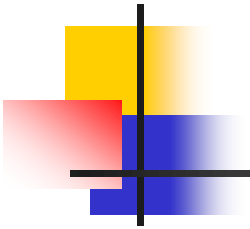


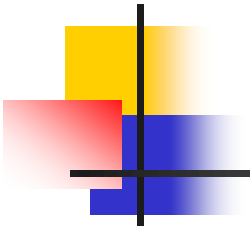
内部排序

- 基本概念
- 内部排序算法
  - 插入排序
    - 直接插入、折半插入、 希尔排序
  - 交换排序
    - 冒泡排序、快速排序
  - 选择排序
    - 简单选择排序、堆排序
  - 归并排序
    - 二路归并排序
  - 分配排序
    - 基数排序
- 内部排序算法的比较

概述

- 排序：
  - 将一组杂乱无章的数据按一定的规律顺次排列起来
- 数据表(*datalist*):
  - 待排序数据元素的有限集合
- 排序码(*key*):
  - 数据元素的组成：多个属性域/多个数据成员
  - 其中一个属性域用来区分元素, 作为排序依据，即排序码
  - 每个数据表选用哪个属性域作为排序码，视具体的应用需要而定

- 排序算法的稳定性:
  - 在元素序列中，有两 个元素$r[i]$和$r[j]$, 它们的排序码 $k[i] == k[j]$
  - 排序之前, 元素$r[i]$排在$r[j]$前面
  - 排序之后, 元素$r[i]$仍在元素$r[j]$的前面, 称排序方法是稳定的；否则，称排序方法是不稳定的
- 内排序
  - 在排序期间数据元素全部存放在内存的排序
- 外排序
  - 在排序期间全部元素个数太多，不能同时存放在内存，必须根据排序过程的要求，不断在内、外存之间移动的排序



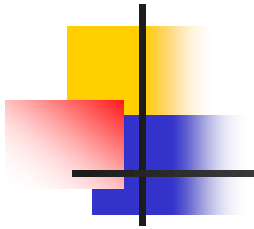
- 时间复杂度
  - 算法执行的时间开销，衡量算法好坏的最重要的标志
  - 用算法执行中的数据比较次数与数据移动次数来衡量
- 一般按平均情况估算算法运行时间代价
- 受元素排序码序列初始排列及元素个数影响较大的算法，需要按最好情况和最坏情况进行估算
- 空间复杂度
  - 算法执行时所需的附加存储，评价算法好坏的标准之一

待排序数据表的类定义

```
#include <iostream.h>
const int  DefaultSize = 100;
template <class T>
class Element {                          //数据表元素定义
public:
    T key;                               //排序码
    field otherdata;                     //其他数据成员
```

```
    Element<T>& operator = (Element<T>& x) {
        key = x.key;  otherdata = x.otherdata;
        return this;
    }
    bool operator == (Element<T>& x)
        { return key == x.key; }      //判*this与x相等
    bool operator <= (Element<T>& x)
        { return key <= x.key; }      //判*this小于或等于x
    bool operator >= (Element<T>& x)
        { return key >= x.key; }      //判*this大于或等于x bool
     operator > (Element<T>& x)
        { return key > x.key; }       //判*this大于x
    bool operator < (Element<T>& x)
        { return key < x.key; }       //判*this小于x
};
```

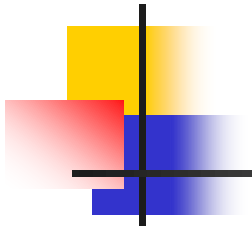
```
template <class T>
class dataList {                          //数据表类定义
private:
    Element <T>* Vector;                  //存储排序元素的向量
    int maxSize;                          //向量中最大元素个数
    int currentSize;                      //当前元素个数
public:
    datalist (int maxSz = DefaultSize) :  //构造函数
    maxSize(maxSz), currentSize(0)
        { Vector = new Element<T>[maxSize]; }
    int Length() { return currentSize; }  //取表长度
```



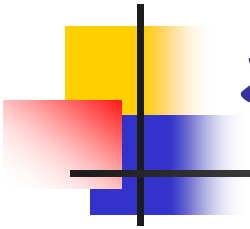
```
    void Swap (Element<T>& x, Element<T>& y)
        { Element<T> temp = x;  x = y;  y = temp; }

    Element<T>& operator [](int i)      //取第i个元素
        { return Vector[i]; }

    int Partition (const int low, const int high);
                                        //快速排序划分
};
```



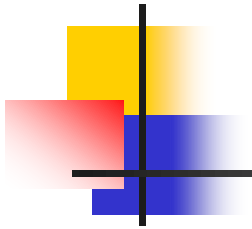
插入排序 (Insert Sorting)



插入排序 (Insert Sorting)

基本方法：

- 每步将一个待排序的元素
- 按其排序码大小，插入到前面已经排好序的一组元素的适当位置上
- 直到元素全部插入为止

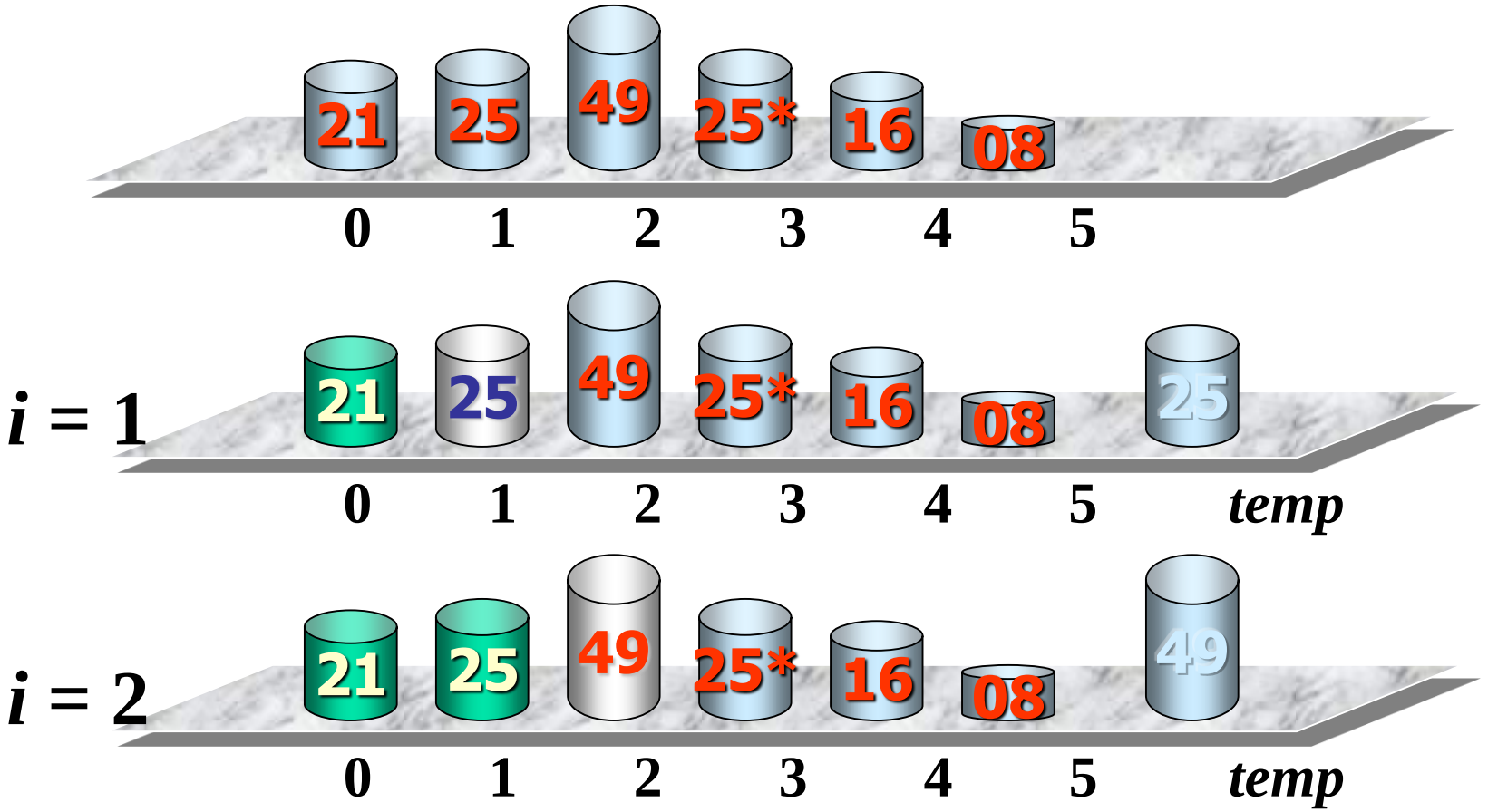


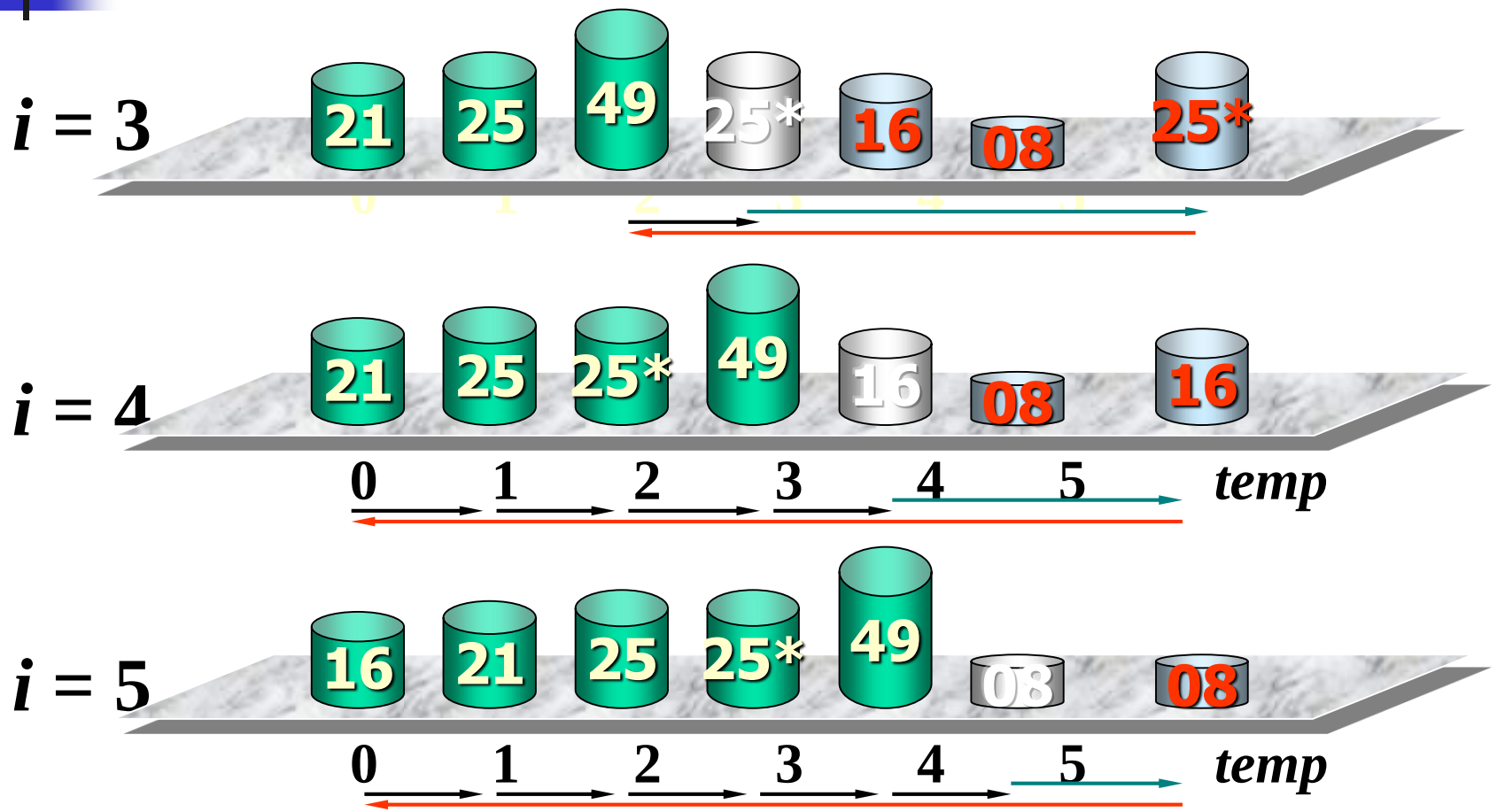
直接插入排序 (Insert Sort)

直接插入排序基本思想

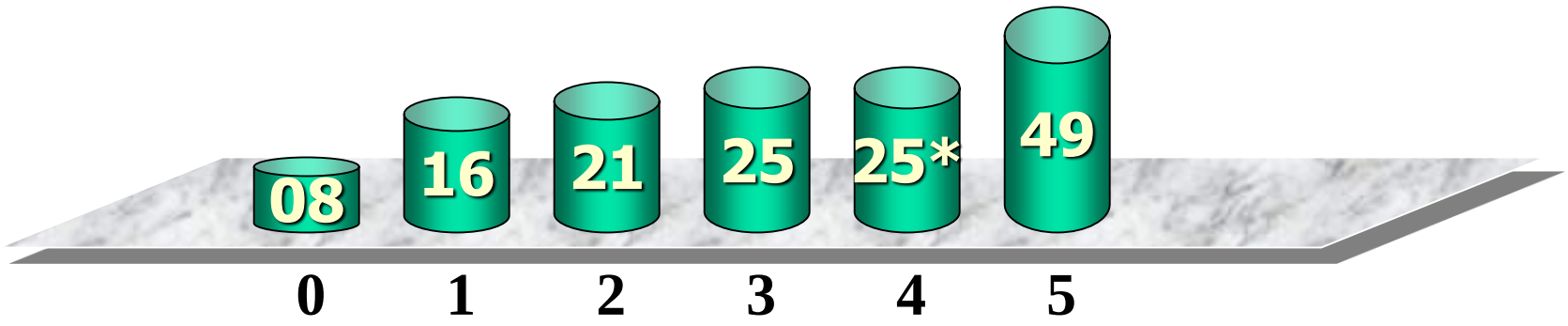
- 插入第i ($i \geq 1$) 个元素时，前面的V[0], V[1], …, V[$i-1$]已经排好序
- 用V[i]的排序码与V[$i-1$], V[$i-2$], …的排序码顺序进行比较
- 找到插入位置，将V[i]插入
- 原来位置上的元素向后顺移

各趟排序结果

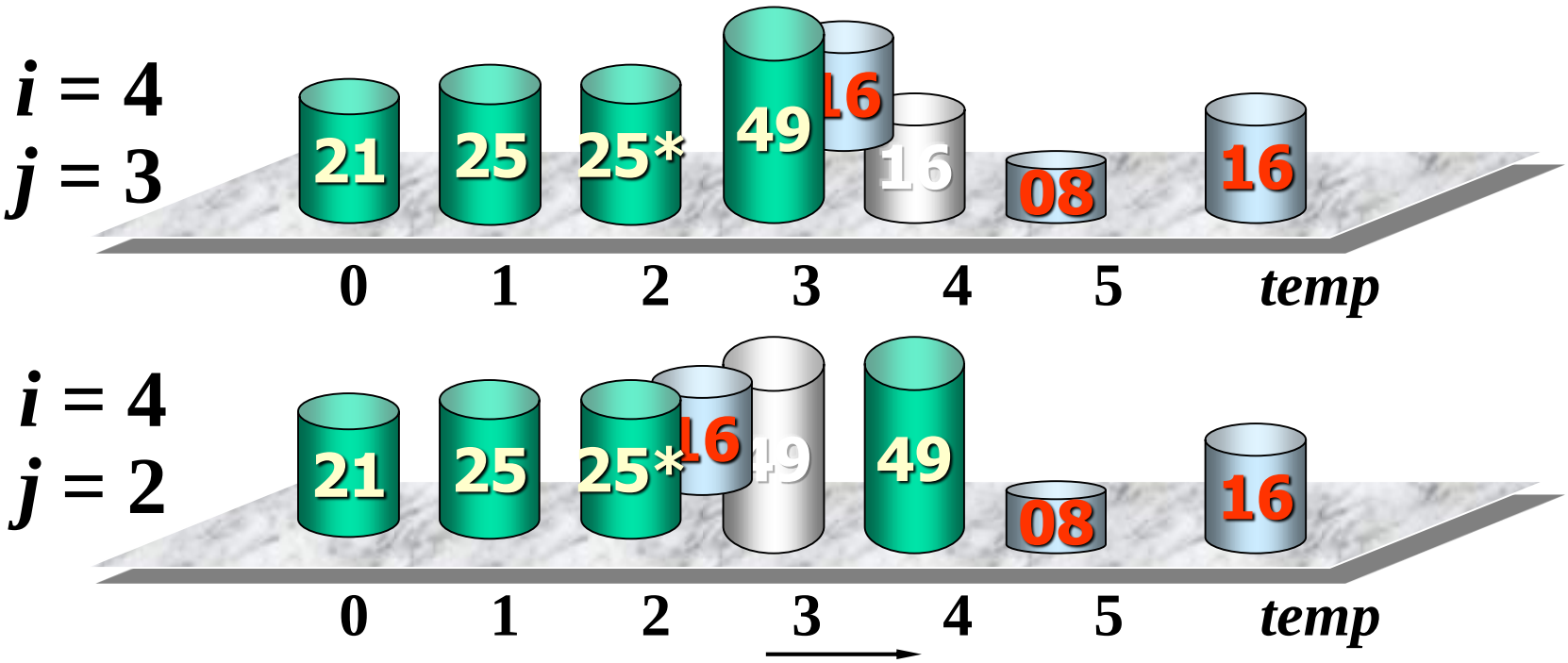

21
25
49
25*
16
08
0
1
2
3
4
5
i = 1
21
25
49
25*
16
08
25
0
1
2
3
4
5
temp
i = 2
21
25
49
25*
16
08
49
0
1
2
3
4
5
temp

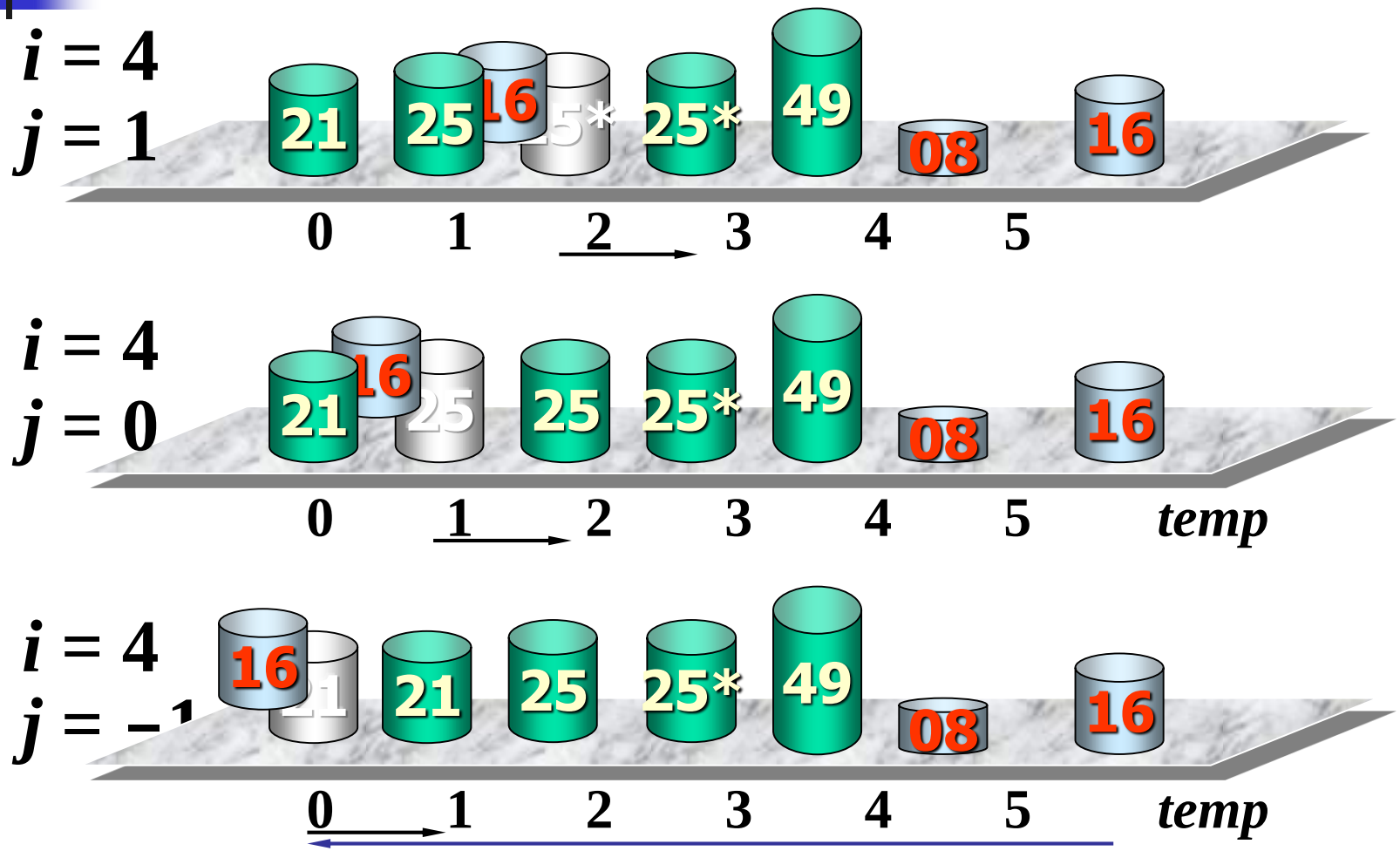

i = 3
21
25
49
25*
16
08
25*
i = 4
21
25
25*
49
16
08
16
0
1
2
3
4
5
temp
i = 5
16
21
25
25*
49
08
08
0
1
2
3
4
5
temp

完成



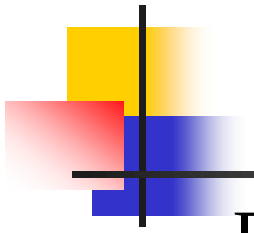
$i = 4$ 时的排序过程


i = 4
j = 3
21
25
25*
49
16
16
08
16
0
1
2
3
4
5
temp
i = 4
j = 2
21
25
25*
16
49
49
08
16
0
1
2
3
4
5
temp


i = 4
j = 1
21
25
16
25*
25*
49
08
16
0
1
2
3
4
5
i = 4
j = 0
21
16
25
25
25*
49
08
16
0
1
2
3
4
5
temp
i = 4
j = –1
16
21
21
25
25*
49
08
16
0
1
2
3
4
5
temp

直接插入排序的算法

```
#include "dataList.h"
template <class T>
void InsertSort (dataList<T>& L, int left, int right)
{
//依次将元素L.Vector[i]按其排序码插入到有序表
//L.Vector[left],…,L.Vector[i-1]中,使得
//L.Vector[left]到L.Vector[i]有序。
```



```
Element<T> temp;  int i, j;
    for (i = left+1; i <= right; i++)
        if (L[i] < L[i−1]) {
        { temp = L[i];  j = i−1;
            do {
                L[j+1] = L[j];  j−−;
            } while (j >= left && temp < L[j]);
            L[j+1] = temp;
        }
}
```

算法分析

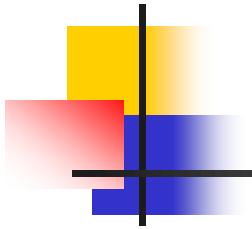
- 设待排序元素个数为currentSize = n
- 该算法的主程序执行n−1趟
- 排序码比较次数和元素移动次数与元素排序码的初始排列有关
- 最好情况下
  - 排序前元素已按排序码从小到大有序
  - 每趟只需与前面有序元素序列的最后一个元素比较1次
  - 总的排序码比较次数为 $n-1$, 元素移动次数为0

- 最坏情况下，
  - 第 i 趟时，第 i 个元素必须与前面 i 个元素都做排序码比较
  - 每做1次比较，做1次数据移动
  - 总排序码比较次数*KCN*和元素移动次数*RMN*分别为：

$$KCN = \sum_{i=1}^{n-1} i = n(n-1)/2 \approx n^2/2,$$

$$RMN = \sum_{i=1}^{n-1} (i+2) = (n+4)(n-1)/2 \approx n^2/2$$

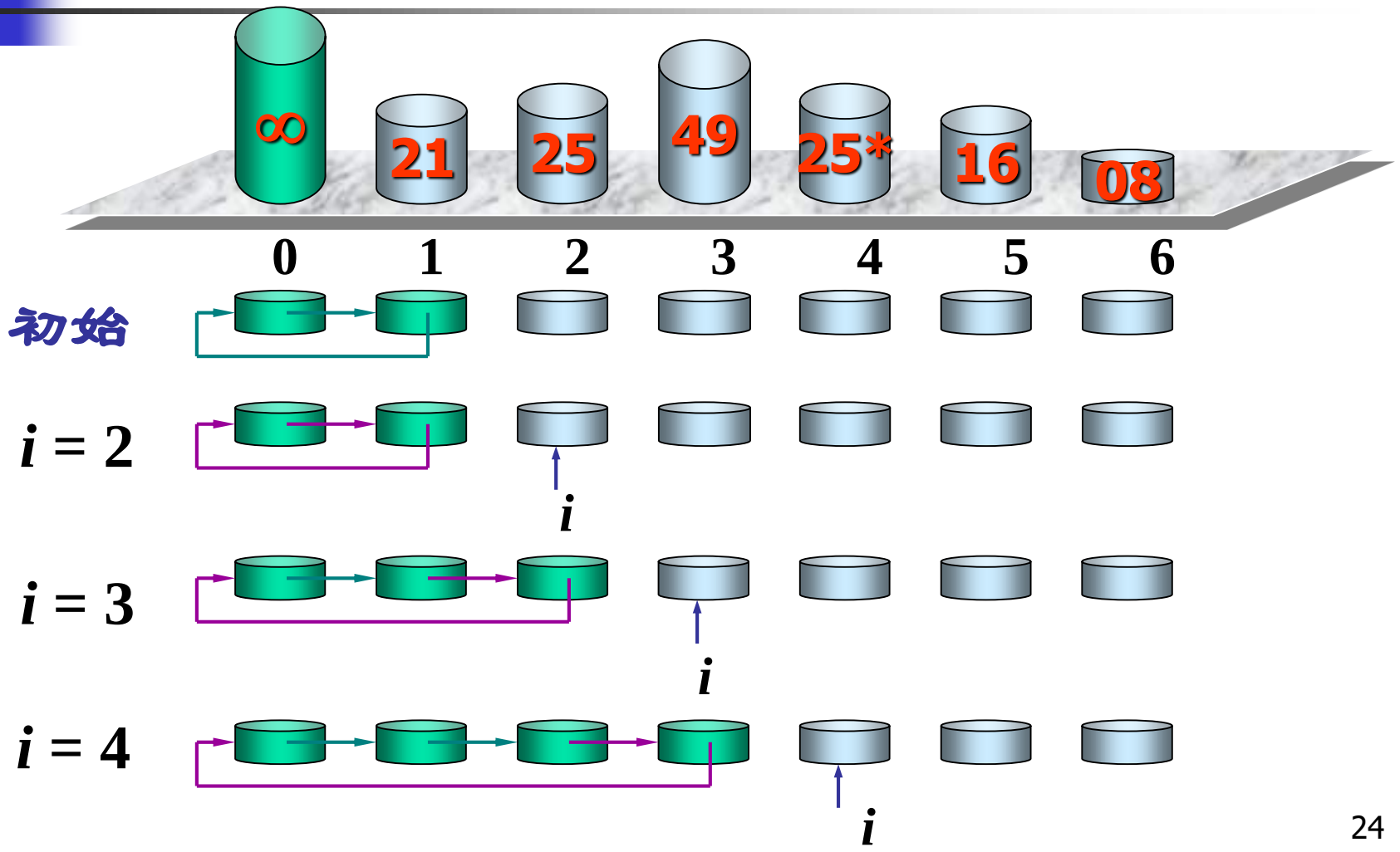
- 平均情况下，排序的时间复杂度为 $o(n^2)$
- 直接插入排序是一种稳定的排序方法

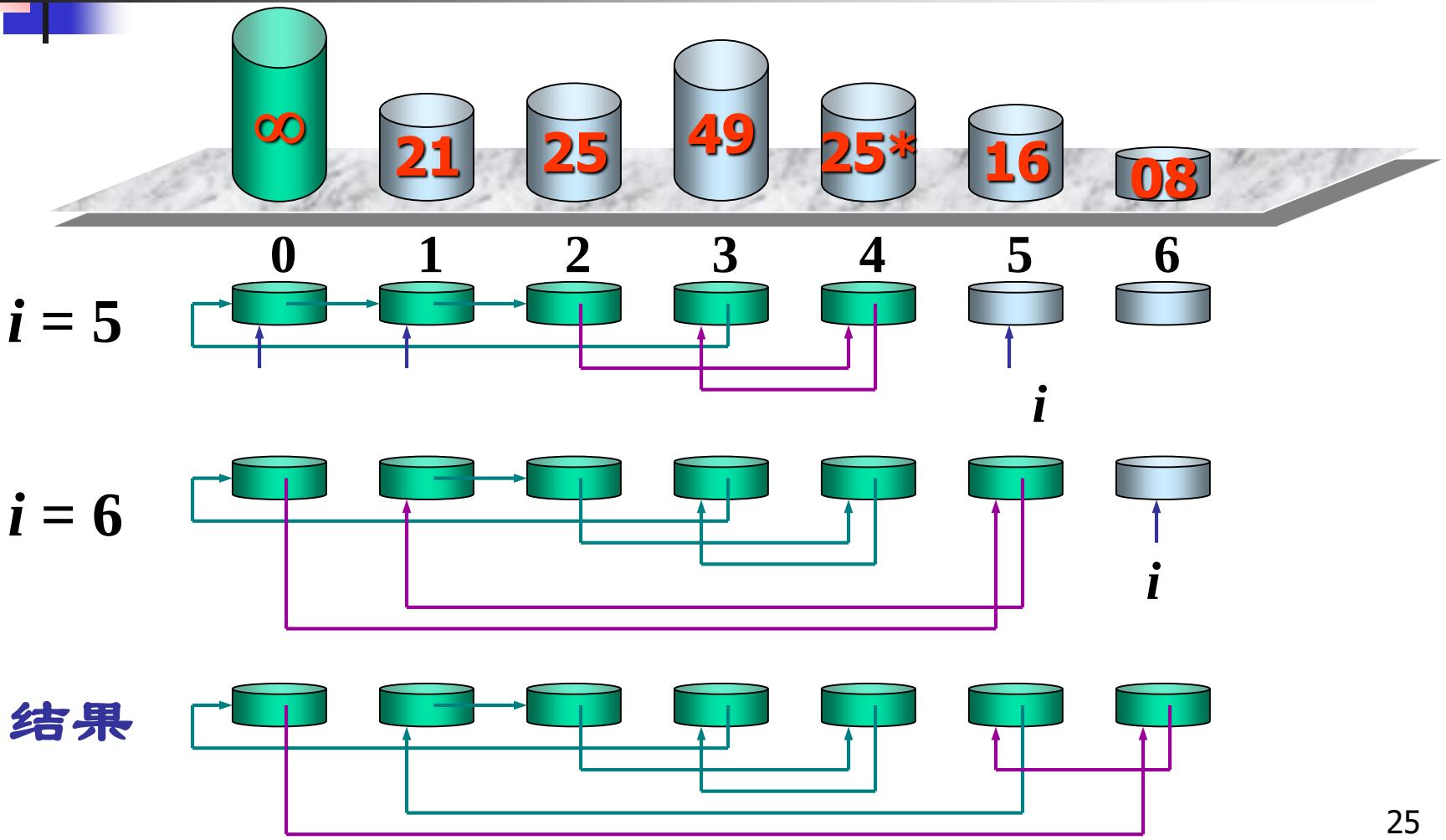


链表插入排序

链表插入排序基本思想

- 在每个元素的结点中，增加一个链接指针数据成员 link
- 数组中的一组元素V[1], …, V[n]
  - V[1], …, V[i−1]已经通过指针link，按其排序码从小到大链接起来
- 插入V[i], $i = 2, 3, \ldots, n$,
- 必须在前面 i−1 个链接起来的元素当中，顺链检测比较，找到V[i] 应插入的位置，把V[i] 链入，并修改相应链接指针
- 得到V[1], …, V[i]的一个通过指针排好的链表
- 如此重复执行，直到把V[n]也插入到链表中排好序为止


∞
21
25
49
25*
16
08
0
1
2
3
4
5
6
初始
i = 2
i
i = 3
i
i = 4
i


∞
21
25
49
25*
16
08
0
1
2
3
4
5
6
i = 5
i
i = 6
i
结果

用于链表排序的静态链表的类定义

```
const int DefaultSize = 10;      //在staticList.h文件中
template <class T>
struct Element
{                          //静态链表元素类定义
    T key;                                    //排序码，其它信息略
    int link;                                 //结点的链接指针
     Element () : link(0) { }          //构造函数
     Element (T x, int next = 0) : key(x), link(next) { }
                                              //构造函数
}
```

```
template <class T>
class staticLinkedList
{          //静态链表的类定义
public:
    staticLinkedList (int sz = DefaultSize) {
       maxSize = sz;  n = 0;
       Vector = new Element<T>[sz];
    }
    Element<T>& operator [](int i) {return Vector[i];}
private:
    Element<T> *Vector;                    //存储元素的向量
    int maxSize;                           //最大元素个数
    int n;                                 //当前元素个数
};
```

链表插入排序的算法

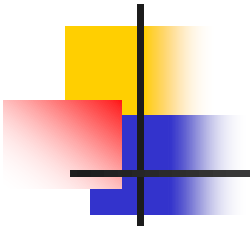
```
#include "staticList.h"
const T maxData;                    //排序码集合中的最大值
template <class T>
int insertSort (staticLinkedList<T>& L)
{
//对L.Vector[1],...,L.Vector[n]按其排序码key排序,
//L.Vector[0] 做为排序后各个元素所构成的有序循
//环链表的附加头结点使用
```

```
    L[0].key = maxData;
    L[0].link = 1;   L[1].link = 0;
                              //形成只有一个元素的循环链表
    int i, pre, p;
    for (i = 2; i <= n; i++) {
          //每趟向有序链表中插入一个结点
       p = L[0].link;                          //p是链表检测指针
       pre = 0;                          //pre指向p的前驱
       while (L[p].key <= L[i].key)    //循链找插入位置
          { pre = p;  p = L[p].link; }
       L[i].link = p;  L[pre].link = i;   //结点i链入
     }
}
```

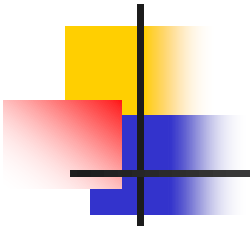
算法分析

- 插入一个元素
  - 最大排序码比较次数等于链表中已排好序的元素个数
  - 最小排序码比较次数为1
- 总的排序码比较次数
  - 最小为 *n*-1
  - 最大为:

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



- 链表插入排序，元素移动次数为0
- 实现链表插入，在每个元素中增加了一个链域 link，使用Vector[0] 作为链表的表头结点
- 使用 *n* 个附加域和一个附加元素
- 算法在Vector[pre].key == Vector[i].key时，将Vector[i]插在Vector[pre]的后面
- 链表插入排序方法是稳定的



折半插入排序 (Binary Insertsort)

折半插入排序基本思想

- 在顺序表中，有一 个元素序列 V[0], V[1], …, V[n−1]
- V[0], V[1], …, V[i−1] 是已经排好序的元素
- 插入V[i] 时, 利用折半搜索法寻找V[i] 的插入位置

折半插入排序的算法

```
#include "dataList.h"
template <class T>
void BinaryInsertSort (dataList<T>& L,
      const int left, const int right) {

//利用折半搜索, 在L.Vector[left]到L.Vector[i-1]中
//查找L.Vector[i]应插入的位置, 再进行插入。
```

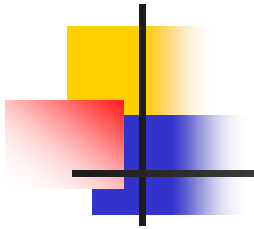
```
    Element<T> temp;
    int i, low, high, middle, k;
    for (i = left+1; i <= right; i++)
      { temp = L[i];  low = left;  high = i-1;
        while (low <= high)      //折半搜索插入位置
        { middle = (low+high)/2;//取中点
          if (temp < L[middle])  //插入值小于中点值
              high = middle-1;      //向左缩小区间
          else low = middle+1;    //否则,向右缩小区间
        }
        for (k = i-1; k >= low; k--)
              L[k+1] = L[k];  //成块移动,空出插入位置
         L[low] = temp;                 //插入
    }
}
```

算法分析

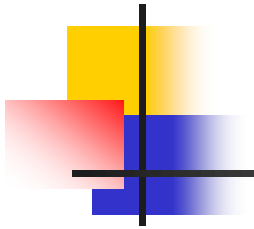
- 折半搜索比顺序搜索快
- 折半插入排序就平均性能，比直接插入排序要快
- 排序码比较次数与待排序元素序列的初始排列无关，仅依赖于元素个数
- 插入第 i 个元素时，经过 $\lfloor \log_2 i \rfloor + 1$ 次排序码比较, 才能确定它应插入的位置
- 将 n 个元素(为推导方便, 设为 $n=2^k$) 用折半插入排序所进行的排序码比较次数为：

$$\sum_{i=1}^{n-1} \left(\lfloor \log_2 i \rfloor + 1 \right) \approx n \cdot \log_2 n$$

- 折半插入排序是一个稳定的排序方法



- 当 n 较大时
  - 总排序码比较次数比直接插入排序的最坏情况要好得多
  - 比其最好情况要差
- 在元素的初始排列已经按排序码排好序或接近有序时
  - 直接插入排序比折半插入排序执行的排序码比较次数要少
- 折半插入排序的元素移动次数与直接插入排序相同
- 依赖于元素的初始排列

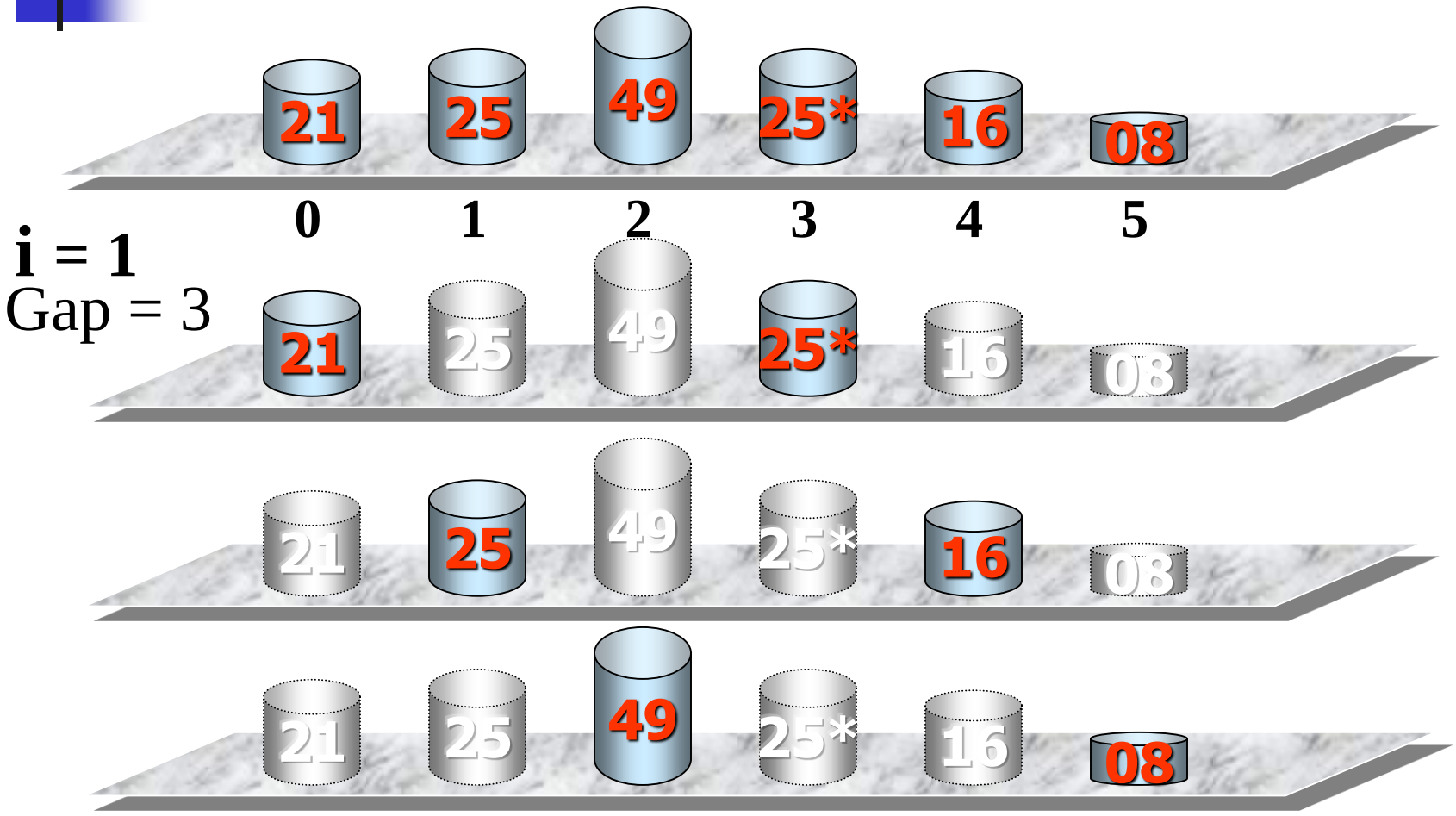


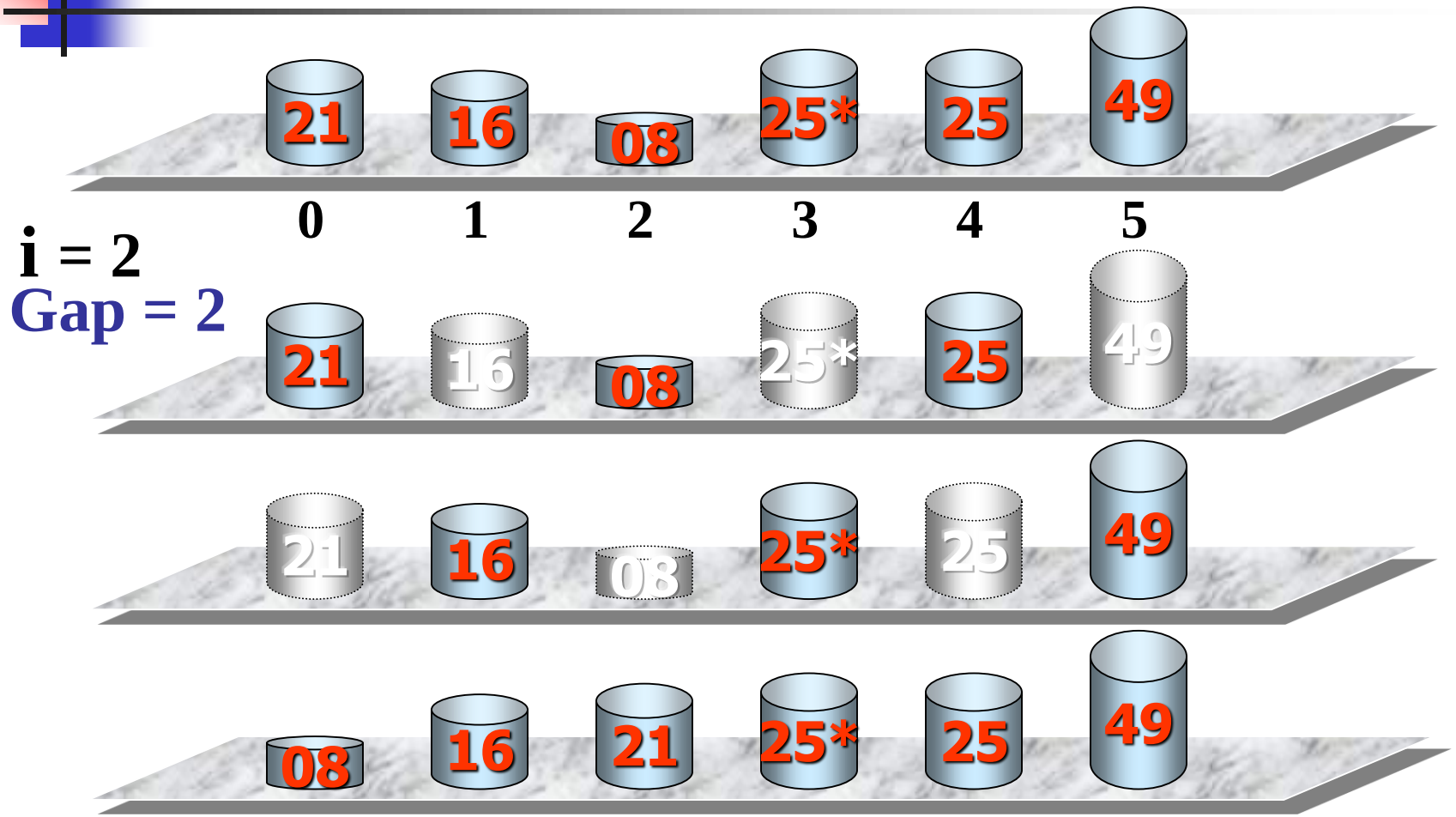
希尔排序 (Shell Sort)

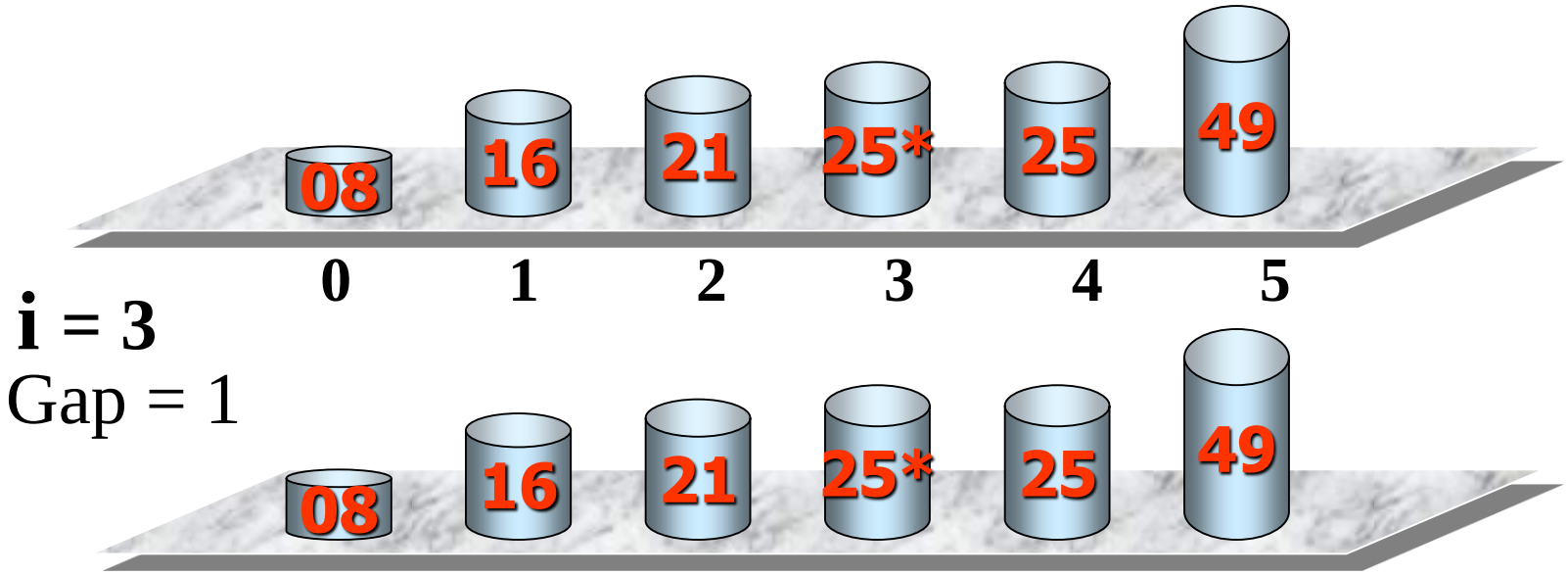
希尔排序：缩小增量排序

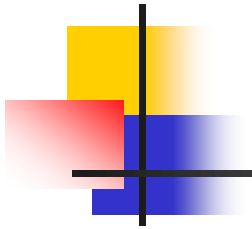
基本思想:

- 待排序元素序列有 n 个元素
- 取一个整数 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，在每一个子序列中分别施行直接插入排序
- 缩小间隔 gap，如取 gap =⌈gap/2⌉，重复上述的子序列划分和排序工作
- 直至最后 gap == 1，将所有元素放在同一个序列中排序为止


21
25
49
25*
16
08
0
1
2
3
4
5
i = 1
Gap = 3
21
25
49
25*
16
08
21
25
49
25*
16
08
21
25
49
25*
16
08


21
16
08
25*
25
49
0
1
2
3
4
5
i = 2
Gap = 2
21
16
08
25*
25
49
21
16
08
25*
25
49
08
16
21
25*
25
49

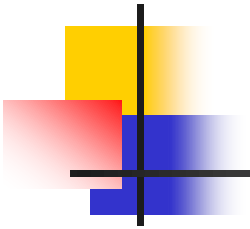

08
16
21
25*
25
49
0
1
2
3
4
5
i = 3
Gap = 1
08
16
21
25*
25
49



- 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, 子序列中元素个数逐渐变多
  - 大多数元素已基本有序，所以排序速度仍然很快

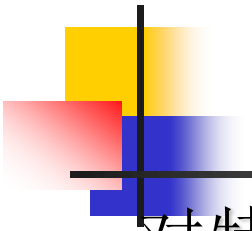
希尔排序的算法

```
#include "dataList.h"
template <class T>
void Shellsort (dataList<T>& L,
        const int left, const int right)
{
   int i, j, gap = right−left+1;                  //增量的初始值
     Element<T> temp;
   do {
      gap = gap/3+1;                                          //求下一增量值
      for (i = left+gap; i <= right; i++)
         if (L[i] < L[i−gap])                     //逆序
         {  temp = L[i];  j = i−gap;
            do {
               L[j+gap] = L[j];  j = j−gap;
            } while (j >= left && temp < L[j]);
            L[j+gap] = temp;    //将vector[i]回送
         }
   } while (gap > 1);
}
```



算法分析

- Gap的取法有多种
  - 最初 shell 提出取 gap = ⌊n/2⌋，gap = ⌊gap/2⌋，直到gap = 1
  - knuth 提出，取 gap = ⌊gap/3⌋ +1
  - 有提出都取奇数为好
  - 有提出各 gap 互质为好



- 对特定的待排序元素序列，可以准确地估算排序码的比较次数和元素移动次数
- 目前无法确定：
  - 排序码比较次数和元素移动次数与增量选择之间的依赖关系，并给出完整的数学分析
- 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元素平均移动次数大约在 $n^{1.25}$ 到 $1.6n^{1.25}$ 的范围内
    - 在利用直接插入排序作为子序列排序方法情况下得到
- 希尔排序是一种不稳定的排序方法