



# Data Structures

Ch8

# 排序 Sort

2024 年 12 月 10 日

学而不厌 诲人不倦

- ➡ **8.1 概述**
- ➡ **8.2 插入排序：直接插入排序、希尔排序**
- ➡ **8.3 交换排序：起泡、快速排序**
- ➡ **8.4 选择排序：简单选择、堆排序**
- ➡ **8.5 归并排序：二路归并排序**
- ➡ **8.6 各种排序方法比较**
- ➡ **8.7 扩展与提高**

## 8.1 概述

## 8.1 概述

### 1. 排序的定义

数据元素、结点、顶点

✚ 排序：给定一组记录的集合  $\{r_1, r_2, \dots, r_n\}$ ，其相应的关键码分别为  $\{k_1, k_2, \dots, k_n\}$ ，将这些记录排列为  $\{r_{s1}, r_{s2}, \dots, r_{sn}\}$  的序列，使得相应的关键码满足  $k_{s1} \leq k_{s2} \leq \dots \leq k_{sn}$ （或  $k_{s1} \geq k_{s2} \geq \dots \geq k_{sn}$ ）。

升序（非降序）

降序（非升序）

✚ 排序码：排序的依据，简单起见，也称关键码。

职工号	姓名	性别	年龄	工作时间
0001	王刚	男	48	1990.4
0002	张亮	男	35	2003.7
0003	刘楠	女	57	1979.9
0004	齐梅	女	35	2003.7
0005	李爽	女	56	1982.9

🕒 排序的数据模型是什么？  
排序是对线性结构的一种操作

## 8.1 概述

### 2. 排序的基本原理

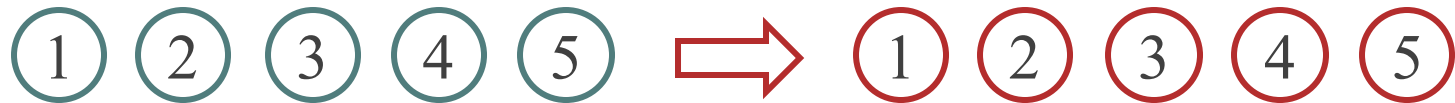
有序序列区

无序序列区

**一趟排序**：将无序序列中的一个或几个加入到有序序列中。

这样经过若干趟排序后，有序序列区越来越大，无序序列区越来越小。

✦ **正序**：待排序序列中的记录已按关键码排好序。



✦ **逆序（反序）**：待排序序列中记录的顺序与排好序的顺序相反。



## 8.1 概述

### 3. 排序算法分类

✦ 根据排序过程中所有记录**是否全部放在内存中**，排序方法分为：

- (1) **内排序**：在排序的整个过程中，待排序的所有记录**全部放在内存中**
- (2) **外排序**：待排序的记录个数较多，整个排序过程需要**在内外存之间**多次交换数据才能得到排序的结果

✦ 根据排序方法是否建立在**关键码比较**的基础上，排序方法分为：

- (1) **基于比较**：主要通过关键码之间的**比较**和记录的**移动**实现

① 插入排序；

├ 直接插入排序  
└ 希尔排序

② 交换排序；

├ 起泡排序  
└ 快速排序

③ 选择排序；

├ 简单选择排序  
└ 堆排序

④ 归并排序

├ 二路归并**递归算法**  
└ 二路归并**非递归算法**

- (2) **不基于比较**：根据待排序数据的特点所采取的其他方法

## 8.1 概述

### 4. 算法稳定性

✦ **排序算法的稳定性**：假定在待排序的记录序列中存在多个具有相同关键码的记录，若经过排序，这些记录的**相对次序**保持不变，则称这种排序算法稳定，否则称为不稳定。

学号	姓名	高数	英语	语文
0001	王 军	85	68	88
0002	李 明	64	72	92
0003	汤晓影	85	78	86
...	...	...	...	...



学号	姓名	高数	英语	语文
0001	李 明	64	68	88
0002	王 军	85	72	92
0003	汤晓影	85	78	86
...	...	...	...	...

排序算法的稳定性只是算法的一种属性，且由具体算法决定

## 8.1 概述

### 5. 排序算法性能

 如何衡量排序算法的性能呢？

(1) **时间性能**：排序算法在**各种情况（最好、最坏、平均）**下的时间复杂度。

例如，**基于比较的内排序**在排序过程中的基本操作：

① **比较**：关键码之间的比较；

② **移动**：记录从一个位置移动到另一个位置。

(2) **空间性能**：排序过程中占用的辅助存储空间。

辅助存储空间是除了存放待排序记录占用的存储空间之外，执行算法所需要的其他存储空间。



## 8.1 概述



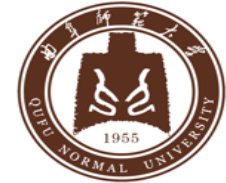
### 6. 排序类定义

```
class Sort
{public:
    Sort(int r[ ], int n);
    ~Sort( );
    void InsertSort( );
    void ShellSort( );
    void BubbleSort( );
    void QuickSort(int first, int last);
    void SelectSort( );
    void HeapSort( );
    void MergeSort1(int first, int last);
    void MergeSort2( );
    void Print( );
```

```
private:
    int Partition(int first, int last);
    void Sift(int k, int last);
    void Merge(int first1, int last1, int last2);
    void MergePass(int h);

    int *data;
    int length;
};
```

## 8.1 概述



### 6. 排序类定义

```
Sort :: Sort(int r[ ], int n)
{
    data = new int[n];
    for (int i = 0; i < n; i++)
        data[i] = r[i];
    length = n;
}
Sort :: ~Sort( )
{
    delete[ ] data;
}
```

```
void Sort :: Print( )
{
    for (int i = 0; i < length; i++)
    {
        cout << data[i] << "\t";
    }
    cout << endl;
}
```

## 8.2 插入排序

### 8-2-1 直接插入排序

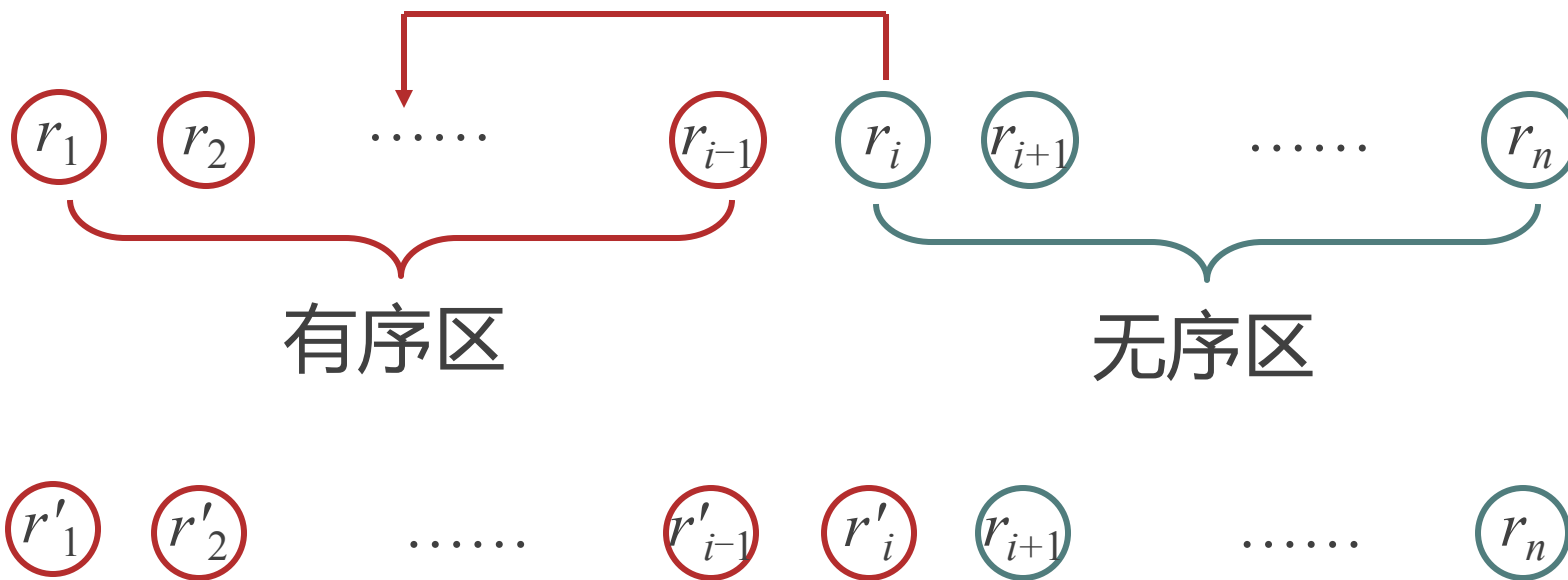
## 8.2 插入排序

### 8-2-1 直接插入排序



#### 1. 直接插入排序

直接插入排序的基本思想：**依次**将待排序序列中的每一个记录插入到**已排好序**的序列中，直到全部记录都排好序。



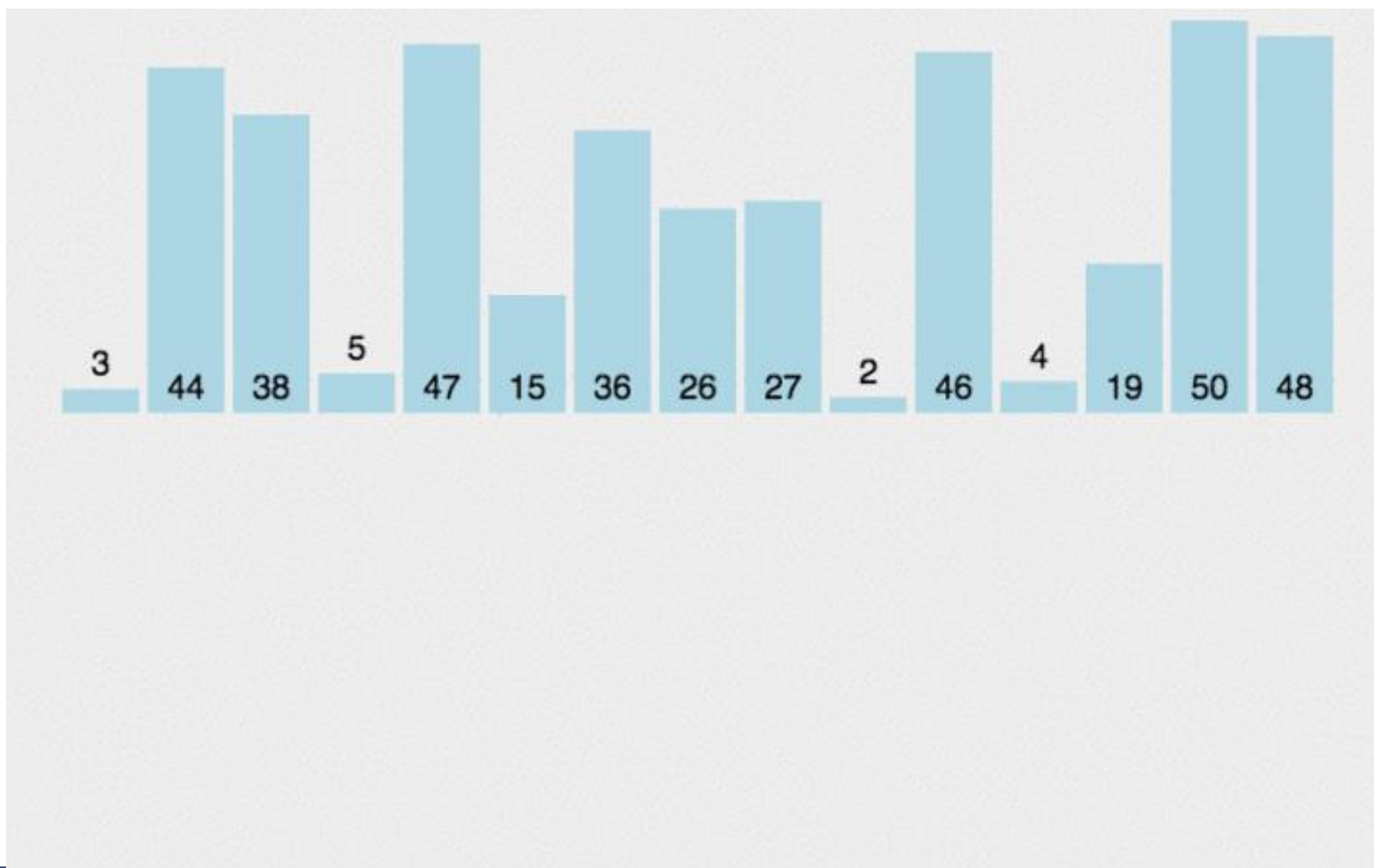
在插入第  $i$  ( $i > 1$ ) 个记录时，前面的  $i-1$  个记录已经排好序

## 8.2 插入排序

### 8-2-1 直接插入排序



### 1. 直接插入排序



## 8.2 插入排序

### 8-2-1 直接插入排序

#### 1. 直接插入排序

实现“一趟插入排序”可分三步进行：


1、在 $R[1..i-1]$ 中查找 $R[i]$ 的插入位置；

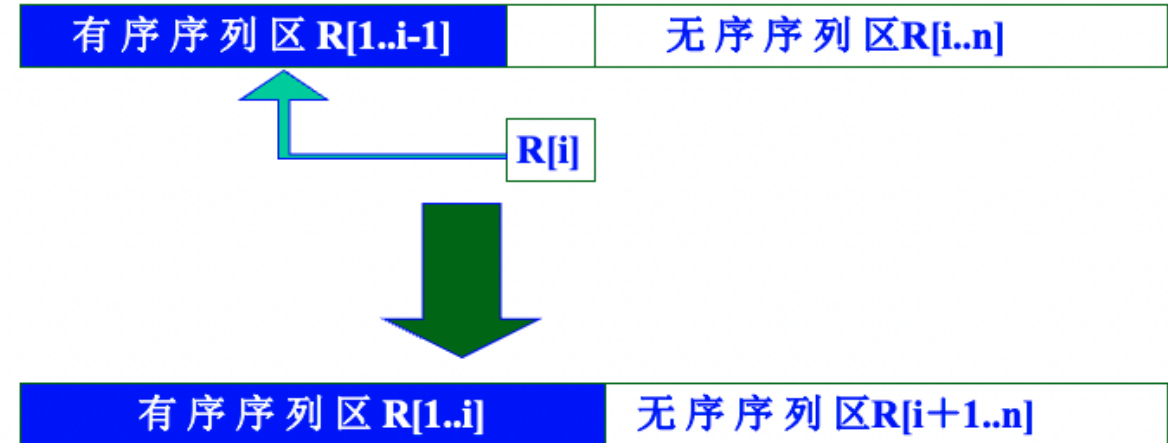
$$R[1..j] \leq R[i] < R[j+1..i-1]$$

2、将 $R[j+1..i-1]$ 中的所有记录均后移一个位置；

3、将 $R[i]$ 复制到 $R[j+1]$ 的位置上。

 如何构造初始的有序序列？

 解决方法：将第 1 个记录看成是初始有序序列，  
然后从第 2 个记录起依次插入到有序序列中，直至将第  $n$  个记录插入。



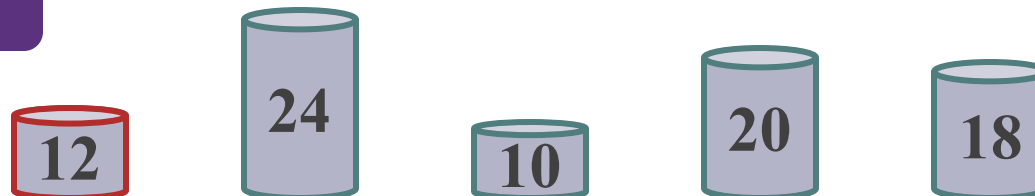
## 8.2 插入排序

### 8-2-1 直接插入排序

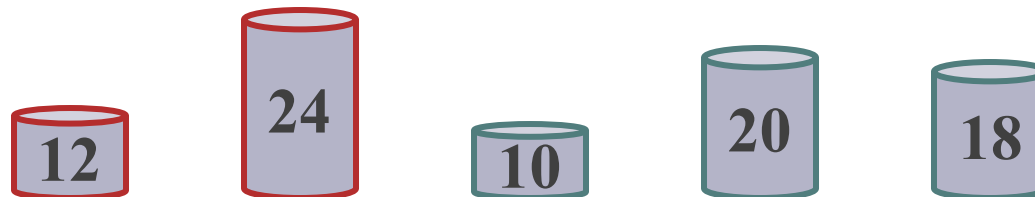


#### 1. 直接插入排序

待排序序列



第一趟排序结果



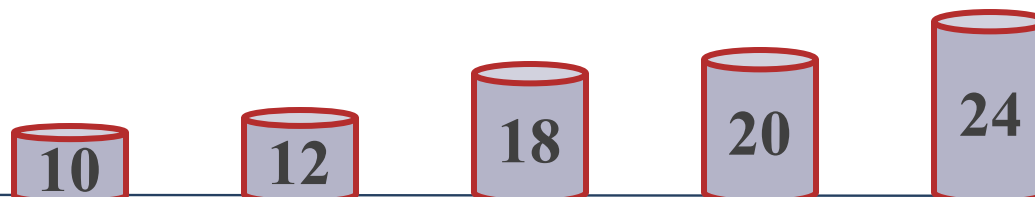
第二趟排序结果



第三趟排序结果



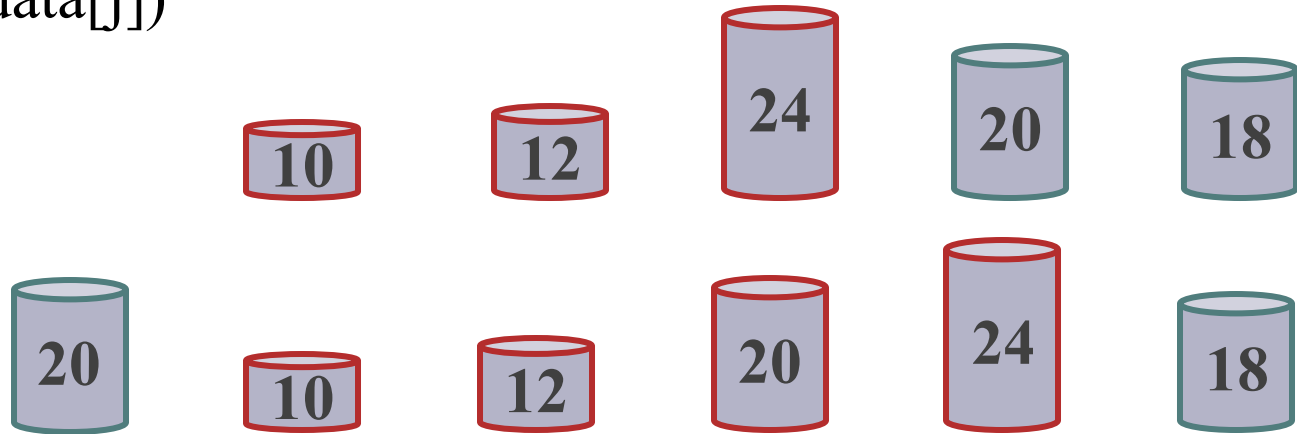
第四趟排序结果





## 2. 算法描述与实现

```
void Sort :: InsertSort( )  
{  
    int i, j, temp;  
    for (i = 1; i < length; i++)  
    {  
        temp = data[i];  
        j = i - 1;  
        while (j >= 0 && temp < data[j])  
        {  
            data[j + 1] = data[j];  
            j--;  
        }  
        data[j + 1] = temp;  
    }  
}
```





### 3. 算法时间性能

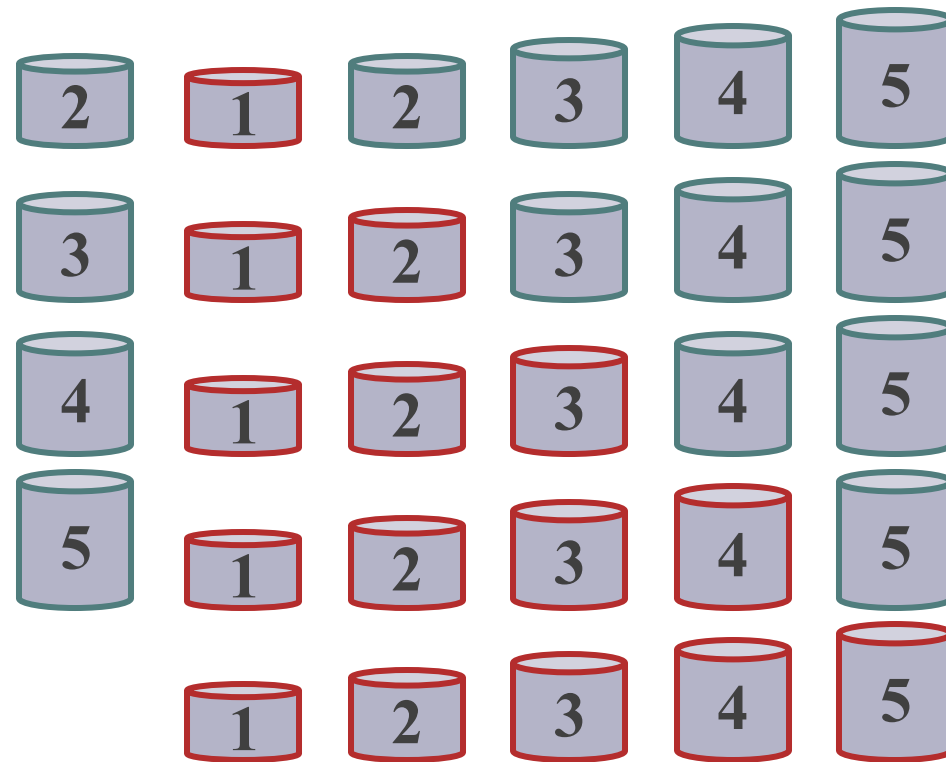
```
void Sort :: InsertSort( )  
{  
    int i, j, temp;  
    for (i = 1; i < length; i++)  
    {  
        temp = data[i];  
        j = i - 1;  
        while (j >= 0 && temp < data[j])  
        {  
            data[j + 1] = data[j];  
            j--;  
        }  
        data[j + 1] = temp;  
    }  
}
```



比较语句? 执行次数?



最好情况:  $n-1$ 次



### 3. 算法时间性能

```
void Sort :: InsertSort( )
{
    int i, j, temp;
    for (i = 1; i < length; i++)
    {
        temp = data[i];
        j = i - 1;
        while (j >= 0 && temp < data[j])
        {
            data[j + 1] = data[j];
            j--;
        }
        data[j + 1] = temp;
    }
}
```



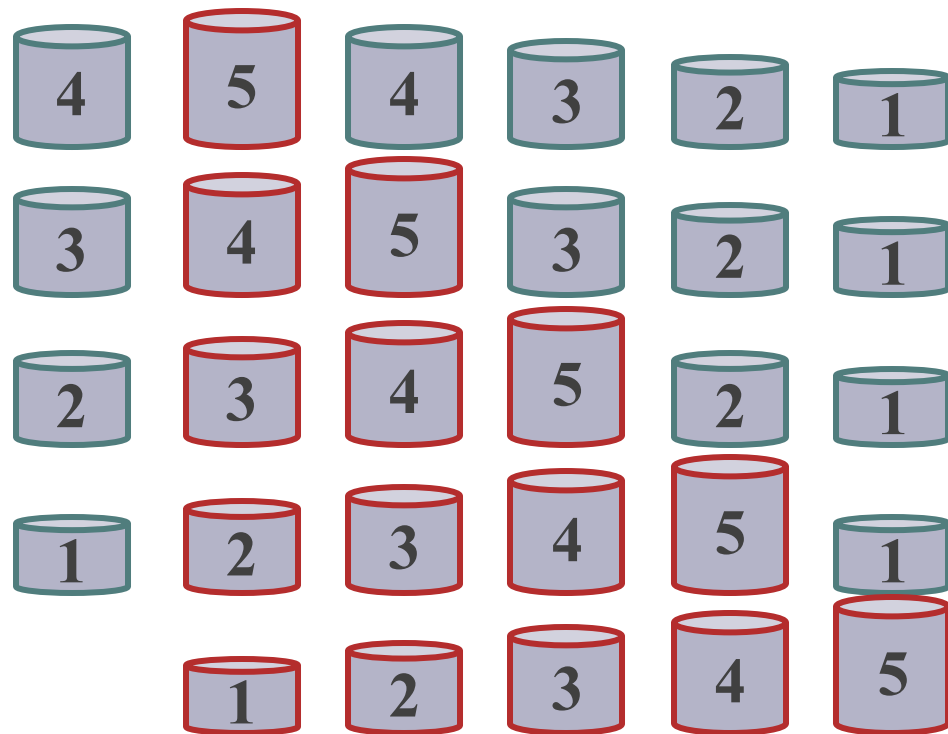
比较语句？ 执行次数？



最好情况：  $n-1$  次



最坏情况：  $2+3+\cdots+n$  次



### 3. 算法时间性能

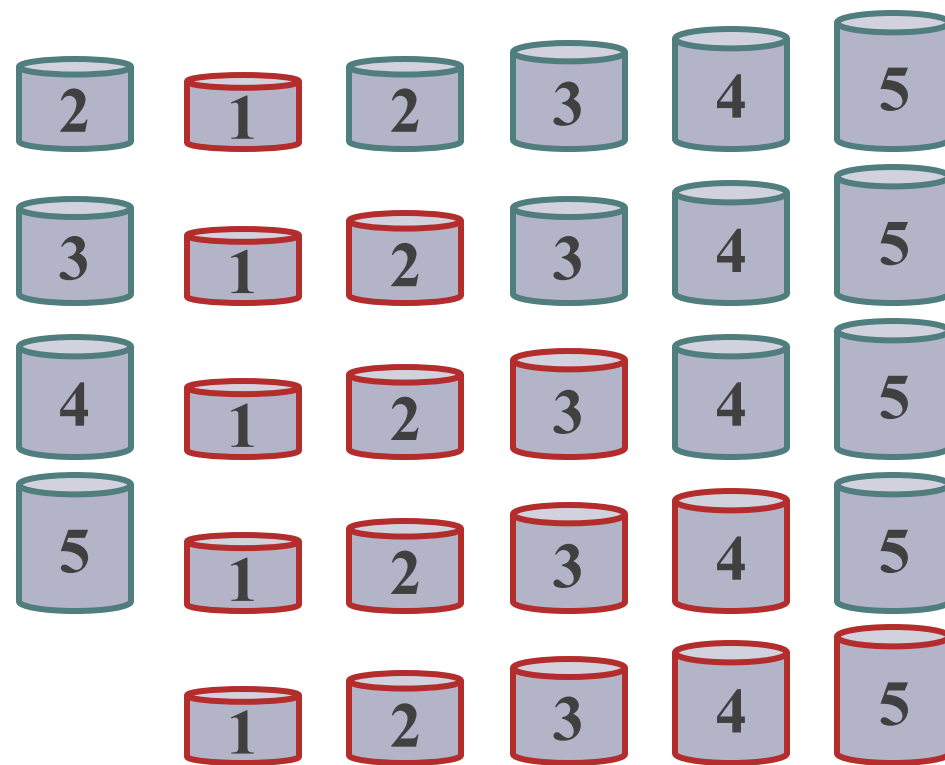
```
void Sort :: InsertSort( )
{
    int i, j, temp;
    for (i = 1; i < length; i++)
    {
        temp = data[i];
        j = i - 1;
        while (j >= 0 && temp < data[j])
        {
            data[j + 1] = data[j];
            j--;
        }
        data[j + 1] = temp;
    }
}
```



移动语句？ 执行次数？

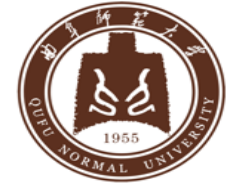


最好情况：  $2(n-1)$  次



## 8.2 插入排序

### 8-2-1 直接插入排序



### 3. 算法时间性能

```
void Sort :: InsertSort( )
{
    int i, j, temp;
    for (i = 1; i < length; i++)
    {
        temp = data[i];
        j = i - 1;
        while (j >= 0 && temp < data[j])
        {
            data[j + 1] = data[j];
            j--;
        }
        data[j + 1] = temp;
    }
}
```



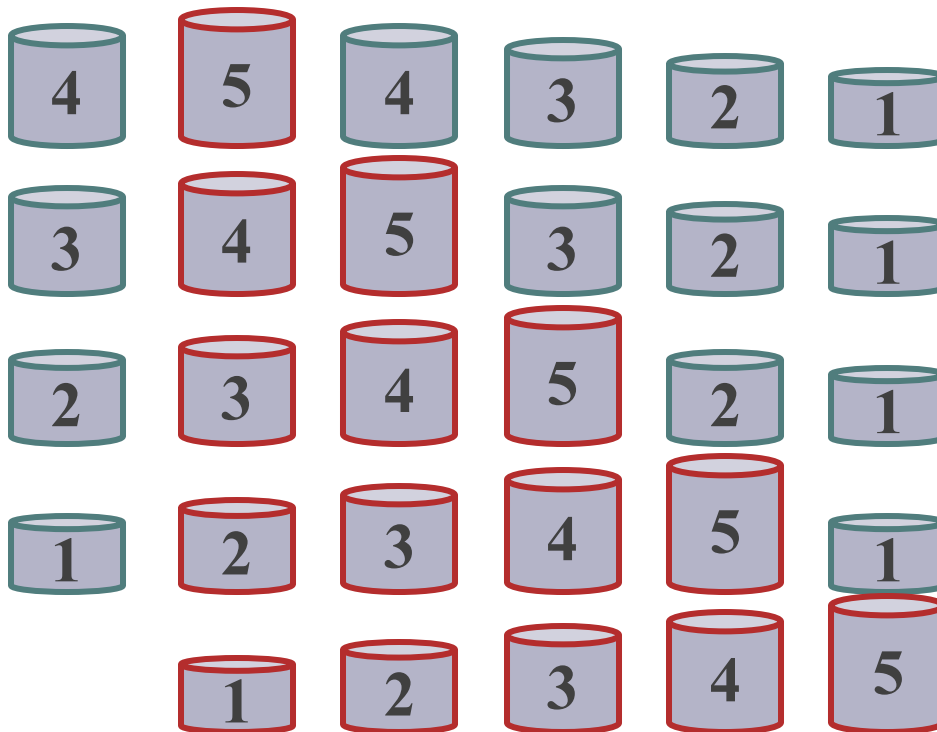
移动语句? 执行次数?



最好情况:  $2(n-1)$ 次



最坏情况:  $3+4+\cdots+n+1$ 次



#### 4. 直接插入排序的效率分析

```
void Sort :: InsertSort( )  
{  
    int i, j, temp;  
    for (i = 1; i < length; i++)  
    {  
        temp = data[i];  
        j = i - 1;  
        while (j >= 0 && temp < data[j])  
        {  
            data[j + 1] = data[j];  
            j--;  
        }  
        data[j + 1] = temp;  
    }  
}
```



最好情况：正序  $O(n)$



比较次数： $n-1$ 次



移动次数： $2(n-1)$ 次



最坏情况：逆序  $O(n^2)$



比较次数： $\sum_{i=2}^n i = \frac{(n+2)(n-1)}{2}$  次



移动次数： $\sum_{i=2}^n (i+1) = \frac{(n+4)(n+1)}{2}$  次



平均情况：随机排列， $O(n^2)$



### 5. 折半插入排序

#### 直接插入排序算法的**移动**改进

由于直接插入排序算法是对有序表进行插入操作，故**顺序查找**操作可以替换为**折半查找**操作。

```
void Sort :: BiInsertionSort( Elem R[], int n)
```

```
{ for(i=2; i<=n; ++i)
```

```
{ R[0]=R[i]; //将R[i]暂存在R[0]
```

```
{ 在R[1..i-1]中折半查找插入位置; }
```

```
for(j=i-1; j>=high-1; --j)
```

```
    R[j+1]=R[j]; //记录后移
```

```
    R[high+1]=R[0]; //插入
```

```
}
```

```
}
```

只考虑比较： $O(n\log_2 n)$



## 8.2 插入排序

### 8-2-1 直接插入排序

#### 6. 二路插入排序

#### 直接插入排序算法的**移动**改进

初始，取第一个元素为**基准元素**；

构造前序列  $S_1$ ，后序列  $S_2$ ；

对第 2, 3, ..., n 个元素依次执行：

与基准元素比较：

若小，插入到前序列  $S_1$  中；

若大，插入到后序列  $S_2$  中；

$S_1$  + **基准元素** +  $S_2$  可得最终有序表。

例，序列 49 38 65 97 76 13 27

↑ ↑ ↑ ↑ ↑ ↑

以 49 为基准

前序列

后序列

{ 13 27 38 }

{ 65 76 97 }

{ 13 27 38 } + { 49 } + { 65 76 97 }

## 8.2 插入排序

### 8-2-2 希尔排序





## 8.2 插入排序

### 8-2-2 希尔排序

#### 1. 希尔排序

#### 缩小增量排序

##### 分析直接插入排序

1. 若待排序记录序列按关键字**基本有序**，则排序效率可大大提高；
2. 待排序记录**总数越少**，排序**效率越高**；

当待排序的记录**个数较多**时，大量的**比较和移动**操作使直接插入排序算法的效率降低。

#### 希尔排序的基本思想：

对待排记录序列先作“宏观”调整，再作“微观”调整。

所谓“宏观”调整，指的是“跳跃式”的插入排序。



## 8.2 插入排序

### 8-2-2 希尔排序

#### 1. 希尔排序

#### 缩小增量排序

先将待排序记录序列分割成为若干子序列分别进行直接插入排序；  
待整个序列中的记录基本有序后，再全体进行一次直接插入排序。

例如：将  $n$  个记录分成  $d$  个子序列

$\{ R[1], R[1+d], R[1+2d], \dots, R[1+kd] \}$

$\{ R[2], R[2+d], R[2+2d], \dots, R[2+kd] \}$

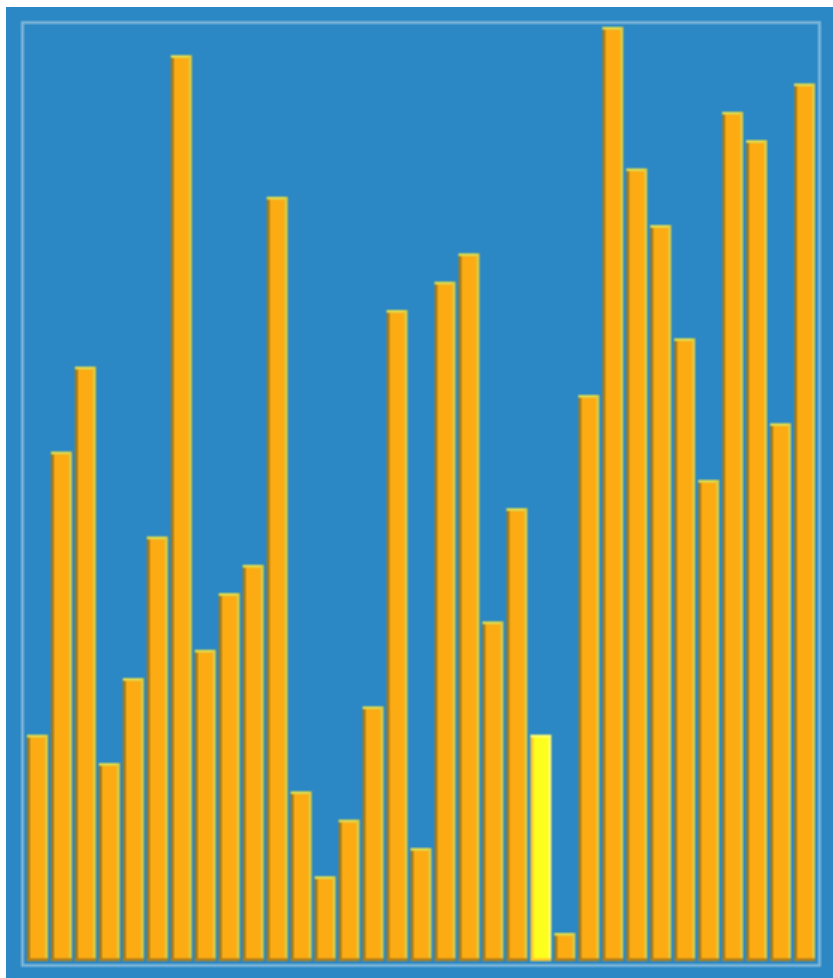
$\dots$

$\{ R[d], R[2d], R[3d], \dots, R[kd], R[(k+1)d] \}$

其中， $d$  称为增量，它的值在排序过程中从大到小逐渐缩小，直到最后一趟排序减为1。

## 8.2 插入排序

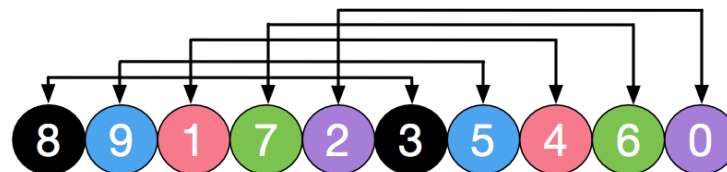
### 1. 希尔排序



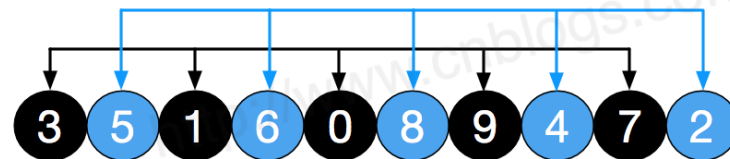
原始数组 以下数据元素颜色相同为一组

8 9 1 7 2 3 5 4 6 0

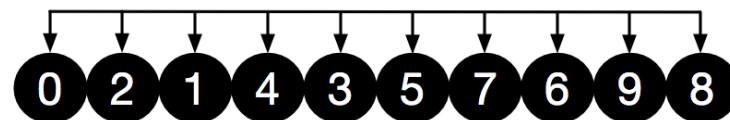
初始增量  $gap=length/2=5$ ，意味着整个数组被分为5组， $[8,3]$   $[9,5]$   $[1,4]$   $[7,6]$   $[2,0]$



对这5组分别进行直接插入排序，结果如下，可以看到，像3，5，6这些小元素都被调到前面了，然后缩小增量  $gap=5/2=2$ ，数组被分为2组  $[3,1,0,9,7]$   $[5,6,8,4,2]$



对以上2组再分别进行直接插入排序，结果如下，可以看到，此时整个数组的有序程度更进一步啦。再缩小增量  $gap=2/2=1$ ，此时，整个数组为1组  $[0,2,1,4,3,5,7,6,9,8]$ ，如下



经过上面的“宏观调控”，整个数组的有序化程度成果喜人。

此时，仅仅需要对以上数列简单微调，无需大量移动操作即可完成整个数组的排序。

0 1 2 3 4 5 6 7 8 9

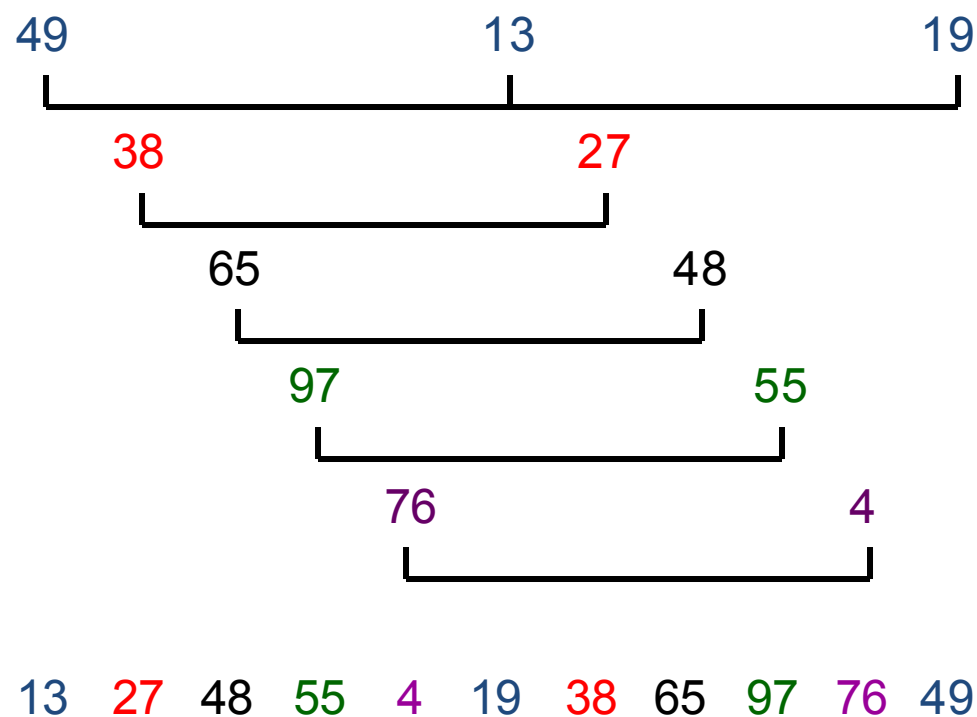


## 2. 希尔排序实例

例，序列 49 38 65 97 76 13 27 48 55 4 19

第一趟排序

$d=5$

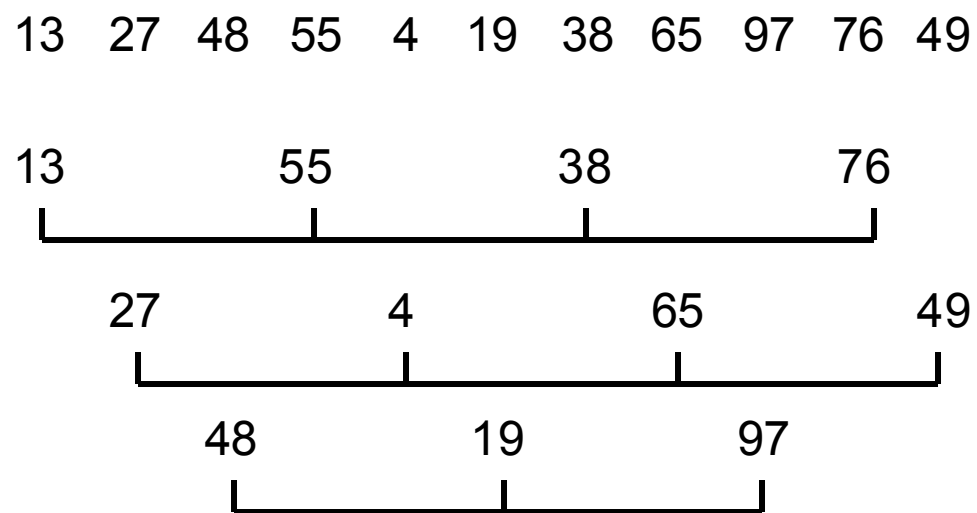




## 2. 希尔排序实例

第二趟排序

$d=3$



13 4 19 38 27 48 55 49 97 76 65

第三趟排序

$d=1$

4 13 19 27 38 48 49 55 65 76 97

## 8.2 插入排序

### 8-2-2 希尔排序

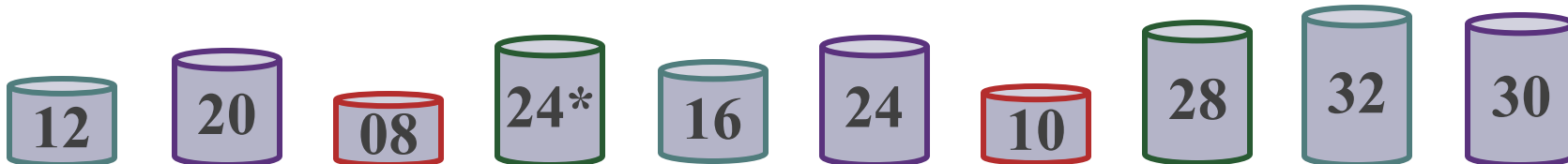


#### 2. 希尔排序实例

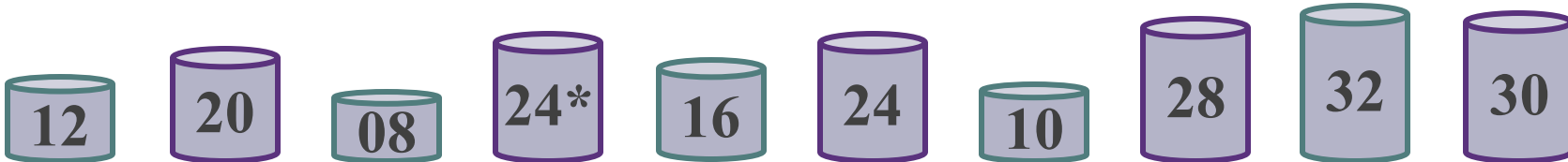
待排序序列



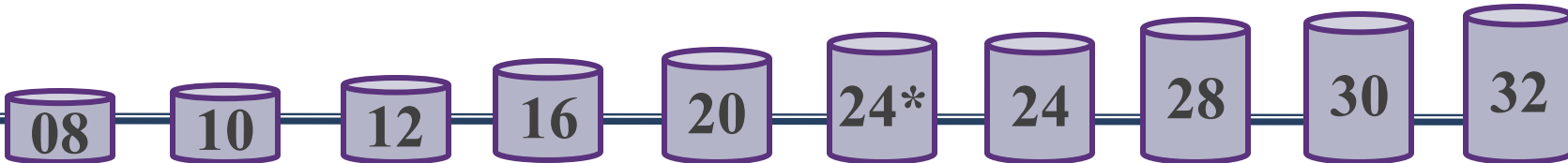
增量  $d = 4$

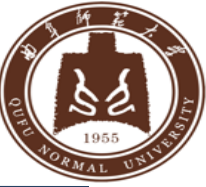


增量  $d = 2$



增量  $d = 1$





### 3. 希尔排序的实现

```
void Sort :: ShellSort( )
{
    int d, i, j, temp;
    for (d = length/2; d >= 1; d = d/2)           //增量为d进行直接插入排序
    {
        for (i = d; i < length; i++)               //进行一趟希尔排序
        {
            temp = data[i];                          //暂存待插入记录
            for (j = i - d; j >= 0 && temp < data[j]; j = j - d)
                data[j + d] = data[j];               //记录后移d个位置
            data[j + d] = temp;
        }
    }
}
```

## 8.2 插入排序

### 8-2-2 希尔排序

#### 3. 希尔排序性能分析

📜 时间性能:  $O(n^2) \sim O(n\log_2 n)$

- (1) 希尔排序算法的时间性能是所取增量的函数;
- (2) 研究表明, 希尔排序的时间性能在  $O(n^2)$  和  $O(n\log_2 n)$  之间;
- (3) 如果选定合适的增量序列, 希尔排序的时间性能可以达到  $O(n^{1.3})$ 。

📜 空间性能:  $O(1)$ ——暂存单元

📜 稳定性: 不稳定

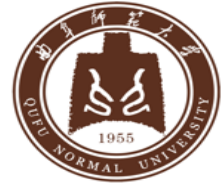
待排序序列



增量  $d = 4$







### 3. 希尔排序性能分析

	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
8-间隔	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
4-间隔	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
2-间隔	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
1-间隔	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16



增量元素不互质，则小增量可能根本不起作用。

## 小结

1. 掌握直接插入排序算法及实现
2. 理解直接插入排序算法的改进方法
3. 掌握希尔排序的基本思路
4. 掌握排序算法性能分析的基本方法

## 小结

1. 掌握直接插入排序算法及实现
2. 理解直接插入排序算法的改进方法
3. 掌握希尔排序的基本思路
4. 掌握排序算法性能分析的基本方法

已知数据序列为：{11, 7, 9, 25, 8, 36, 24, 16, 28, 25\*},

(1) 写出**直接插入排序算法**每趟的结果。

(2) 令  $d_1 = \lfloor n/2 \rfloor$ ,  $d_{i+1} = \lfloor d_i/2 \rfloor$ , 且增量序列互质,  $i$  为排序趟数,

$n$  为数据序列元素个数, 给出**希尔排序**的增量序列和每趟的结果。



*Thank You !*

*Q & A*