



# 排序与查找算法

---

北京亚嵌教育研究中心

# 排序与查找算法

## 排序算法：

冒泡排序

插入排序

快速排序

选择排序

归并排序

## 查找算法：

顺序查找

折半查找



## Section 1

---

# 冒泡、选择排序

## 冒泡排序

冒泡排序：

升序排列：若一组数据存储在某一数组中，完成升序排列后，小下标处存放小值，大下标处存放大数。

# 冒泡排序

## 冒泡排序：

算法描述：将一组数据中**相邻两元素**两两比较，若小下标处存放了大数则进行数据交换，若从待排序元素起始处到结束处元素全部比较完毕，则数列中的最大值将被排列到数列的最尾端，这样的一次操作叫做一次冒泡。

# 冒泡排序



下标	0	1	2	3	4
内容	8	7	9	6	5
第一趟排序	8	8	9	6	5
	7	8	9	6	5
	7	8	6	6	5
	7	8	6	9	9

## 冒泡排序

下标  
第一趟排序  
后

0	1	2	3	4
7	8	6	5	9

第一趟排序后，数列中的最大值 9 被排在了数列的最尾端，则数列被分为两部分，一部分是无序的数列：

7、8、5、6，另一部分是有序数列 9，接下来的第二、三、...趟排序就是将无序数列中的最大值排到数列最尾端，每进行一趟排序无序数列中的数据就会减少一个，有序数列中的数据就会增加一个。

若有 **n** 个数需要排序，则经过 **n-1** 趟后，该数列变为有序数列。

## 插入排序

补充：

- 1、将一个整型数组名做为参数时，可以不传递数组长度而在自定义函数内部通过 **sizeof** 计算来数组长度吗，若自定义函数处理的是一个存储字符串的字符数组，可以不传递数组长度吗？

```
char *p = "hello"; char str[] = "hello";  
void *p = malloc(100);
```



# 插入排序

补充：

**2、调试输出：使用函数宏代替单纯的 `printf`**

**3、模块的测试方法**



## 选择排序：

**算法描述：**对一组数据进行选择排序时总是寻找数列中最小值的下标，找到最小值下标后和待排序数列的实际最小下标进行比较，如两者不相同，说明不满足最小下标处存小值的要求，则进行数据交换，这样完成一趟排序，总是将待排序数列中的最小值放到最小下标处。

下标

0

1

2

3

4

内容

8

7

9

6

5

先记录待排序数列的实际最小下标：**0**，然后寻找下标**1-4**的位置上是否出现过待排序数列中的真正最小值，若出现（下标**4**上的数值**5**），记录最小值下标，遍历完数据后，发现最小值下标**4**和待排序数列实际最小下标**0**不相同，则将**0**下标上的内容**8**和下标**4**上的内容**5**交换，这样待排序数列中的最小值被存放在了数列最前端，完成了一趟排序。

若有 **n** 个数需要排序，则经过 **n-1** 趟后，该数列变为有序数列。



## Section 2

---

# 插入排序

## 插入排序

### 插入排序：

算法描述：将待插入数据和已经排列好的数据由右向左依次比较，若待插入数据比某一数据小，说明待插入数据应该被放到该数据的前端，反复比较直到在有序数列中找到一个比待插入数据小的数或有序数列中所有数据被比较完毕，则待插入数据的存放位置被确定下来。

# 插入排序



## 插入排序

插入排序：

向数组中插入数据的操作不能完全等价于扑克牌的操作，因为数组元素的存放位置是连续的，若想向两个元素进行插入新的数据，则需要将后一个元素及其后续各元素往后挪移一个位置。

# 插入排序

插入排序：

若有如下数据，将要向该数列中插入数据 **12**

下标	0	1	2	3	4
内容	5	8	14	16	

下标	0	1	2	3	4
内容	5	8	<b>12</b>	<b>14</b>	<b>16</b>



## 插入排序

练习：

随意输入 **5** 个整数并存入数组，完成对这 **5** 个整数的升序排列，然后在输入一个数，插入到此有序数列中来，要求插入后数组仍有序。请对以下三种情况完成程序测试

- 1**、最前插入数据
- 2**、在中间插入数据
- 3**、在最后插入数据

## 插入排序

插入排序：若有如下数据，**5**、**4**、**2**、**3**、**1** 需要通过插入排序完成升序排列

第一步：将 **4** 插入到已排好序的数列中：**4** 和 **5** 比较，发现 **4** 小于 **5**，并且 **5** 已经是数列最前端的数，则 **4** 应该被放在 **0** 号下标处，**5** 应该向后移动一个位置

下标	0	1	2	3	4
内容	<b>5</b>	<b>5</b>	2	3	1

# 插入排序

下标

0	1	2	3	4
---	---	---	---	---

内容

2	8	5	3	1
---	---	---	---	---

下标

0	1	2	3	4
---	---	---	---	---

内容

2	4	8	5	1
---	---	---	---	---

下标

0	1	2	3	4
---	---	---	---	---

内容

2	3	4	8	5
---	---	---	---	---

## 插入排序

算法： $a_0, a_1, \dots, a_{n-1}$  排序，也即将  $a_i (1 \leq i \leq n-1)$  插入到数列中  $a_i$  和之前的数据  $a_j (j = i-1, \dots, 0)$  顺序比较 若发现比  $a_i$  小的数字或数列已经到了尽头，则停止比较， $a_i$  就应该插入到  $a_j$  之后

# 插入排序

练习：

在前一个练习的基础上实现插入排序。

## 插入排序

插入排序算法的时间复杂度：

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

时间复杂度： **$O(x)$**

- 1、对算法执行时间的描述（估算值）**
- 2、描述的时算法执行的上限**
- 3、只是估算执行时间的量级**



## Section 3

---



## 递归

**1、如果定义一个概念时有用到了概念本身，则我们称这个概念是递归定义的。**

**如：数列的阶乘的定义：**

**$n!$  定义为： $n! = n * (n - 1)!$**





如：数的阶乘的定义：

**$n!$  定义为： $n! = n * (n - 1)!$**

$$(n - 1)! = (n - 1) * (n - 2)!$$

....

$$1! = 1 * 0!$$

$$0! = 1$$



```
void recursion(void)
```

---

```
{  
    static int n = 0;  
    if(n == 3)  
        return;  
    n++;  
    printf("%d: calling recursion...\n",  
n);  
    recursion();  
    printf("%d return...\n", n);  
}
```

第一次

n = 0	
n++	
printf ( “calling\n” )	
调用 recursion	将printf ( “return\n” ) 语句地址压栈
	第二次进入 recursion执行

第二次


n++	
printf ( “calling\n” )	
调用 recursion	将printf ( “return\n” ) 语句地址压栈
	第三次进入 recursion执行

第三次

n++	
printf ( “calling\n” )	
调用 recursion	将printf ( “return\n” ) 语句地址压栈
	第四次进入 recursion执行

第四次

测试发现n == 3	
执行return语句返回，回到第三次调用的 recursion() 语句之后，则 执行 printf( “return\n” )	



数的阶乘的定义中又用到了阶乘概念本身，如果阶乘的求解就一直这么递归下去，将会是一个无解，正是因为定义了 **0!** 为 **1**，才使得 **n!** 能够得到正确结论，因此在递归问题中必然会有一个基础条件 - 能让递归结束的条件。

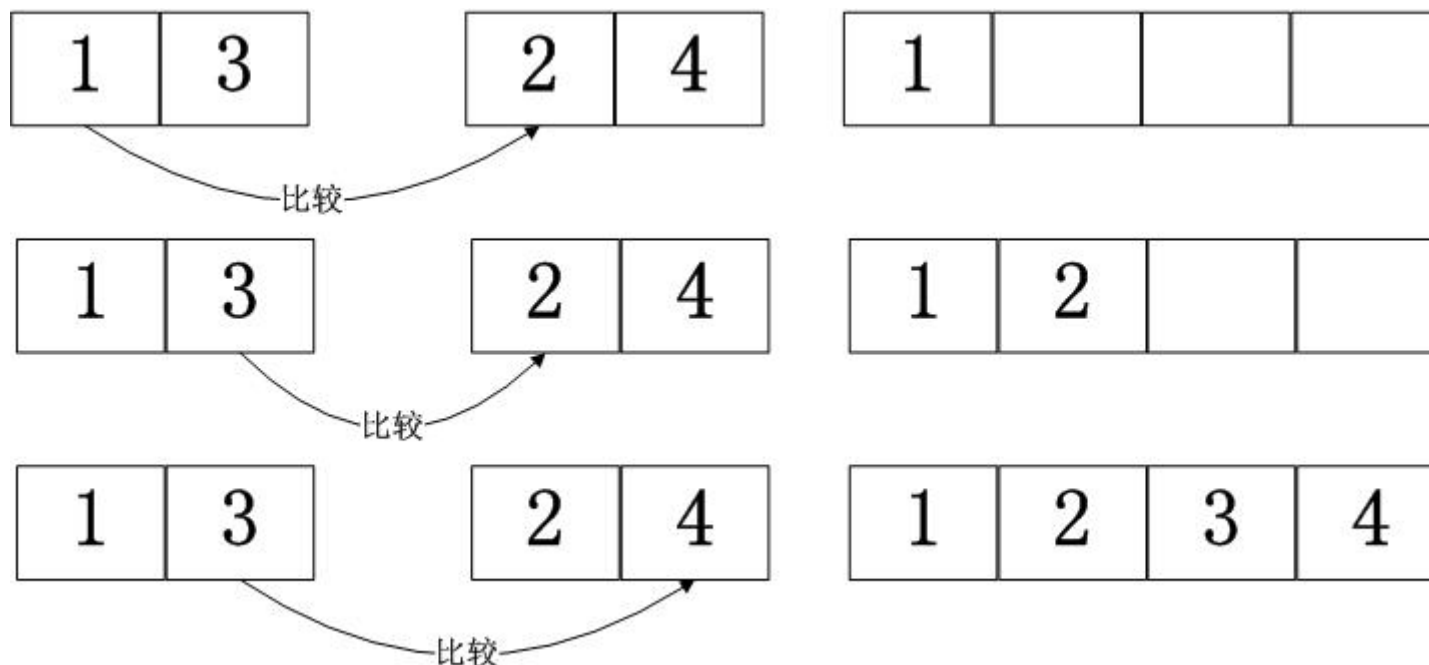


---

## 归并算法：

- 1、把长度为  $n$  的输入序列分成两个长度为  $n/2$  的子序列。**
- 2、对这两个子序列分别采用归并排序。**
- 3、将两个排序好的子序列合并成一个最终的排序序列（有序归并）。**

## 有序归并：





有两个字符串（有效字符不超过 **9** 个）

**str1, str2**，将其对应字符按从小到

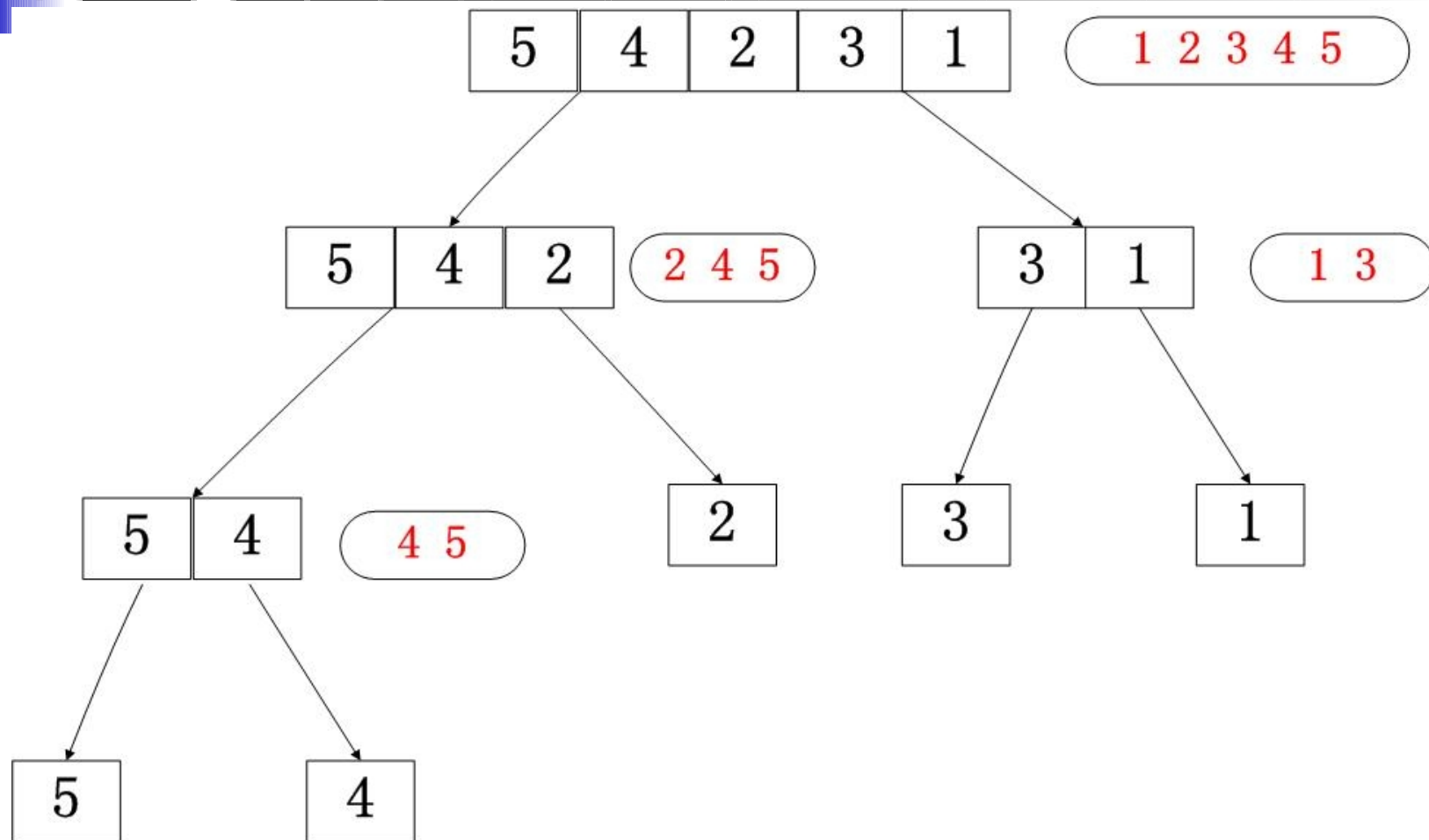
大的顺序存放到一个数组的对应位置

上。如：**char \*str1 = "hello";**

**char \*str2 = "akaedu";** 完成组

合后，新生成的字符串

为：**"ahekaedllou"**



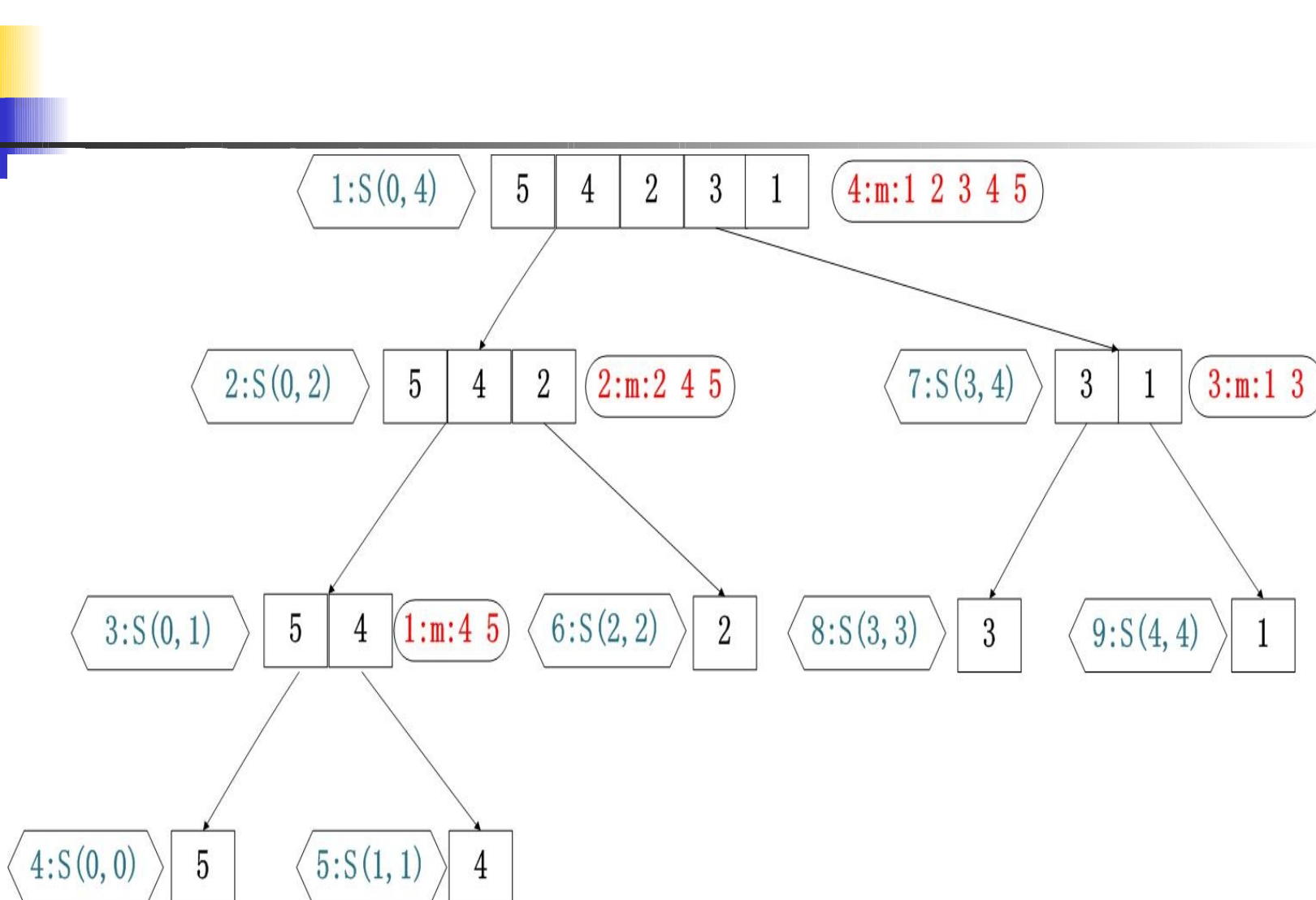




---

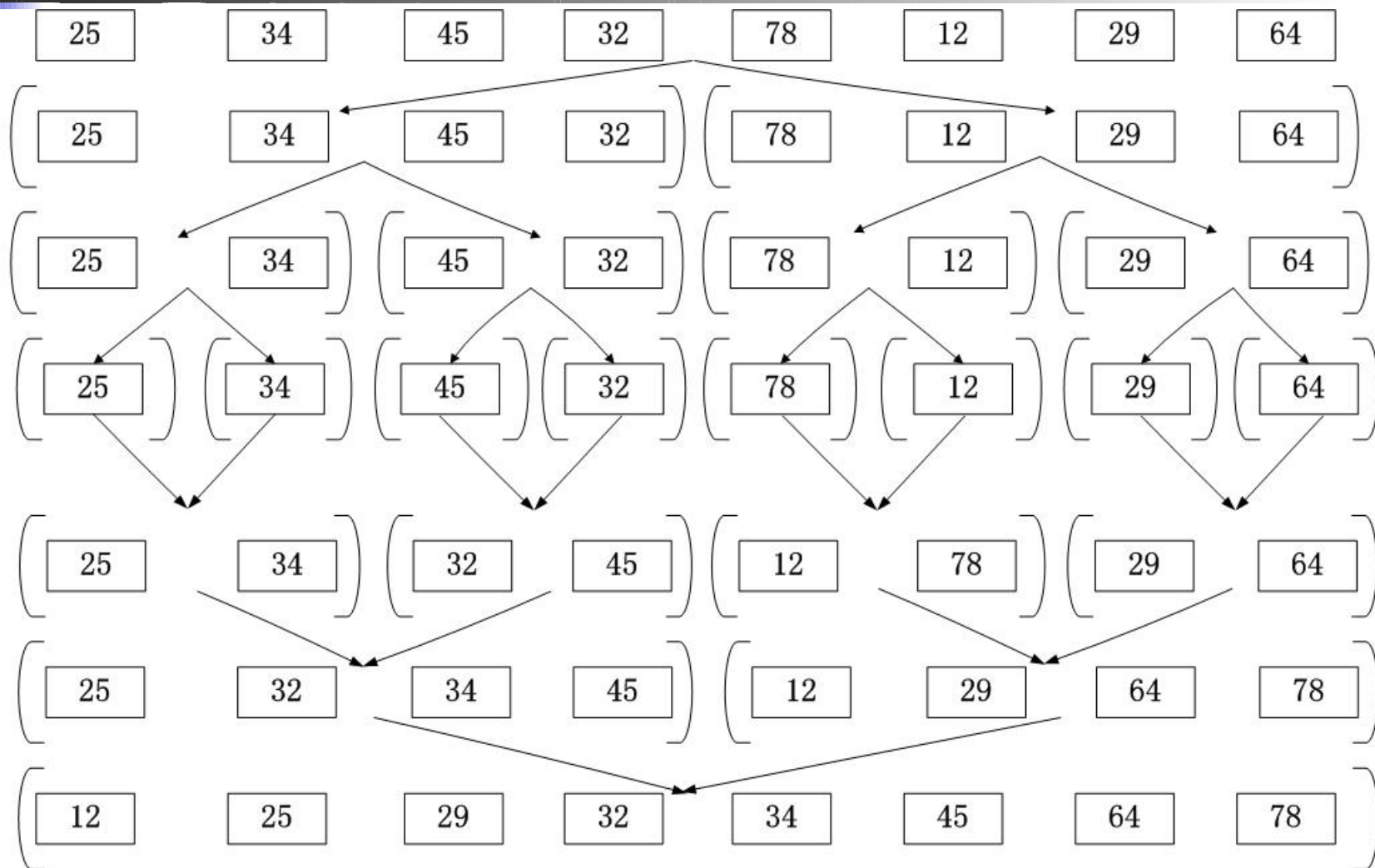
## 归并算法：

递归是不断将任务下推的过程，当任务不能再细分的时候递归达到上限，在回溯的过程中是真正完成任务的时候。



先  
划  
分

再  
归  
并



## 插入排序

归并排序算法的时间复杂度：

**merge** 函数时间复杂度： $O(n)$

归并排序的时间复杂度： $O(n * \lg n)$

$$T(n) = T(n/2) + T(n / 2) + cn$$

$$T(n / 2) = 2 * T(n/4) + cn/2$$

## 插入排序

归并排序算法的空间复杂度：在排序过程中，除去原始数据占用的空间之外，排序算法使用的额外空间

插入排序 ---- 空间复杂度  **$O(1)$**

归并排序 ---- 空间复杂度  **$O(n)$**



## 练习：递归实现

**1、求解  $1+2+\dots+n$  的和**

**2、求 Fibonacci 数列的第  $n$  项。**

**$\text{fib}(0) = 1$**


**$\text{fib}(1) = 1$**

**$\text{fib}(n) = \text{fib}(n - 1) + \text{fib}(n - 2)$**



## Section 4

---



**1、**把长度为 **n** 的输入序列分成两部分，分割条件是先从数列中找一个哨兵（通常可以将数列中的首元素作为哨兵），要求数列前半部分的值全部比哨兵值小，后半部分全部比哨兵值大。

**2、**将分割完的两部分分别进行快速排序。





## Section 5

---



---

线性查找：时间复杂度  $O(n)$

访问数列中的每个值，若某个值与被查找值相等则说明该数存在于数列中，  
否则说明在数列中找不到被查找值。



## Section 6

---

# 折半查找

# 折半查找

折半查找：

前提：数列有序

算法：将数列中间位置上的值与被查找数相比较，若相等则说明找到，若不相等则缩小查找范围，直到查找完数列。

## 折半查找

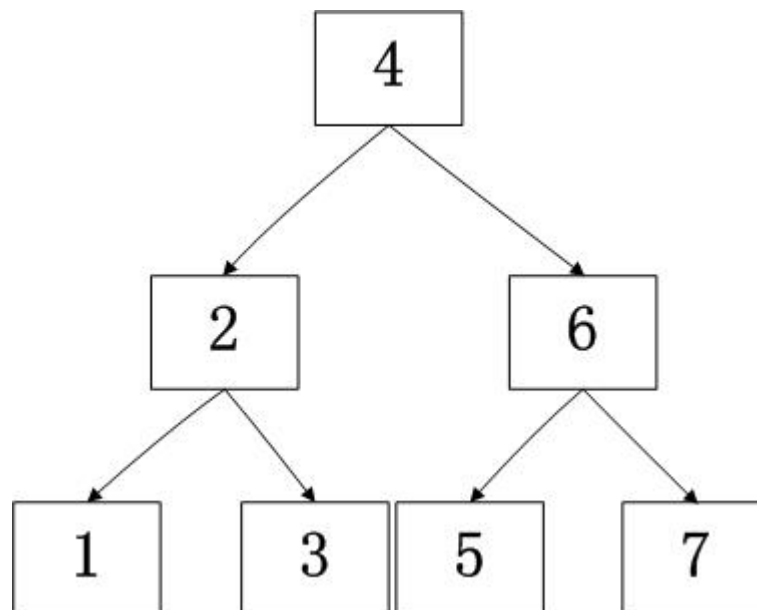
算法：**1、确定中点元素**

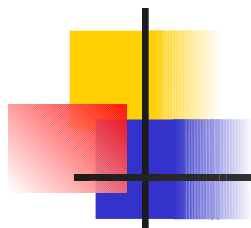
**2、和中点元素比较，若找到则结束，否则重新确定查询序列，对新序列按照折半查找算法重新查找**

**3、如果数据序列长度小于一，结束**

# 折半查找

折半查找：时间复杂度  $O(\lg n)$





# Let's DO it!

---

# Thanks for listening!

