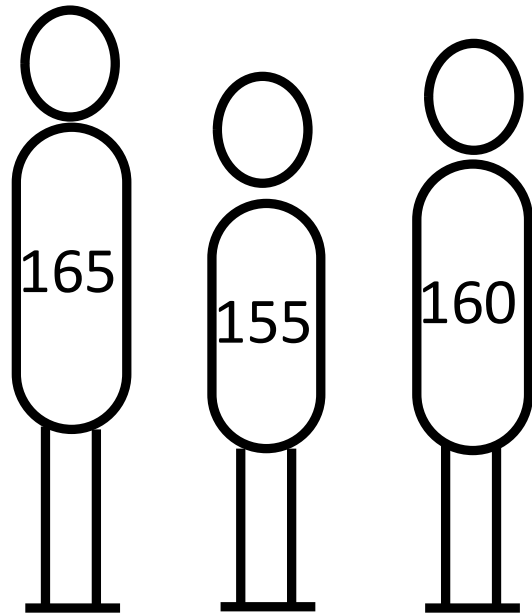


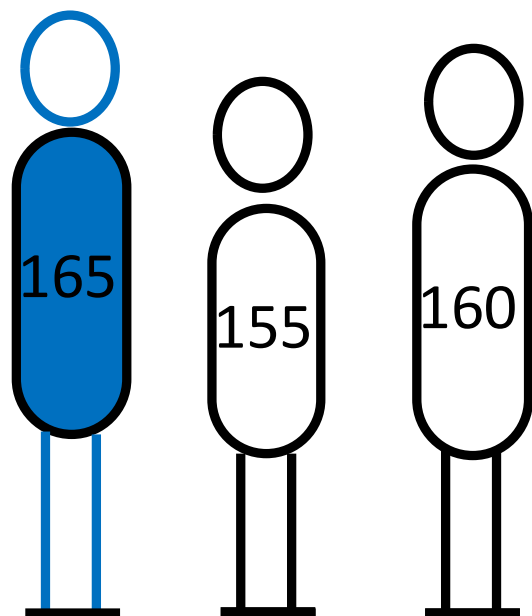
基准元素位置

快速排序

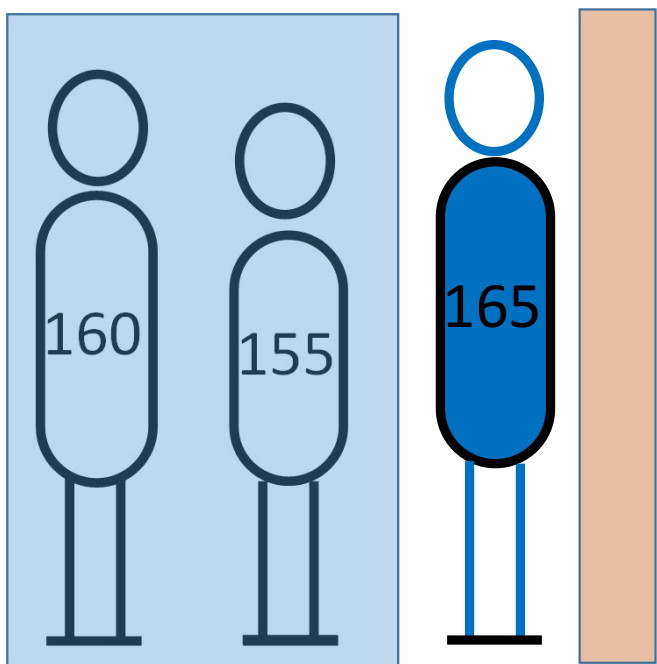


快速排序



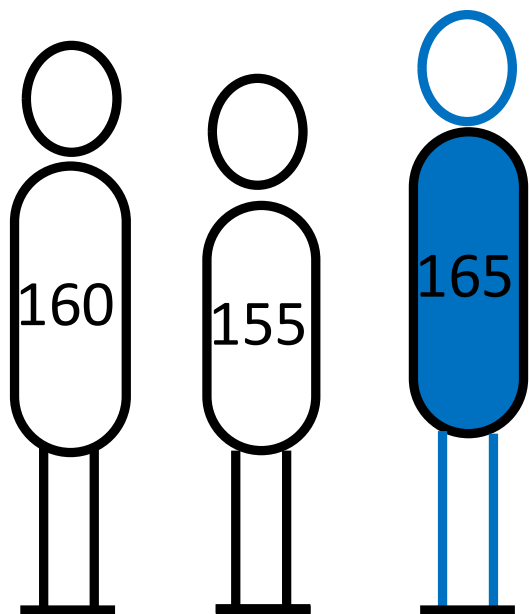


基准元素

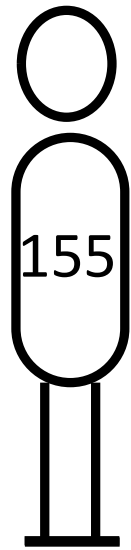
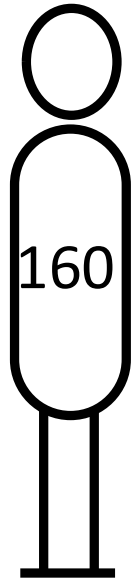


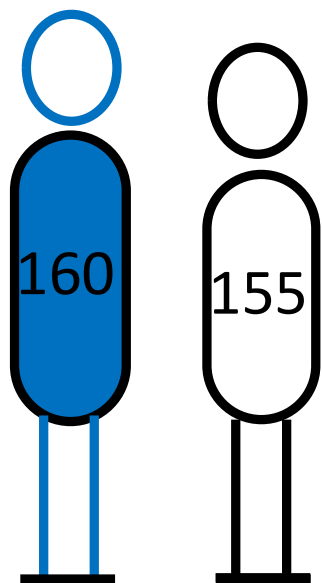
基准元素位置

一次划分

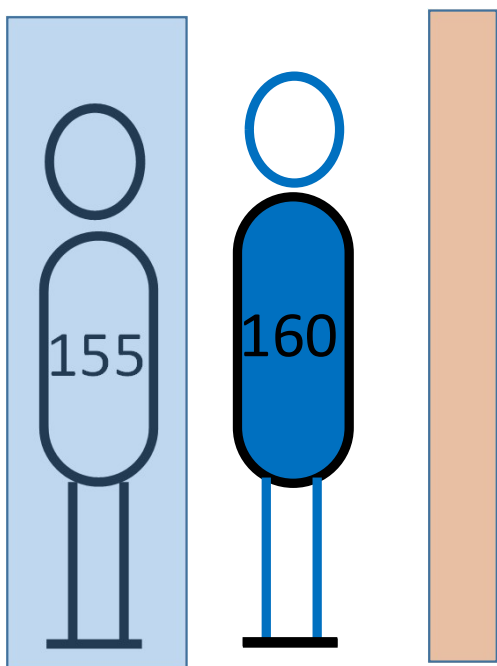


基准元素位置



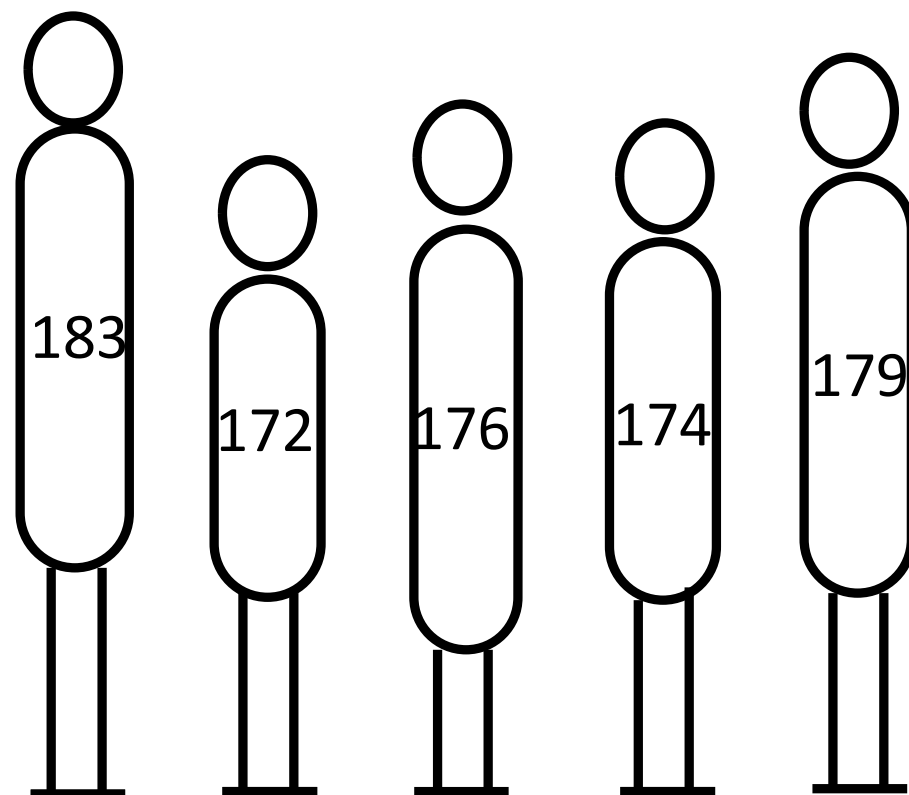


基准元素

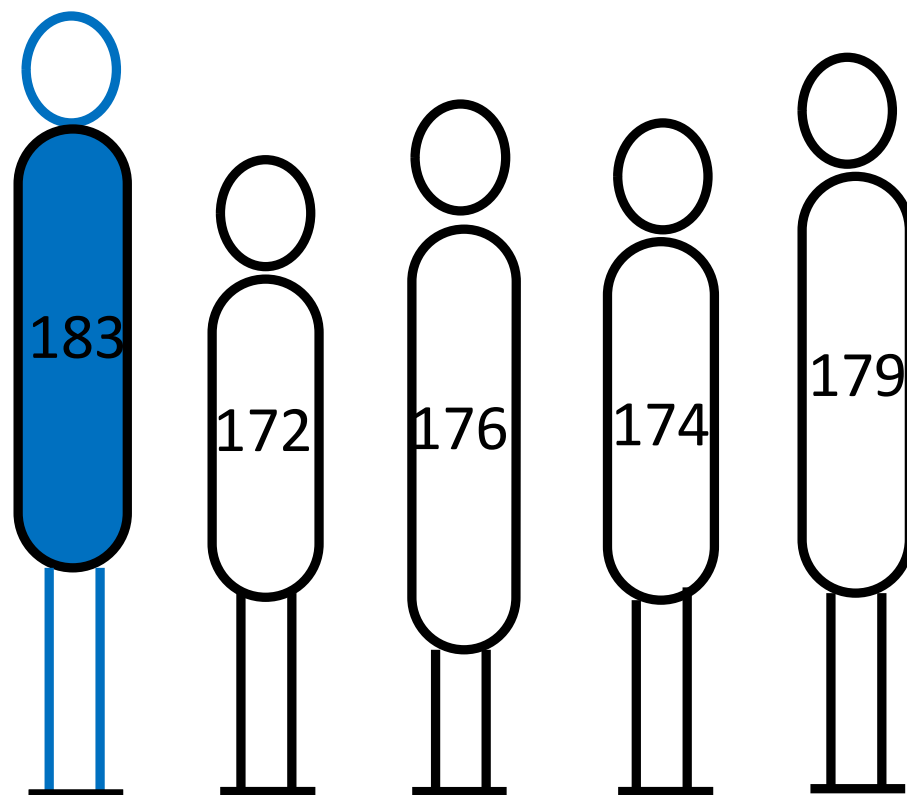


基准元素位置

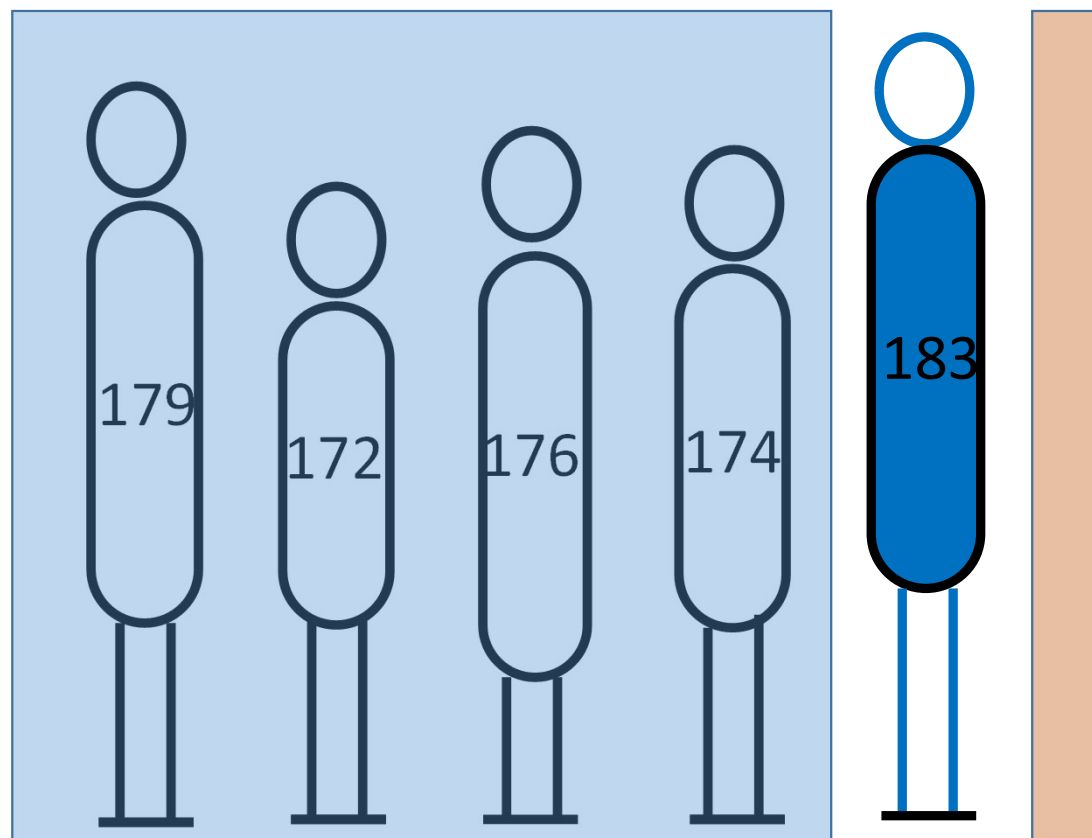
快速排序



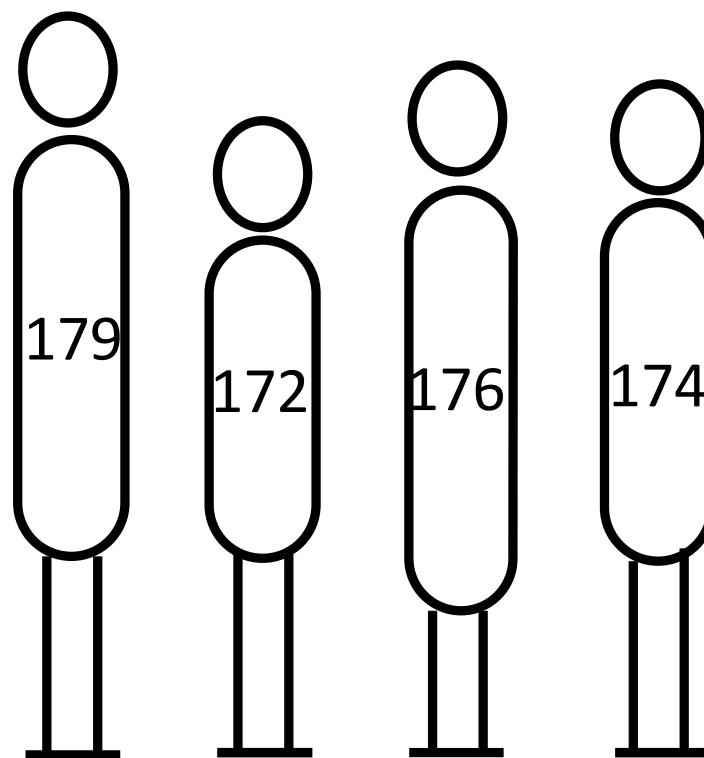
快速排序



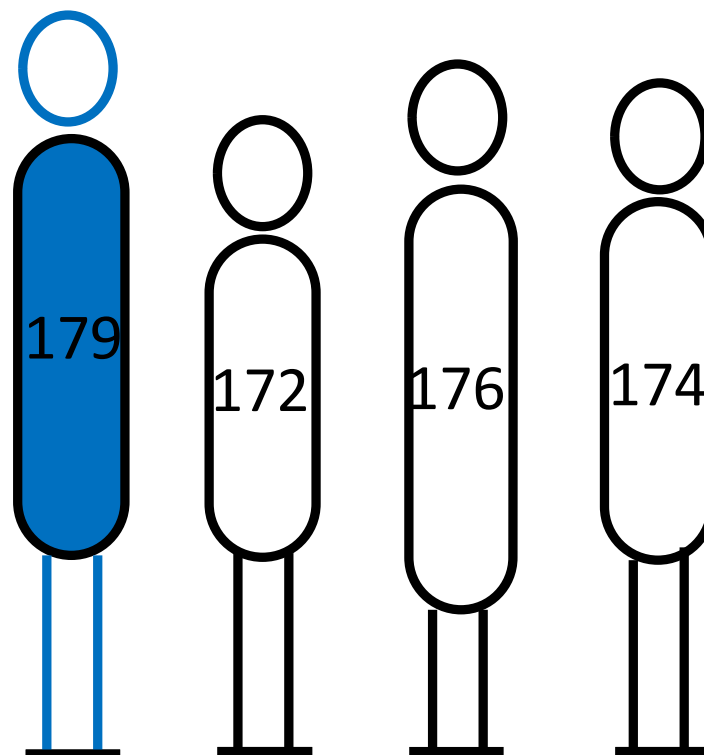
快速排序



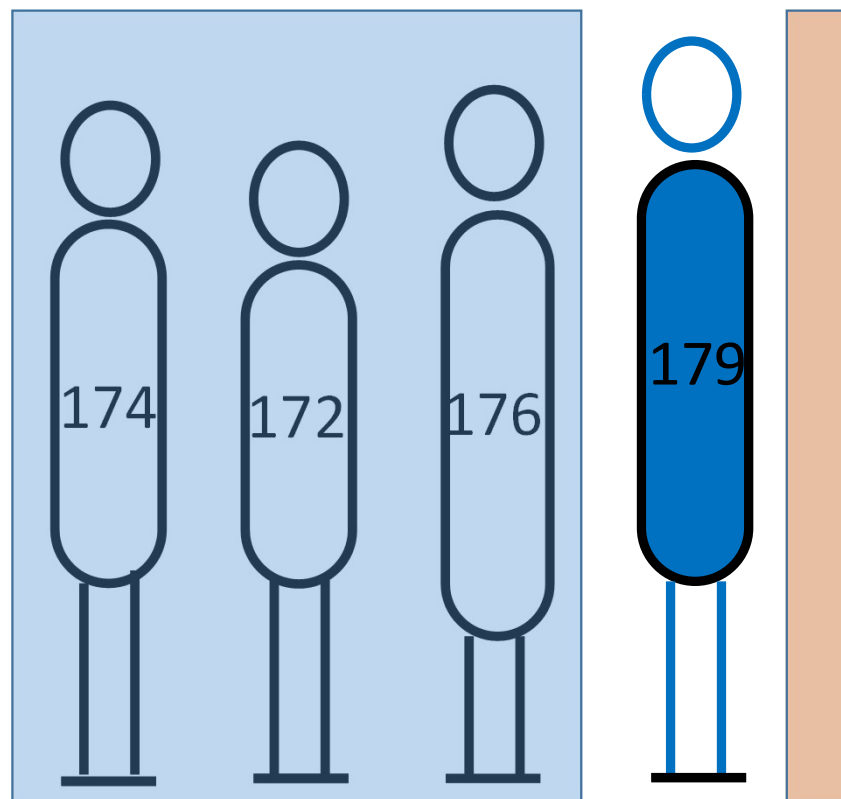
快速排序



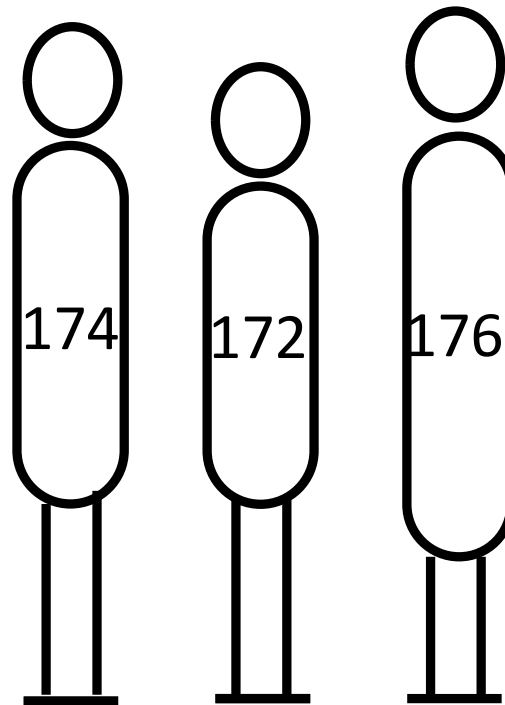
快速排序



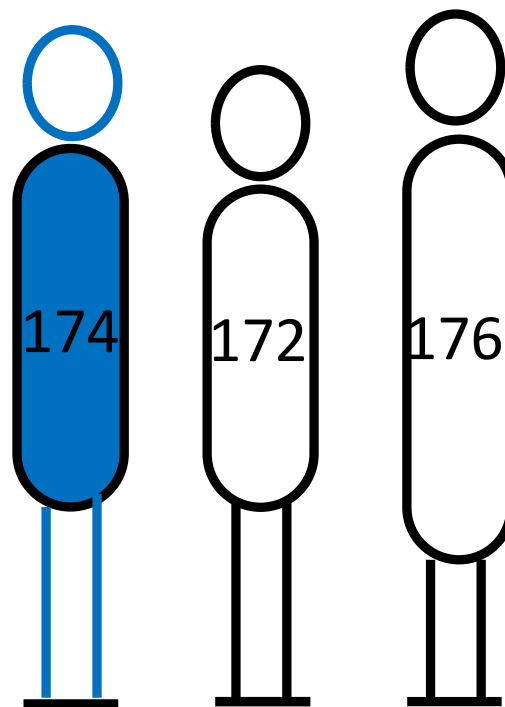
快速排序



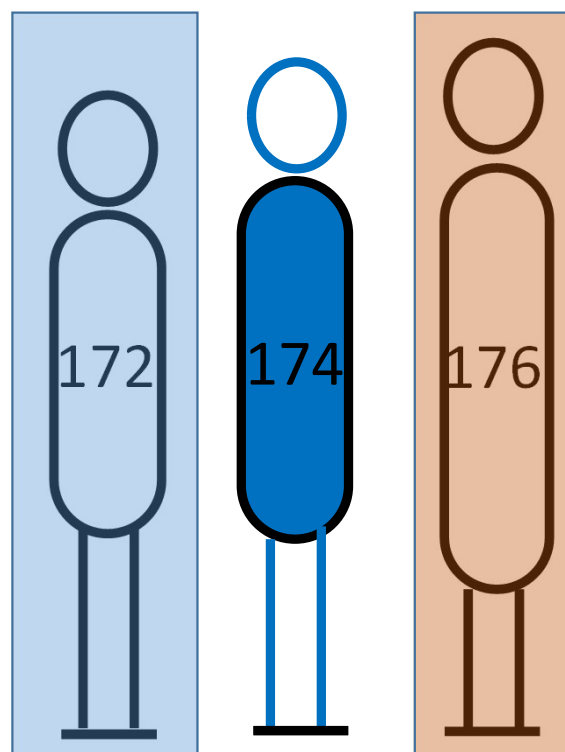
快速排序



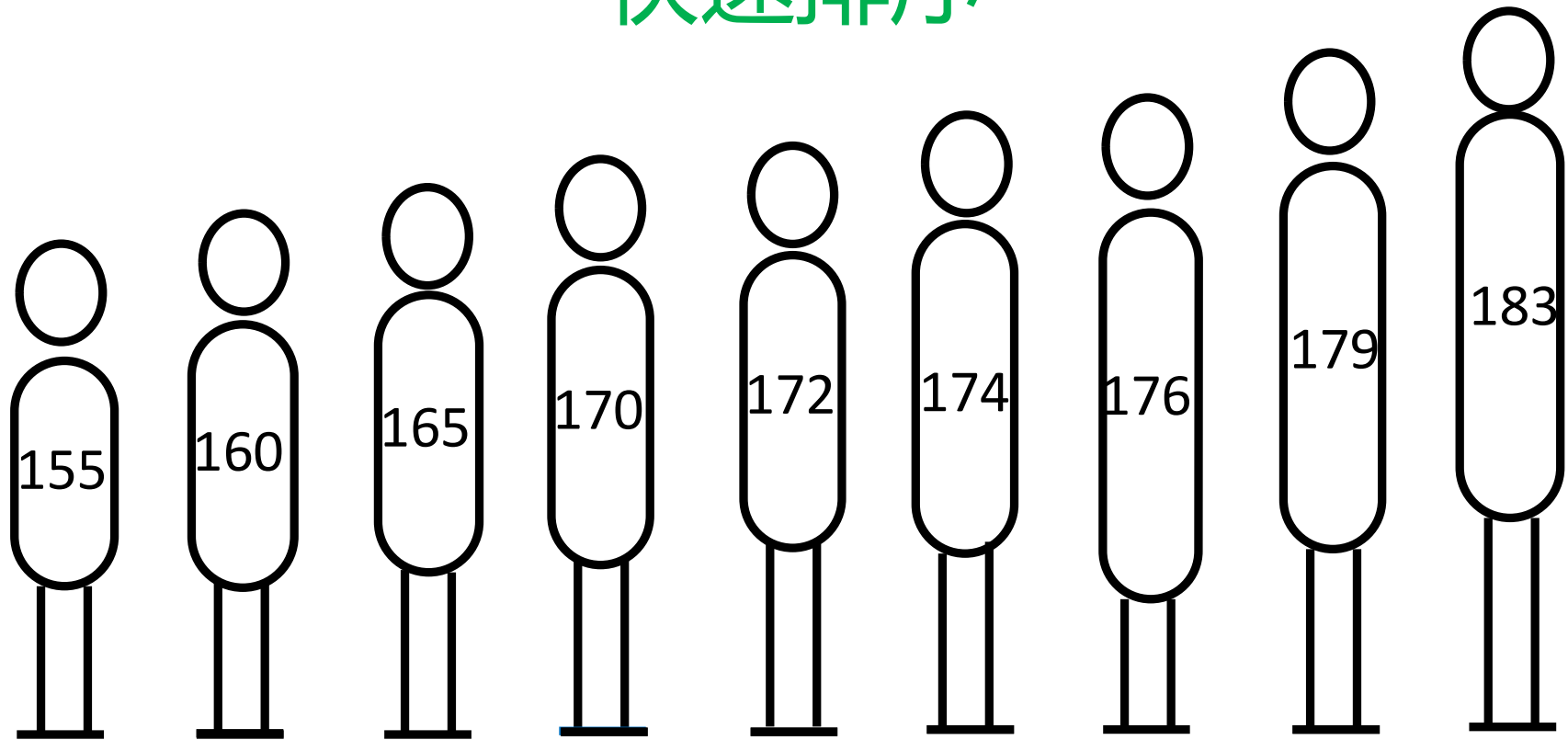
快速排序



快速排序



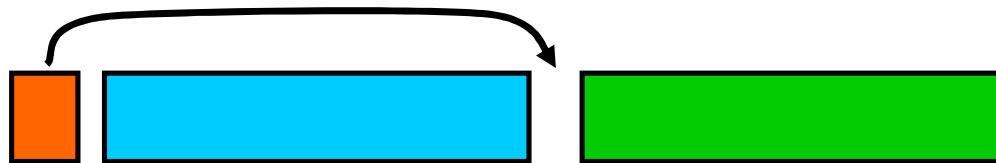
快速排序



快速排序

- 基本思想

- 以某一个数据(例如第一个)作为基准, 将整个序列 “划分” 为左右两个子序列:
 - 左侧子序列中所有数据都小于基准数据
 - 右侧子序列中所有数据都不小于基准数据
- 这时基准对象就排在这两个子序列中间
- 然后分别对这两个子序列重复施行上述方法, 直到排序完毕



快速排序

- 递归函数

```
void QSort(T a[], int L, int H) {  
    if (L < H) { //待排序数列长度大于1  
        int pivotloc = Partition(a, L, H);  
        //对左子序列进行快速排序  
        QSort(a, L, pivotloc - 1);  
        //对右子序列进行快速排序  
        QSort(a, pivotloc + 1, H);  
    }  
}
```

Qsort([70, 74, 60, 76, 83, 72, 55, 65, 79])

划分: [65, 55, 60] , 70, [83, 72, 76, 74, 79]

Qsort([65, 55, 60])

```
void QSort(T a[], int L, int H) {  
    if(L < H){ //待排序数列长度大于1  
        int pivotloc = Partition(a, L, H);  
        //对左子序列进行快速排序  
        QSort(a, L, pivotloc - 1);  
        //对右子序列进行快速排序  
        QSort(a, pivotloc + 1, H);  
    }  
}
```

Qsort([70, 74, 60, 76, 83, 72, 55, 65, 79])

划分: [65, 55, 60] , 70, [83, 72, 76, 74, 79]

Qsort([65, 55, 60])

划分: [60, 55], 65, []

Qsort([60, 55])

```
void QSort(T a[], int L, int H) {  
    if(L < H){ //待排序数列长度大于1  
        int pivotloc = Partition(a, L, H);  
        //对左子序列进行快速排序  
        QSort(a, L, pivotloc - 1);  
        //对右子序列进行快速排序  
        QSort(a, pivotloc + 1, H);  
    }  
}
```

Qsort([70, 74, 60, 76, 83, 72, 55, 65, 79])

划分: [65, 55, 60] , 70, [83, 72, 76, 74, 79]

Qsort([65, 55, 60])

划分: [60, 55], 65, []

Qsort([60, 55])

划分: [55], 60, []

Qsort([55])

Qsort([])

```
void QSort(T a[], int L, int H) {  
    if(L < H){ //待排序数列长度大于1  
        int pivotloc = Partition(a, L, H);  
        //对左子序列进行快速排序  
        QSort(a, L, pivotloc - 1);  
        //对右子序列进行快速排序  
        QSort(a, pivotloc + 1, H);  
    }  
}
```

Qsort([70, 74, 60, 76, 83, 72, 55, 65, 79])

划分: [65, 55, 60] , 70, [83, 72, 76, 74, 79]

Qsort([65, 55, 60])

划分: [60, 55], 65, []

Qsort([60, 55])

划分: [55], 60, []

Qsort([55])

Qsort([])

Qsort([])

```
void QSort(T a[], int L, int H) {  
    if(L < H){ //待排序数列长度大于1  
        int pivotloc = Partition(a, L, H);  
        //对左子序列进行快速排序  
        QSort(a, L, pivotloc - 1);  
        //对右子序列进行快速排序  
        QSort(a, pivotloc + 1, H);  
    }  
}
```

Qsort([70, 74, 60, 76, 83, 72, 55, 65, 79])

划分: [65, 55, 60], 70, [83, 72, 76, 74, 79]

Qsort([65, 55, 60])

划分: [60, 55], 65, []

Qsort([60, 55])

划分: [55], 60, []

Qsort([55])

Qsort([])

Qsort([])

```
void QSort(T a[], int L, int H) {  
    if(L < H){ //待排序数列长度大于1  
        int pivotloc = Partition(a, L, H);  
        //对左子序列进行快速排序  
        QSort(a, L, pivotloc - 1);  
        //对右子序列进行快速排序  
        QSort(a, pivotloc + 1, H);  
    }  
}
```

Qsort([70, 74, 60, 76, 83, 72, 55, 65, 79])

划分: [65, 55, 60] , 70, [83, 72, 76, 74, 79]

Qsort([65, 55, 60])

划分: [60, 55], 65, []

Qsort([60, 55])

划分: [55], 60, []

Qsort([55])

Qsort([])

Qsort([])

Qsort([83, 72, 76, 74, 79])

```
void QSort(T a[], int L, int H) {  
    if(L < H){ //待排序数列长度大于1  
        int pivotloc = Partition(a, L, H);  
        //对左子序列进行快速排序  
        QSort(a, L, pivotloc - 1);  
        //对右子序列进行快速排序  
        QSort(a, pivotloc + 1, H);  
    }  
}
```


Qsort([70, 74, 60, 76, 83, 72, 55, 65, 79])

划分: [65, 55, 60], 70, [83, 72, 76, 74, 79]

Qsort([65, 55, 60])

划分: [60, 55], 65, []

Qsort([60, 55])

划分: [55], 60, []

Qsort([55])

Qsort([])

Qsort([])

Qsort([83, 72, 76, 74, 79])

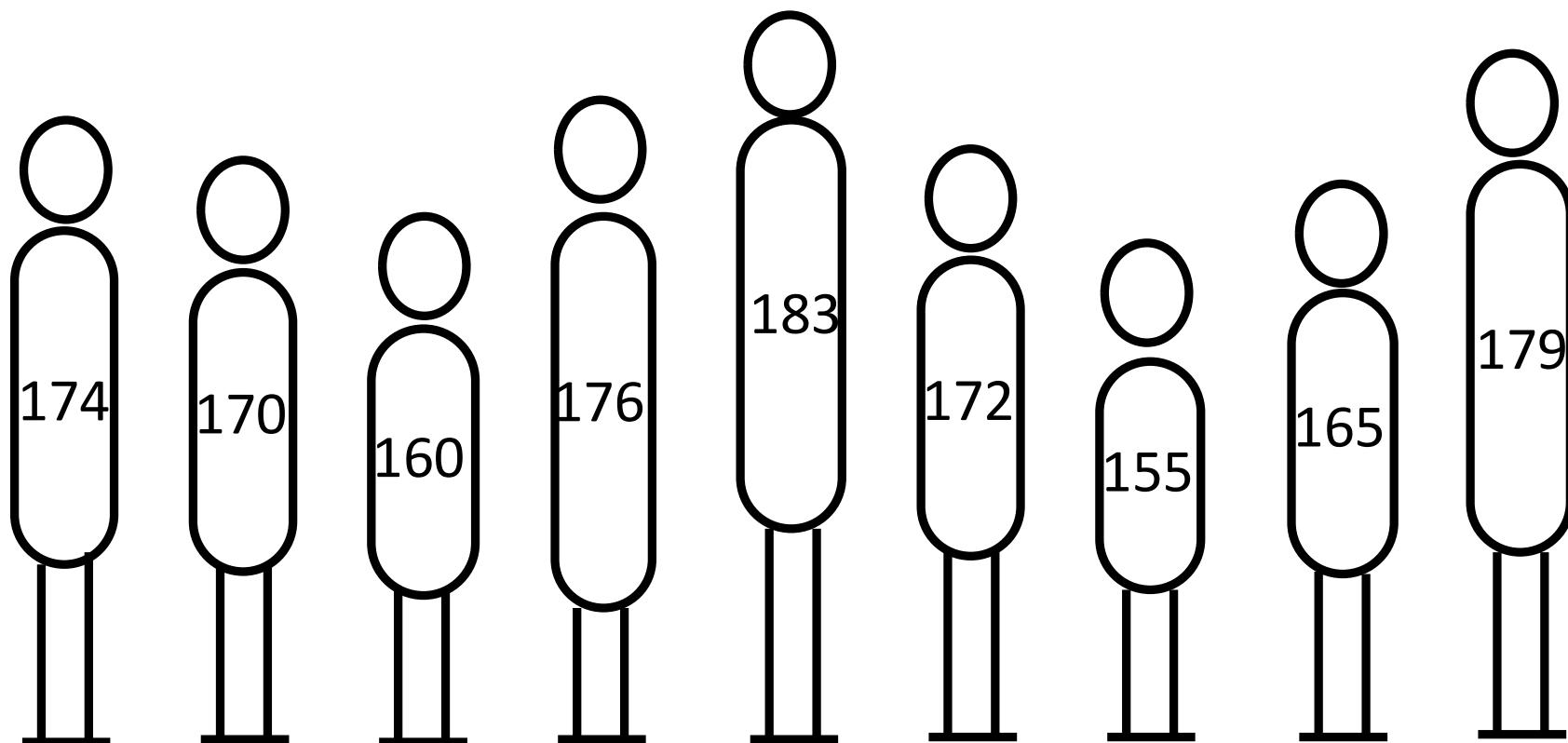
.

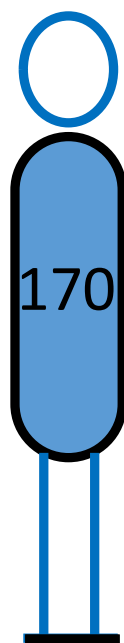
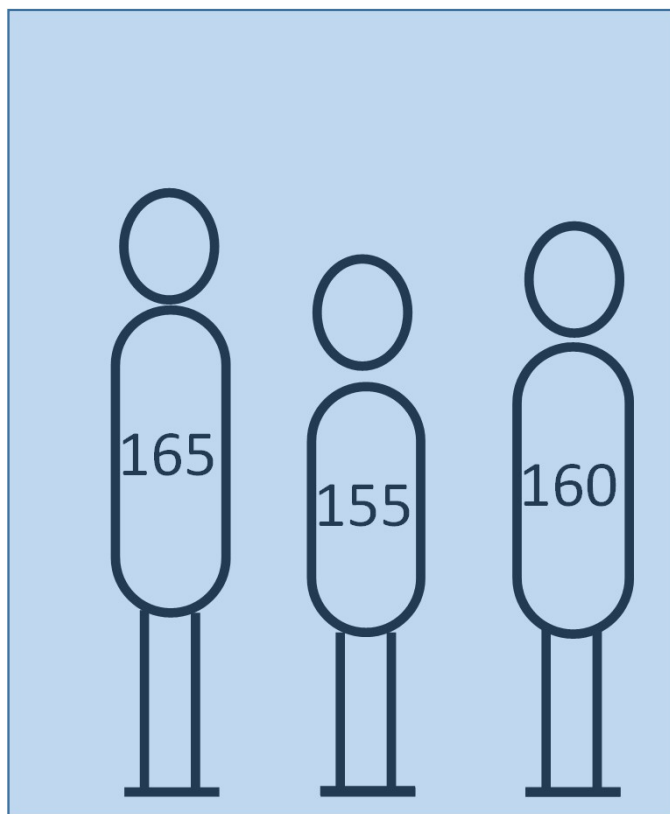
.

.

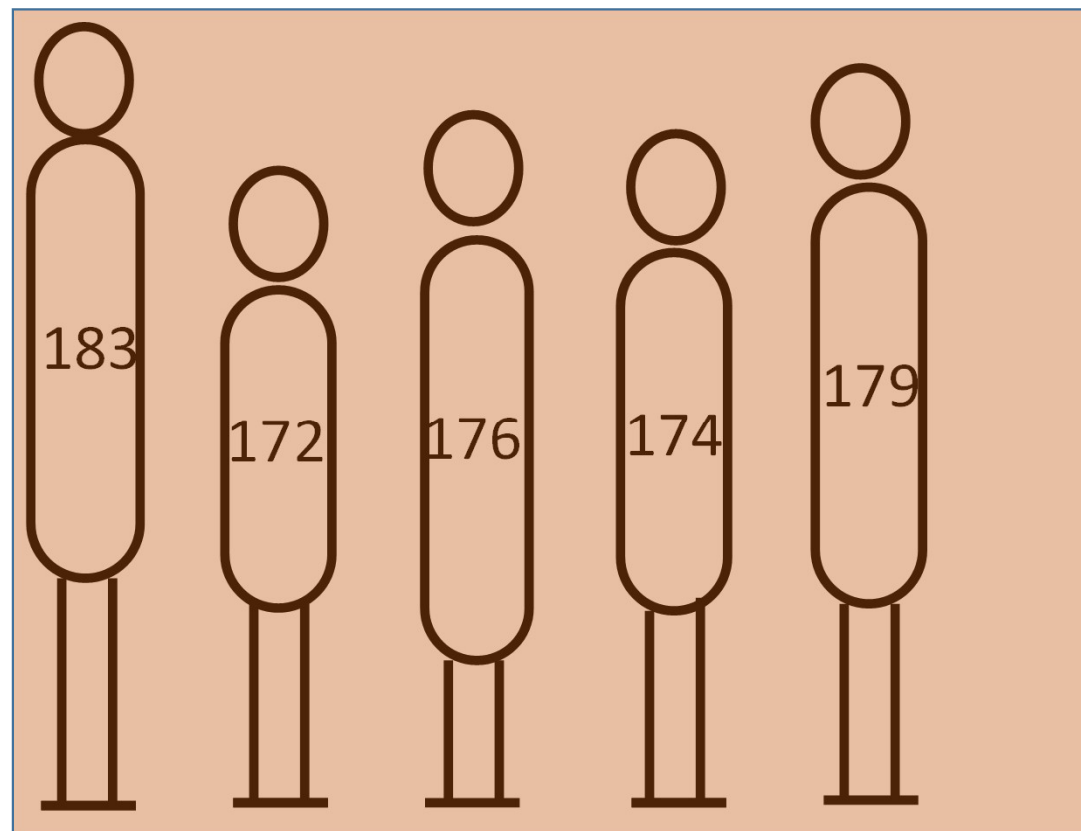
```
void QSort(T a[], int L, int H) {  
    if(L < H){ //待排序数列长度大于1  
        int pivotloc = Partition(a, L, H);  
        //对左子序列进行快速排序  
        QSort(a, L, pivotloc - 1);  
        //对右子序列进行快速排序  
        QSort(a, pivotloc + 1, H);  
    }  
}
```

一次划分

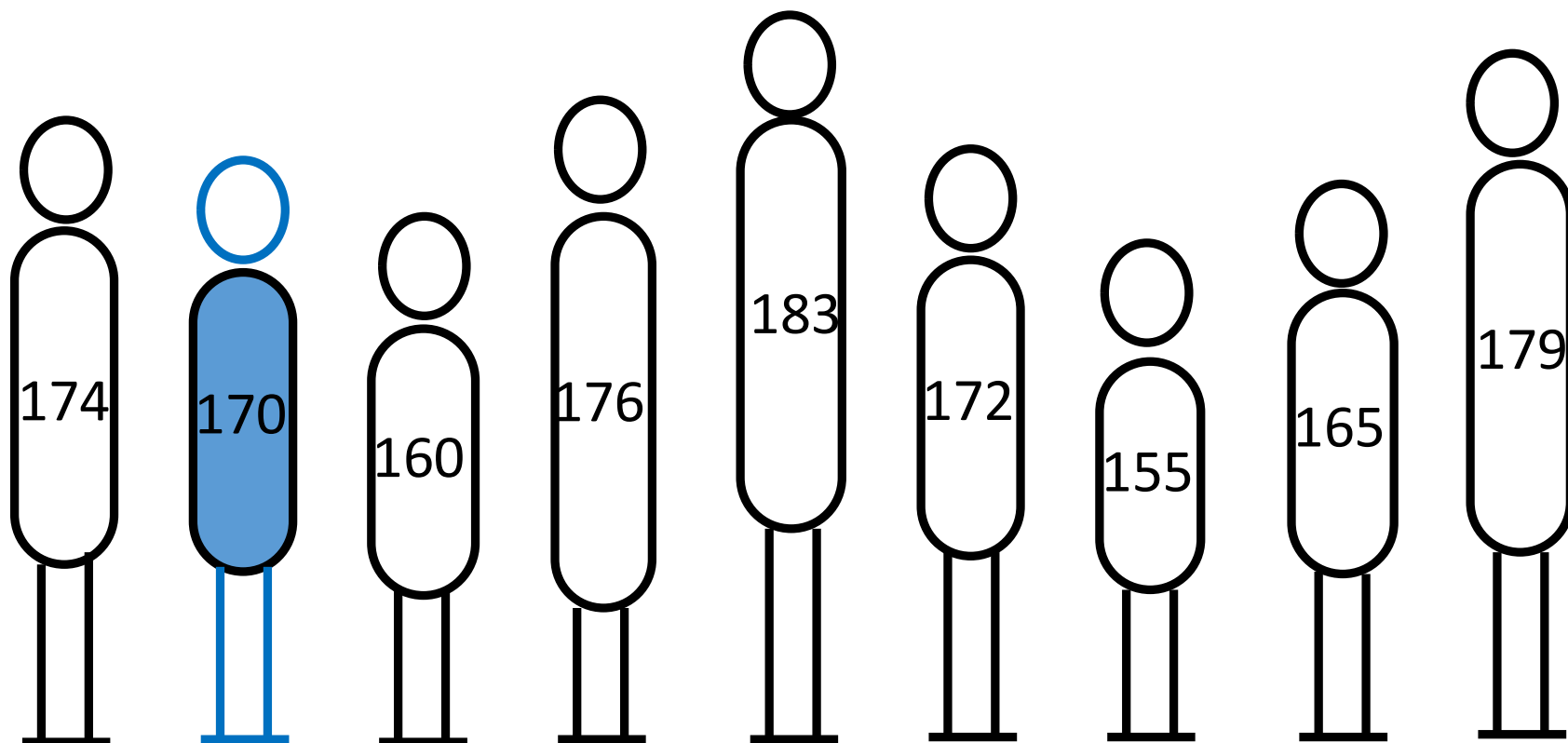


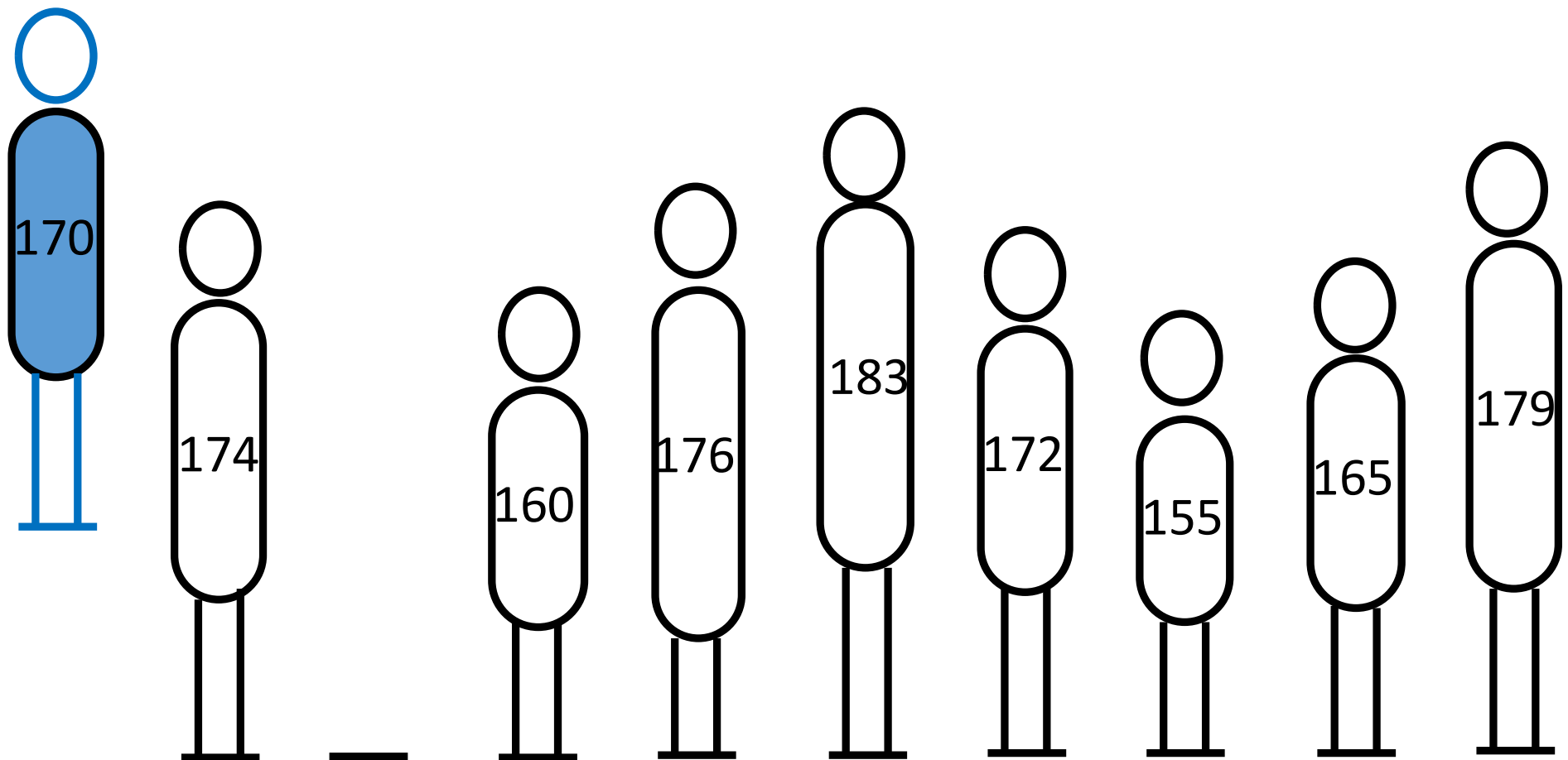


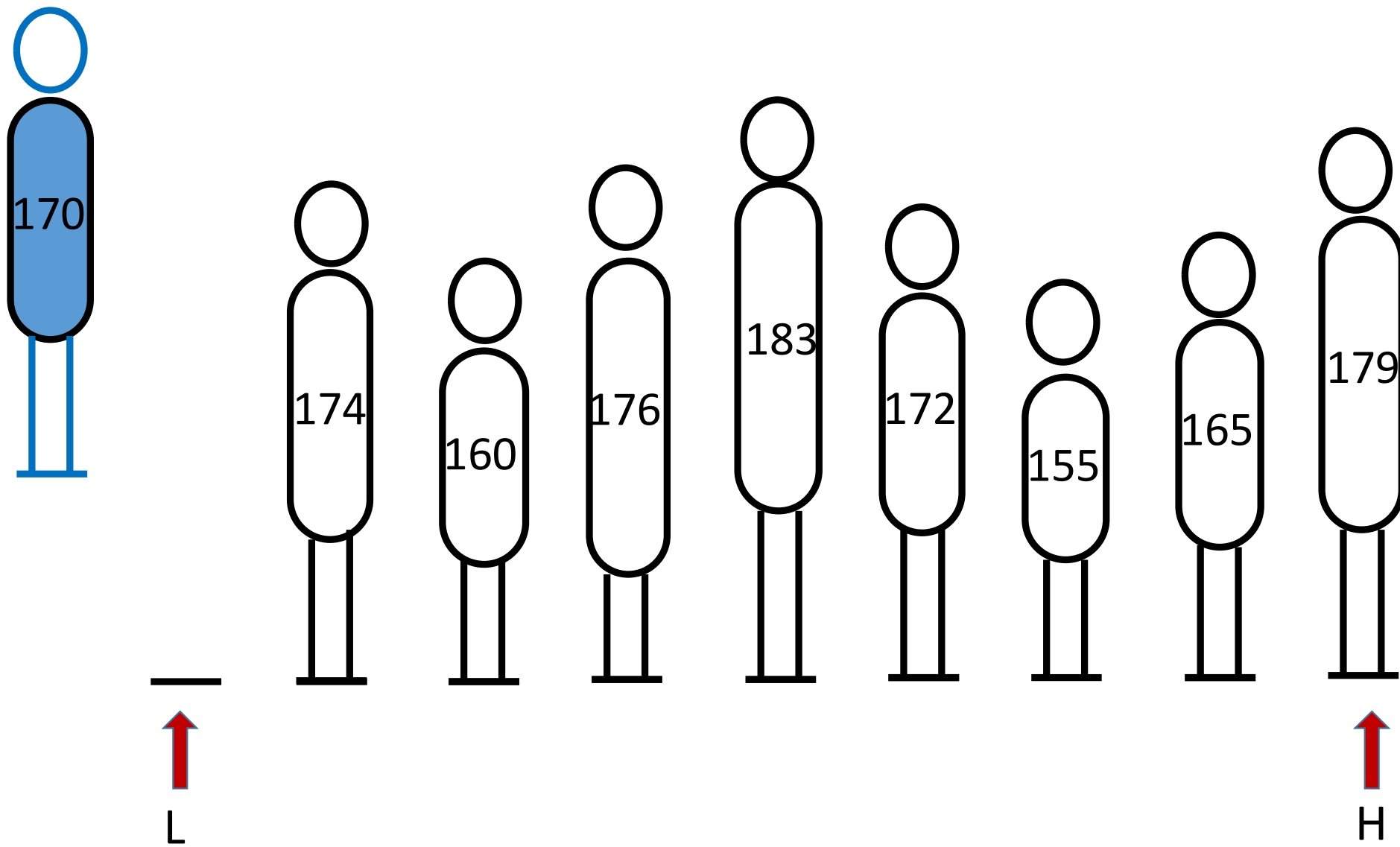
↑
UH

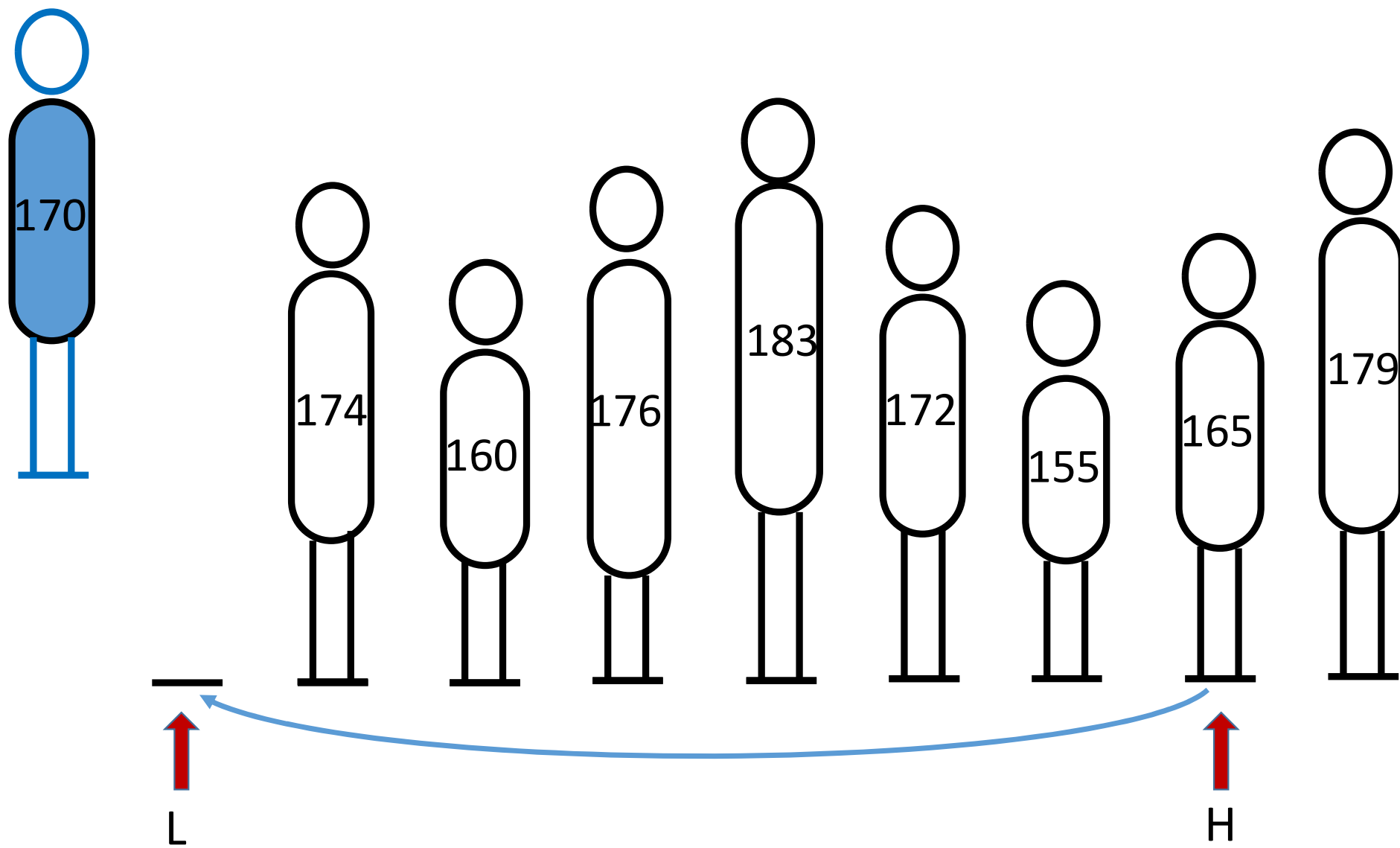


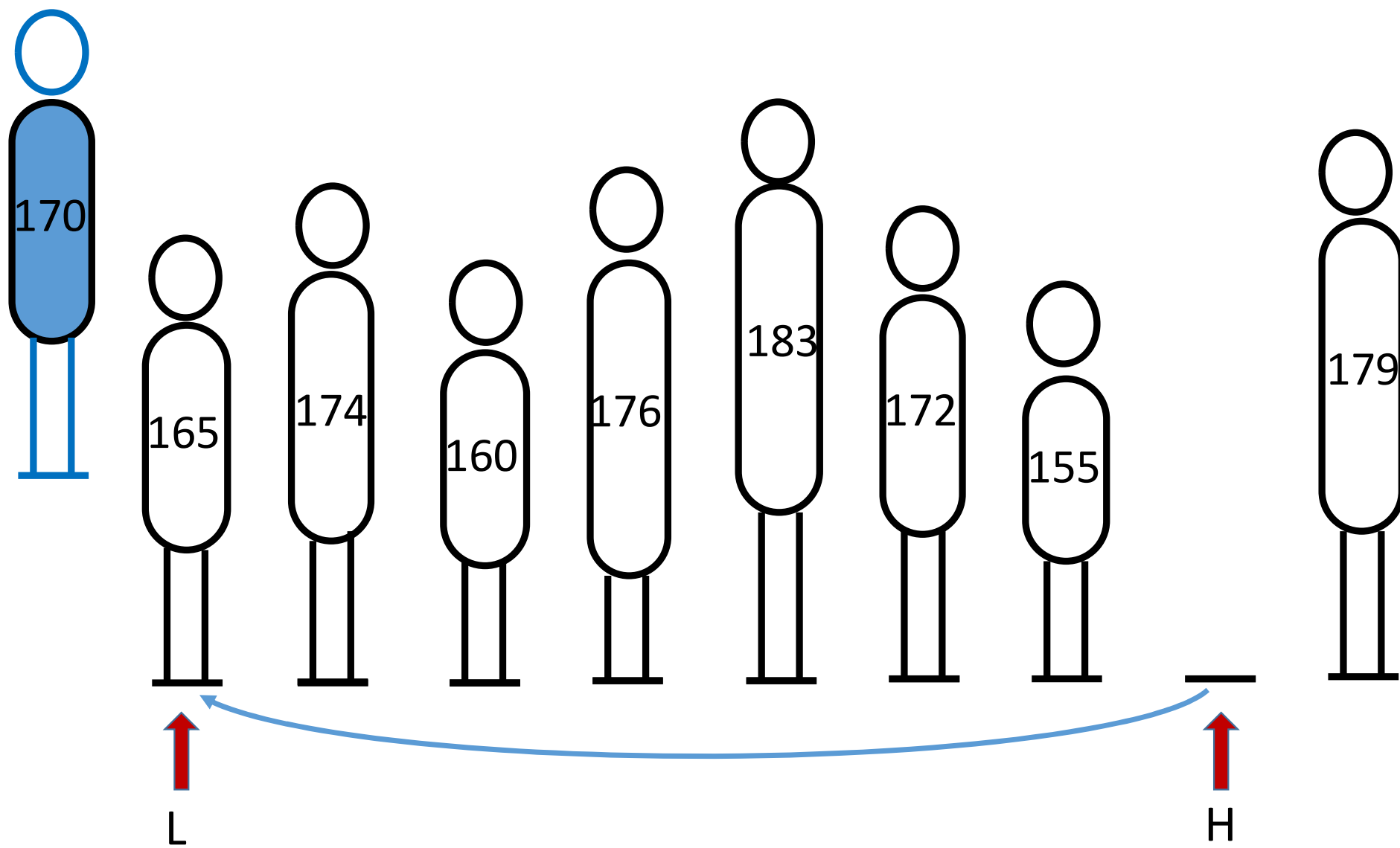
一次划分

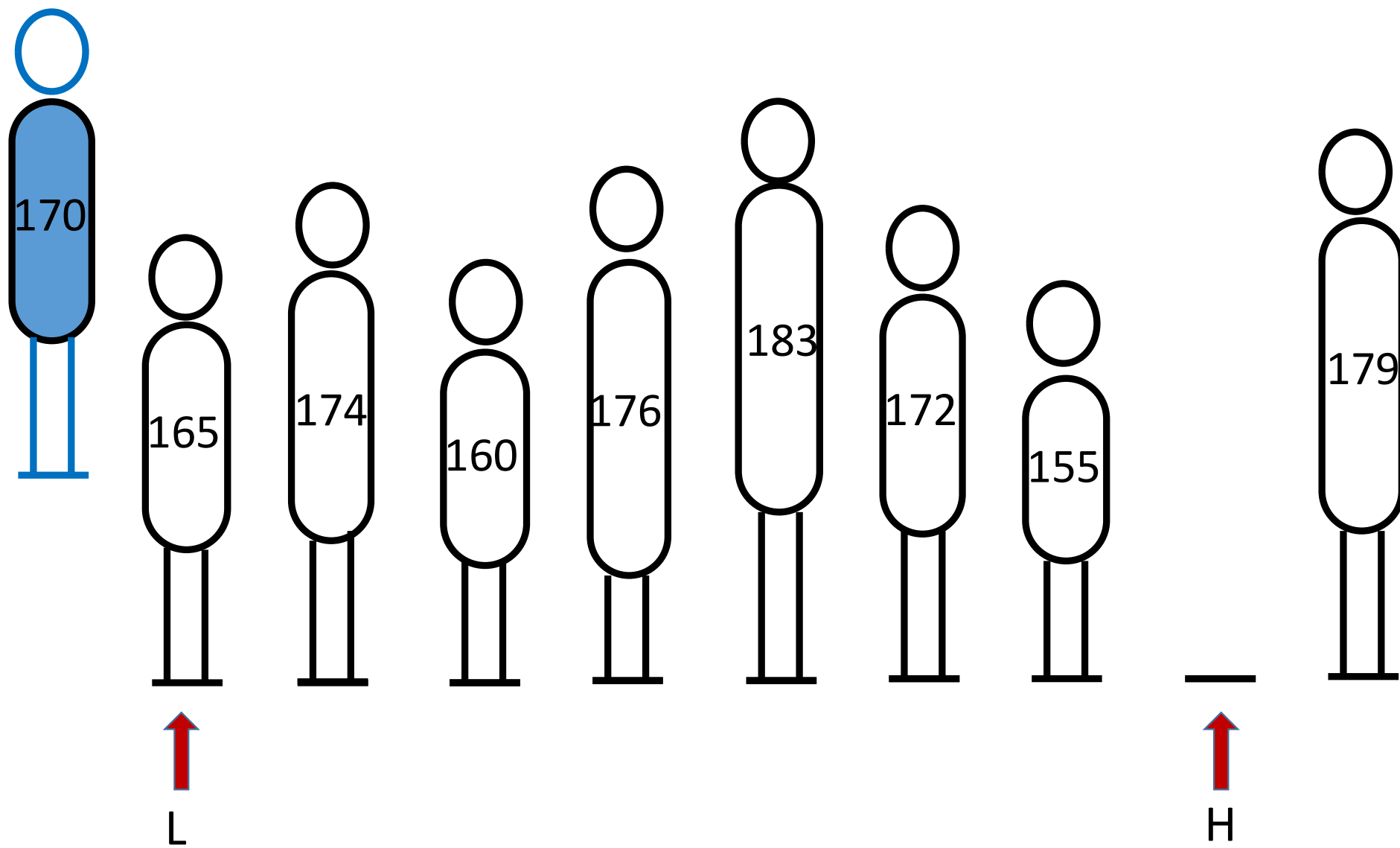


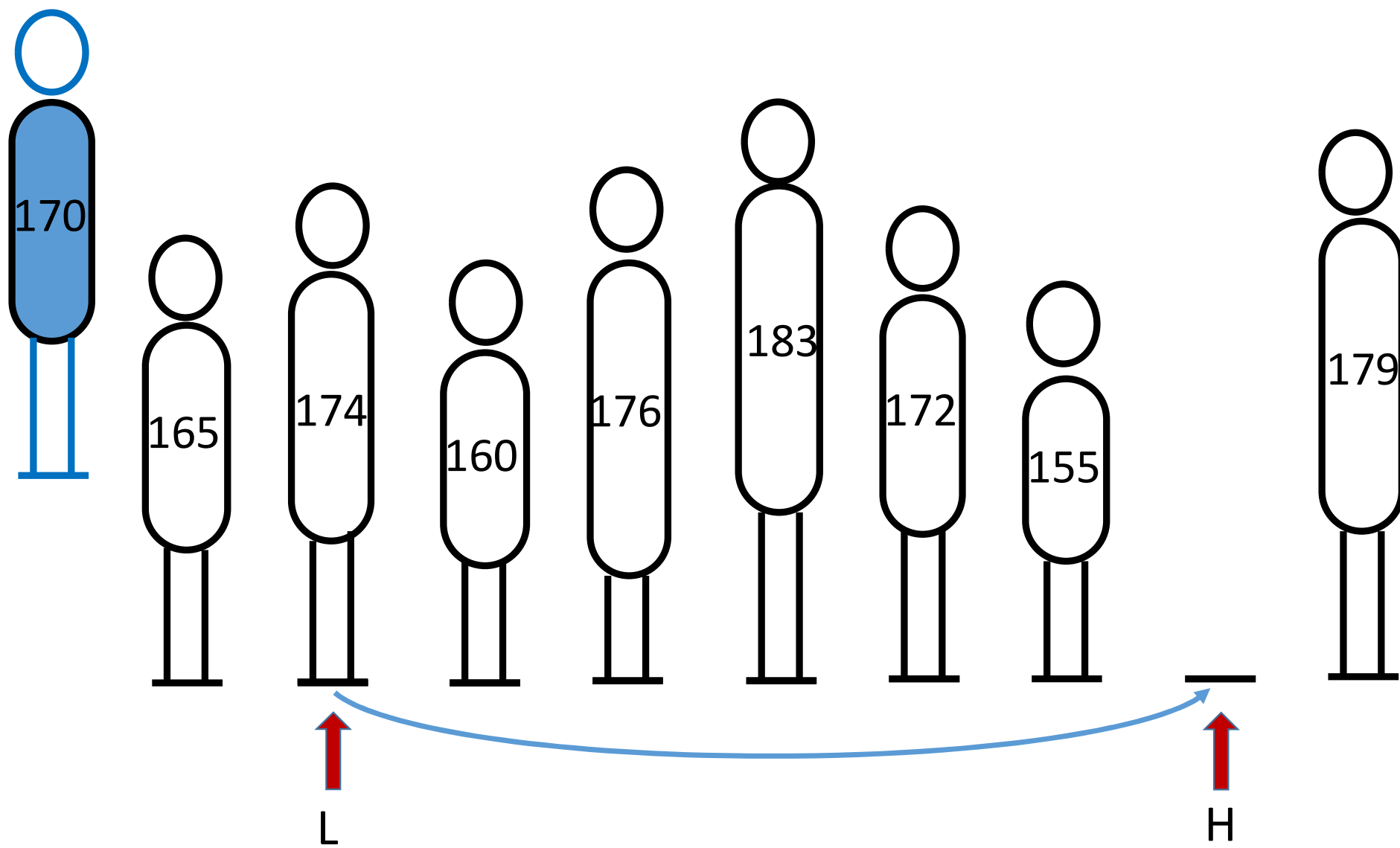


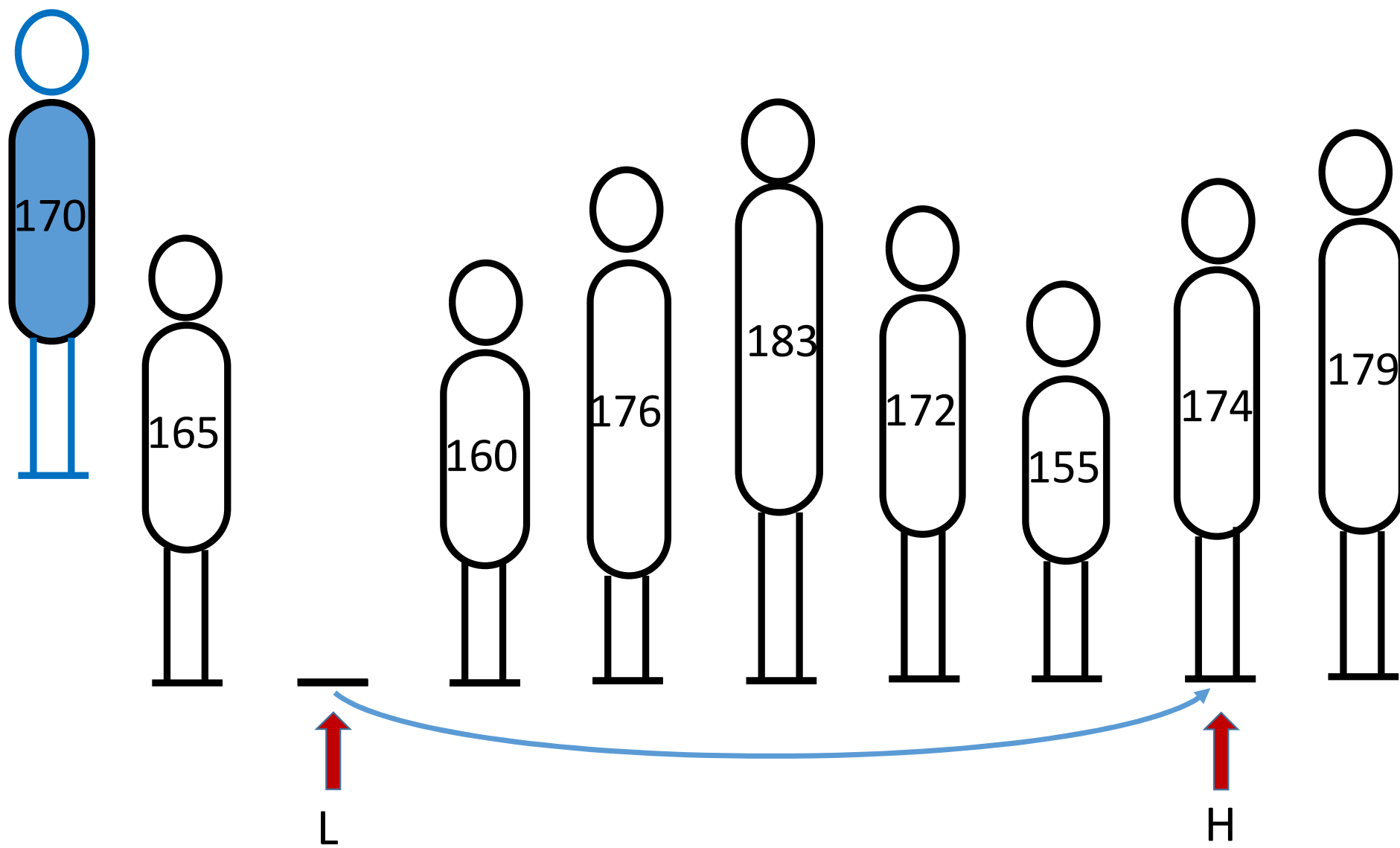


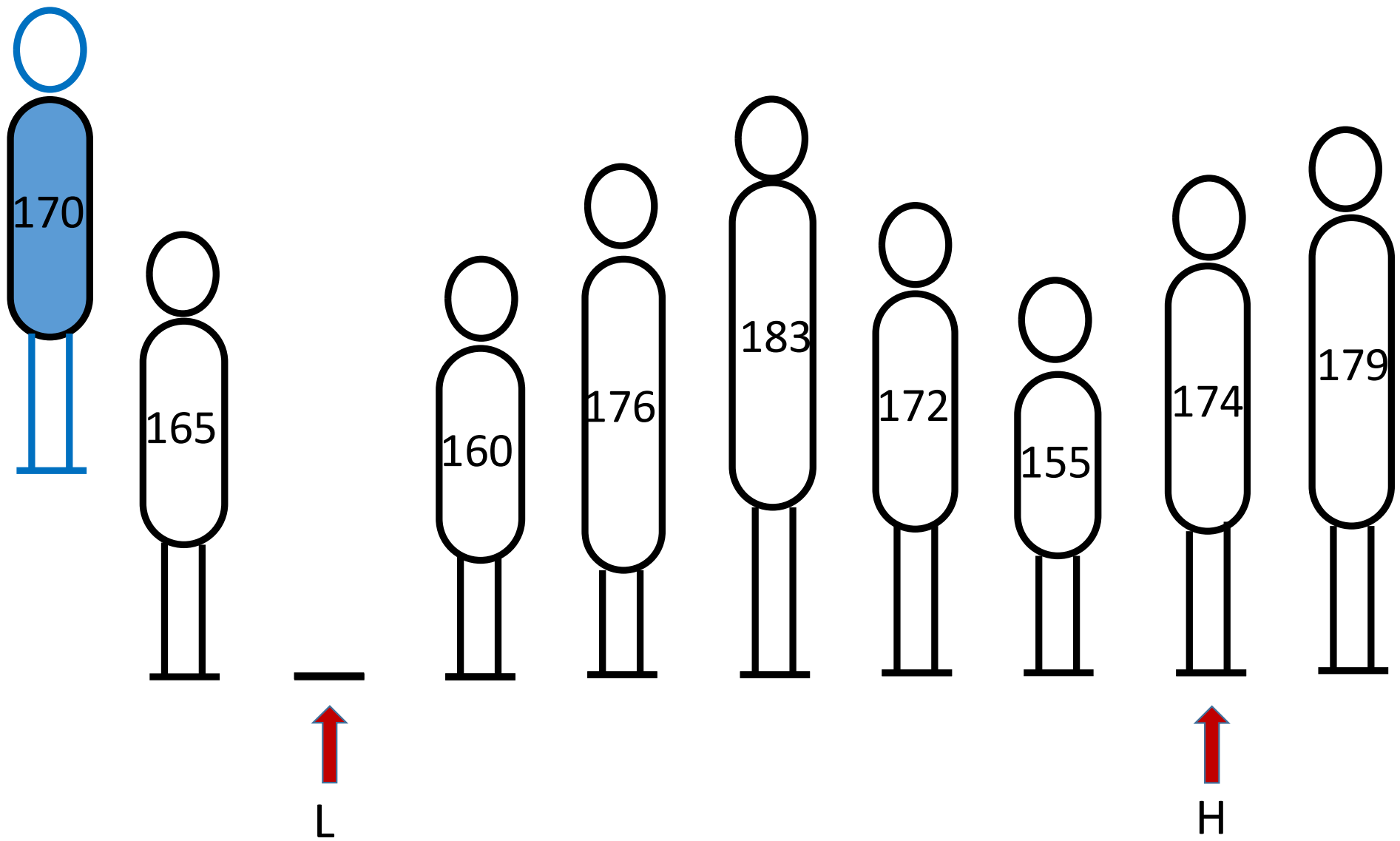


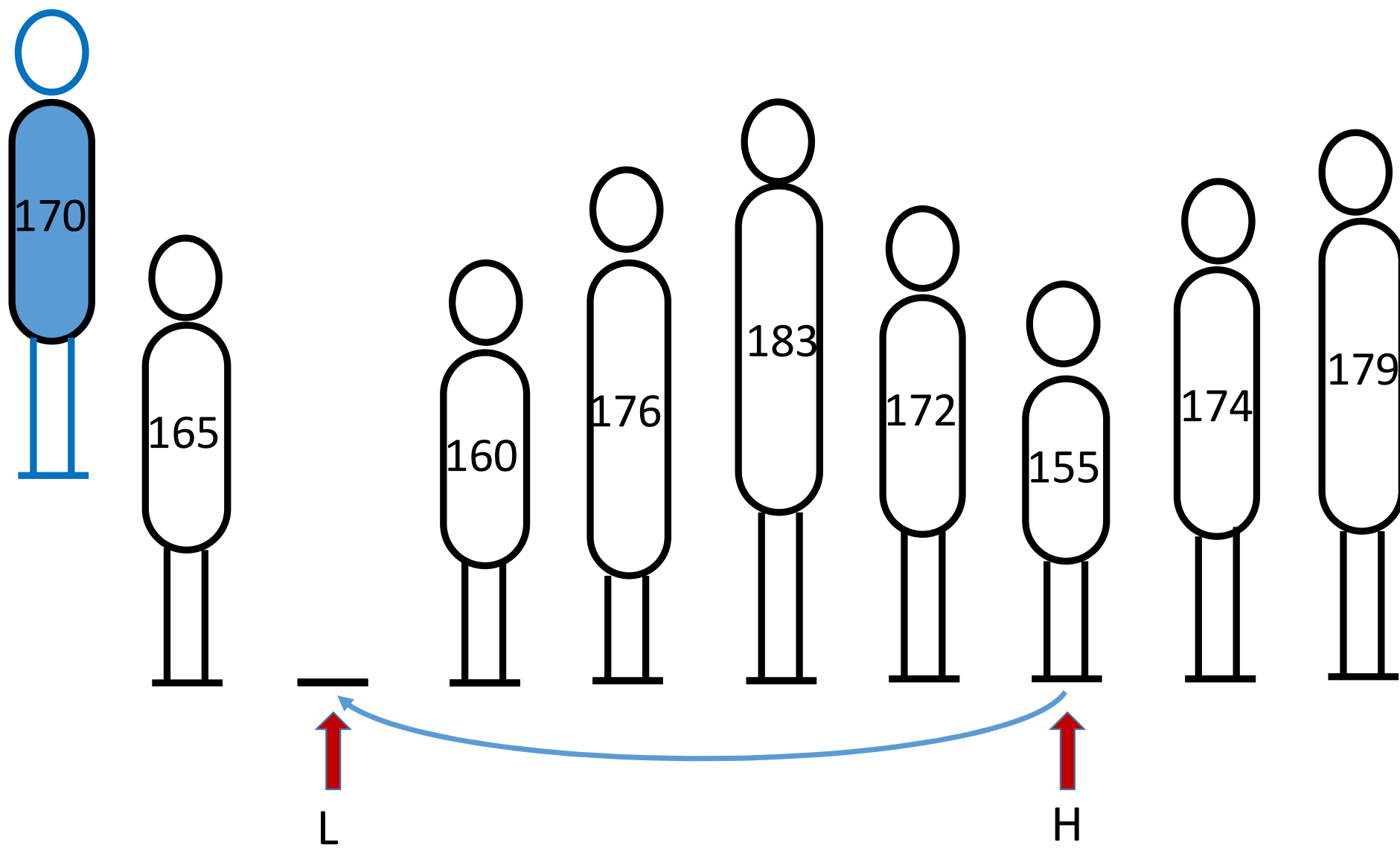


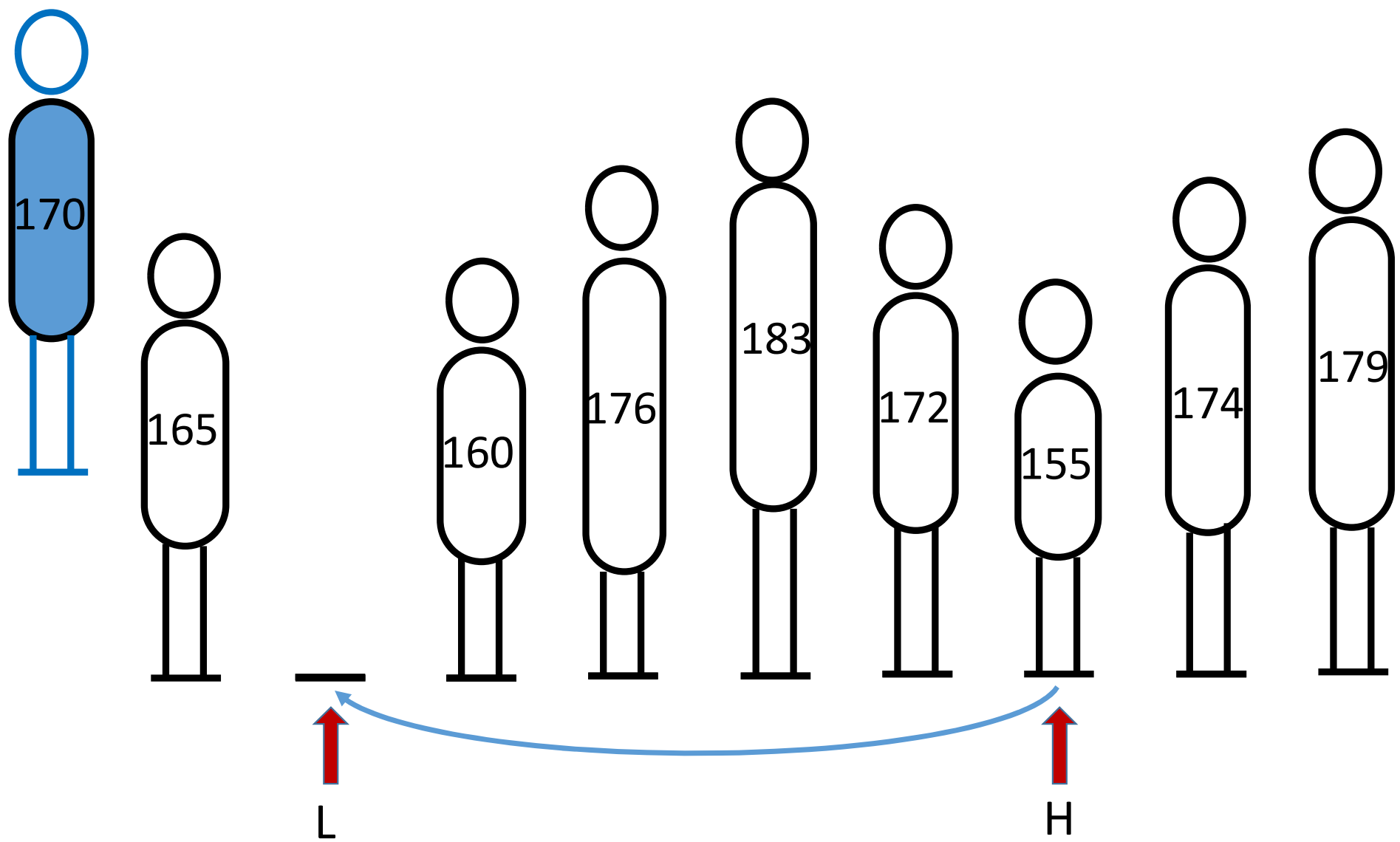


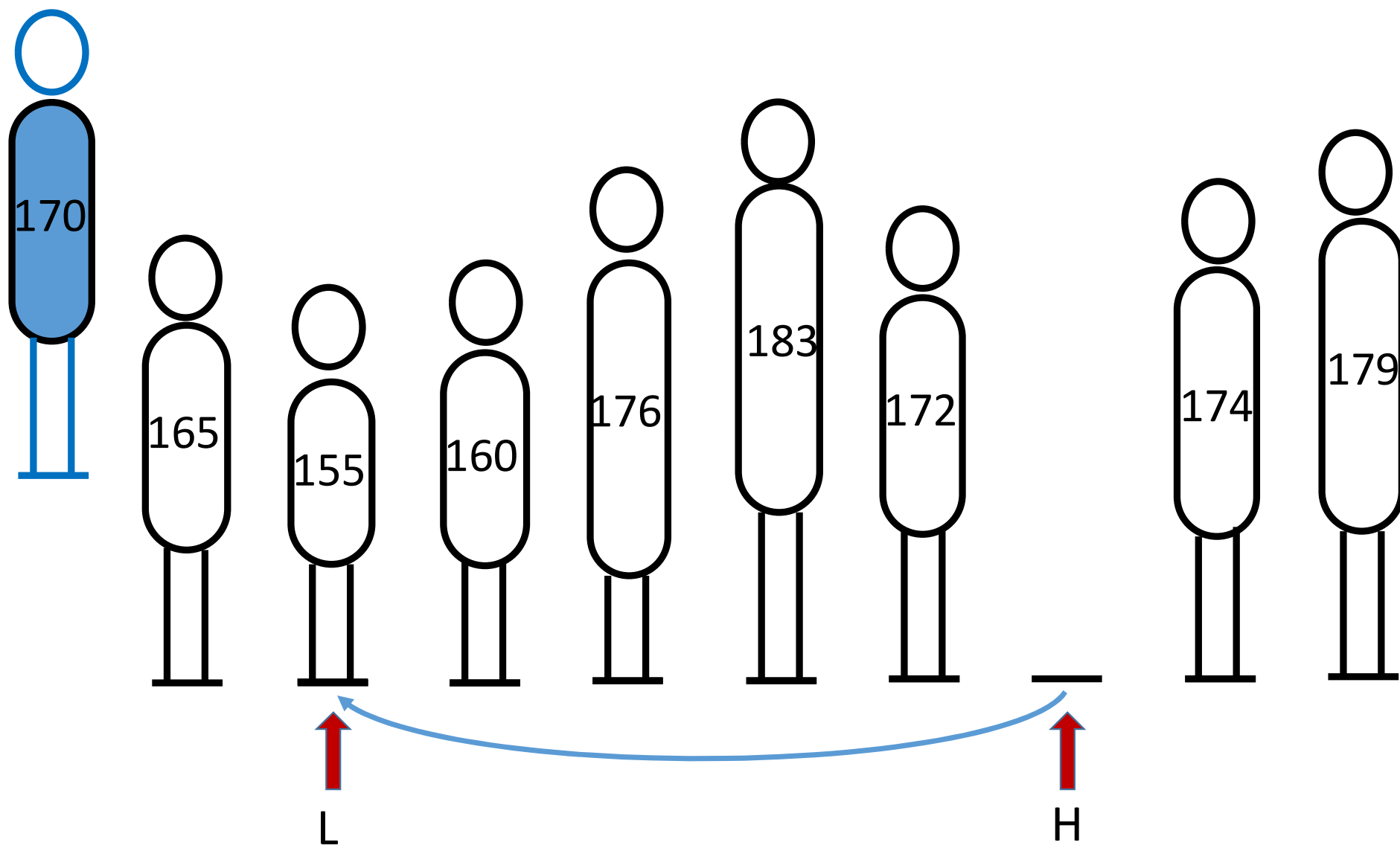


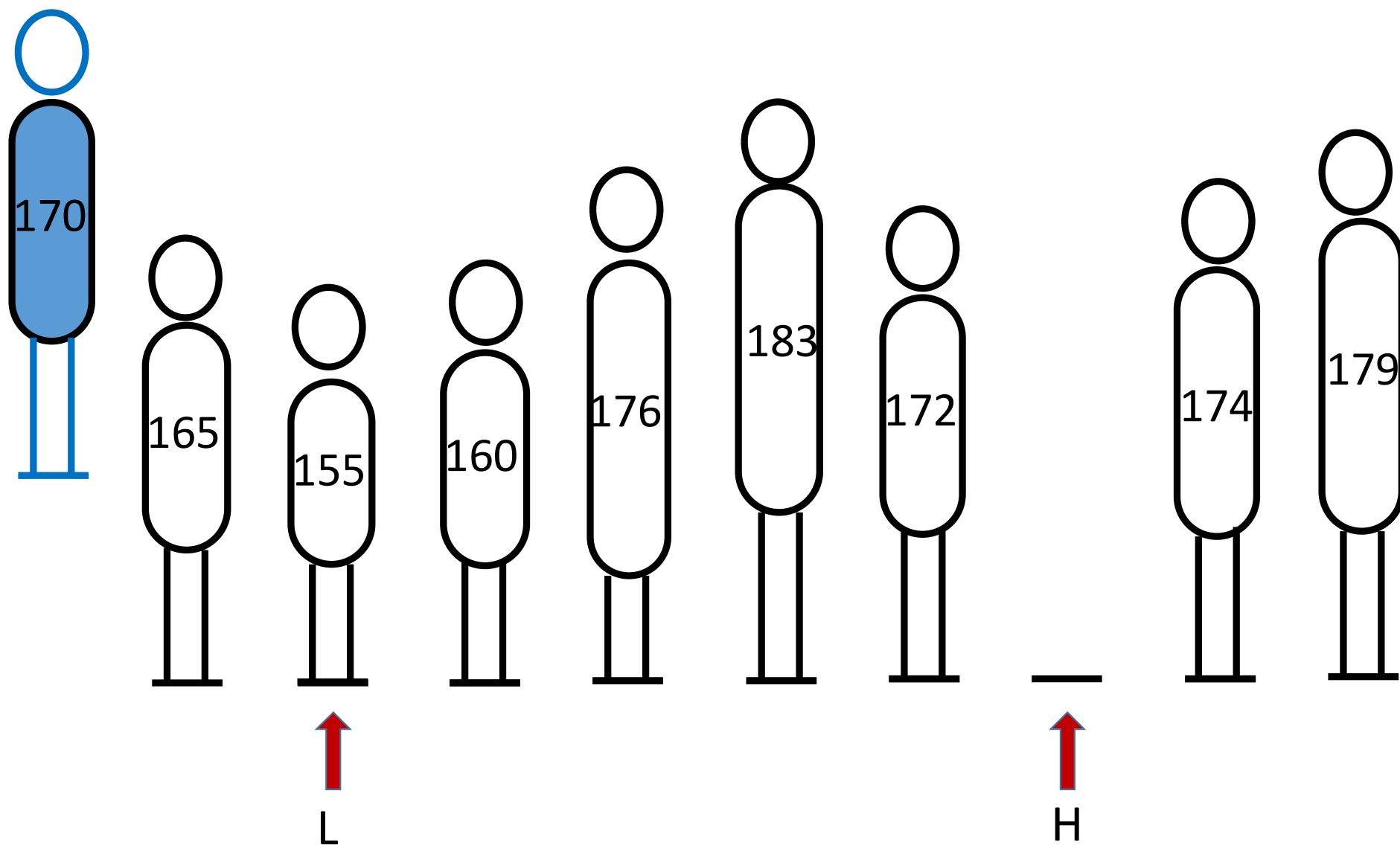


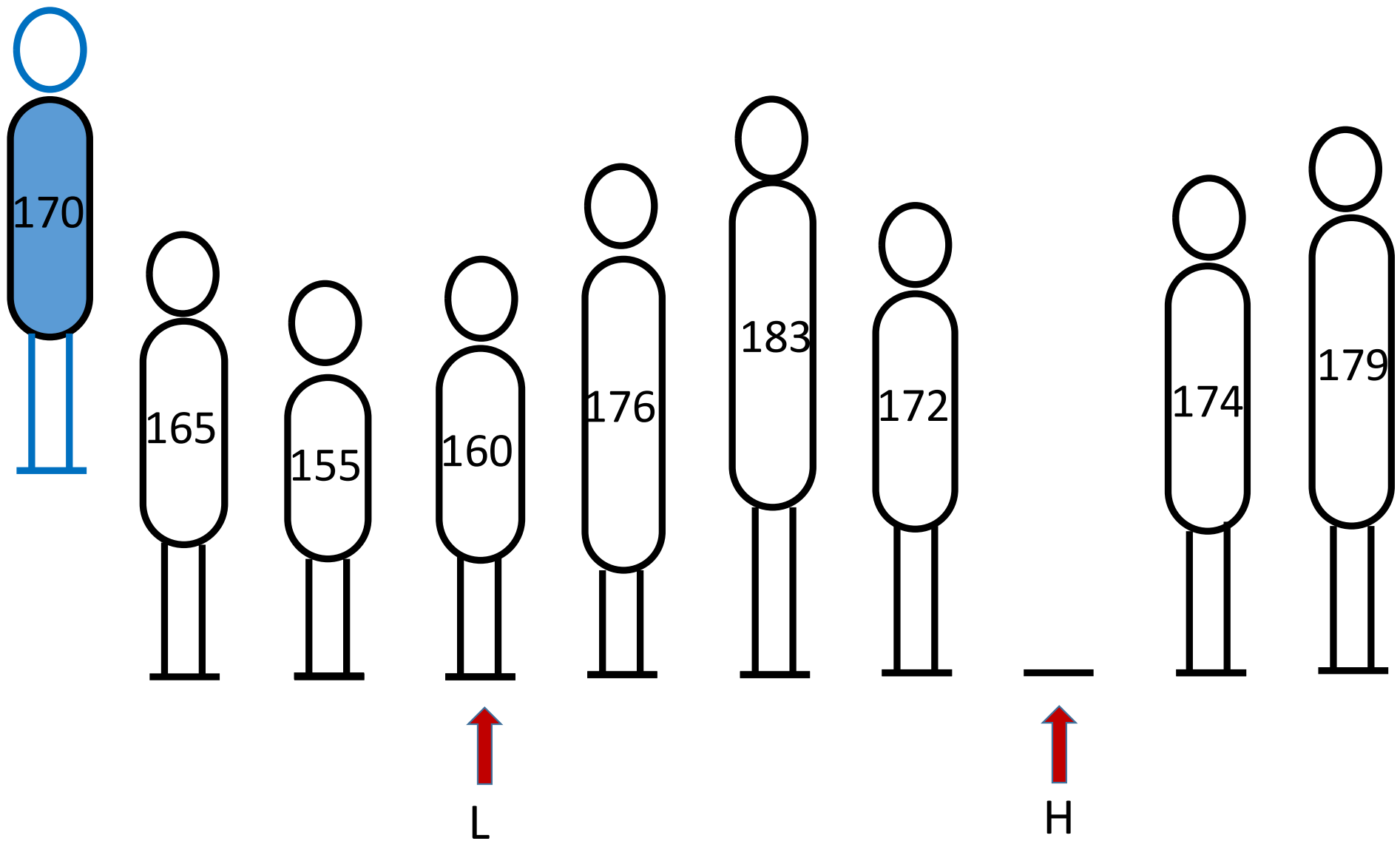


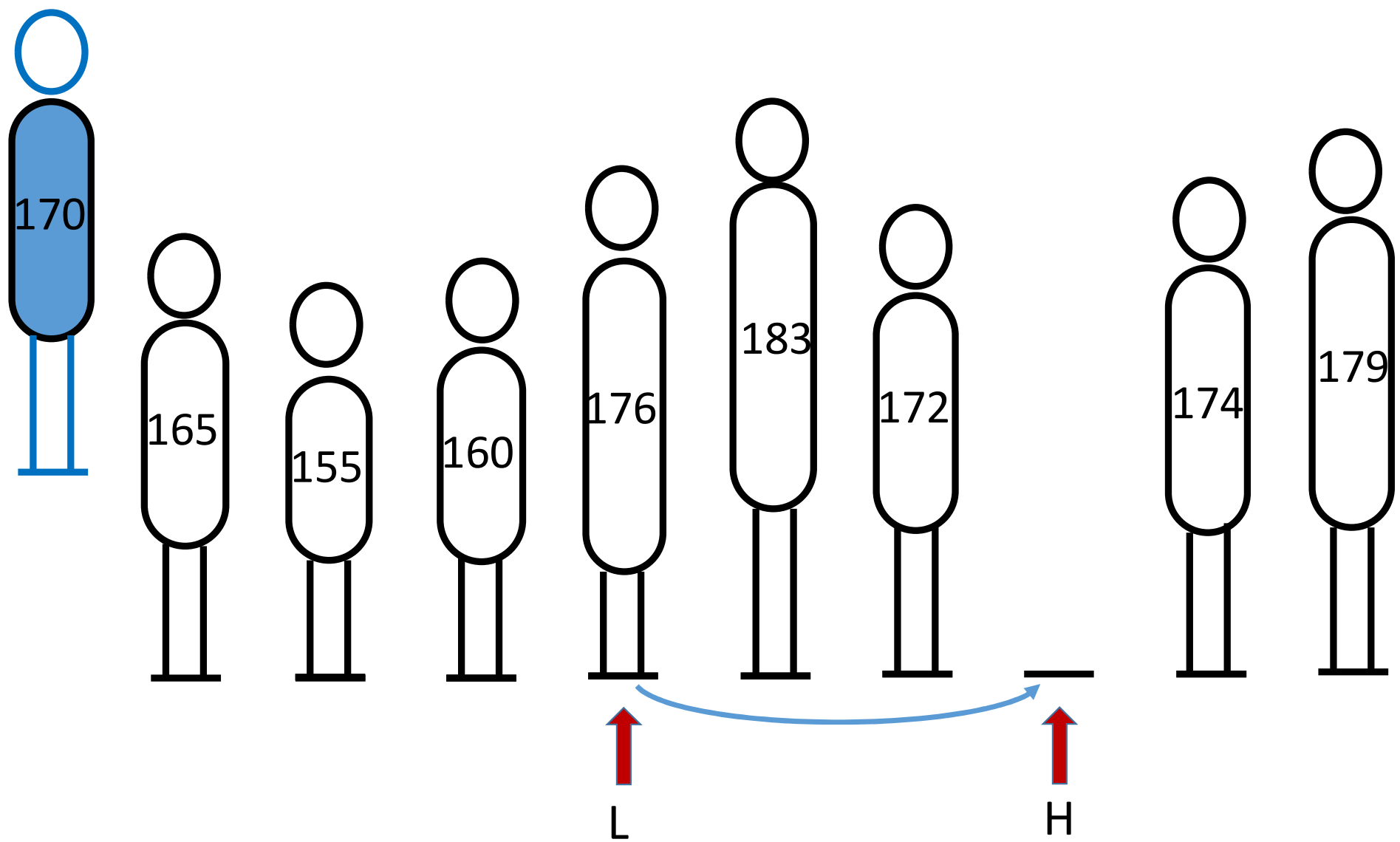


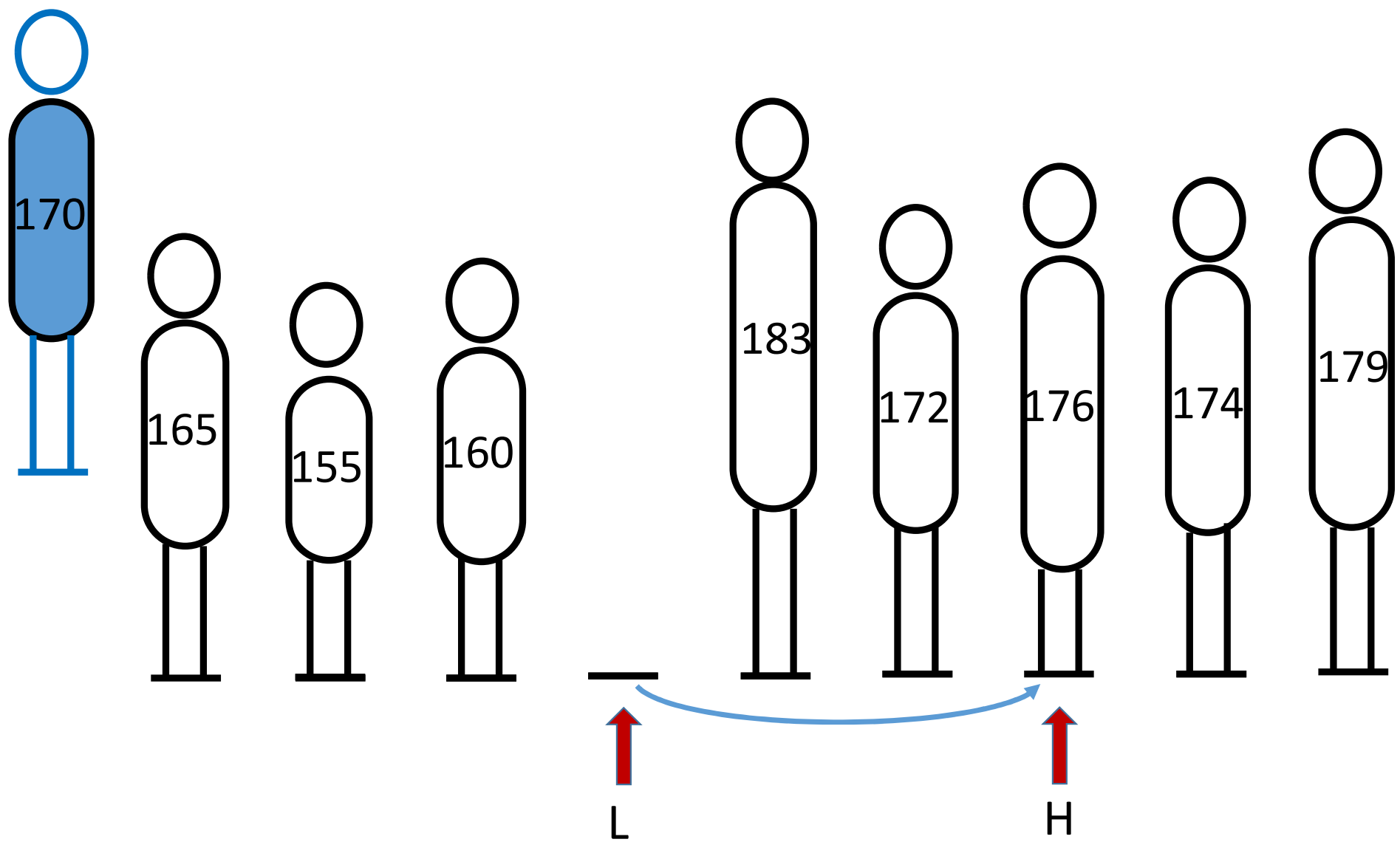


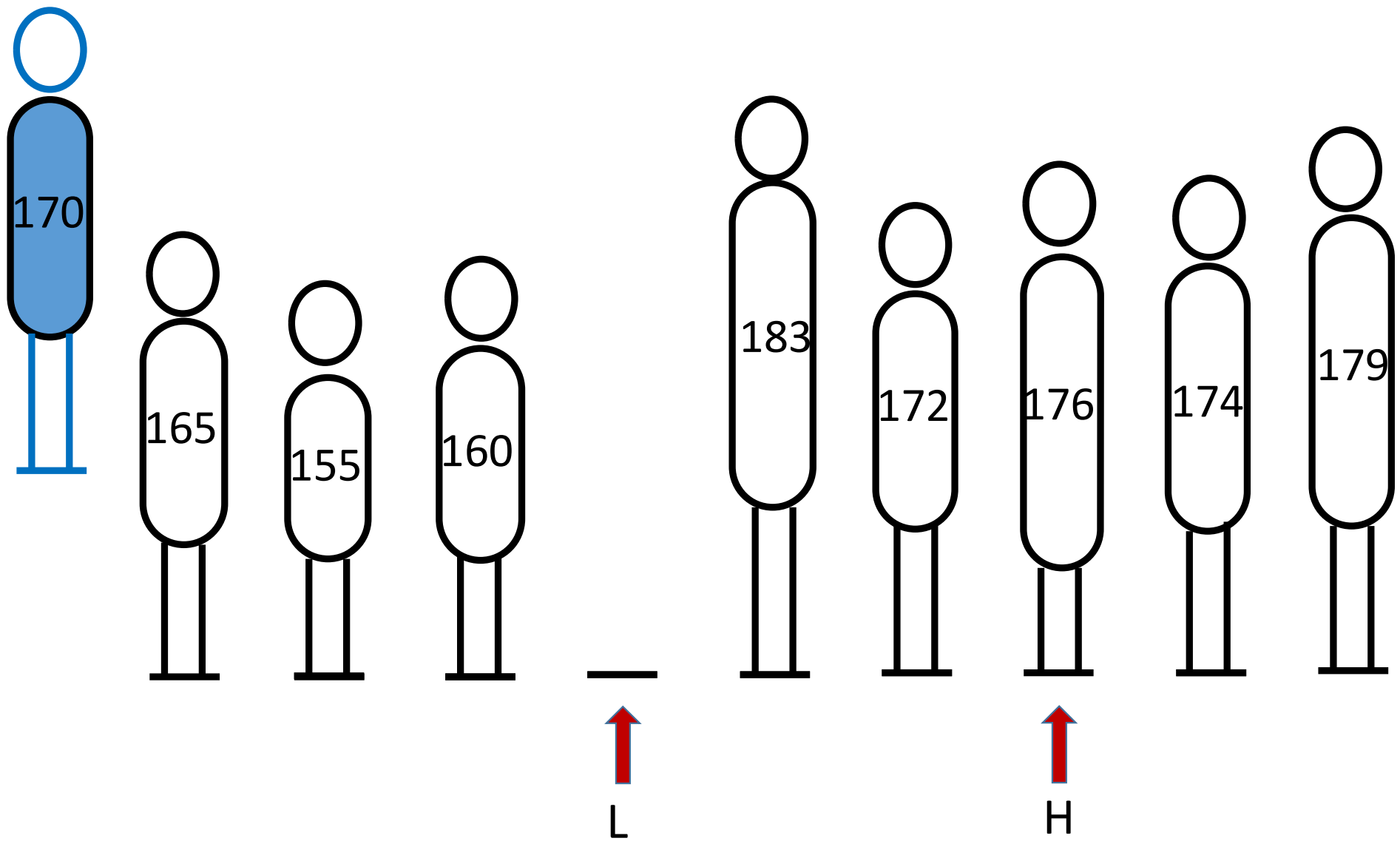


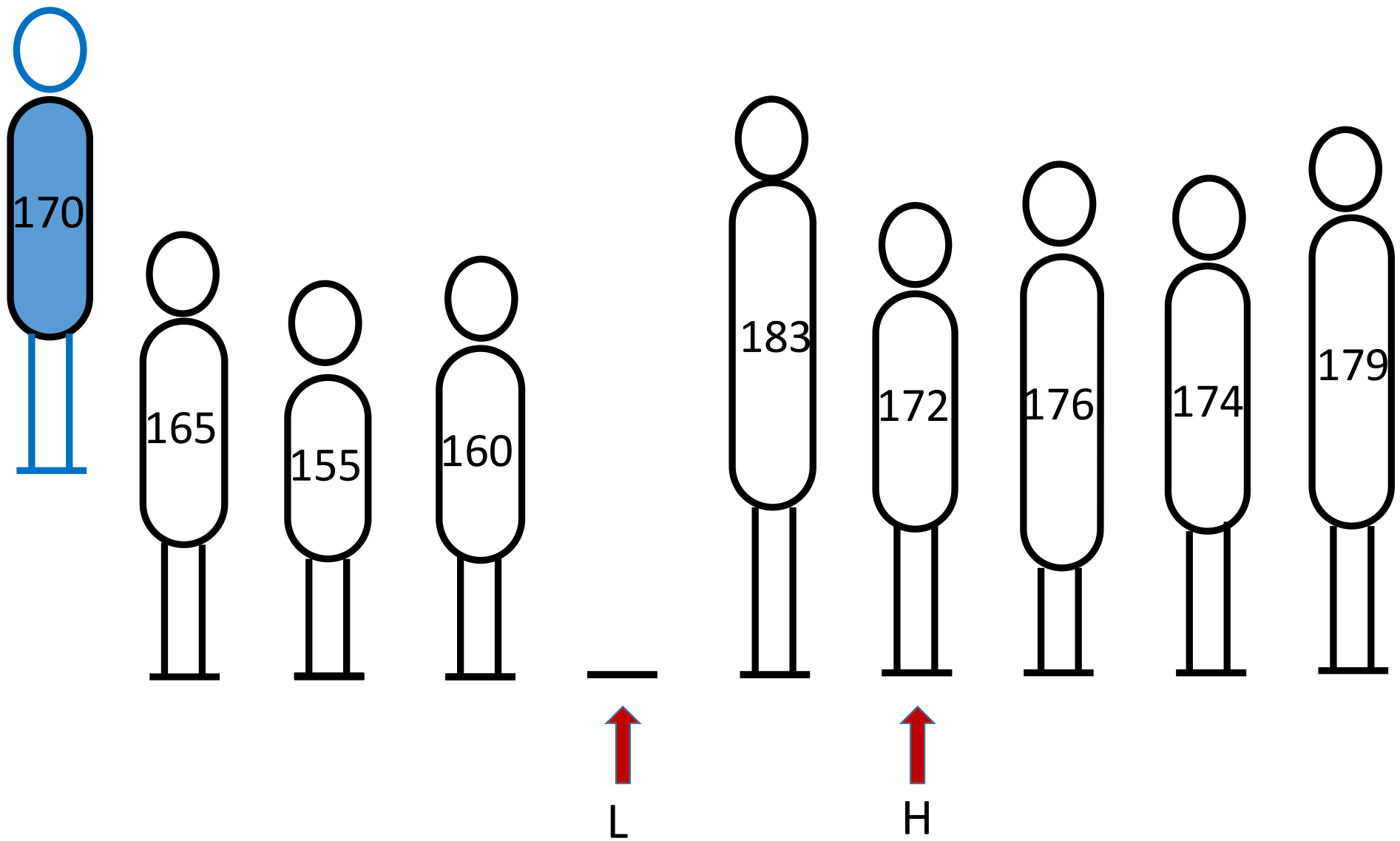


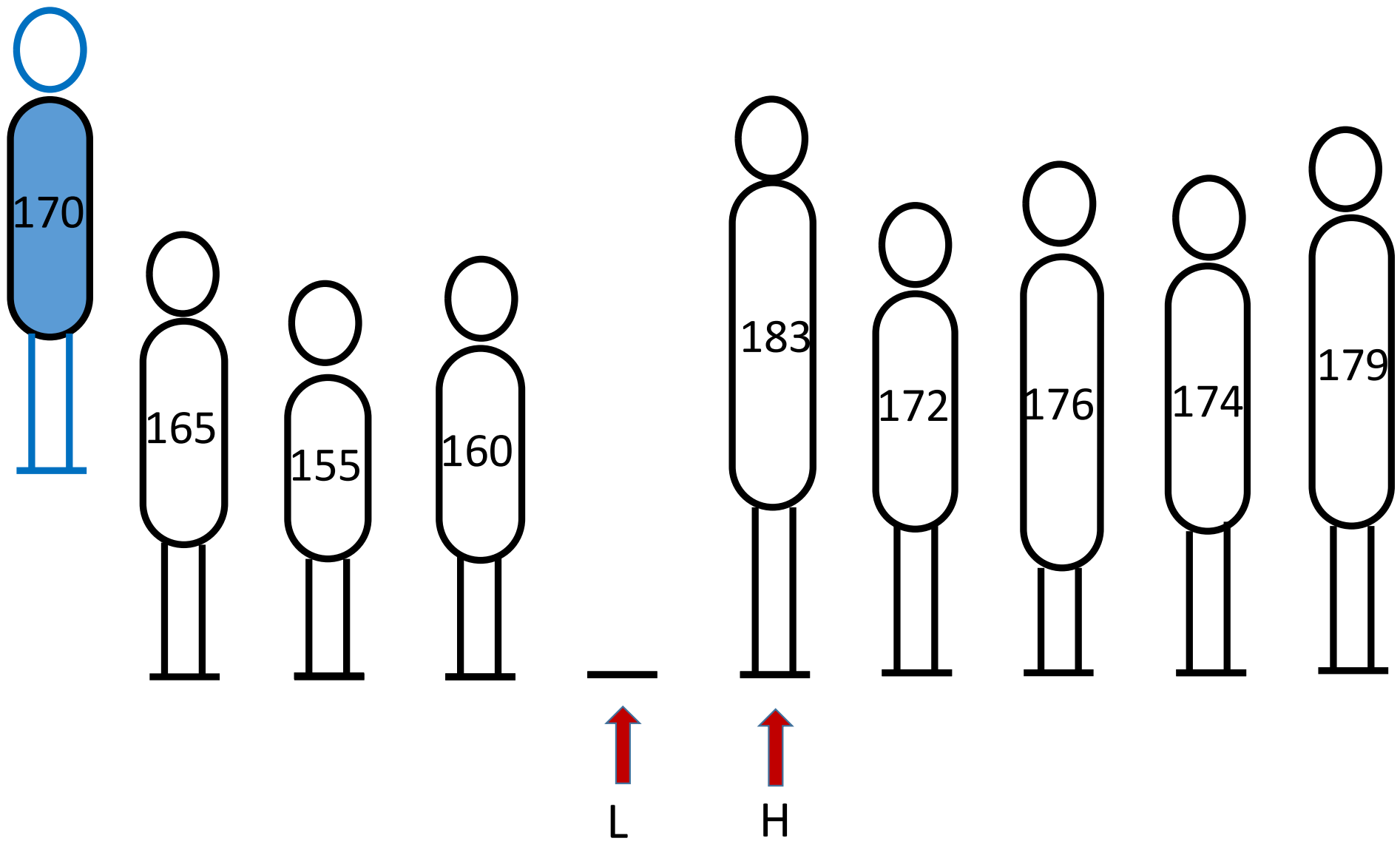


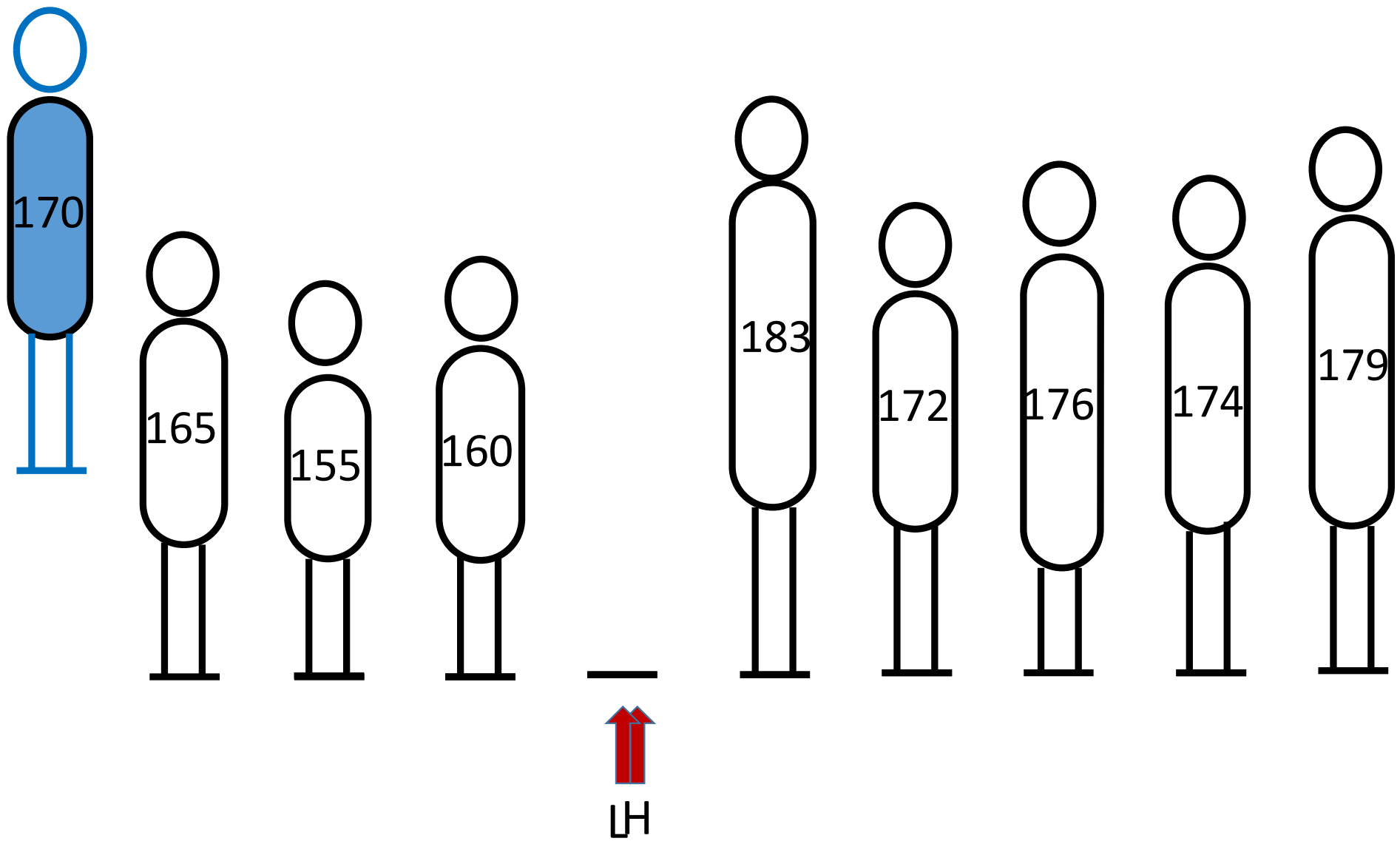


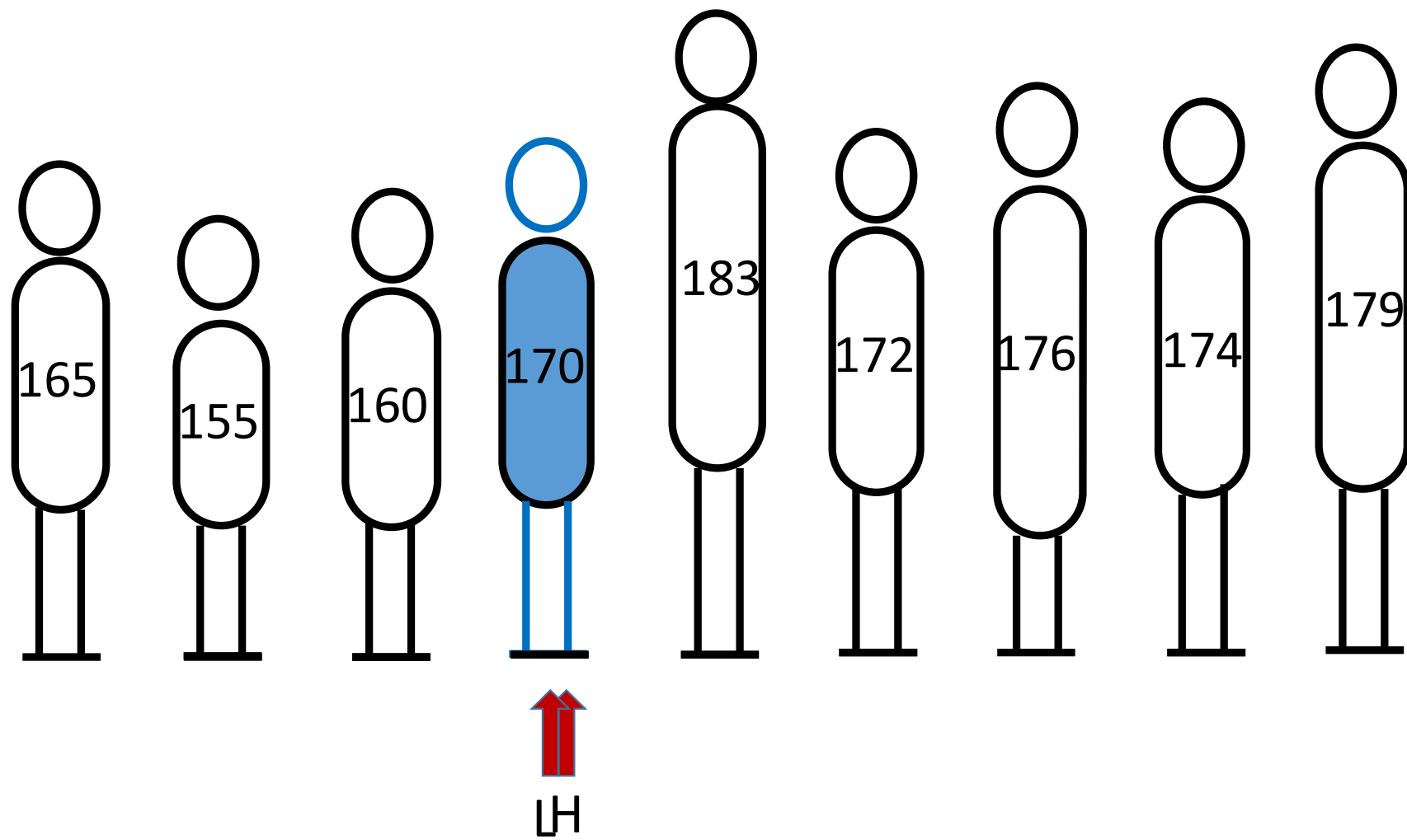












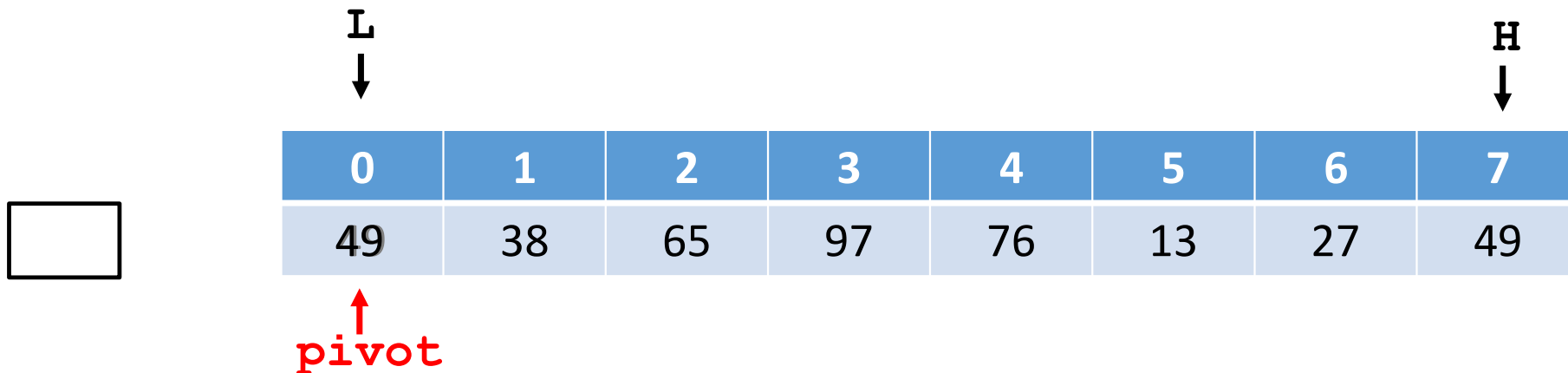

```
int Partition (T a[], int L, int H){
    T t = a[L];    //把最左元素当作基准
    while(L < H){
        //H向左, 直到遇见比pivot小的
        while(L < H && !(a[H] < t))
            H --;
        a[L] = a[H];

        //L向右, 直到遇见比pivot大的
        while(L < H && a[L] < t )
            L++;
        a[H] = a[L];
    }
    a[L] = t;
    return L;
}
```

```

T  t = a[L];           //把最左元素当作基准
while(L < H) {
    while(L < H && !(a[H]<t) ) //H向左, 直到遇见比pivot小的
        H --;
    a[L] = a[H];
    while(L<H && a[L]<t ) //L向右, 直到遇见比pivot大的
        L++;
    a[H] = a[L];
}
a[L] = t;    return L;

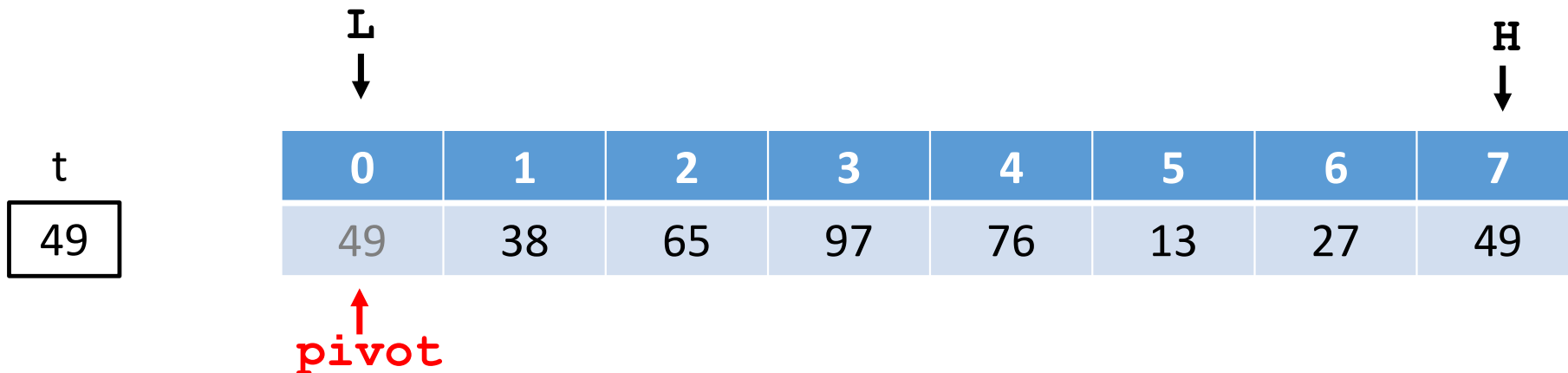
```



```

T   t = a[L];           //把最左元素当作基准
while(L < H) {
    while(L < H && !(a[H]<t) ) //H向左, 直到遇见比pivot小的
        H--;
    a[L] = a[H];
    while(L<H && a[L]<t) //L向右, 直到遇见比pivot大的
        L++;
    a[H] = a[L];
}
a[L] = t;   return L;

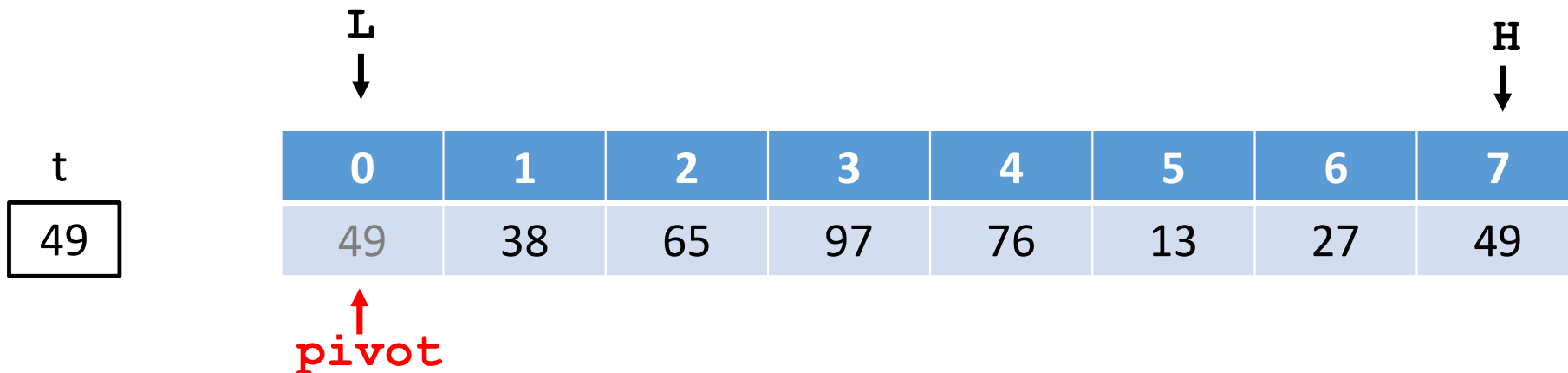
```



```

T   t = a[L];           //把最左元素当作基准
while(L < H){
    while(L < H && !(a[H]<t) ) //H向左, 直到遇见比pivot小的
        H --;
    a[L] = a[H];
    while(L<H && a[L]<t ) //L向右, 直到遇见比pivot大的
        L++;
    a[H] = a[L];
}
a[L] = t;   return L;

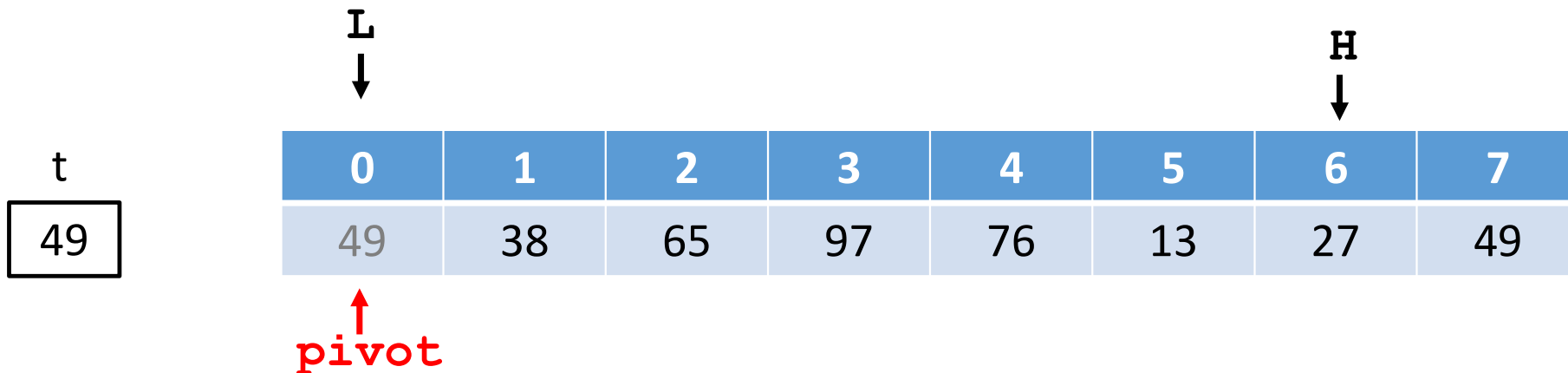
```



```

T  t = a[L];      //把最左元素当作基准
while(L < H){
    while(L < H && !(a[H]<t) ) //H向左, 直到遇见比pivot小的
        H --;
    a[L] = a[H];
    while(L<H && a[L]<t ) //L向右, 直到遇见比pivot大的
        L++;
    a[H] = a[L];
}
a[L] = t;    return L;

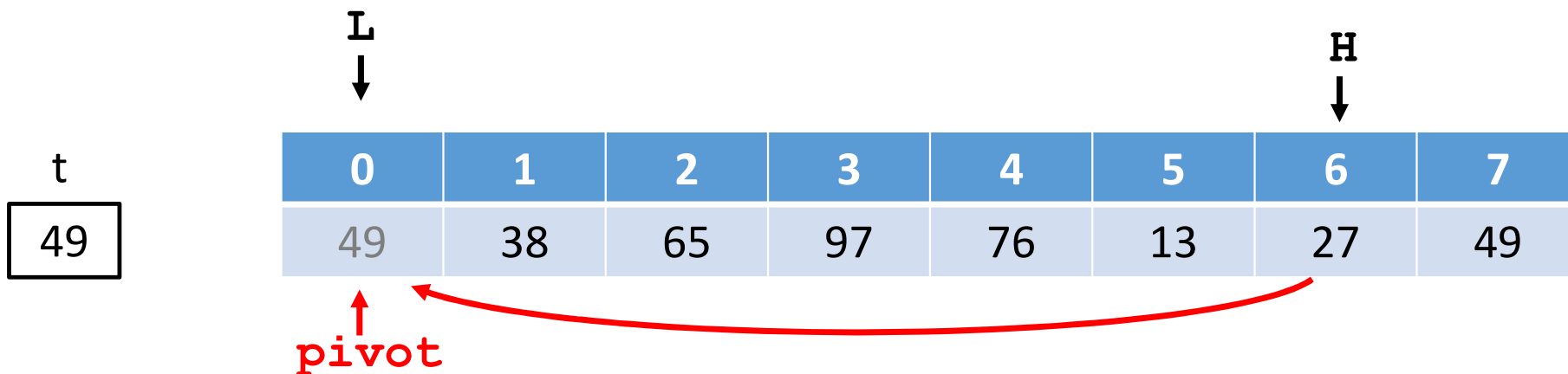
```



```

T  t = a[L];      //把最左元素当作基准
while(L < H){
    while(L < H && !(a[H]<t) ) //H向左, 直到遇见比pivot小的
        H --;
    a[L] = a[H];
    while(L<H && a[L]<t ) //L向右, 直到遇见比pivot大的
        L++;
    a[H] = a[L];
}
a[L] = t;    return L;

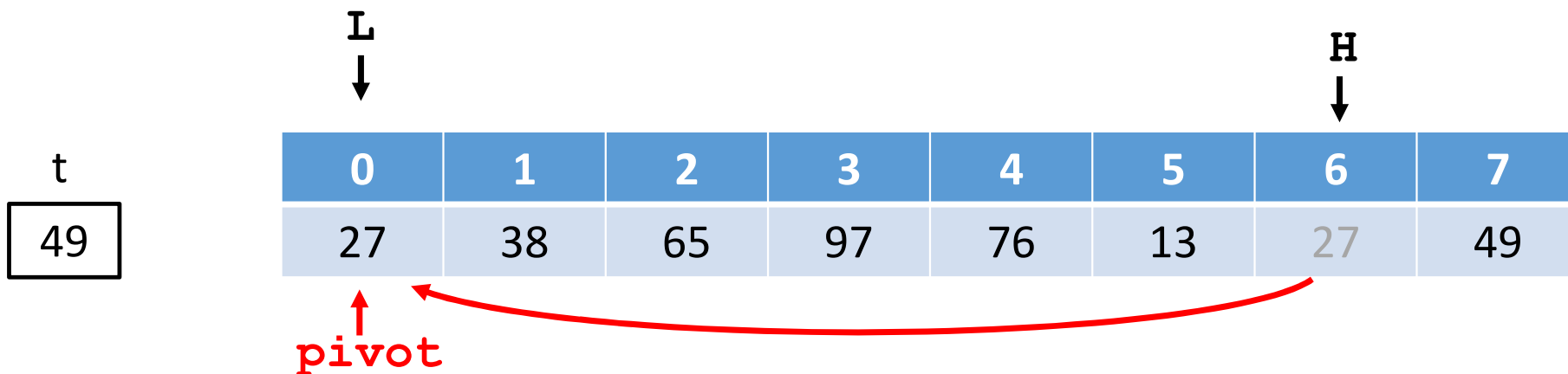
```



```

T   t = a[L];      //把最左元素当作基准
while(L < H){
    while(L < H && !(a[H]<t) ) //H向左, 直到遇见比pivot小的
        H --;
    a[L] = a[H];
    while(L<H && a[L]<t ) //L向右, 直到遇见比pivot大的
        L++;
    a[H] = a[L];
}
a[L] = t;    return L;

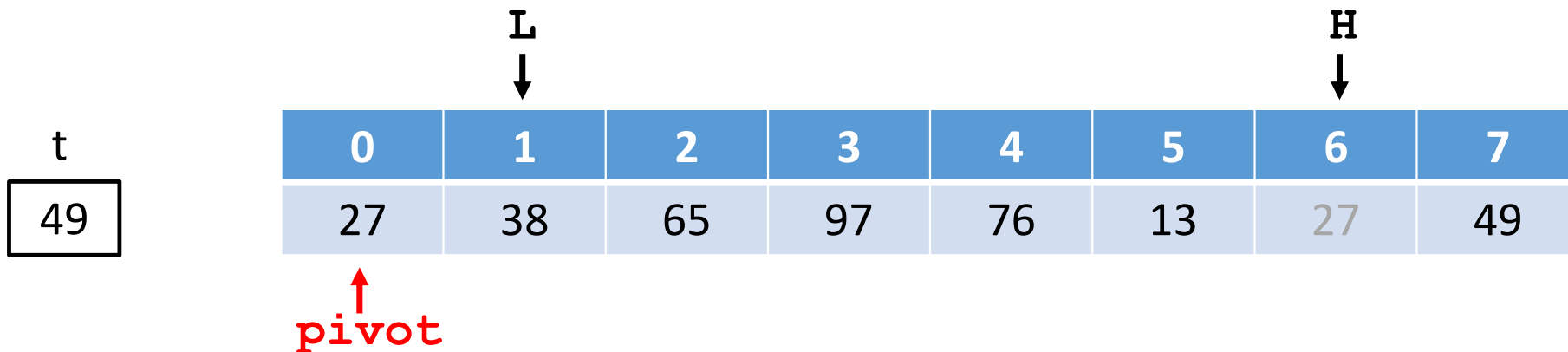
```



```

T  t = a[L];      //把最左元素当作基准
while(L < H){
    while(L < H &&  !(a[H]<t) ) //H向左, 直到遇见比pivot小的
        H --;
    a[L] = a[H];
    while(L<H && a[L]<t ) //L向右, 直到遇见比pivot大的
        L++;
    a[H] = a[L];
}
a[L] = t;    return L;

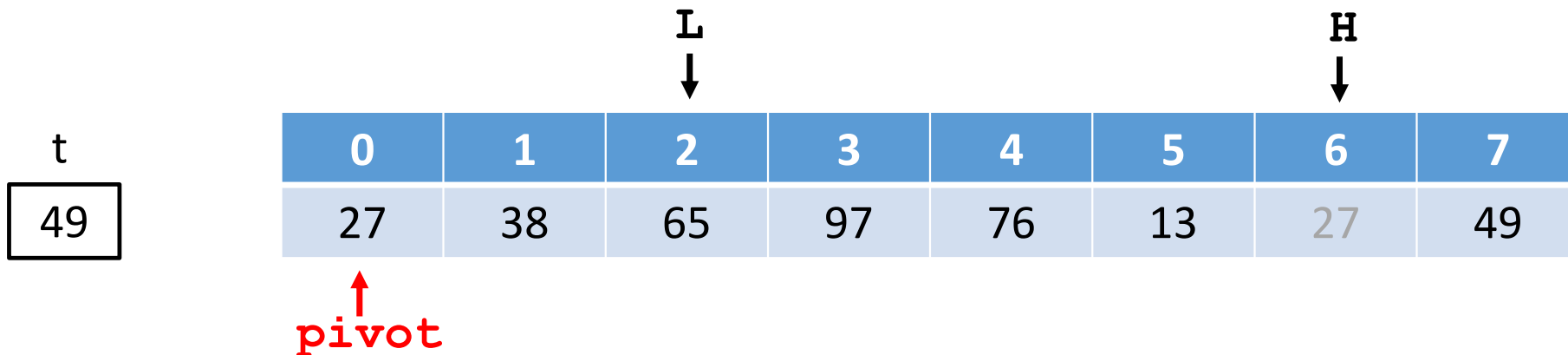
```




```

T   t = a[L];           //把最左元素当作基准
while(L < H){
    while(L < H && !(a[H]<t) ) //H向左, 直到遇见比pivot小的
        H --;
    a[L] = a[H];
    while(L<H && a[L]<t ) //L向右, 直到遇见比pivot大的
        L++;
    a[H] = a[L];
}
a[L] = t;   return L;

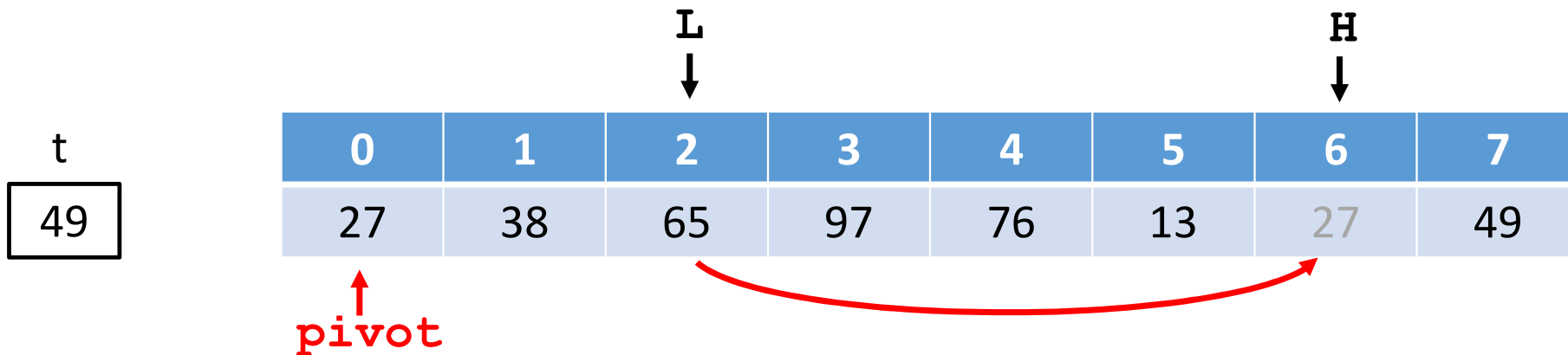
```



```

T   t = a[L];      //把最左元素当作基准
while(L < H){
    while(L < H && !(a[H]<t) ) //H向左, 直到遇见比pivot小的
        H --;
    a[L] = a[H];
    while(L<H && a[L]<t ) //L向右, 直到遇见比pivot大的
        L++;
    a[H] = a[L];
}
a[L] = t;    return L;

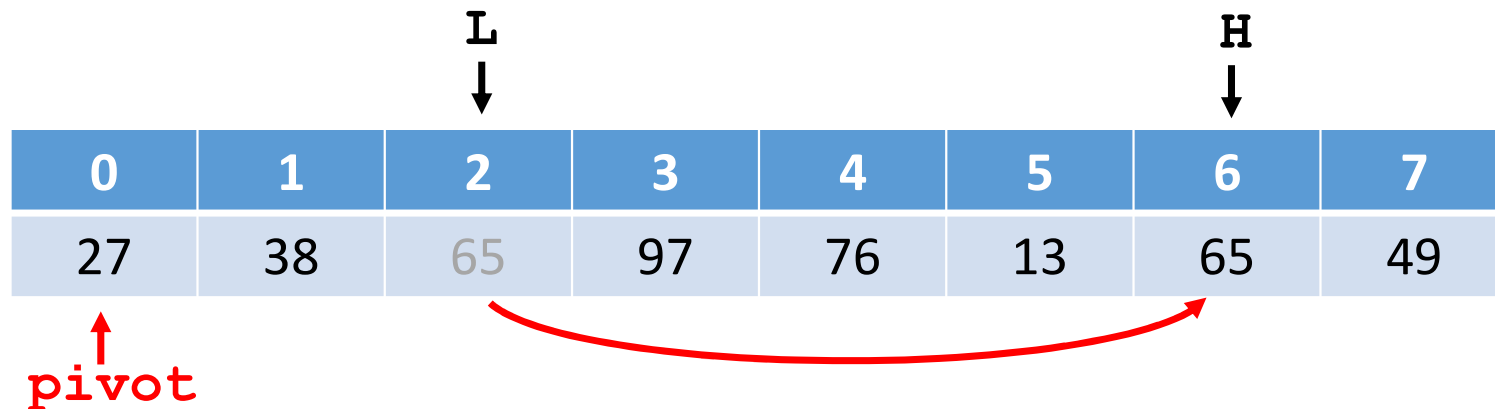
```



```

T   t = a[L];      //把最左元素当作基准
while(L < H){
    while(L < H &&  !(a[H]<t) ) //H向左, 直到遇见比pivot小的
        H --;
    a[L] = a[H];
    while(L<H && a[L]<t ) //L向右, 直到遇见比pivot大的
        L++;
    a[H] = a[L];
}
a[L] = t;    return L;

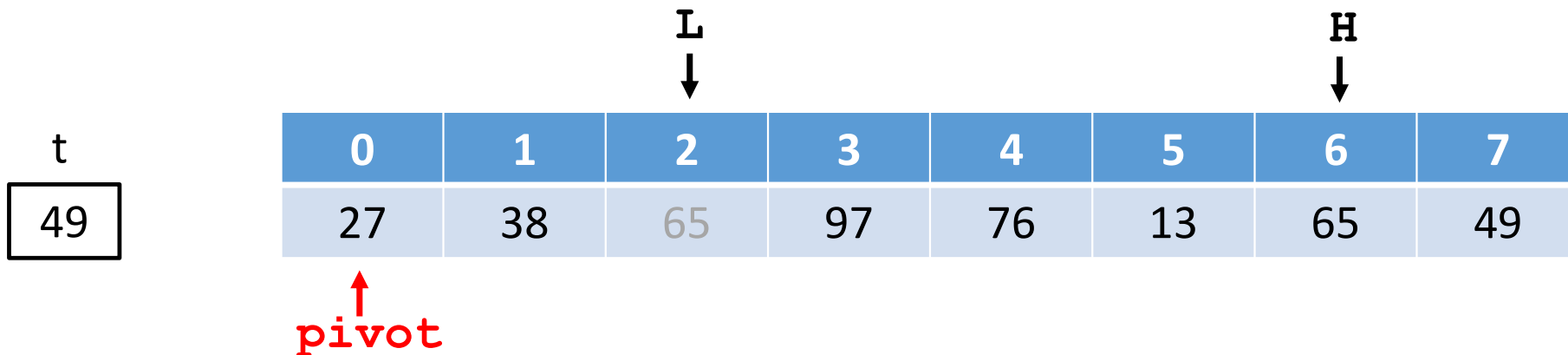
```



```

T   t = a[L];           //把最左元素当作基准
while(L < H) {
    while(L < H && !(a[H]<t) ) //H向左, 直到遇见比pivot小的
        H--;
    a[L] = a[H];
    while(L<H && a[L]<t ) //L向右, 直到遇见比pivot大的
        L++;
    a[H] = a[L];
}
a[L] = t;   return L;

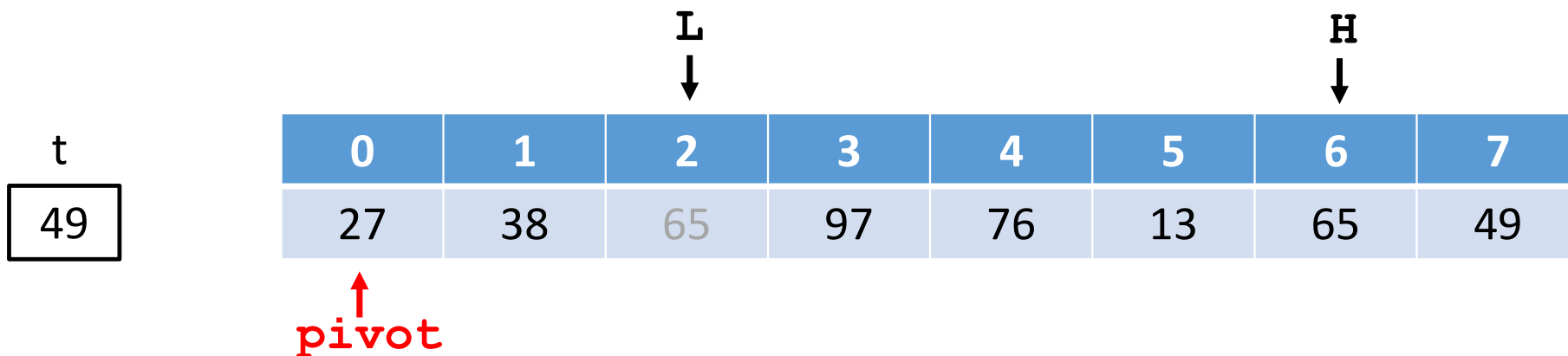
```



```

T  t = a[L];    //把最左元素当作基准
while(L < H){
    while(L < H && !(a[H]<t) ) //H向左, 直到遇见比pivot小的
        H --;
    a[L] = a[H];
    while(L<H && a[L]<t ) //L向右, 直到遇见比pivot大的
        L++;
    a[H] = a[L];
}
a[L] = t;    return L;

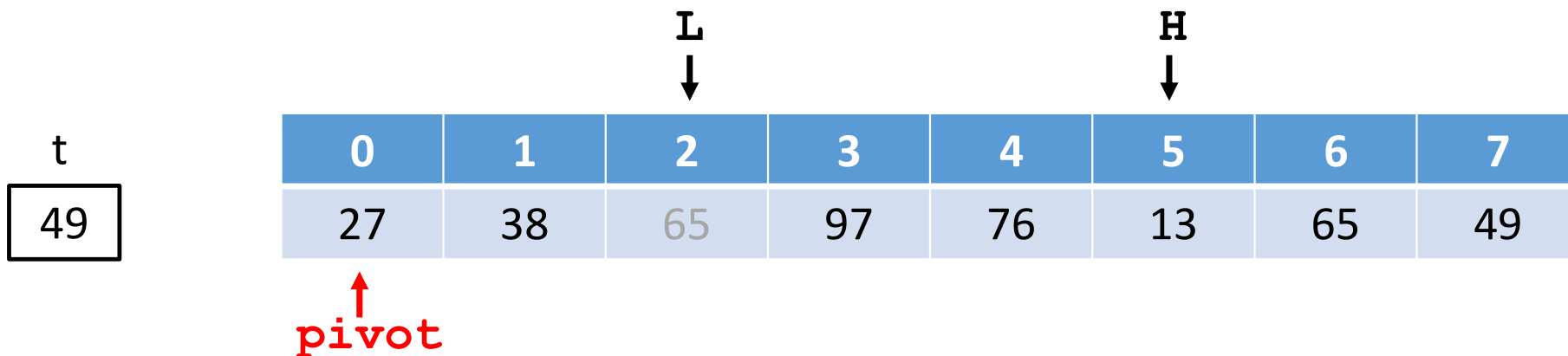
```



```

T   t = a[L];           //把最左元素当作基准
while(L < H){
    while(L < H && !(a[H]<t) ) //H向左, 直到遇见比pivot小的
        H --;
    a[L] = a[H];
    while(L<H && a[L]<t ) //L向右, 直到遇见比pivot大的
        L++;
    a[H] = a[L];
}
a[L] = t;   return L;

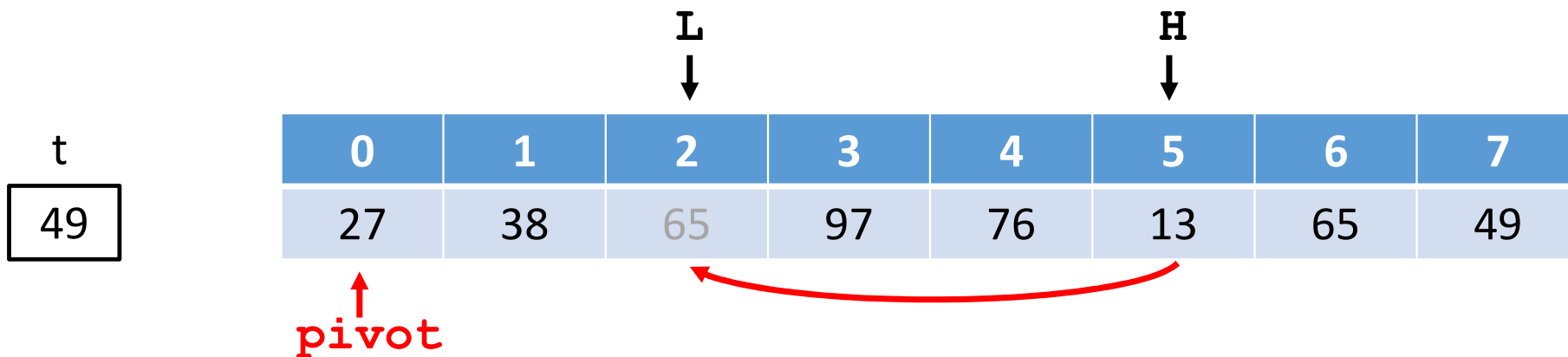
```



```

T  t = a[L];      //把最左元素当作基准
while(L < H){
    while(L < H && !(a[H]<t) ) //H向左, 直到遇见比pivot小的
        H --;
    a[L] = a[H];
    while(L<H && a[L]<t ) //L向右, 直到遇见比pivot大的
        L++;
    a[H] = a[L];
}
a[L] = t;    return L;

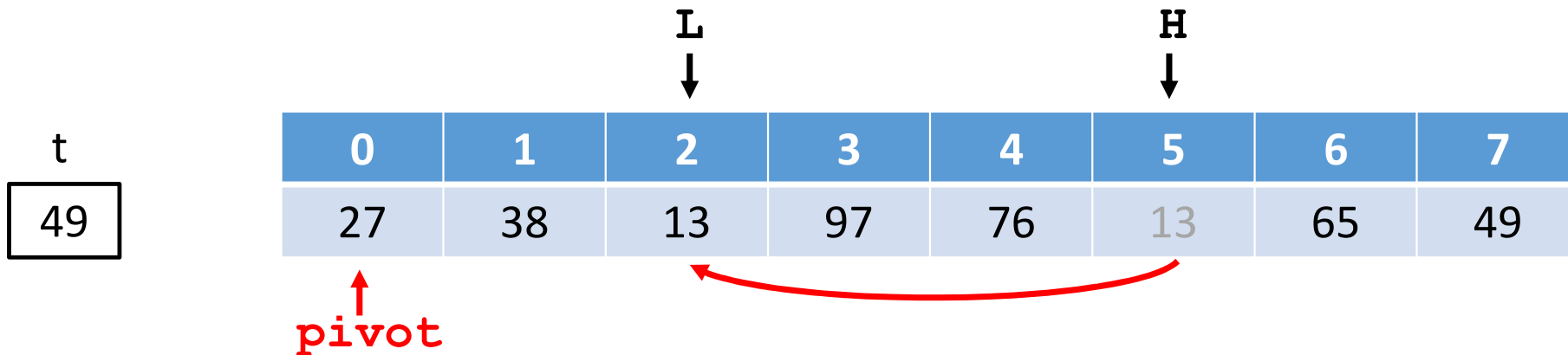
```



```

T   t = a[L];      //把最左元素当作基准
while(L < H){
    while(L < H && !(a[H]<t) ) //H向左, 直到遇见比pivot小的
        H--;
    a[L] = a[H];
    while(L<H && a[L]<t ) //L向右, 直到遇见比pivot大的
        L++;
    a[H] = a[L];
}
a[L] = t;    return L;

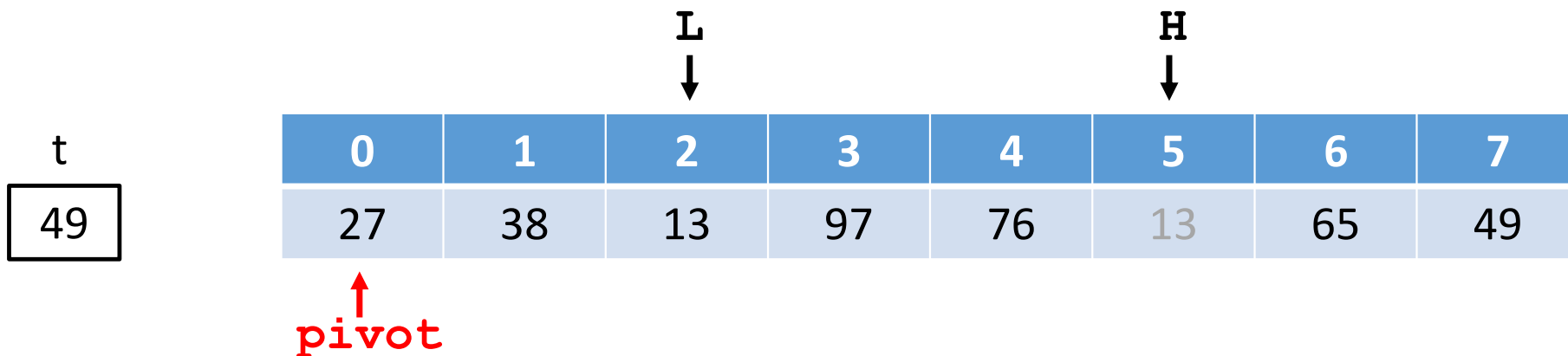
```




```

T   t = a[L];           //把最左元素当作基准
while(L < H) {
    while(L < H && !(a[H]<t) ) //H向左, 直到遇见比pivot小的
        H --;
    a[L] = a[H];
    while(L<H && a[L]<t ) //L向右, 直到遇见比pivot大的
        L++;
    a[H] = a[L];
}
a[L] = t;   return L;

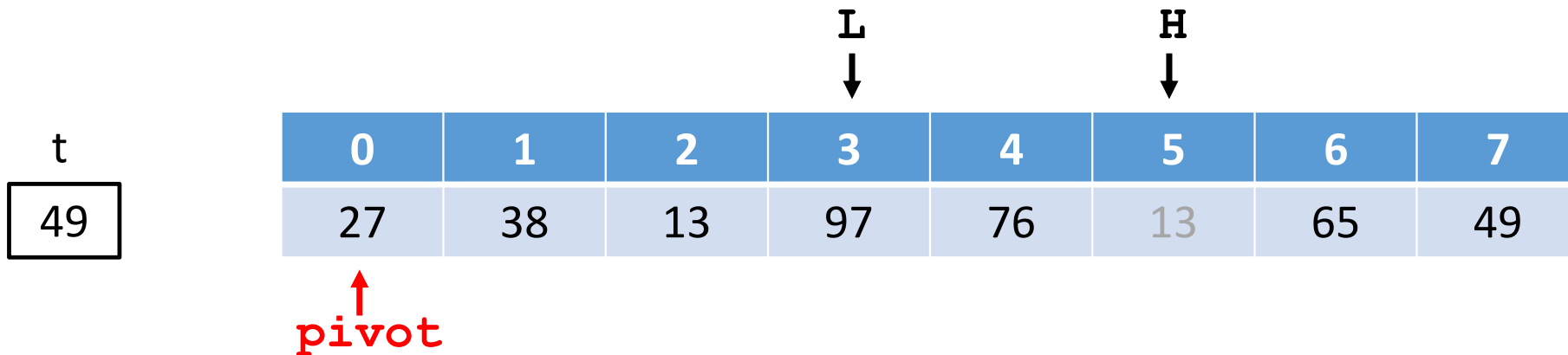
```



```

T   t = a[L];           //把最左元素当作基准
while(L < H) {
    while(L < H && !(a[H]<t) ) //H向左, 直到遇见比pivot小的
        H --;
    a[L] = a[H];
    while(L<H && a[L]<t) //L向右, 直到遇见比pivot大的
        L++;
    a[H] = a[L];
}
a[L] = t;   return L;

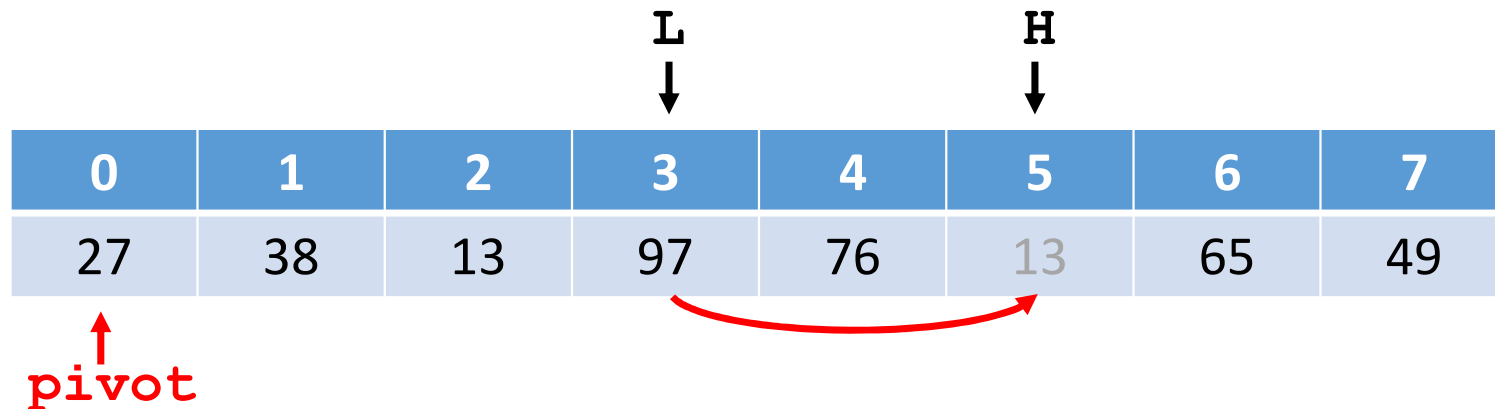
```



```

T   t = a[L];      //把最左元素当作基准
while(L < H){
    while(L < H && !(a[H]<t) ) //H向左, 直到遇见比pivot小的
        H --;
    a[L] = a[H];
    while(L<H && a[L]<t ) //L向右, 直到遇见比pivot大的
        L++;
    a[H] = a[L];
}
a[L] = t;    return L;

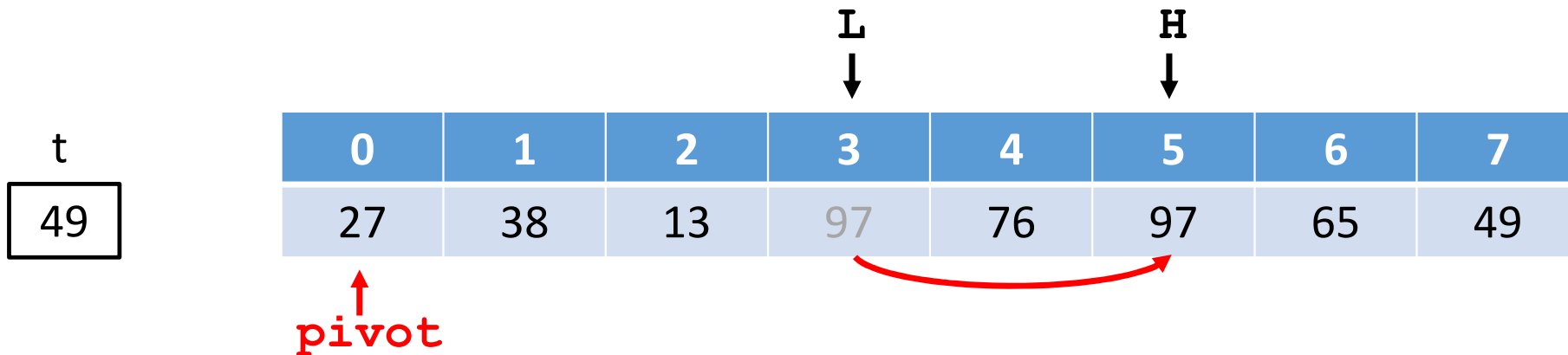
```



```

T   t = a[L];      //把最左元素当作基准
while(L < H){
    while(L < H && !(a[H]<t) ) //H向左, 直到遇见比pivot小的
        H --;
    a[L] = a[H];
    while(L<H && a[L]<t ) //L向右, 直到遇见比pivot大的
        L++;
    a[H] = a[L];
}
a[L] = t;    return L;

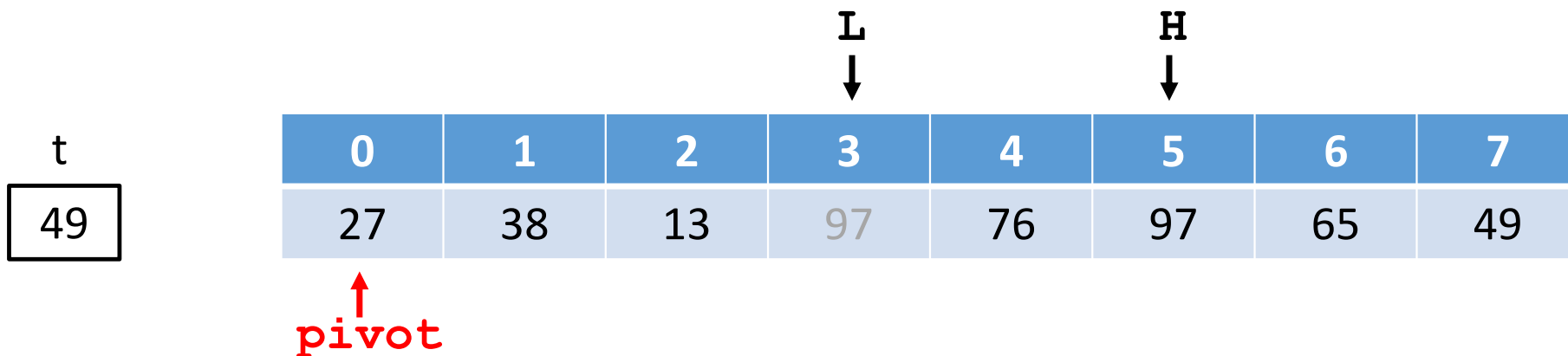
```



```

T   t = a[L];           //把最左元素当作基准
while(L < H) {
    while(L < H && !(a[H]<t) ) //H向左, 直到遇见比pivot小的
        H --;
    a[L] = a[H];
    while(L<H && a[L]<t ) //L向右, 直到遇见比pivot大的
        L++;
    a[H] = a[L];
}
a[L] = t;    return L;

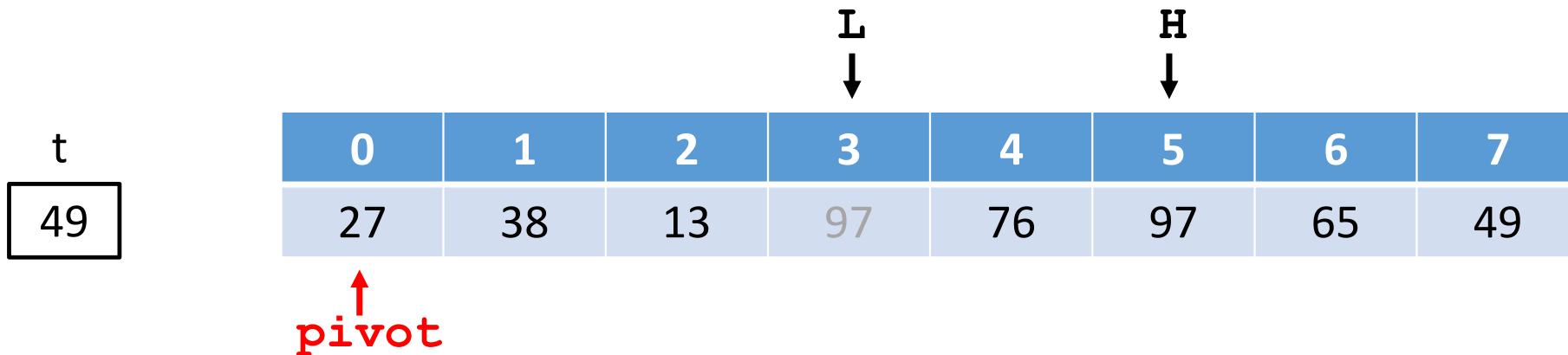
```



```

T  t = a[L];    //把最左元素当作基准
while(L < H){
    while(L < H && !(a[H]<t) ) //H向左, 直到遇见比pivot小的
        H --;
    a[L] = a[H];
    while(L<H && a[L]<t ) //L向右, 直到遇见比pivot大的
        L++;
    a[H] = a[L];
}
a[L] = t;    return L;

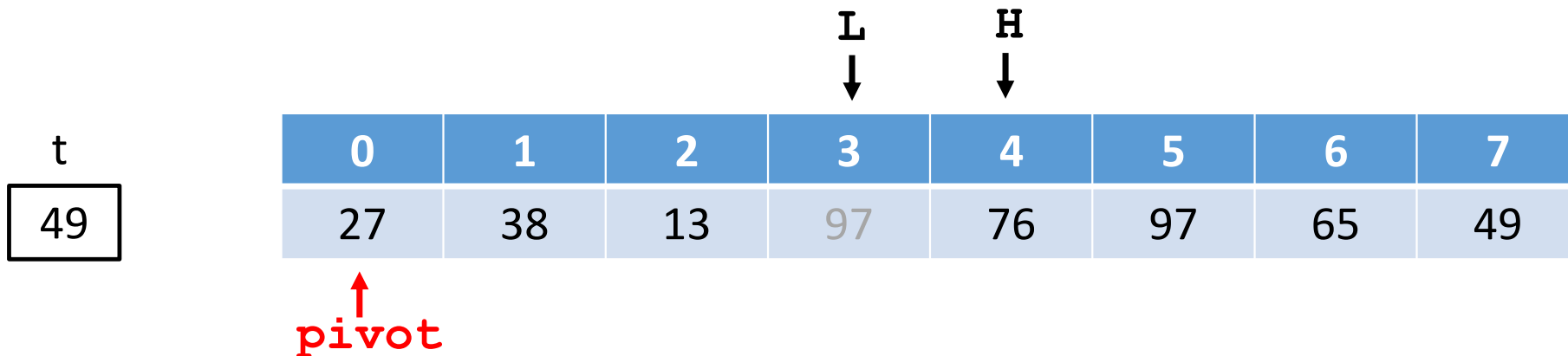
```



```

T  t = a[L];      //把最左元素当作基准
while(L < H){
    while(L < H && !(a[H]<t) ) //H向左, 直到遇见比pivot小的
        H --;
    a[L] = a[H];
    while(L<H && a[L]<t ) //L向右, 直到遇见比pivot大的
        L++;
    a[H] = a[L];
}
a[L] = t;    return L;

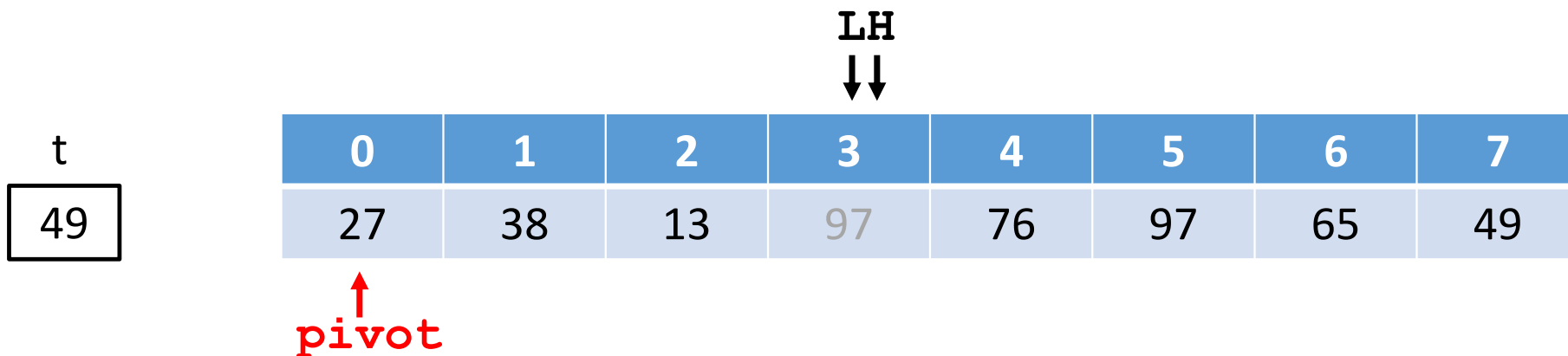
```



```

T  t = a[L];      //把最左元素当作基准
while(L < H){
    while(L < H && !(a[H]<t) ) //H向左, 直到遇见比pivot小的
        H --;
    a[L] = a[H];
    while(L<H && a[L]<t ) //L向右, 直到遇见比pivot大的
        L++;
    a[H] = a[L];
}
a[L] = t;    return L;

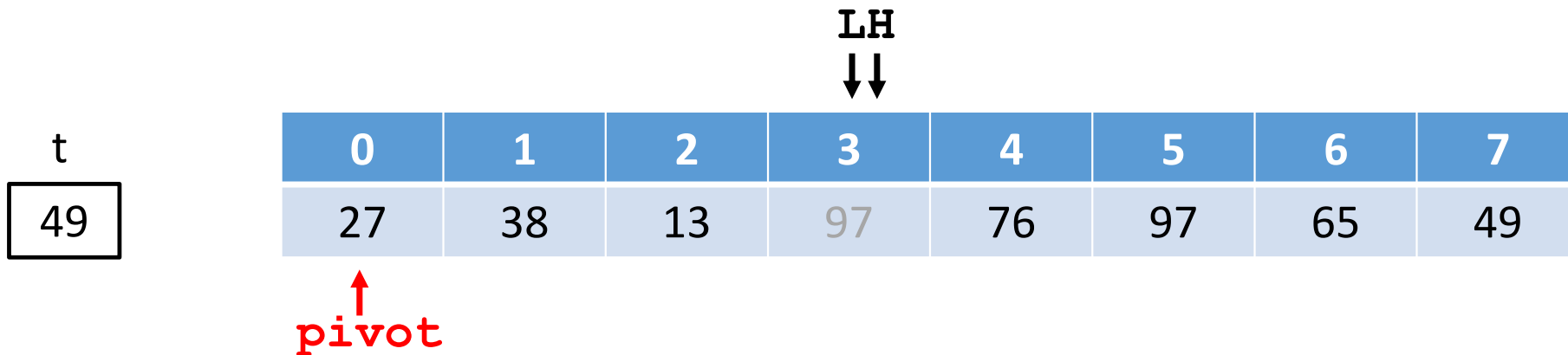
```




```

T  t = a[L];    //把最左元素当作基准
while(L < H){
    while(L < H && !(a[H]<t) ) //H向左, 直到遇见比pivot小的
        H--;
    a[L] = a[H];
    while(L<H && a[L]<t ) //L向右, 直到遇见比pivot大的
        L++;
    a[H] = a[L];
}
a[L] = t;    return L;

```



```

T  t = a[L];    //把最左元素当作基准
while(L < H){
    while(L < H && !(a[H]<t) ) //H向左, 直到遇见比pivot小的
        H --;
    a[L] = a[H];
    while(L<H && a[L]<t ) //L向右, 直到遇见比pivot大的
        L++;
    a[H] = a[L];
}
a[L] = t;    return L;

```

49	0	1	2	3	4	5	6	7
	27	38	13	97	76	97	65	49

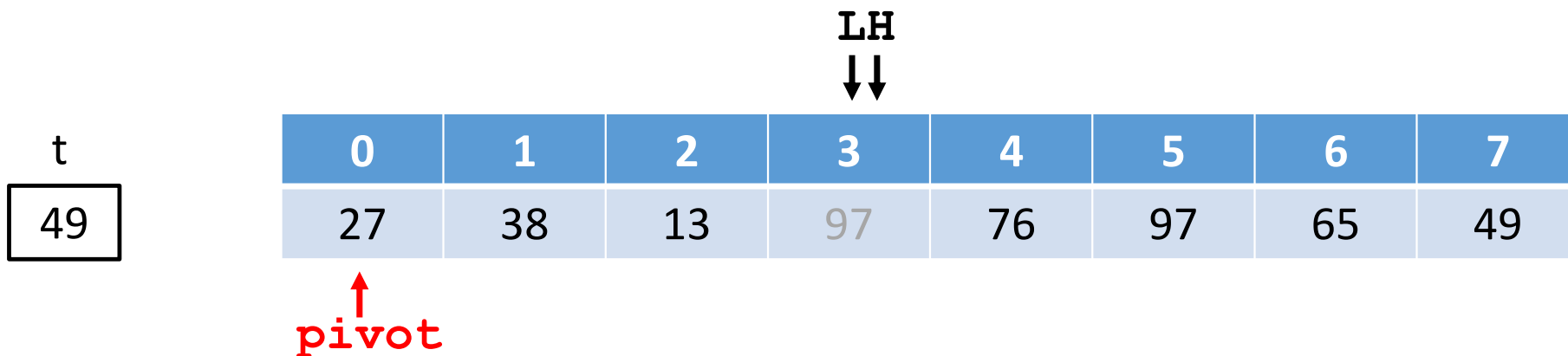
LH
↓
↓

↑
pivot


```

T   t = a[L];           //把最左元素当作基准
while(L < H) {
    while(L < H && !(a[H]<t) ) //H向左, 直到遇见比pivot小的
        H--;
    a[L] = a[H];
    while(L<H && a[L]<t ) //L向右, 直到遇见比pivot大的
        L++;
    a[H] = a[L];
}
a[L] = t;   return L;

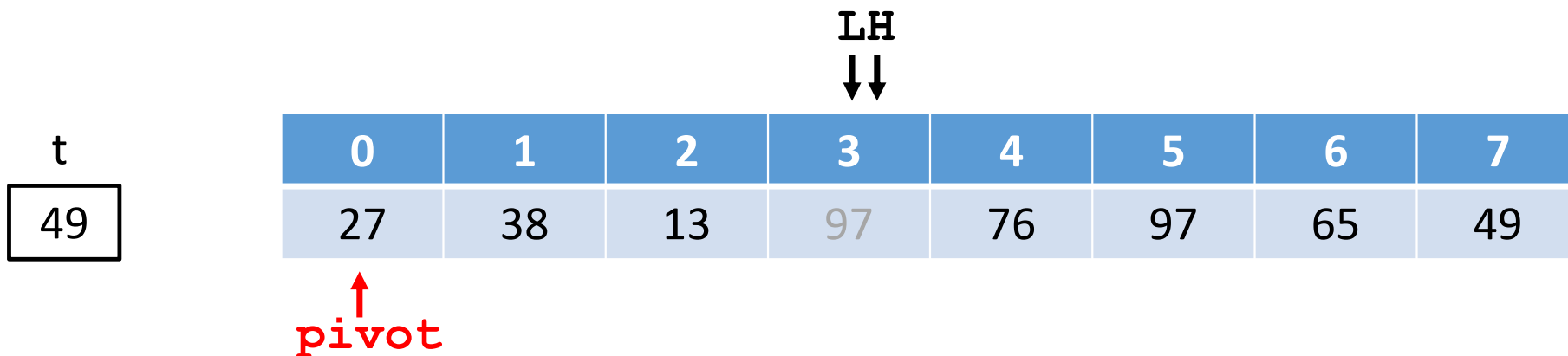
```



```

T  t = a[L];      //把最左元素当作基准
while(L < H){
    while(L < H && !(a[H]<t) ) //H向左, 直到遇见比pivot小的
        H --;
    a[L] = a[H];
    while(L<H && a[L]<t ) //L向右, 直到遇见比pivot大的
        L++;
    a[H] = a[L];
}
a[L] = t; return L;

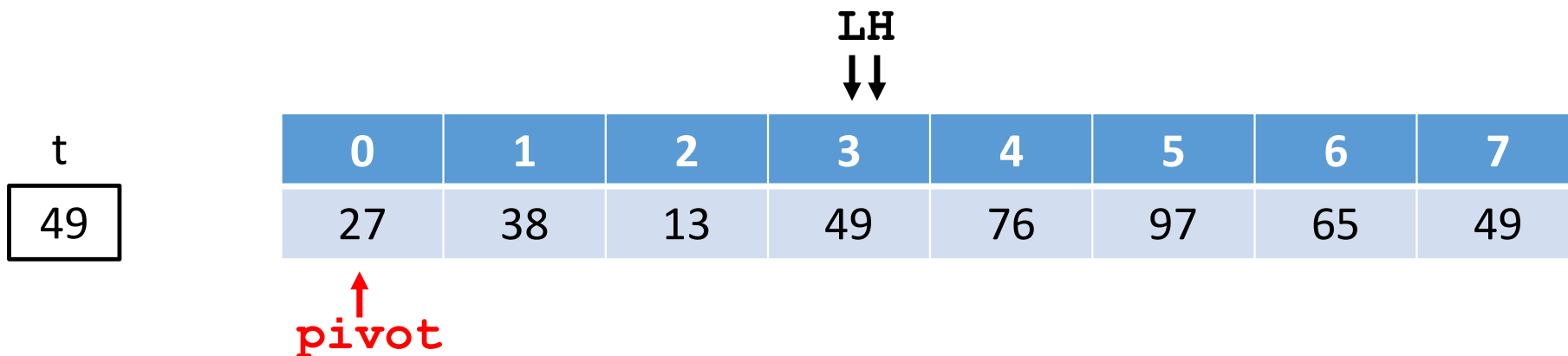
```



```

T  t = a[L];      //把最左元素当作基准
while(L < H){
    while(L < H && !(a[H]<t) ) //H向左, 直到遇见比pivot小的
        H --;
    a[L] = a[H];
    while(L<H && a[L]<t ) //L向右, 直到遇见比pivot大的
        L++;
    a[H] = a[L];
}
a[L] = t; return L;

```



```

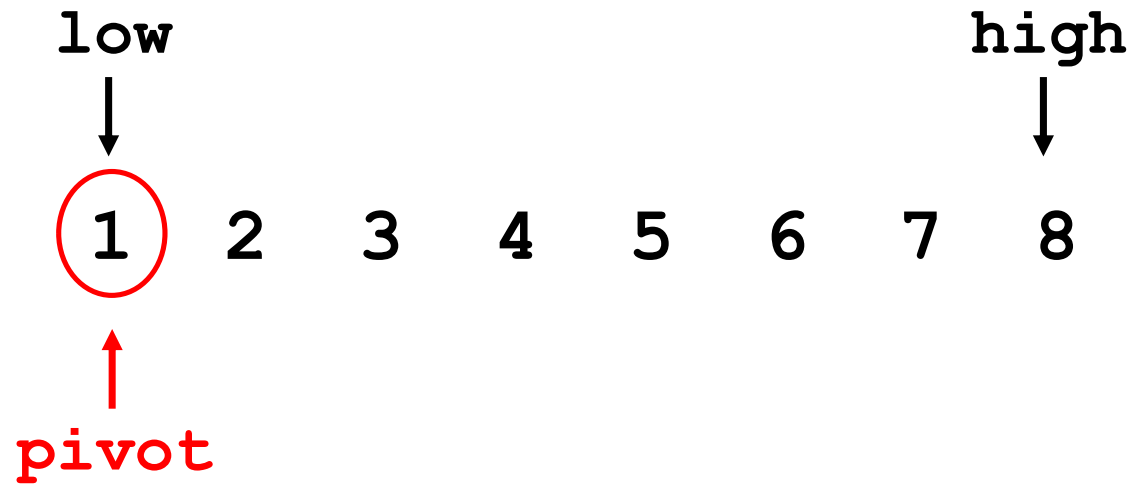
T   t = a[L];           //把最左元素当作基准
while(L < H){
    while(L < H && !(a[H]<t) ) //H向左, 直到遇见比pivot小的
        H--;
    a[L] = a[H];
    while(L<H && a[L]<t ) //L向右, 直到遇见比pivot大的
        L++;
    a[H] = a[L];
}
a[L] = t;   return L;

```

<div>t</div> <div>49</div>				<div>LH</div> <div>↓</div> <div>↓</div> <div>3</div> <div>49</div>				
	0	1	2		4	5	6	7
	27	38	13		76	97	65	49

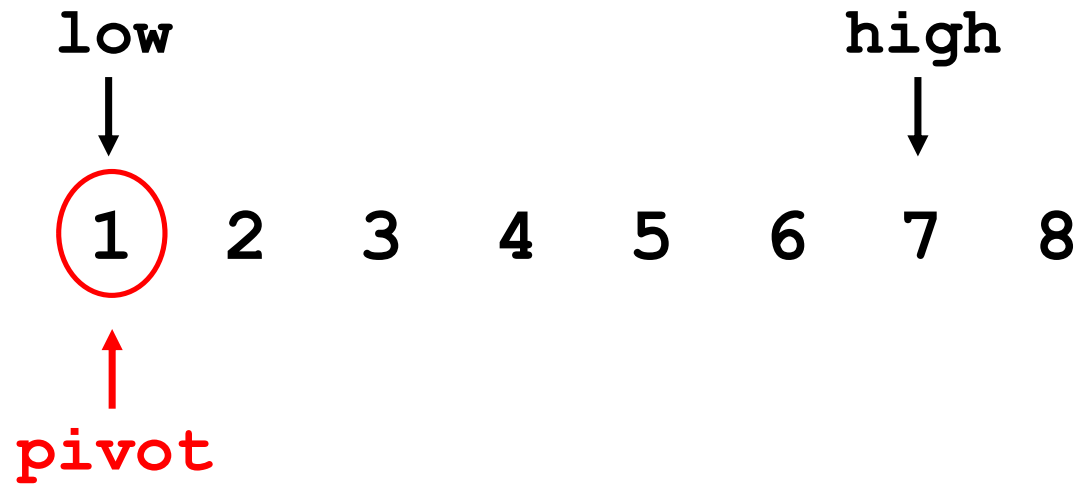
时间复杂度

• 例



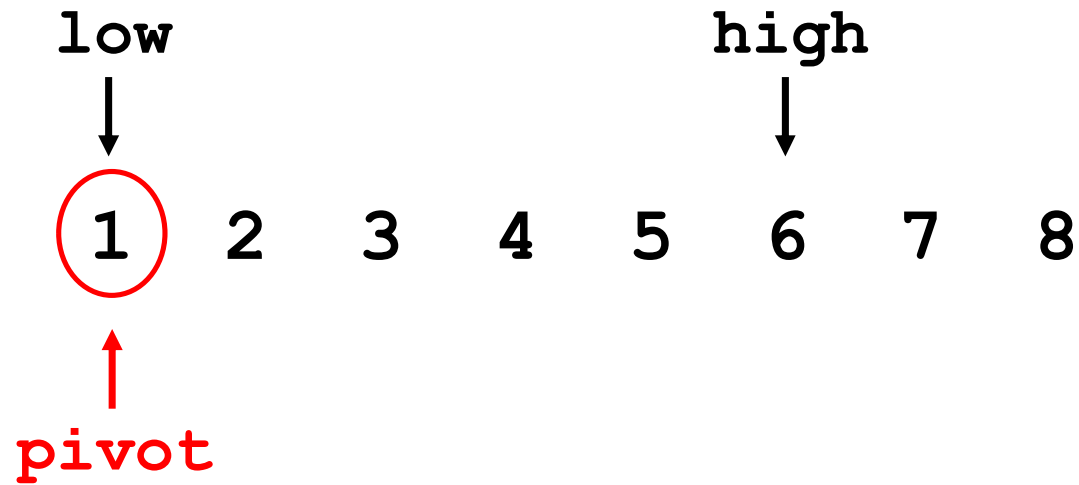
时间复杂度

• 例



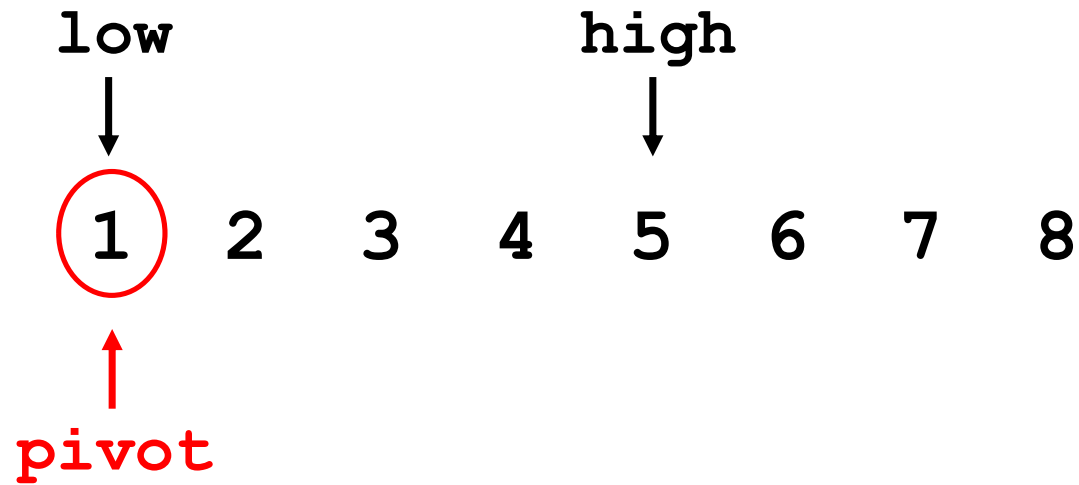
时间复杂度

• 例



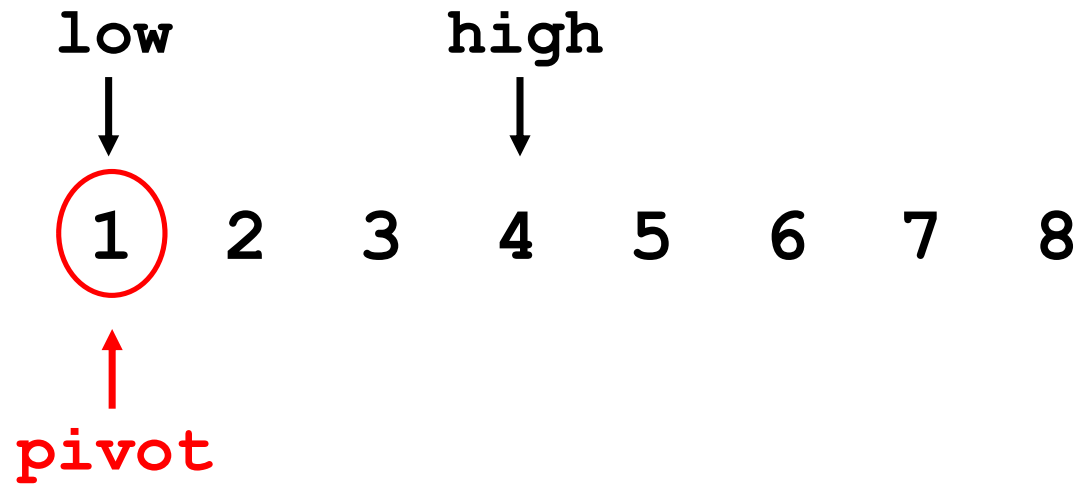
时间复杂度

• 例



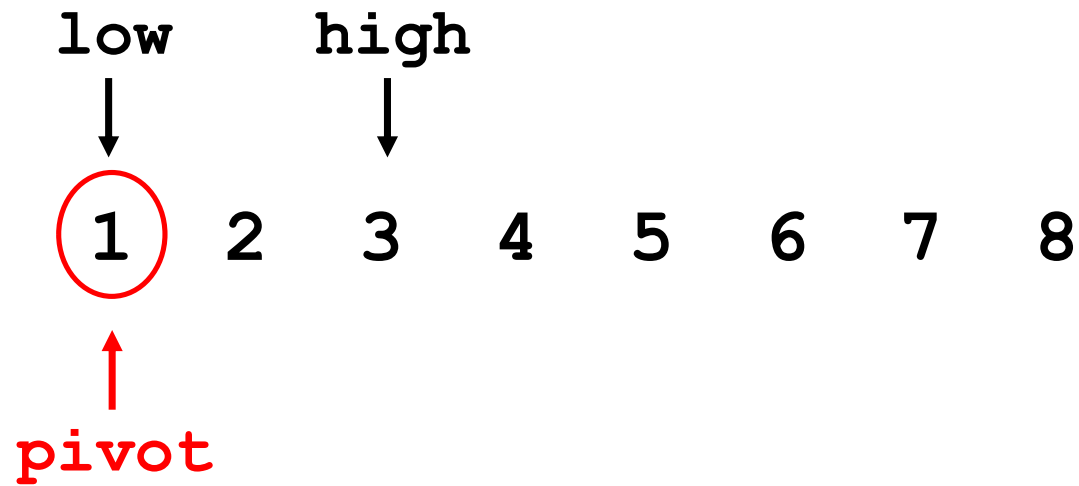
时间复杂度

• 例



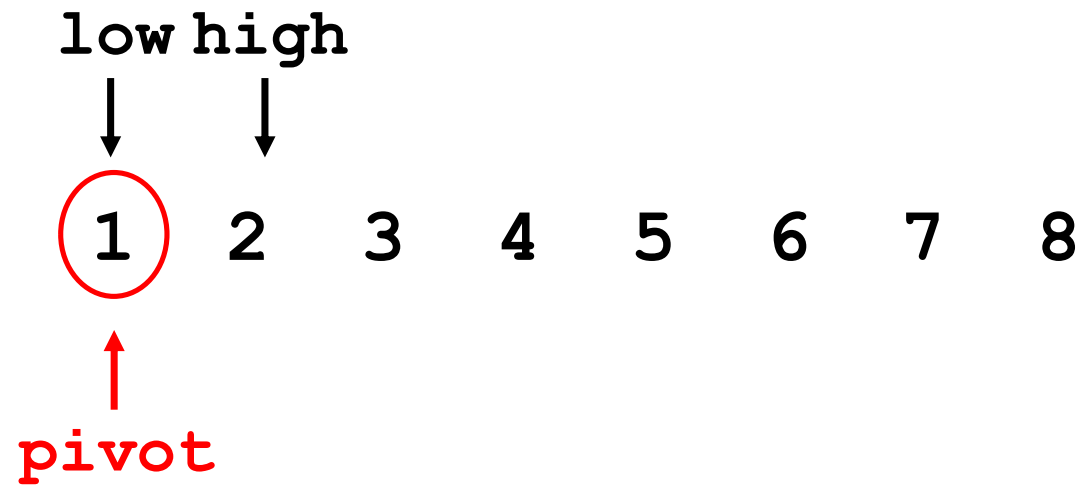
时间复杂度

• 例



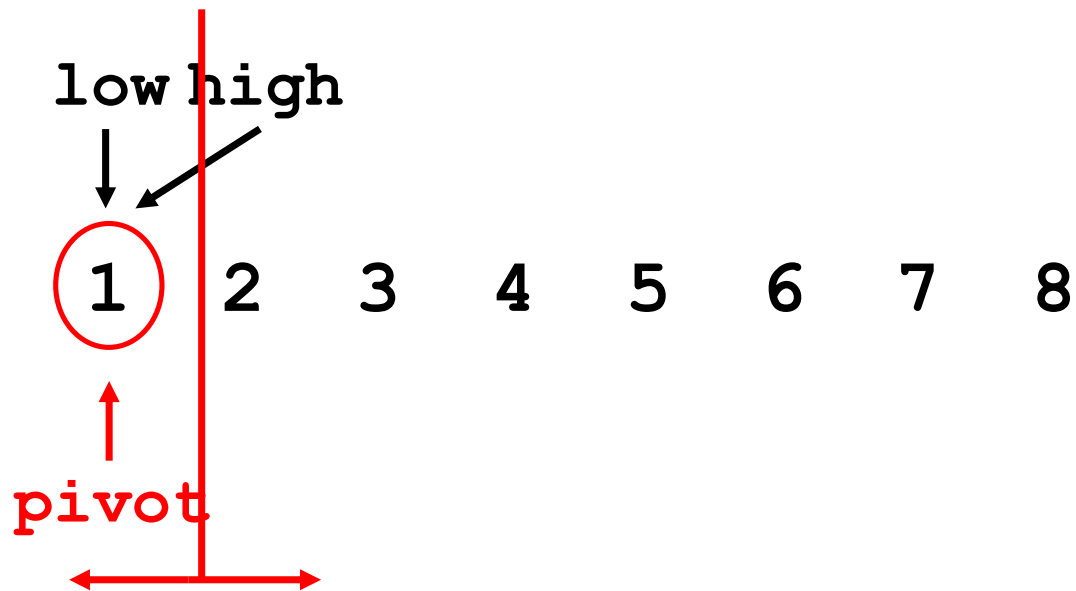
时间复杂度

• 例



时间复杂度

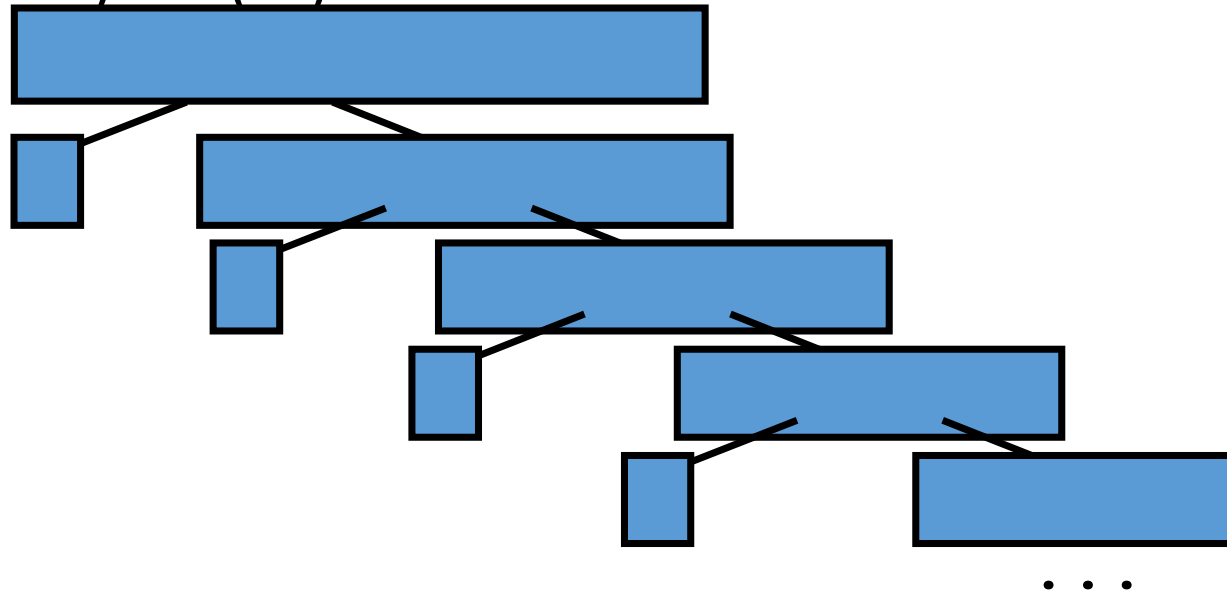
• 例



时间复杂度

- 每次划分都只分出一个元素
- 需要 n 层递归

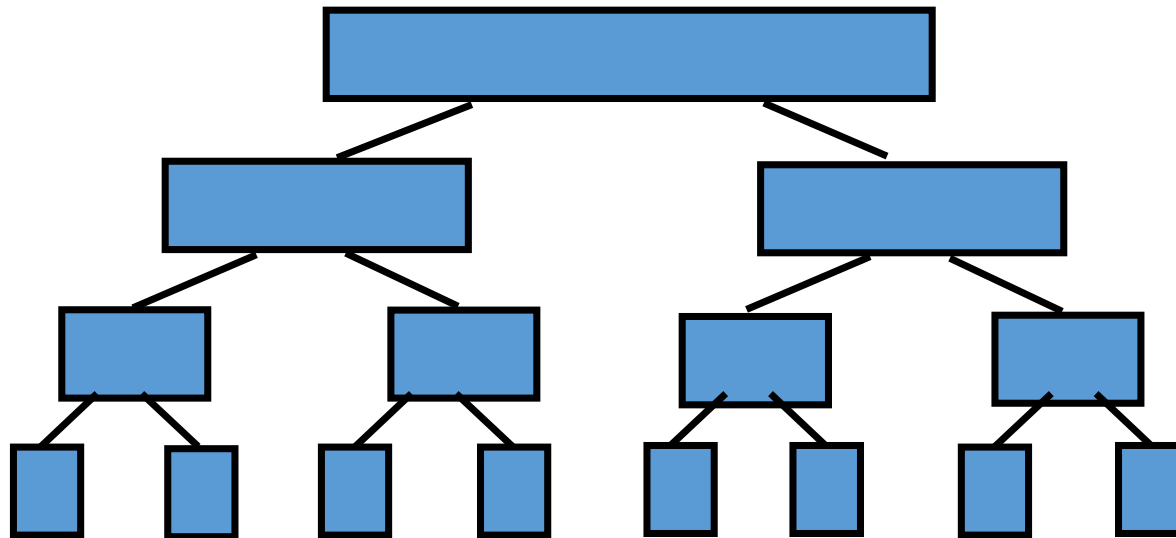
$$O(n*n) = O(n^2)$$



- **有序性好的原始数据不适合用快速排序**

时间复杂度

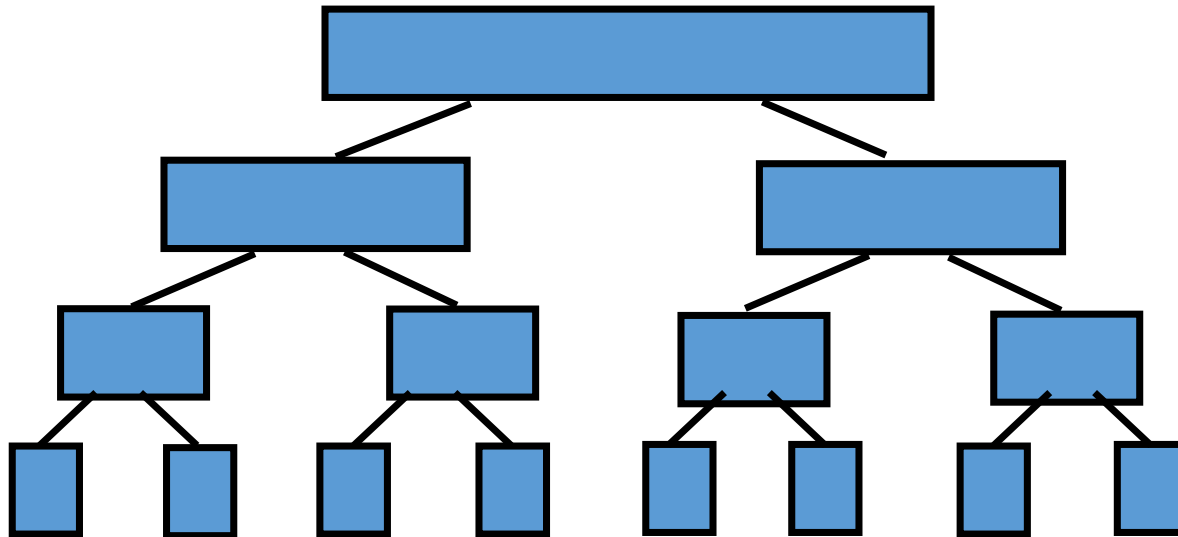
- 每次划分，分成相等的2部分
- 只要 $\log_2 n$ 层。 $O(n * \log_2 n)$ $2^k \geq n$



- 杂乱无章的原始数据适合用快速排序

空间复杂度

- 使用了递归，相当于增加了一个堆栈
- 堆栈的深度 = 递归的层数
- 最少 $\log_2 n$ ：每一次都切在中间



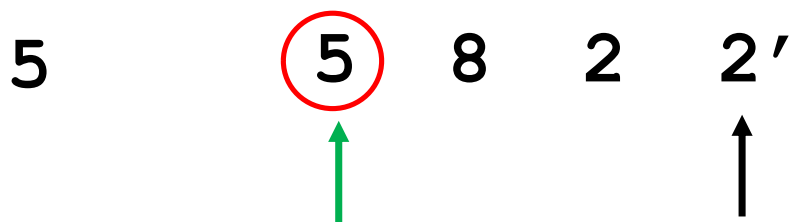
快速排序

- 不稳定:

⑤ 8 2 2'

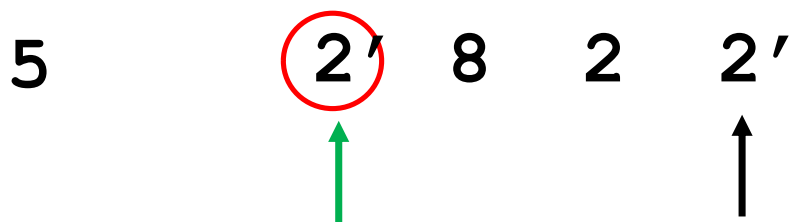
快速排序

• 不稳定:



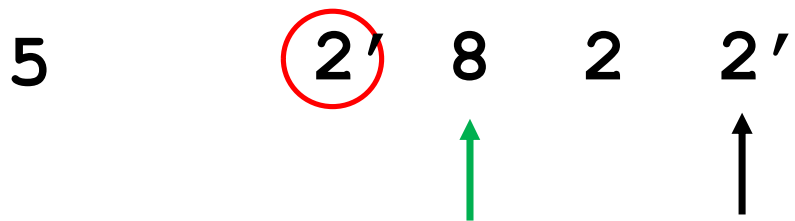
快速排序

• 不稳定:



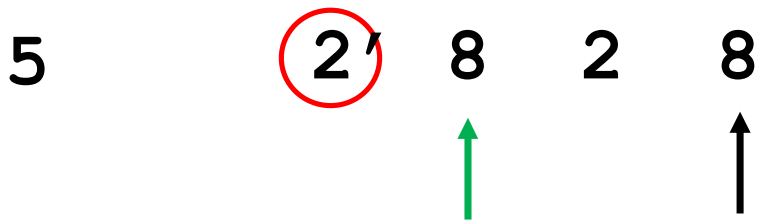
快速排序

• 不稳定:



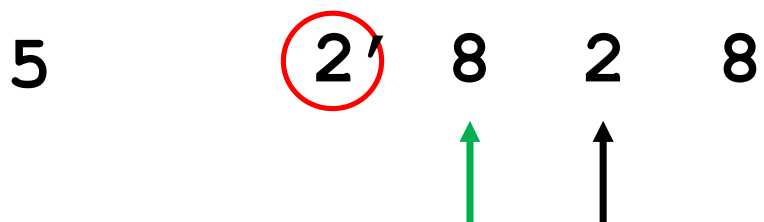
快速排序

• 不稳定:



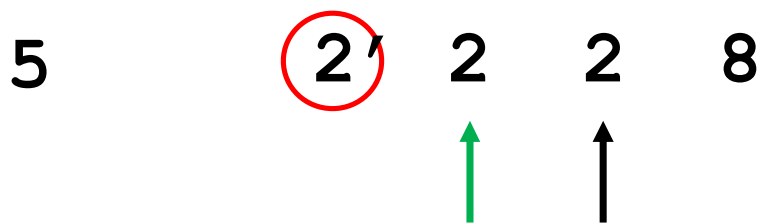
快速排序

- 不稳定:



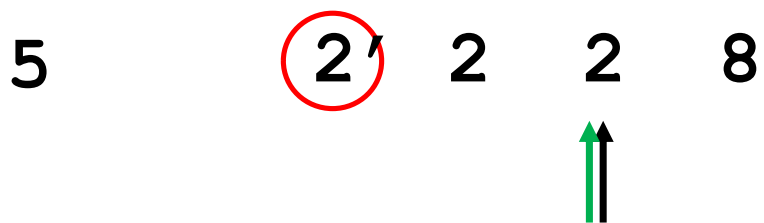
快速排序

- 不稳定:



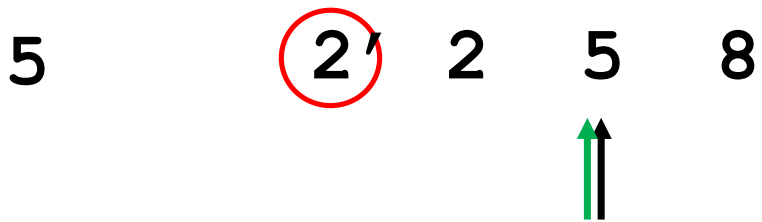
快速排序

• 不稳定:



快速排序

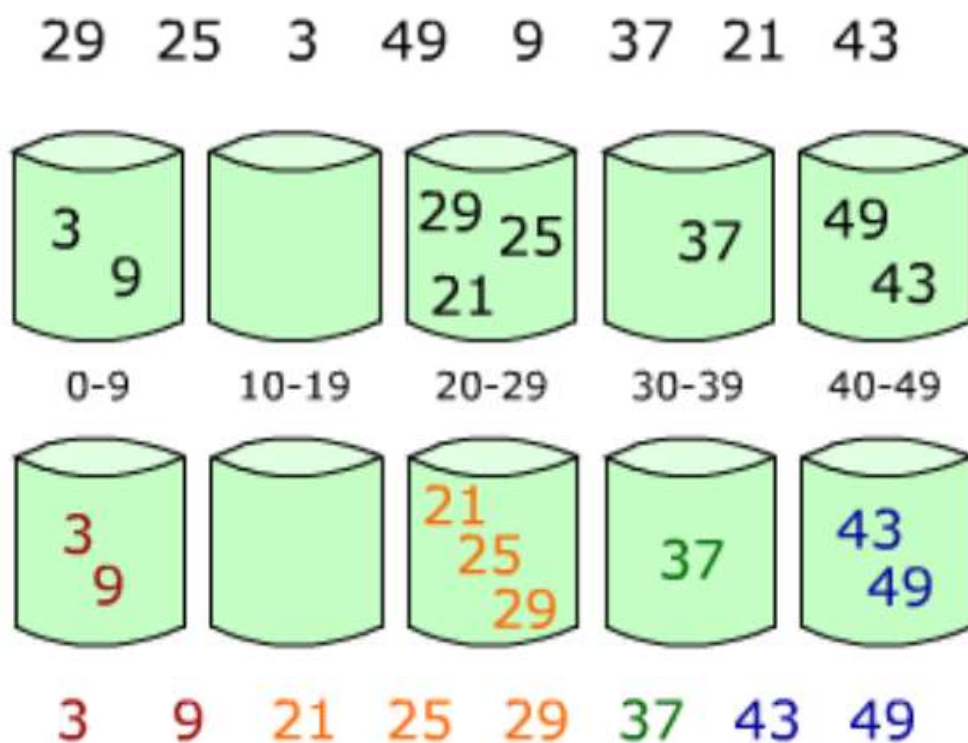
• 不稳定:



桶排序

桶排序

- 将数组元素分配到多个桶中。然后，每个桶使用任何其他排序算法或通过递归应用桶排序单独排序。最后将桶中数据收集起来。



$$O(n^2) = 8^2 = 64$$

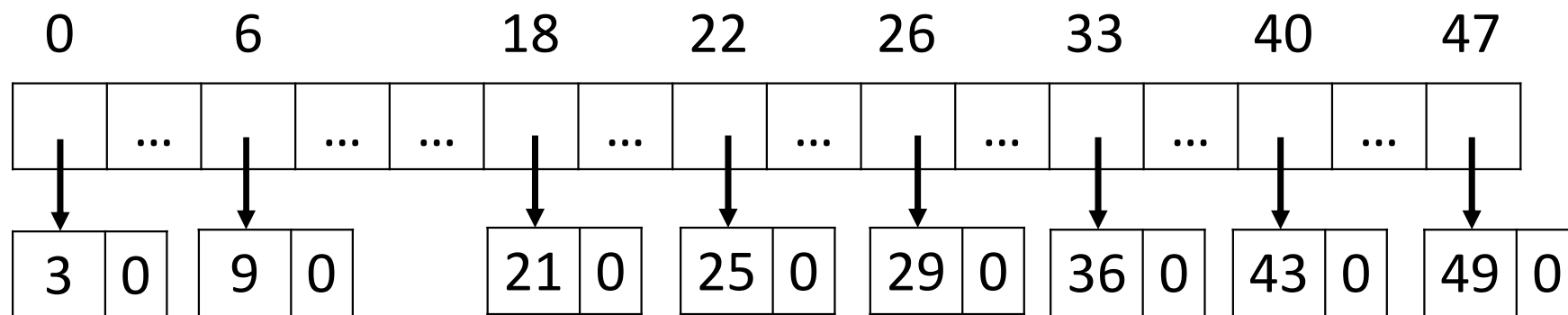
$$\begin{aligned} &2^2 + 3^2 + 1^2 + 2^2 \\ &= 4 + 9 + 1 + 4 \\ &= 16 \end{aligned}$$

$$O(n)$$

桶排序

- 最初是对整数排序：确定这组整数的取值范围 $[\min, \max]$ ，然后建 $\max - \min + 1$ 个桶。将这组整数分配到这些桶(桶用链表表示)。

29 25 3 49 9 37 21 43



{ -1, 25, -58964, 8547, -119, 0, 78596 };

```

template<typename T>
void BucketSort(vector<T> &data, const int bucket_num=0) {
    T minValue = data[0], maxValue = data[0];
    for (int i = 1; i < data.size(); i++) {
        if (data[i] > maxValue)
            maxValue = data[i];
        if (data[i] < minValue)
            minValue = data[i];
    }
    int bucket_n = maxValue - minValue + 1;
    if (bucket_num != 0) bucket_n = bucket_num;
    //  $D = M - m$      $D/S + 1 = N$      $S = D / (N - 1)$ 
    T interval = (maxValue - minValue) / (bucket_n - 1);
}

```

计算值的范围 $O(n)$

```
vector<vector<T>> buckets(bucket_n);  
for (auto e : data)  
    buckets[(e - minValue) / interval].push_back(e);
```

将元素分配到桶中
 $O(n)$


```

int k = 0;
for (int i = 0; i < bucket_n; i++) {
    int bucketSize = buckets[i].size();
    if (bucketSize > 0) {
        if (bucketSize > 1) {
            insert_sort(buckets[i]);
        }
        for (int j = 0; j < bucketSize; j++)
        {
            data[k] = buckets[i][j];
            k++;
        }
    }
}

```

$O(k)$

$O((n/k)^2)$

$O(n^2/k)$

时间复杂度

$$O\left(n + \frac{n^2}{k} + k\right)$$

元素分配到桶中

对所有桶排序
 $O(n^2/k)$

收集k个链式桶

$k=n$ ，理论上线性时间复杂度 $O(n)$

但如果不均匀分配到桶中，可能聚集于一个桶中，退化为 $O(n^2)$

空间复杂度 $O(n+k)$

{ -1, 25, -58964, 8547, -119, 0, 78596 };

多关键字排序-基数排序

多关键字排序

- 前面的排序方法只有一个关键字（排序码）
- 有时候可能存在多个关键字，比如扑克牌

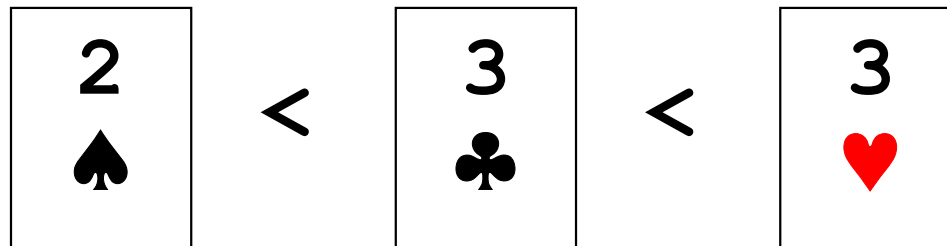
- 关键字1：面值

2 < 3 < ... < K < A

- 关键字2：花色

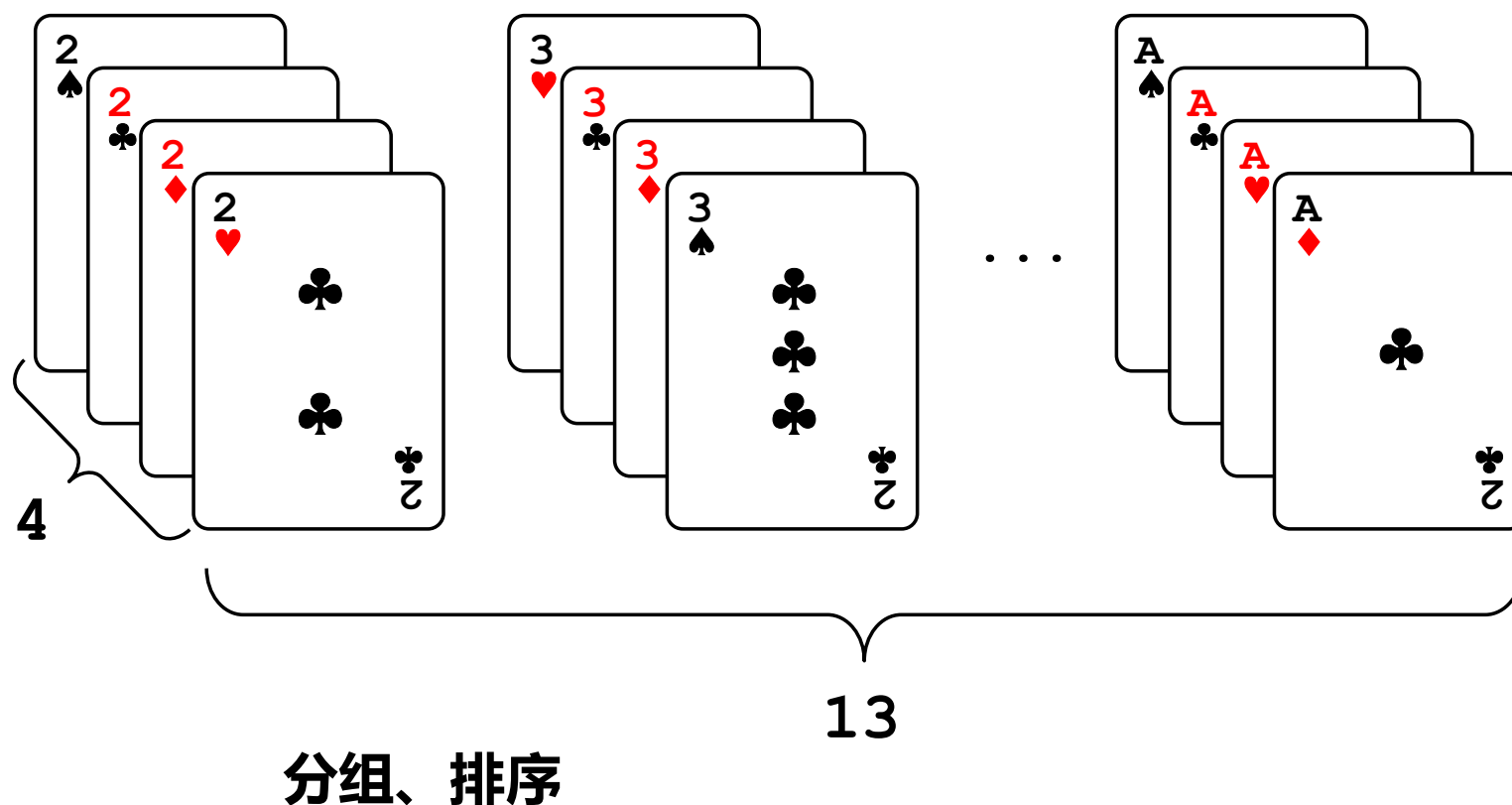
♣ < ♦ < ♥ < ♠

- 在扑克牌中，面值是主关键字，当主关键字相同时再比较次关键字



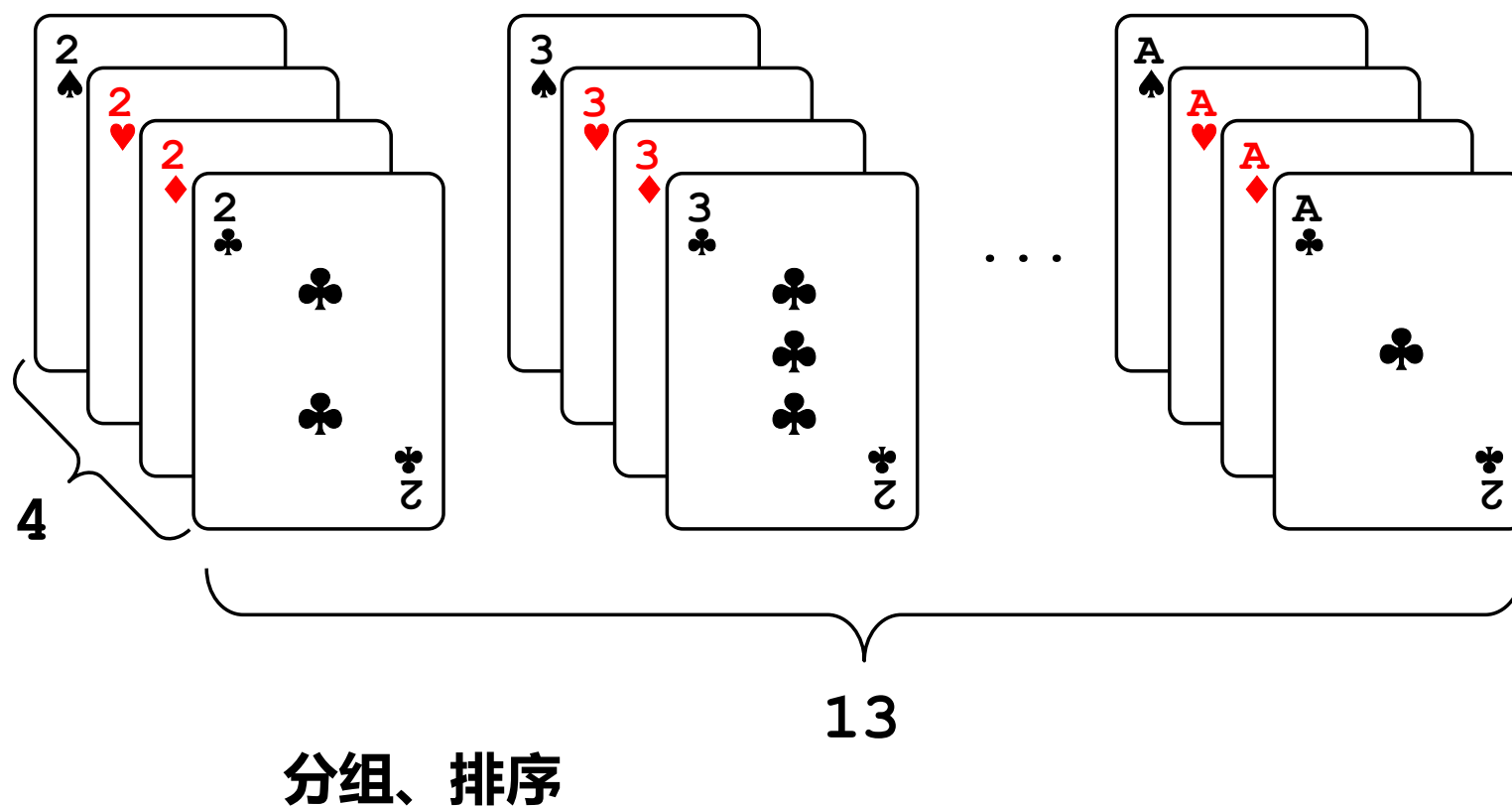
多关键字排序：高位优先法，简称MSD法

- 主关键字优先对扑克牌进行排序



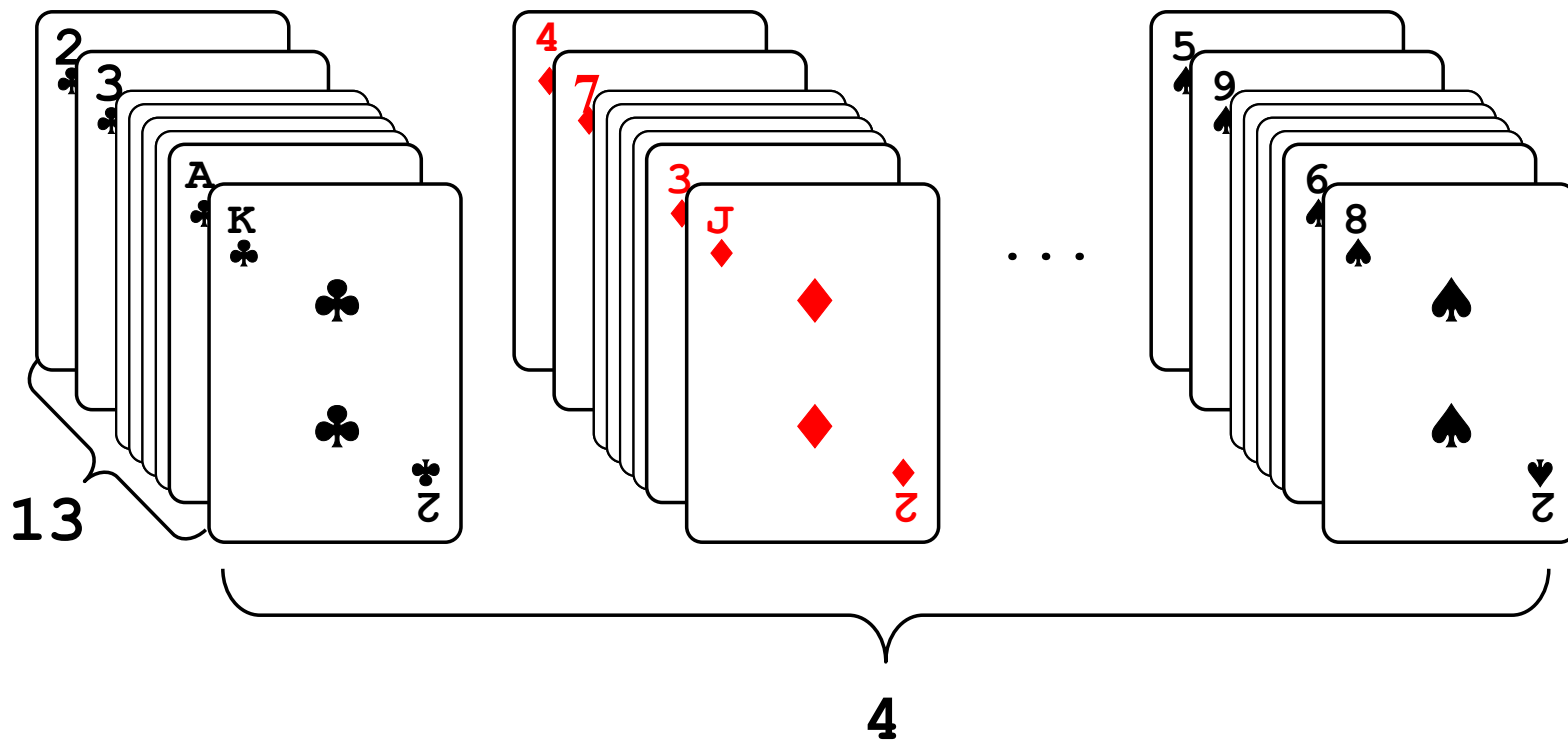
多关键字排序：高位优先法，简称MSD法

- 主关键字优先对扑克牌进行排序



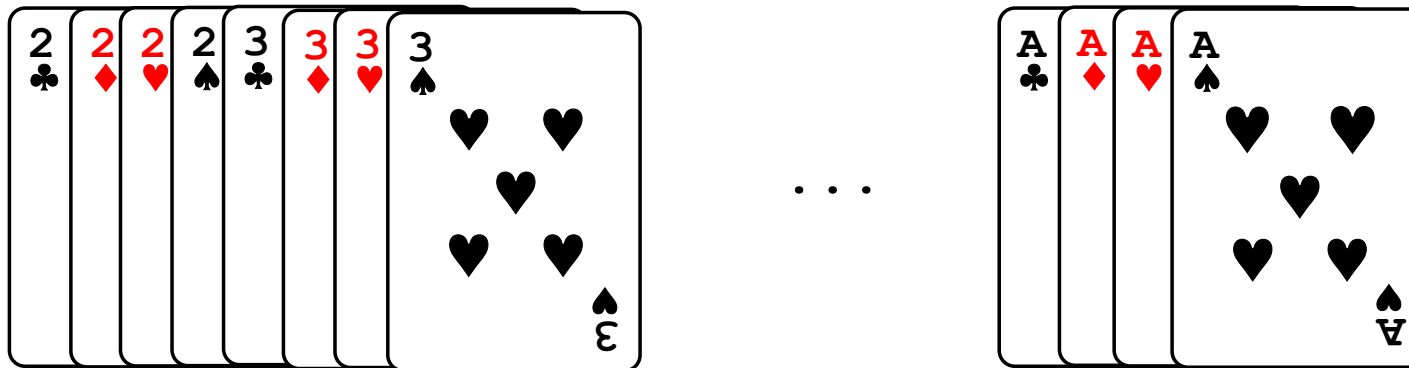
多关键排序：低位优先法，简称 LSD 法

- 次关键字优先对扑克牌进行排序



多关键排序：低位优先法，简称 LSD 法

- 次关键字优先对扑克牌进行排序



要求对每个关键字的排序是稳定的，否则会打乱！

基数排序

- 整数排序为例
 - 在算术中, $300 > 299$, 因为前者的百位数比后者的百位数大
 - 也就是我们比较两个数字的大小, 总是先看最高位, 再看次高位, 以此类推
 - 因此可以把每一位数看作是一个关键字
 - 每个关键字的取值范围是0~9
 - 这里有3个关键字

基数排序

- 字典排序
 - $CBAD < CDAB$
 - 可以把这里的每一位看作一个关键字
 - 每一个关键字的取值范围是A~Z
 - 这里有4个关键字

基数排序

- 链式基数排序：对一组整数排序

278、109、63、930、589、184、505、269、8、83

- 使它们具有相同的位数，高位不足的补0：

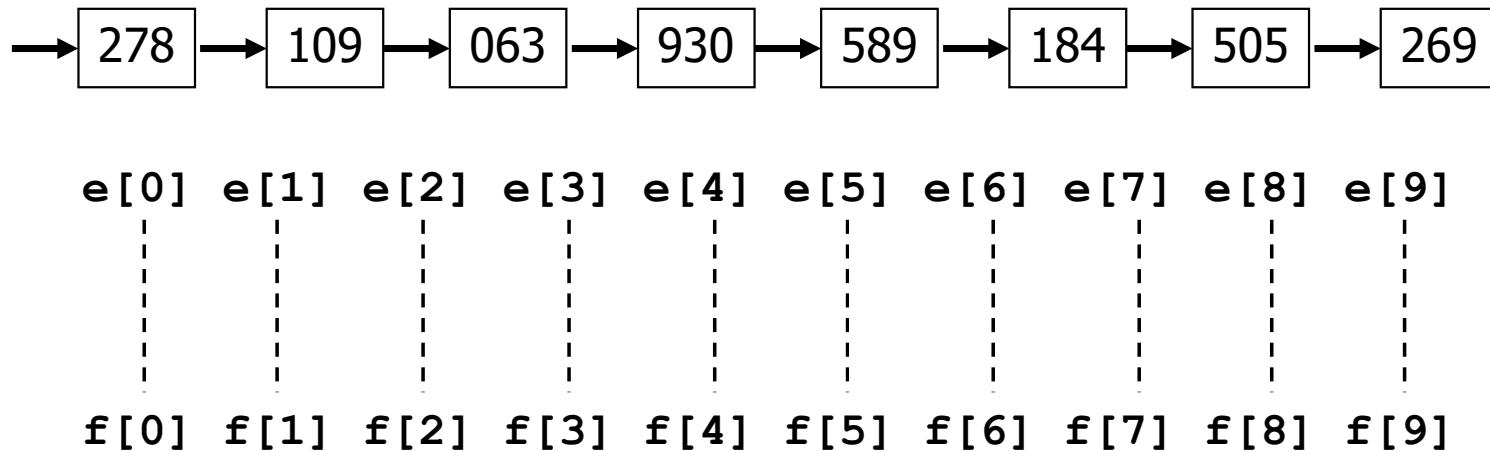
278、109、063、930、589、184、505、269、008、083

- 注意：

- 有3个关键字，每个关键字取值范围是：0,1,2,...,9

基数排序

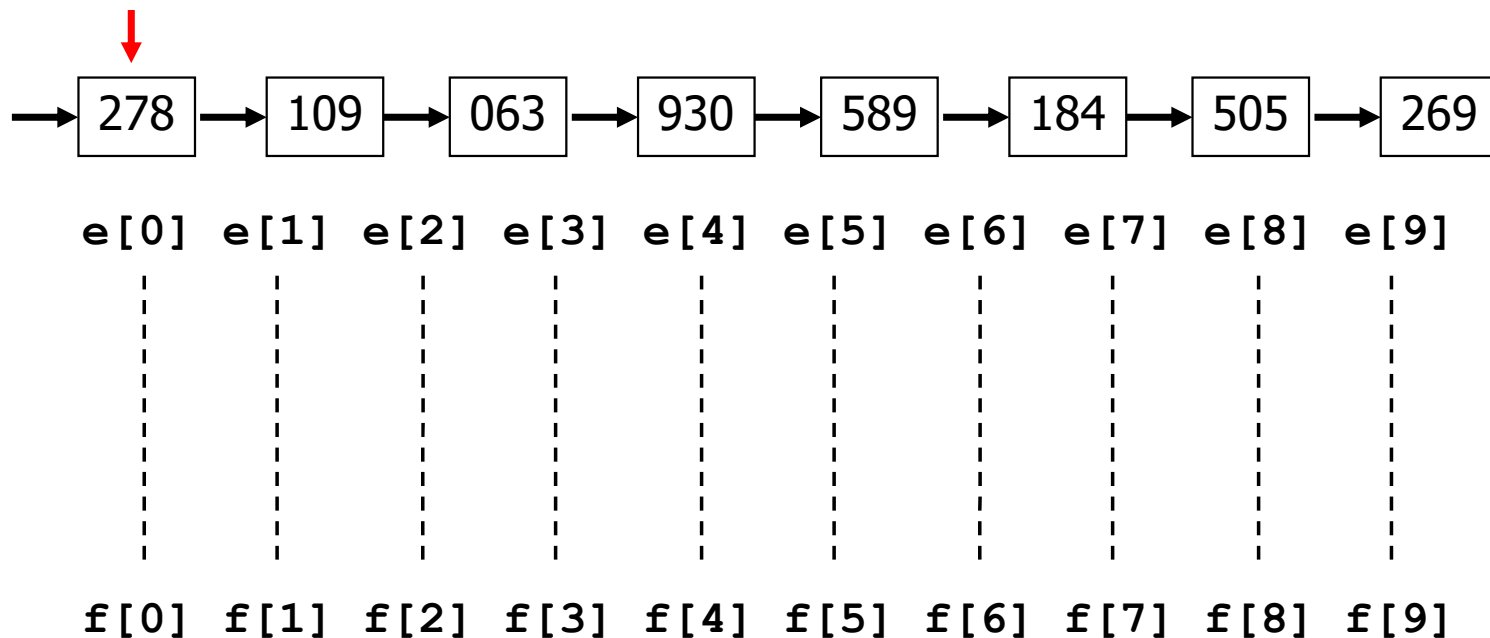
- 基本思想：“分配” + “收集”
 - 首先原始数据被保存在链表中
 - 关键字的取值范围是0~9，因此再准备10个队列（也用链表实现）



基数排序

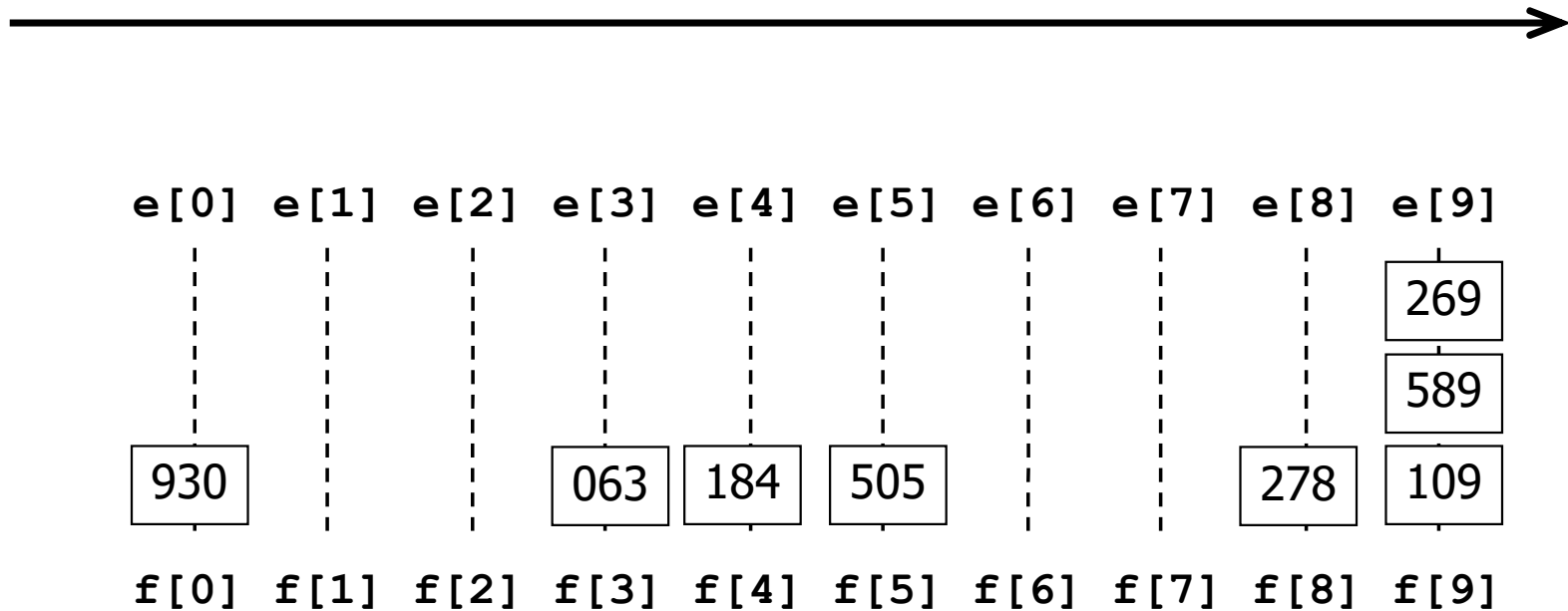
- 第1趟：针对个位数

- “分配”：扫描每一个记录，按照个位数分别放到相应的队列中



基数排序

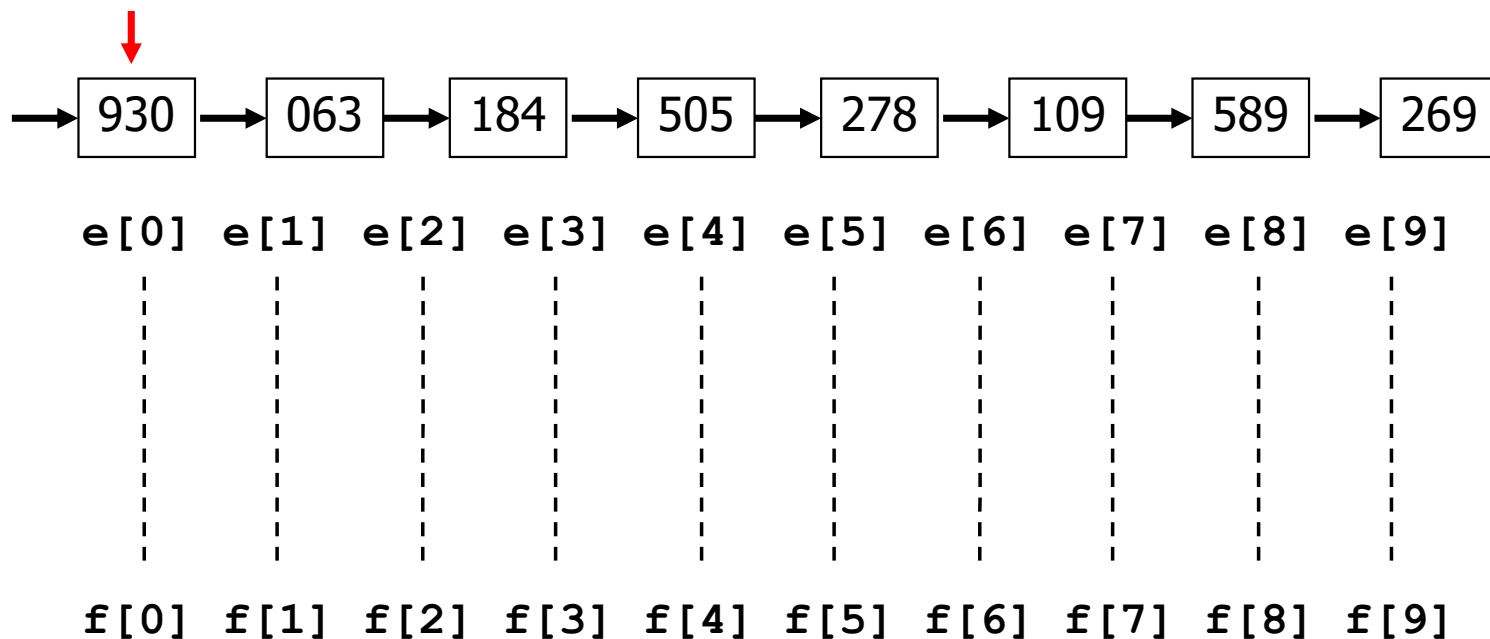
- 第1趟：针对个位数
 - “收集”：将各队列的记录重新组织成一个链表



基数排序

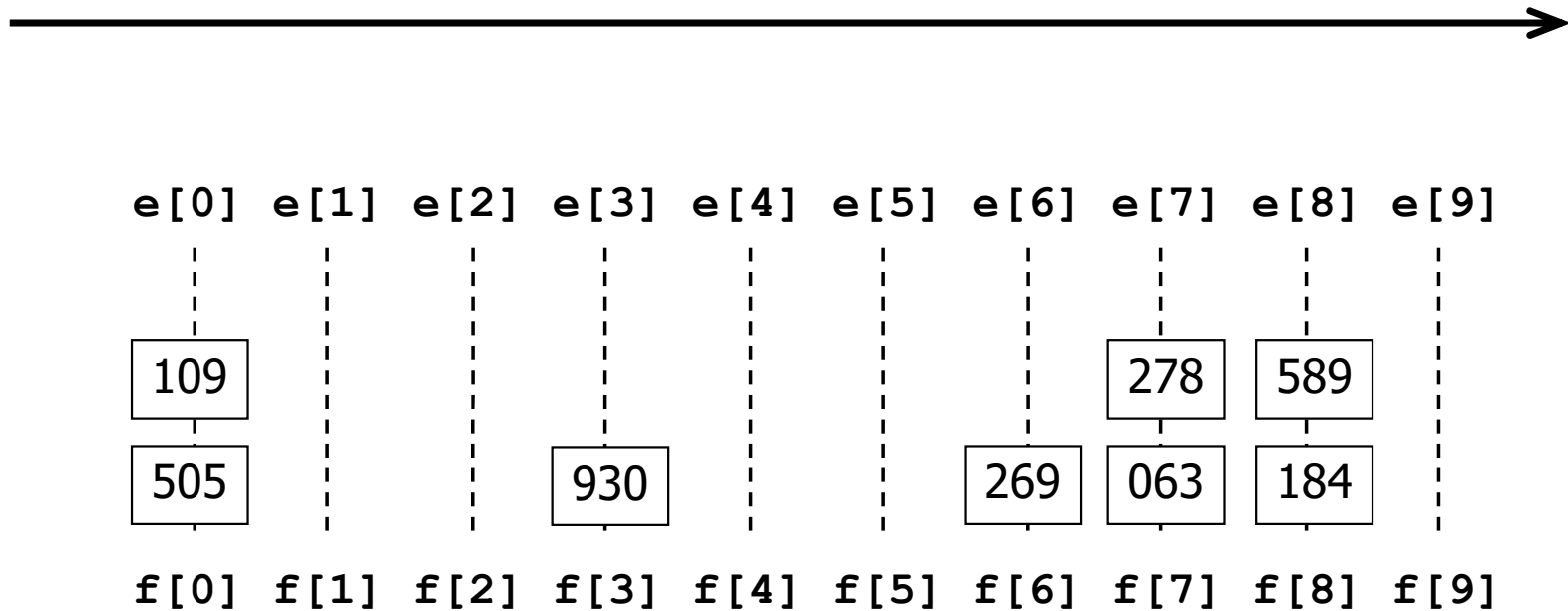
- 第2趟：针对十位数

- “分配”：扫描每一个记录，按照十位数分别放到相应的队列中



基数排序

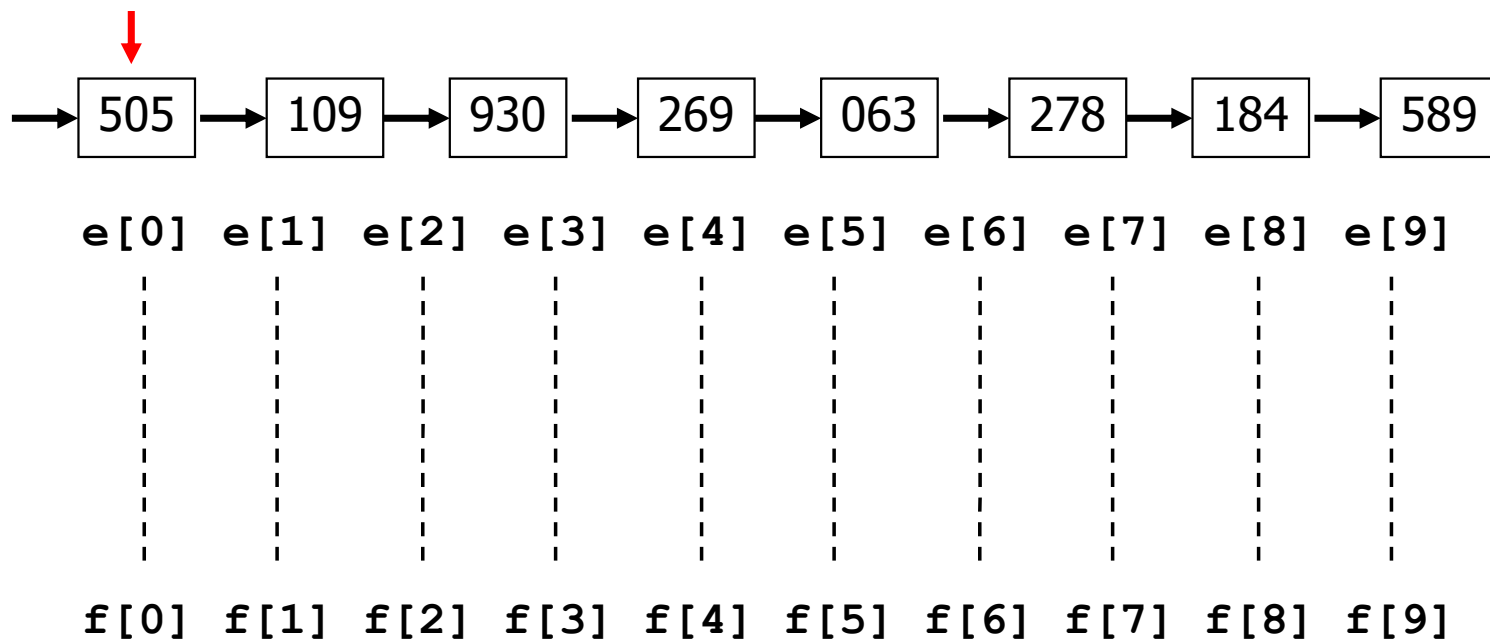
- 第2趟：针对十位数
 - “收集”：将各队列的记录重新组织成一个链表



基数排序

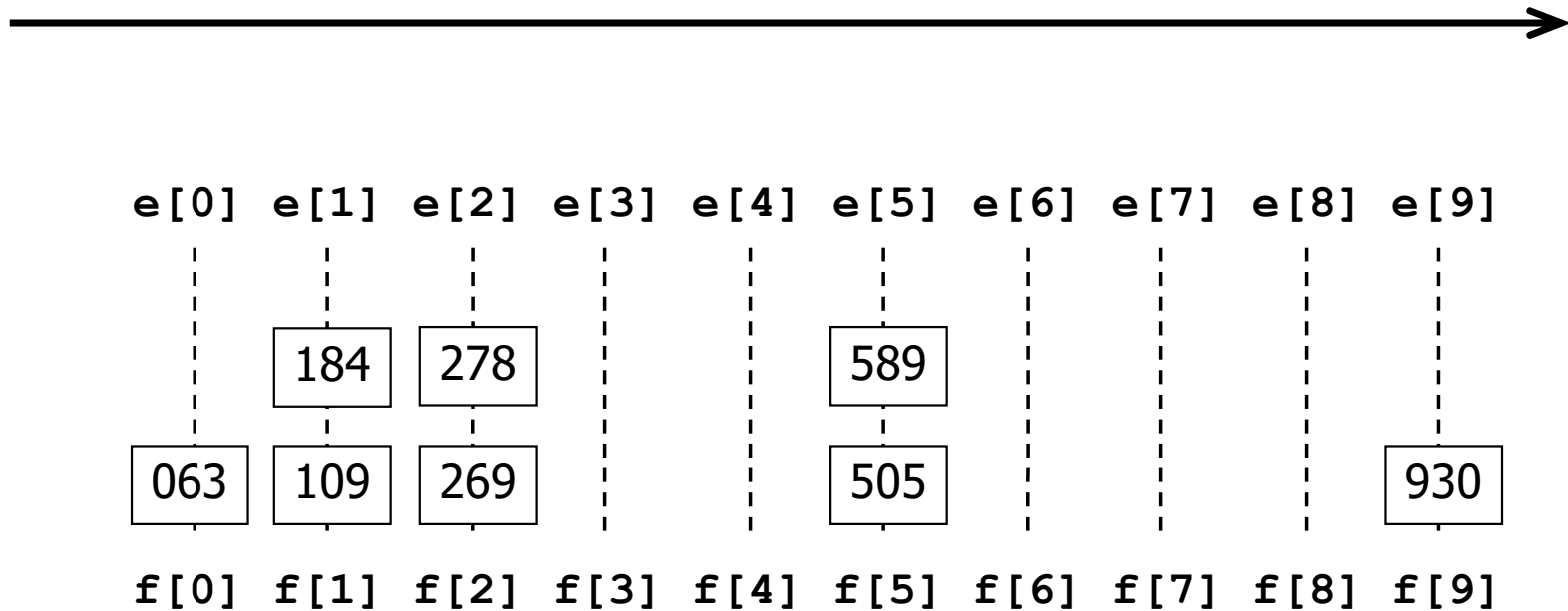
- 第3趟：针对百位数

- “分配”：扫描每一个记录，按照百位数分别放到相应的队列中



基数排序

- 第3趟：针对百位数
 - “收集”：将各队列的记录重新组织成一个链表



时间复杂度

- 假设：记录数 n ,关键字数 d , 关键字取值范围 rd (如十进制为10)
- 分配（每趟）： $O(n)$, n 个记录要分配
- 收集（每趟）： $O(rd)$, rd 个桶(链队列)要收集
- 总的时间复杂度： $T(n) = O(d*(n+rd))$

空间复杂度

- rd个链表各2个（队头、队尾）指针，n个元素各n个结点(数据+指针)
- 空间复杂度： $S(n) = O(n+2rd)$

关注

<https://hwdong-net.github.io>

youtube: hwdong

网易云课堂: hwdong

腾讯课堂: hwdong.ke.qq.com

B站: hw-dong

QQ群: 376330161