

Topic4 查找/排序

程序设计专题

学习目标

- 了解算法效率的度量方法和大O记法
- 掌握简单的线性查找法和二分查找法
- 掌握简单的选择排序法和冒泡排序法
- 了解分而治之策略，基本掌握归并排序法

一、算法效率的度量

■ 算法设计的要求

- 正确性：算法至少应具有输入/出和加工处理无歧义性、能正确反映问题的需求、能够得到问题的正确答案。
- 可读性：便于阅读、理解和交流
- 健壮性：当输入数据不合法时，算法能做出相关处理，而非产生异常或莫名其妙的结果
- 时间效率高和存储量低

■ 算法效率的度量方法

- 事后统计方法（有缺点，较少使用）
- 事前分析估算方法

一、算法效率

■ 算法时间复杂度

➤ 与高级语言程序执行时间相关的因素

- 算法采用的策略、方法 << 算法好坏的根本
- 编译产生的代码质量 << 软件
- 问题的输入规模 << 输入量的多少
- 机器执行指令的速度 << 硬件

```
int i, sum = 0, n = 100;          /* 执行1次 */
for (i = 1; i <= n; i++)          /* 执行了n+1次 */
{
    sum = sum + i;                /* 执行n次 */
}
printf("%d", sum);                /* 执行1次 */
```

执行 $2n+3$ 次

sum = 1 + 2 + 3 + ... + 99 + 100
sum = 100 + 99 + 98 + ... + 2 + 1
2×sum = 101 + 101 + 101 + ... + 101 + 101
共 100 个

```
int sum = 0, n = 100;              /* 执行一次 */
sum = (1 + n) * n / 2;             /* 执行一次 */
printf("%d", sum);                 /* 执行一次 */
```

执行 3 次

一、算法效率

■ 算法时间复杂度

```
int i, sum = 0, n = 100;          /* 执行1次 */
for (i = 1; i <= n; i++)          /* 执行了n+1次 */
{
    sum = sum + i;                /* 执行n次 */
}
printf("%d", sum);                /* 执行1次 */
```

sum = 1 + 2 + 3 + ... + 99 + 100
sum = 100 + 99 + 98 + ... + 2 + 1
2×sum = 101 + 101 + 101 + ... + 101 + 101
共100个

```
int sum = 0, n = 100;              /* 执行一次 */
sum = (1 + n) * n / 2;             /* 执行一次 */
printf("%d", sum);                /* 执行一次 */
```

```
int i, j, x = 0, sum = 0, n = 100; /* 执行一次 */
for (i = 1; i <= n; i++)
{
    for (j = 1; j <= n; j++)
    {
        x++;                      f(n) = n² /* 执行n×n次 */
        sum = sum + x;
    }
}
printf("%d", sum);                /* 执行一次 */
```

- 执行次数对同样的输入规模，多于前两个算法；随着n的增加执行次数也将远远多于前面两个
- 测定运行时间最可靠的方法就是计算对运行时间有消耗的基本操作的执行次数。运行时间与这个计数成正比。

- 分析一个算法的运行时间时，重要的是把基本操作的数量与输入规模关联起来

一、算法效率

■ 算法时间复杂度

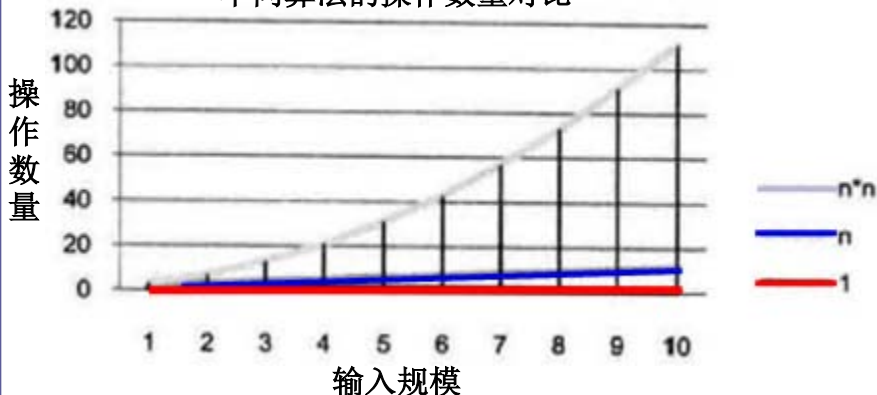
```
int i, sum = 0, n = 100;          /* 执行1次 */
for (i = 1; i <= n; i++)          /* 执行了n+1次 */
{
    sum = sum + i;                f(n) = n /* 执行n次 */
}
printf("%d", sum);                /* 执行1次 */
```

sum = 1 + 2 + 3 + ... + 99 + 100
sum = 100 + 99 + 98 + ... + 2 + 1
2×sum = 101 + 101 + 101 + ... + 101 + 101
共 100 个

```
int sum = 0, n = 100;              /* 执行一次 */
sum = (1 + n) * n / 2;             f(n) = 1 /* 执行一次 */
printf("%d", sum);                /* 执行一次 */
```

```
int i, j, x = 0, sum = 0, n = 100; /* 执行一次 */
for (i = 1; i <= n; i++)
{
    for (j = 1; j <= n; j++)
    {
        x++;                      f(n) = n2 /* 执行n×n次 */
        sum = sum + x;
    }
}
printf("%d", sum);                /* 执行一次 */
```

不同算法的操作数量对比



- 分析一个算法的运行时间时，重要的是把基本操作的数量与输入规模关联起来

一、算法效率

■ 算法时间复杂度

➤ 算法时间复杂度定义

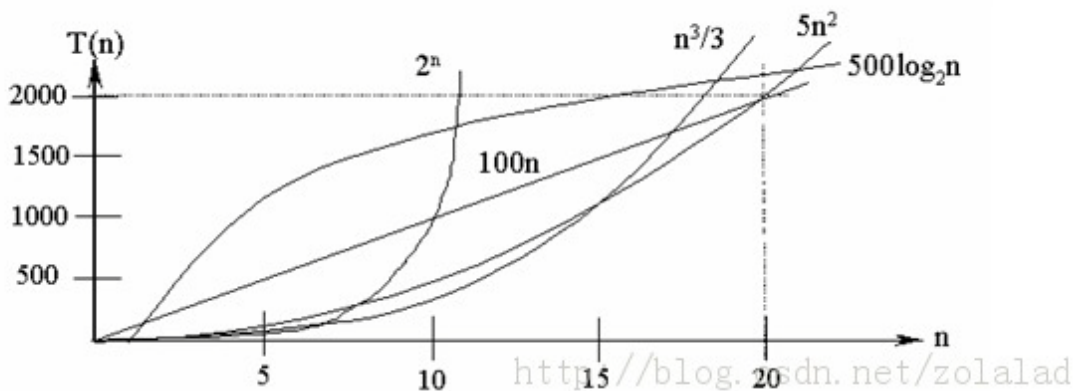
- 进行算法分析时，一般从算法中选取一种对所研究问题是基本/主要的原操作（多数情况下取自最深层次循环体内的语句），以该基本操作在算法中重复执行的次数 $T(n)$ 作为算法运行时间的衡量准则。
- 算法的渐进时间复杂度简称算法的时间复杂度，就是算法的时间度量，记作 $T(n)=O(f(n))$ （即大O记法）。表示随问题规模 n 的增大，算法执行时间的增长率和 $f(n)$ 增长率相同（或同数量级的）。
- $T(n)=O(f(n))$ 表示存在一个常数 C ，使得在当 n 趋于正无穷时总有 $T(n) \leq C * f(n)$ 。简单说就是 $T(n)$ 在 n 趋于正无穷时最大也就跟 $f(n)$ 差不多大。也就是说当 n 趋于正无穷时 $T(n)$ 的上界是 $C * f(n)$ 。
- 对 $f(n)$ 没有规定，但 $f(n)$ 一般都是取尽可能简单的函数。例如，
$$O(2n^2+n+1) = O(3n^2+n+3) = O(7n^2+n) = O(n^2)$$

一、算法效率

■ 算法时间复杂度

➤ 算法时间复杂度定义

按数量级递增排列，常见的时间复杂度有：常数阶 $O(1)$ ，对数阶 $O(\log_2 n)$ ，线性阶 $O(n)$ ，线性对数阶 $O(n\log_2 n)$ ，平方阶 $O(n^2)$ ，立方阶 $O(n^3)$ ，...， k 次方阶 $O(n^k)$ ，指数阶 $O(2^n)$ ，阶乘阶 $O(n!)$ 。随着问题规模 n 的不断增大，上述时间复杂度不断增大，算法的执行效率越低



一、算法效率

■ 算法时间复杂度

➤ 算法时间复杂度实例

$T(n)=O(1)$

```
Temp=i; i=j; j=temp;
```

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

```
i=1;  
while (i<=n)    i=i*2;
```

$T(n)=O(n)$

```
a=0;  
b=1;  
for (i=1;i<=n;i++)  
{  
    s=a+b;  
    b=a;  
    a=s;  
}
```

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

```
for (i=1;i<=n;i++)  
{  
    y=y+1;  
    for (j=0;j<=(2*n);j++)  
        x++;  
}
```

一、算法效率

■ 算法时间复杂度

➤ 我们要确定能反映出算法在各种情况下工作的数据集，选取的数据要能够反映、代表各种计算情况下的估算，包括最好情况下的时间复杂度(时间复杂度下界，一般记为 T_{\max})、最坏情况下的时间复杂度(时间复杂度上界，一般记为 T_{\min})、平均情况下的时间复杂度(一般记为 T_{avg})和有代表性的情况，通过使用这些数据配置来运行算法，以了解算法的性能。

➤ 除非特别指定，提到的都是最坏情况时间复杂度

一、算法效率

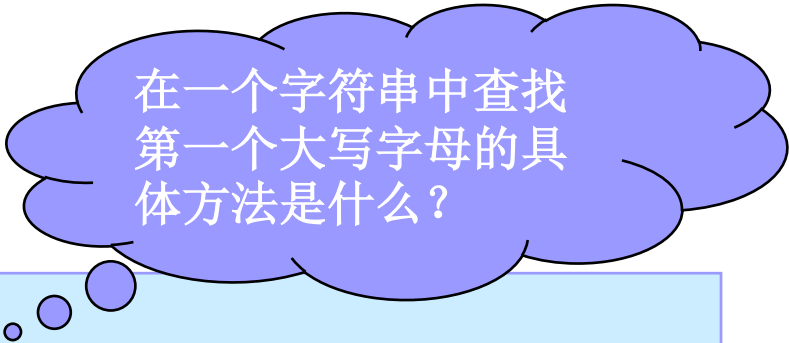
■ 算法空间复杂度

➤ 算法的空间复杂度 $S(n)$ 定义为该算法所耗费的存储空间，它也是问题规模 n 的函数。渐近空间复杂度也常常简称为空间复杂度。

➤ 一个算法在计算机存储器上所占用的存储空间，包括存储算法本身所占用的存储空间，算法的输入输出数据所占用的存储空间和算法在运行过程中临时占用的存储空间这三个方面。算法的空间复杂度 $S(n)$ 算法是对一个算法在运行过程中临时占用存储空间大小的量度。

二、查找算法

查找/检索：在一组数据中找出**满足某种条件的数据**的过程



在一个字符串中查找
第一个大写字母的具
体方法是什么？

```
int FindFirstUpperLetter( char * string )  
{  
    int i;  
  
    for ( i = 0; i < strlen(string); i++ )  
        if ( isUpper(string[i]) ) return i;  
  
    return -1;  
}
```

二、查找算法

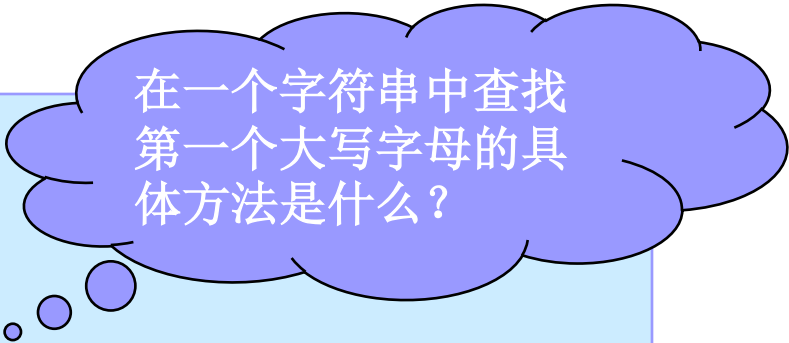
查找/检索：在一组数据中找出**满足某种条件的数据**的过程

■ 线性查找法

```
int FindFirstUpperLetter( char * string )
{
    int i;

    for ( i = 0; i < strlen(string); i++ )
        if ( isUpper(string[i]) ) return i;

    return -1;
}
```



在一个字符串中查找
第一个大写字母的具
体方法是什么？

二、查找算法

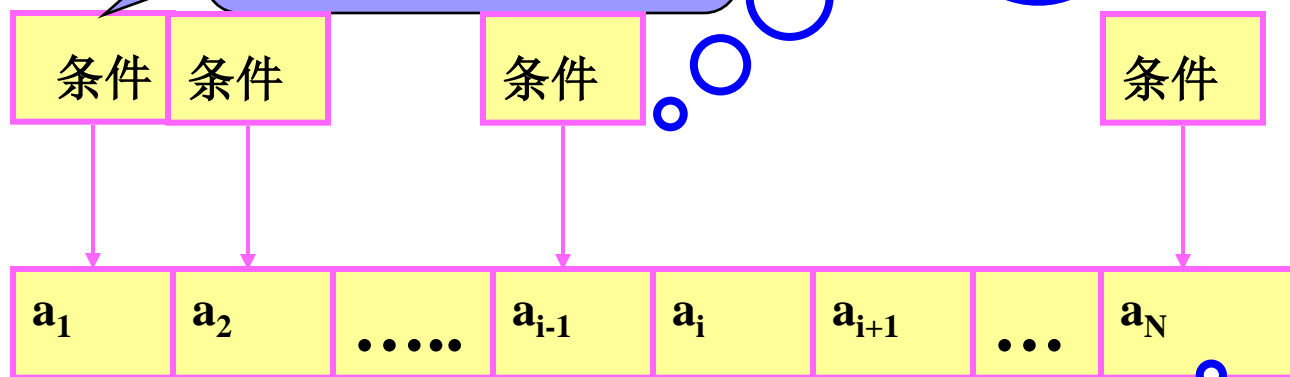
查找/检索：在一组数据中找出满足

如果 x 和 a_{i-1} 相同，
则找到并停止查
找；否则按照前
面的步骤继续下
去。

程

■ 线性查找法

从第一个元素起逐个
元素进行比较



如果此时仍然没有
找到，返回未找到
信息并停止

二、查找算法

查找/检索：在一组数据中找出**满足某种条件的数据**的过程

■ 线性查找法

```
/* Example: linear search */
```

```
/* 要求编写程序来显示美国的硬币名称和它对应的值*/
```

✓ 需定义一个函数 `int FindIntegerInArray (int key, int array[], int n)` 用于在一个有效长度为 `n` 的整数数组中查找整数 `Key`。

```
static int FindIntegerInArray (int key, int array[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        if (key == array[i]) return (i);
    return (-1);
}
```

二、查找算法

查找/检索：在一组数据中找出**满足某种条件的数据**的过程

■ 线性查找法

```
/* Example: linear search */
```

```
/* 要求编写程序来显示美国的硬币名称和它对应的值*/
```

✓ 需定义一个函数 `int FindIntegerInArray (int key, int array[], int n)` 用于在一个有效长度为 `n` 的整数数组中查找整数 `Key`。

✓ 若美国的标准硬币有五种，则可用两个数组分别表示币值和币名。

	coinValues		coinNames
0	1	0	penny
1	5	1	nickel
2	10	2	dime
3	25	3	quarter
4	50	4	half-dollar

/* Example: linear search */

/* 要求编写程序来显示美国的硬币名称和它对应的值*/

```
#include <stdio.h>
```

```
static char* coinNames[ ] = {"penny", "nickel", "dime", "quarter", "half-dollar"};
```

```
static int coinVlaues[ ] = { 1, 5, 10, 25, 50};
```

```
static int nCoins = sizeof coinValues / sizeof coinValues[0];
```

```
static int FindIntegerInArray (int key, int array[ ], int n) ;
```

```
/* Main program */
```

```
main()
```

```
{
```

```
    int value, index;
```

```
    printf ("This program looks up names of U.S. coins.\n");
```

```
    printf ("Enter coin value: ");
```

```
    scanf ("%d",&value);
```

```
    index = FindIntegerInArray (value, coinVlaues, nCoins);
```

```
    if (index == -1) {
```

```
        printf ("There is no such coin.\n");
```

```
    } else {
```

```
        printf ("That's called a %s.\n", coinNames[index]);
```

```
    }
```

```
}
```

二、查找算法

查找/检索：在一组数据中找出**满足某种条件的数据**的过程

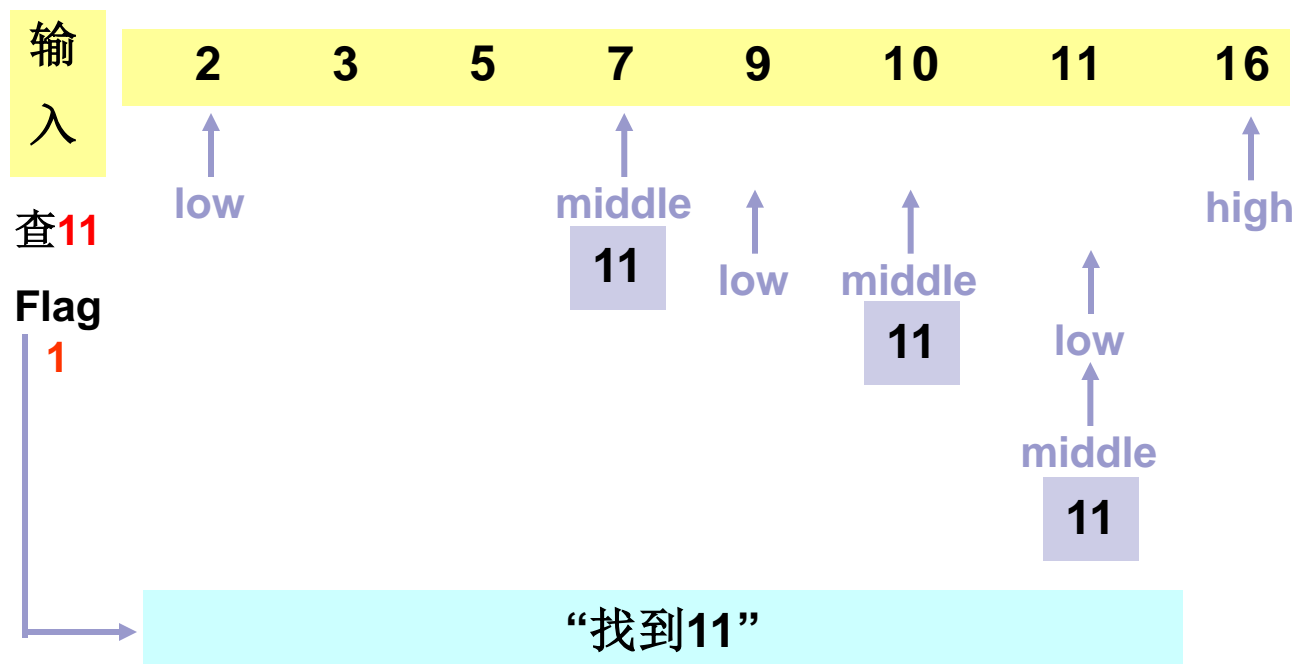
- 线性查找法
- 二分查找法

二、查找算法

查找/检索：在一组数据中找出**满足某种条件的数据**的过程

■ 二分查找法

线性查找要逐个比较表中元素。当表中元素非常多时，顺序查找的效率是很低的。二分查找在速度方面有所改进，但是**只适用于已排好序**的数组。

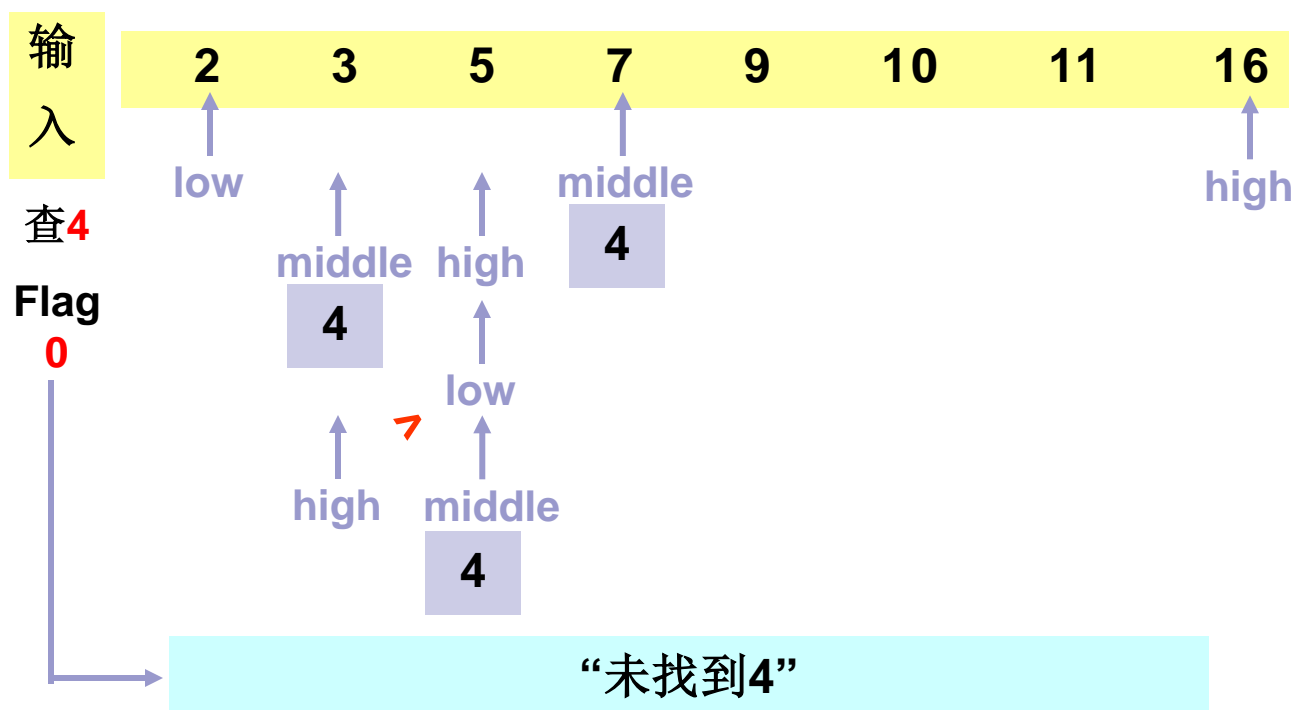


二、查找算法

查找/检索：在一组数据中找出**满足某种条件的数据**的过程

■ 二分查找法

线性查找要逐个比较表中元素。当表中元素非常多时，顺序查找的效率是很低的。二分查找在速度方面有所改进，但是**只适用于已排好序**的数组。



/* Example: **binary search** */

/* 要求在一组整数中查找一个值 */

```
# include <stdio.h>
```

```
# define N 8
```

```
main()
```

```
{
```

```
    int i, a[N], searchKey, low, high, middle, flag;
```

```
    for (i=0; i<N; i++) a[i]=2*i;    /*create data*/
```

```
    printf("Enter interger search key:\n");
```

```
    scanf("%d", &searchKey);
```

```
    low=0;    high=N-1;    flag=0;
```

```
    while (low<=high) {
```

```
        middle=(low+high)/2;
```

```
        if (searchKey == a[middle] )
```

```
        {    flag=1; printf("Found value in a[%d]\n", middle); break;    }
```

```
        else if (searchKey <a[middle] )
```

```
            high=middle-1;    /* lower half */
```

```
        else
```

```
            low=middle+1;    /* upper half */
```

```
    }
```

```
    if (flag==0) printf("Value not found\n");
```

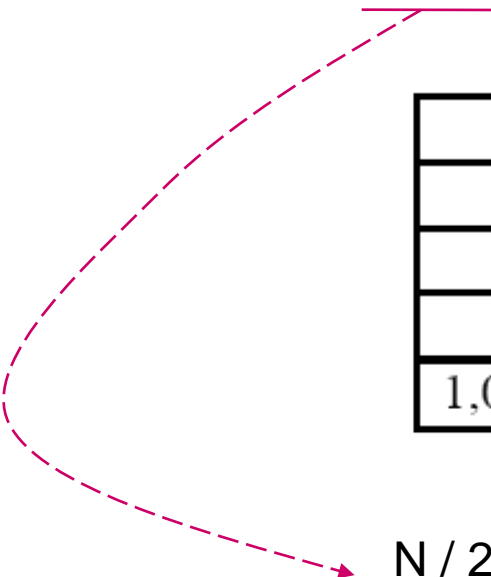
```
}
```

二、查找算法

查找/检索：在一组数据中找出**满足某种条件的数据**的过程

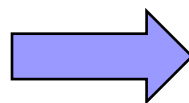
■ 查找算法的效率

在最坏情况下，线性查找法查找N个元素的数组需要**N**次比较，而二分查找法需要 **$\log_2 N$** 次比较



N	$\log_2 N$
10	3
100	7
1000	10
1,000,000	20
1,000,000,000	30

$$N / \underbrace{2 / 2 / \dots / 2 / 2}_{K\text{次}} = 1$$


$$N = 2^K$$

三、排序算法

排序：重排一组数据使它们按某一事先确定的顺序存储的过程

- 选择排序法
- 冒泡排序法
- 归并排序法

三、排序算法

排序：重排一组数据使它们按某一事先确定的顺序存储的过程

■ 选择排序法

➤ 对一组数据，每次将其中的一个数据放在它最终要放的位置。第一步是找到整个数据中最小的数据并把它放在最终要放的第一个位置上，第二步是在剩下的数据中找最小的数据并把它放在第二个位置上。对所有数据重复这个过程，最终将得到按从小到大顺序排列的一组数据。

三、排序算法

排序：重排一组数据使它们按某一事先确定的顺序存储的过程

■ 选择排序法

2 6 1 8 7 4 5	第1次 选择	在a[0]-a[6]中找最小值a[k], 比较后确定k为0; 交换a[0]~a[k]与的值, 结果: <u>1</u> 6 2 8 7 4 5
	第2次 选择	在a[1]-a[6]中找最小值a[k], 比较后确定k为2; 交换a[1]~a[k]与的值, 结果: <u>1</u> <u>2</u> 6 8 7 4 5
	第3次 选择	在a[2]-a[6]中找最小值a[k], 比较后确定k为5; 交换a[2]~a[k]与的值, 结果: <u>1</u> <u>2</u> <u>4</u> 8 7 6 5
	第4次 选择	在a[3]-a[6]中找最小值a[k], 比较后确定k为6; 交换a[3]~a[k]与的值, 结果: <u>1</u> <u>2</u> <u>4</u> <u>5</u> 7 6 8
	第5次 选择	在a[4]-a[6]中找最小值a[k], 比较后确定k为5; 交换a[4]~a[k]与的值, 结果: <u>1</u> <u>2</u> <u>4</u> <u>5</u> <u>6</u> 7 8
	第6次 选择	在a[5]-a[6]中找最小值a[k], 比较后确定k为5; 交换a[5]~a[k]与的值, 结果: <u>1</u> <u>2</u> <u>4</u> <u>5</u> <u>6</u> <u>7</u> 8

三、排序算法

排序：重排一组数据使它们按某一事先确定的顺序存储的过程

■ 选择排序法

```
/* Example: selection sort */
```

```
/* 要求对一组整数进行由小到大的排序*/
```

```
#include <stdio.h>
void main()
{ int i, index, k, n, temp, a[10];
  scanf("%d", &n);
  for(i=0; i<n; i++)    scanf("%d", &a[i]);
  for(k=0; k<n-1; k++){
    index=k;
    for(i=k+1; i<n; i++)
      if(a[i]< a[index]) index=i;
    temp=a[index]; a[index]=a[k]; a[k]=temp;
  }
  for(i=0; i<n; i++)    printf("%d ", a[i]);
}
```

三、排序算法

排序：重排一组数据使它们按某一事先确定的顺序存储的过程

■ 选择排序：

算法效率：

N	Running time (unit: ms)
10	0.13
20	0.33
30	1.00
40	1.47
50	2.40
100	9.67
200	37.33
400	146.67
800	596.67

随着N的增大，执行时间显著增加

实验结论：数组的元素个数增加一倍，所需要的时间是原来的4倍。这个算法具有**平方律**的性质，即执行时间和输入数据的个数的平方成正比。

算法分析：对N个数据排序，需执行N-1次外层for循环：第一次找出N个数中最小值，下一次是在剩下的N-1个元素中找最小值，依次类推。程序执行的比较次数和数据个数成正比，需要执行的总次数为：

$$\sum_{i=1}^{N-1} (N-i) = \frac{1}{2} (N^2 - N)$$

三、排序算法

排序：重排一组数据使它们按某一事先确定的顺序存储的过程

■ 冒泡排序法

➤ 就是将数组中较小的数比喻为气泡，是一个小数上冒、大数下沉的过程。基本思想是通过相邻元素的比较和交换，使全部记录排列有序。

➤ 过程：含有 n 个元素的数组原则上要进行 $n-1$ 次排序。对于每一次排序，从第一个数开始，依次比较前一个数与后一个数的大小。如果前一个数比后一个数大，则进行交换。这样一轮过后，最大的数将会“沉到底”，成为最末位的元素。第二轮则去掉最后一个数，对前 $n-1$ 个数再按照上面的步骤找出最大数，该数将成为倒数第二位的元素…… $n-1$ 轮过后，就完成了排序。

排序：重排一组数据使它们按某一事先确定的顺序存储的过程

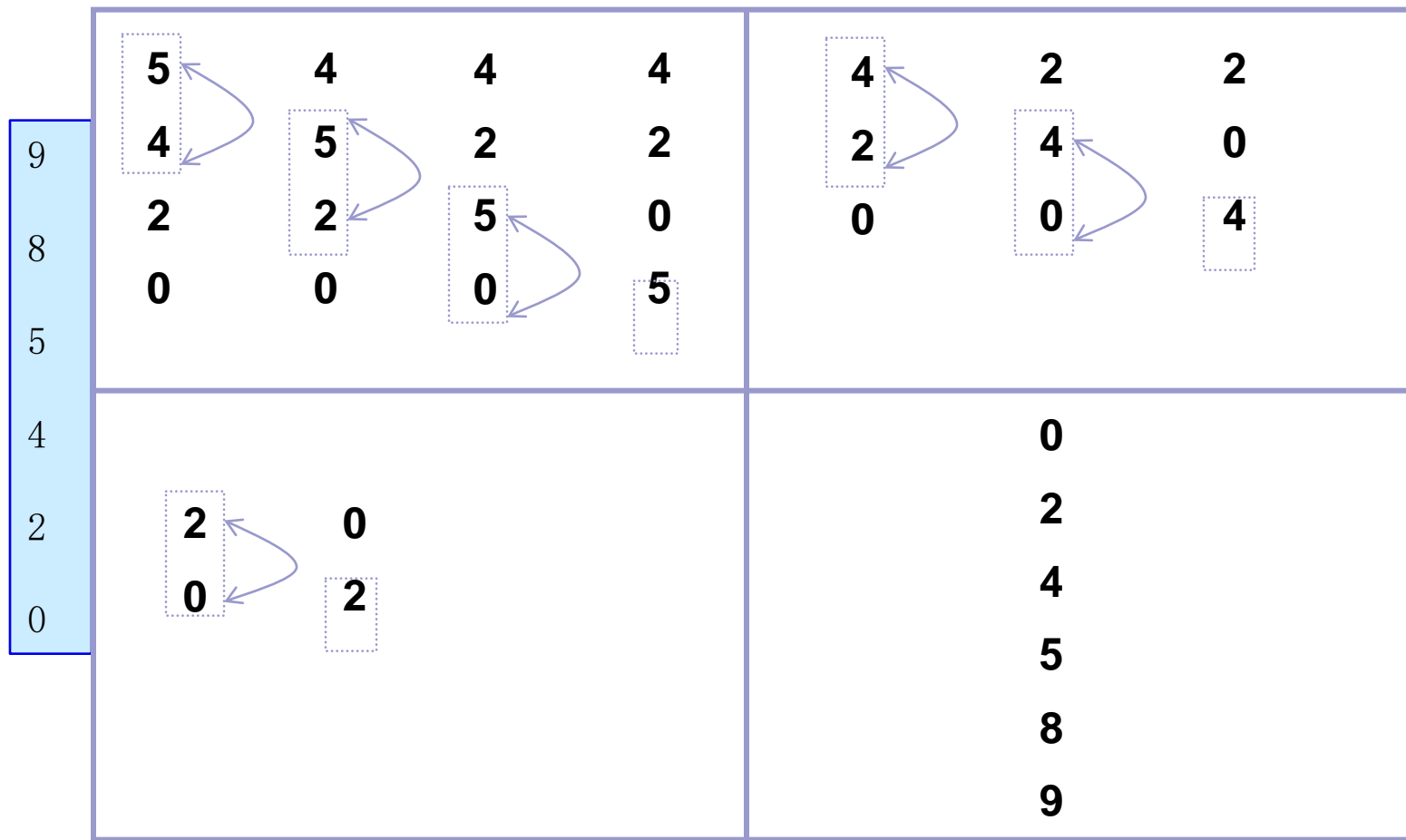
9	9	8	8	8	8	8
8	8	9	5	5	5	5
5	5	4	9	4	4	4
4	4	2	4	2	2	2
2	2	0	2	0	0	0
0	0	0	0	0	0	0

8	5	5	5	5
5	8	4	4	4
4	4	8	2	2
2	2	2	8	0
0	0	0	0	8

三、排序算法

排序：重排一组数据使它们按某一事先确定的顺序存储的过程

■ 冒泡排序法



三、排序算法

排序：重排一组数据使它们按某一事先确定的顺序存储的过程

■ 冒泡排序法

```
/* Example: Bubble sort */
```

```
/* 要求对一组整数进行由小到大的排序*/
```

```
# include <stdio.h>
# define N 6
void main()
{
    int a[N], temp;
    int i, j;
    for (i=0; i<N; i++) scanf("%d", &a[i]);
    printf("\n");
    for (i=0; i<N-1; i++)
        for (j=0; j<N-i-1; j++)
            if (a[j]>a[j+1])
            {
                temp=a[j];  a[j]=a[j+1];  a[j+1]=temp;
            }
    for (i=0; i<N; i++) printf("%d", a[i]);
}
```

最坏情况下总比较次数：

$$\sum_{i=1}^{N-1} (N-i) = \frac{1}{2} (N^2 - N)$$

三、排序算法

排序：重排一组数据使它们按某一事先确定的顺序存储的过程

■ 归并排序法

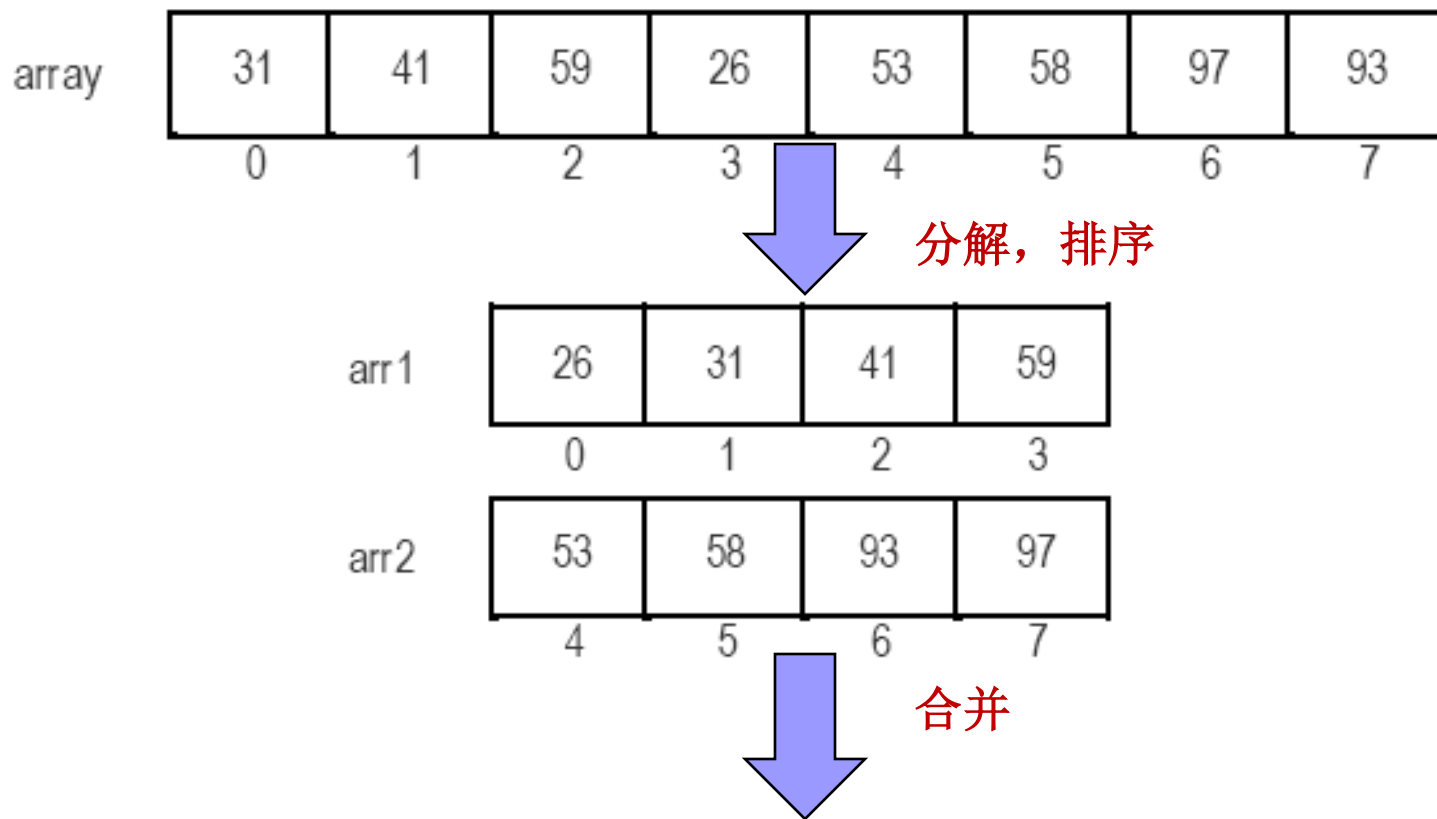
➤ 对于一个规模为 n 的问题，若该问题可以容易地解决（比如说规模 n 较小）则直接解决，否则将其分解为 k 个规模较小的子问题，这些子问题互相独立且与原问题形式相同，递归地解这些子问题，然后将各子问题的解合并得到原问题的解。这种算法设计策略叫做**分治法**。

➤ **归并排序**就是一个典型的分治法的例子，将一个数列分成两部分，分别排序，然后合并。

三、排序算法

排序：重排一组数据使它们按某一事先确定的顺序存储的过程

■ 归并排序法

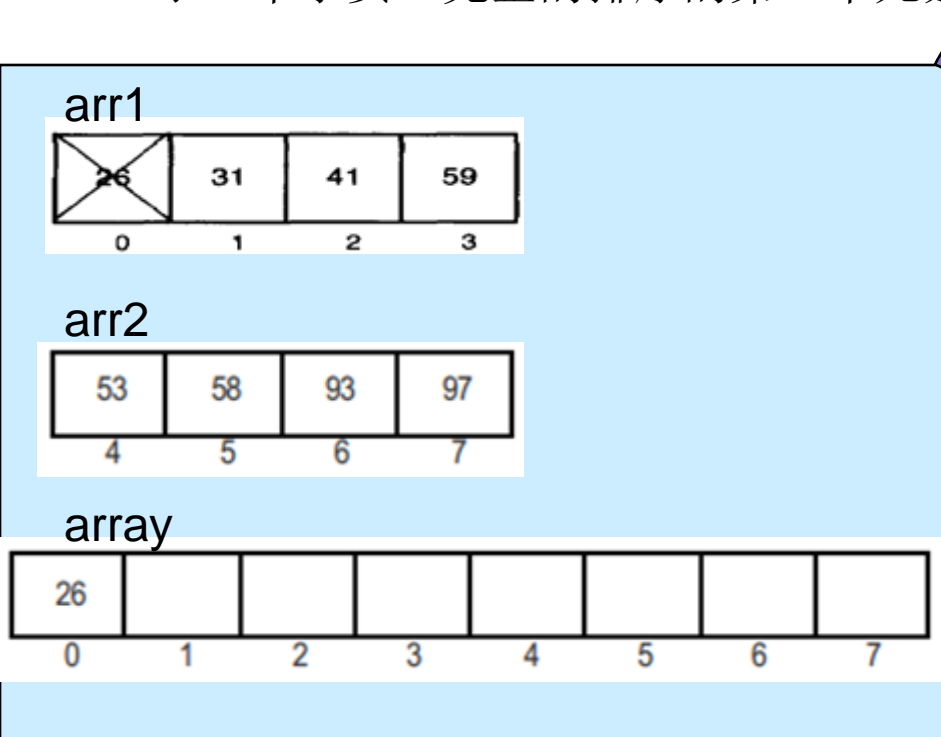


三、排序算法

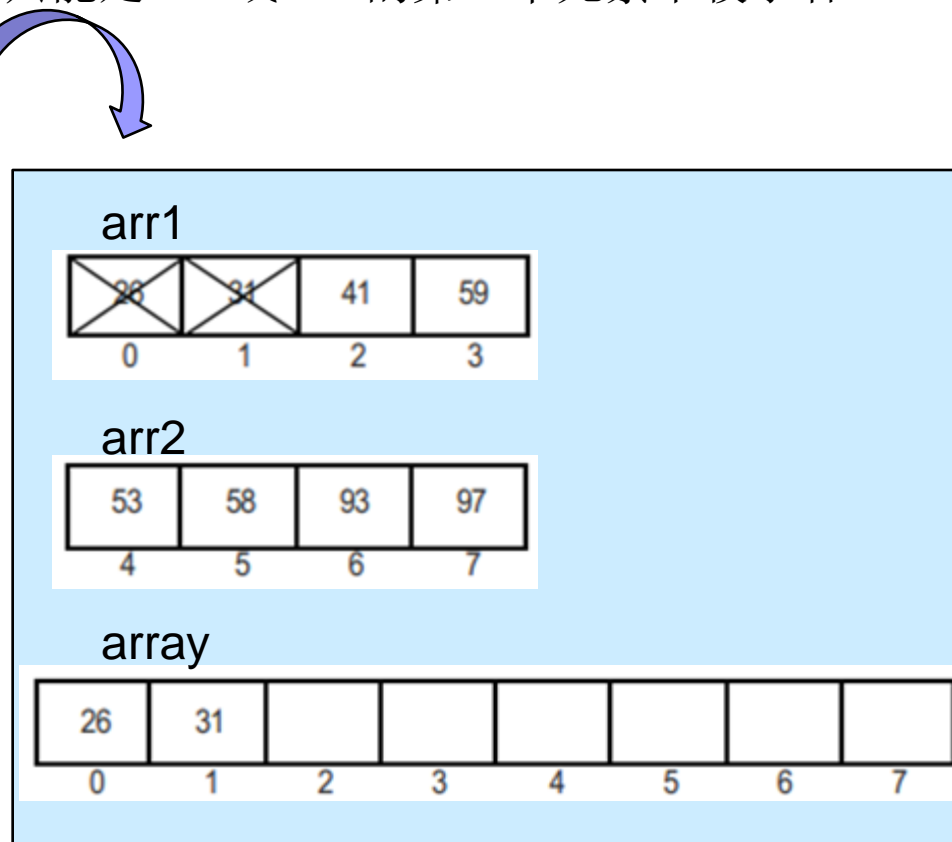
排序：重排一组数据使它们按某一事先确定的顺序存储的过程

■ 归并排序法

把已排序的子数组重组成一个完整数组比排序本身简单，这种技术叫**合并**。它基于一个事实：完整的排序的第一个元素只能是arr1或arr2的第一个元素中较小者。



可以继续**在arr1或arr2中选择较小值**的过程，直到整个数组被填满为止。



三、排序算法

排序：重排一组数据使它们按某一事先确定的顺序存储的过程

■ 归并排序法

➤ 合并操作与递归分解相结合，产生了新的排序算法——归并排序。

➤ 算法描述：首先判断数组大小，若无元素或只有一个元素，则数组必然已被排序；若包含的元素多于1个，则执行下列步骤：

step1: 把数组分为大小相等的两个子数组；

step2: 对每个子数组递归采用归并算法进行排序；

step3: 合并两个排序后的子数组。

三、排序算法

排序：重排一组数据使其有序

■ 归并排序法

```
void SortIntegerArray (int array[], int n)
{
    int i, n1, n2, *arr1, *arr2;
    if (n > 1) {
        n1 = n / 2;
        n2 = n - n1;
        arr1 = NewArray (n1, int);
        arr2 = NewArray (n2, int);
        for (i = 0; i < n1; i++) arr1[i] = array[i];
        for (i = 0; i < n2; i++) arr2[i] = array[n1 + i];
        SortIntegerArray (arr1, n1);
        SortIntegerArray (arr2, n2);
        Merge (array, arr1, n1, arr2, n2);
        FreeBlock (arr1);
        FreeBlock (arr2);
    }
}
```

```
void Merge (int array[], int arr1[], int n1, int arr2[], int n2)
{
    int p, p1, p2;
    p = p1 = p2 = 0;
    while (p1 < n1 && p2 < n2) {
        if (arr1[p1] < arr2[p2])
            array[p++] = arr1[p1++];
        else
            array[p++] = arr2[p2++];
    }
    while (p1 < n1) array[p++] = arr1[p1++];
    while (p2 < n2) array[p++] = arr2[p2++];
}
```

三、排序算法

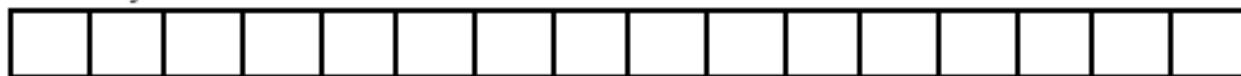
排序：重排一组数据使它们按某一事先确定的顺序存储的过程

■ 归并排序法

算法分析：运行时间可分为两个部分

- 1) 在当前的递归分解层次上，执行操作所需的所有时间
工作量与 N 成正比
- 2) 执行递归调用的时间

sorting an array of size N



N operations

requires sorting two arrays of size $N/2$



$2 \times N/2$ operations

which requires sorting four arrays of size $N/4$



$4 \times N/4$ operations

确定工作总量就转化为确定层数的问题， $M\log N$

四种排序方法性能比较

排序方法	最好时间	平均时间	最坏时间	辅助空间	稳定性
简单选择	$o(n^2)$	$o(n^2)$	$o(n^2)$	$o(1)$	不稳定
冒泡排序	$o(n)$	$o(n^2)$	$o(n^2)$	$o(1)$	稳定
直接插入	$o(n)$	$o(n^2)$	$o(n^2)$	$o(1)$	稳定
归并排序	$o(n \log_2 n)$	$o(n \log_2 n)$	$o(n \log_2 n)$	$o(n)$	稳定

N	N^2	$N \log N$
10	100	33
100	10,000	664
1000	1,000,000	9,965
10,000	100,000,000	132,877

N	<i>Selection sort</i>	<i>Merge sort</i>
10	0.00013	.00094
100	0.00967	.012
1000	1.08	.14
10,000	110.0	1.6