程序设计与算法基础2
数据结构与算法

主讲教师： 刘峤

第2章 排序(上)

第2章 内容提要

★ 这一章主要介绍排序算法的四种设计思路

- 插入、交换、选择、归并
- 算法改进：希尔排序（插入）、快速排序（交换）
- 学习目的：排序是基本操作；进一步理解算法复杂度

★ 其中部分算法的实现需要其他数据结构的支持

- 堆排序，树形选择排序、基数排序
- 这几个算法推迟到相应的章节讲解

排序的基本概念

- 排序的定义
  - 将一个数据元素（记录）的任意序列
  - 重新排列成一个按关键字有序的序列，称为排序
- 设：给定一个包含n个记录的序列｛**R1, R2, …， Rn**｝
  - 其相应的关键字序列为 ｛**K1, K2, …，Kn**｝
  - 这些关键字之间存在偏序关系（可相互比较）

$$K_{p1} \le K_{p2} \le \dots \le K_{pn}$$

  - 按此偏序关系将上式记录序列重新排列为

$$\{ R_{p1}, R_{p2}, \dots, R_{pn} \}$$

  - 将上述操作称为排序

排序的基本概念

- 内部排序与外部排序
  - 是否存在内外存交换
- 稳定排序和不稳定排序
  - 对于任意的数据元素序列
  - 若在排序前后相同关键字数据的相对位置都保持不变
  - 这样的排序方法称为稳定的排序方法
  - 否则称为不稳定的排序方法
  - 例如：对于关键字序列 **3**，**2**，**3**，**4**
    - 若某种排序方法排序后变为 **2**，**3**，**3**，**4**
    - 则此排序方法就不稳定

排序算法的四种设计思路

交换、插入、选择、归并

交换法：冒泡排序（回顾）

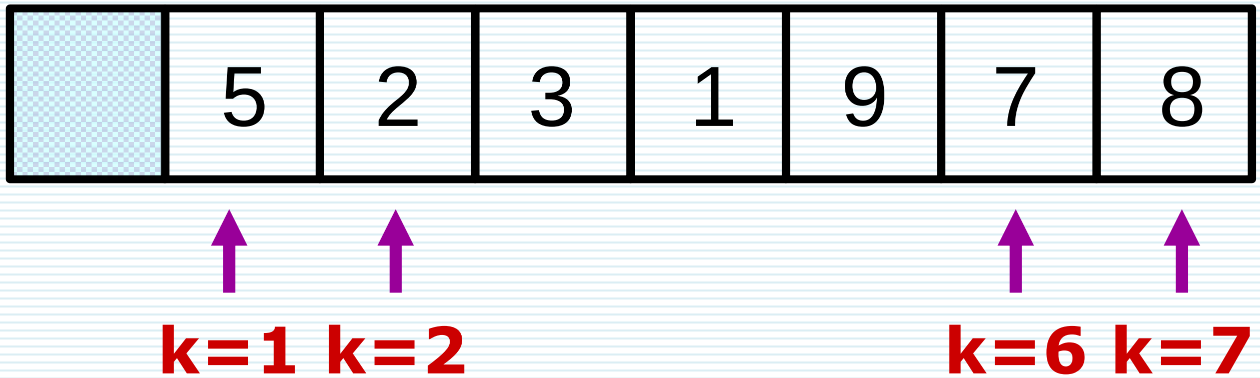
冒泡排序（Bubble Sort)

- 第一趟冒泡
  - 将第一个记录与第二个记录的关键字进行比较
    - 若为逆序：r[1].key>r[2].key，则交换记录值
  - 然后比较第二个记录与第三个记录，依次类推……
    - 直至第n-1个记录和第n个记录比较为止
  - 结果使关键字最大的记录被安置在最后一个记录上
- 对前n-1个记录进行第二趟冒泡排序
  - 结果使关键字次大的记录被安置在第n-1个记录位置
- 重复上述过程
  - 直到在一趟排序过程中没有进行过交换记录的操作为止

冒泡排序

```
// 待排序序列为R[1..n](R[0]闲置)
void bubble_sort (int* R, int n){
    int i, j, flag;
    for( i = 1; i < n; ++i ){
        flag = 0;  // 元素交换标志
        for( j = 1; j <= (n-i); ++j ){
            if( R[j] > R[j+1] ){
                R[0] = R[j+1];  R[j+1] = R[j];
                R[j] = R[0];  flag = 1; // 发生交换
            }
        }
        if( flag == 0) break;
    }
}
```

冒泡排序算法的改进



- 冒泡排序的结束条件为：最后一轮没有发生“记录交换”

  for (j = 1; j <= n-i; j++) if (R[j+1] < R[j]) {元素交换}

- 一般情况下：每经过一轮冒泡，k=(n-i) 的值减1
  - 但并不是在任何情况下都需要逐一递减 **k**！

改进的冒泡排序算法

```
// 待排序序列为R[1..n](R[0]闲置)
void bubble_sort (int* R, int n){
    int i = n,  j,  idx; // 本轮发生交换的最后一个记录的位置
    while( i > 1 ) {
        idx = 1;
        for( j = 1; j < i; ++j ){
            if( R[j] > R[j+1] ){
                R[0] = R[j+1];  R[j+1] = R[j];
                R[j] = R[0];  idx = j; // 发生交换
            }
        }
        i = idx;
    }
}
```

冒泡排序法算法的特点

- 算法的空间复杂度：**S(n)=O(1)**
- 算法的时间复杂度：**T(n)=O(n^2)**
  - 最好情况下（正序）
    - 比较次数： **n-1 次**
    - 移动次数： **0 次**
  - 最坏情况下（逆序）
    - 比较次数： $\sum_{i=1}^{n-1}(n-i)=\frac{1}{2}(n^2-n)$
    - 移动次数： $3\sum_{i=1}^{n}(n-i)=\frac{3}{2}(n^2-n)$

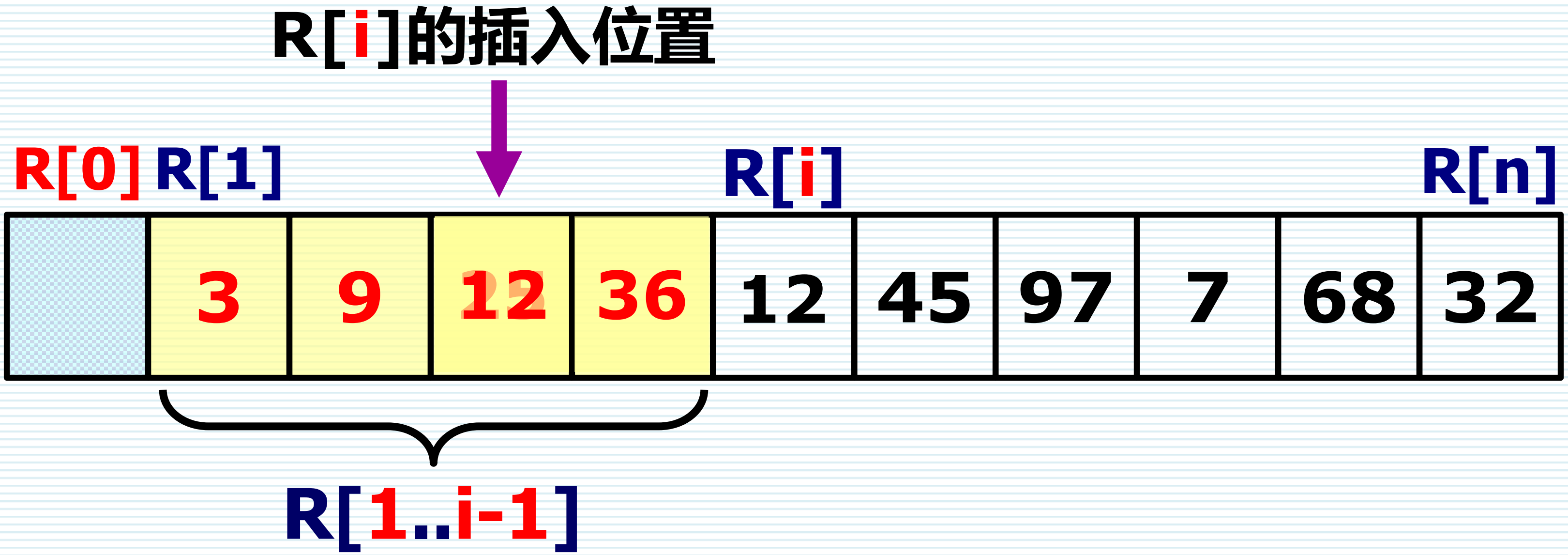
排序算法的四种设计思路

交换、插入、选择、归并

插入法：直接插入排序

直接插入排序（Insertion Sort）

- 利用**顺序查找**实现：在R[1..i-1]中查找R[i]的插入位置



直接插入排序

R[0]	R[1]					R[i]				R[n]
	3	9	12	25	36	45	97	7	68	32

- 排序算法：整个排序过程由n-1轮插入操作构成
  - 首先将序列中第1个记录看成是一个有序子序列
  - 然后从第2个记录开始，逐个将其插入前面的有序子序列
    - 查找过程中找到的那些关键字不小于**R[i]**的记录
    - 在查找的同时实现记录向后移动
  - 直至整个序列有序

直接插入排序

```
// R为顺序表，len为表长（R[0]闲置)
void insert_sort(int *R, int len){
    int i;
    for(i = 2; i <= len; i++){
        insert (R, i);  // 将R[i]插入到R[1..i]合适的位置上
    }
}
```

直接插入排序

```
// R[1..n-1]为有序表，n为表长
void insert (int *R, int n){
    int pos = n; R[0] = R[n];  // 设置监视哨
    // 从右至左查找第一个比R[n]小的数的位置
    while(R[0] < R[pos-1]) {
        R[pos] = R[pos-1];    // 元素移动
        pos--;
    }
    R[pos] = R[0]; // 将R[n]插入到合适的位置
}
```

直接插入排序算法的性能分析

- 空间性能分析　**S(n)=O(1)**
  - 需要一个辅助空间：R[0]
- 时间性能分析　**T(n)=O(n^2)**
  - 实现直接插入排序的基本操作有两个
    - **比较**：序列中两条记录的关键字大小
    - **移动**：序列中的记录以腾出插入位置

直接插入排序算法的性能分析

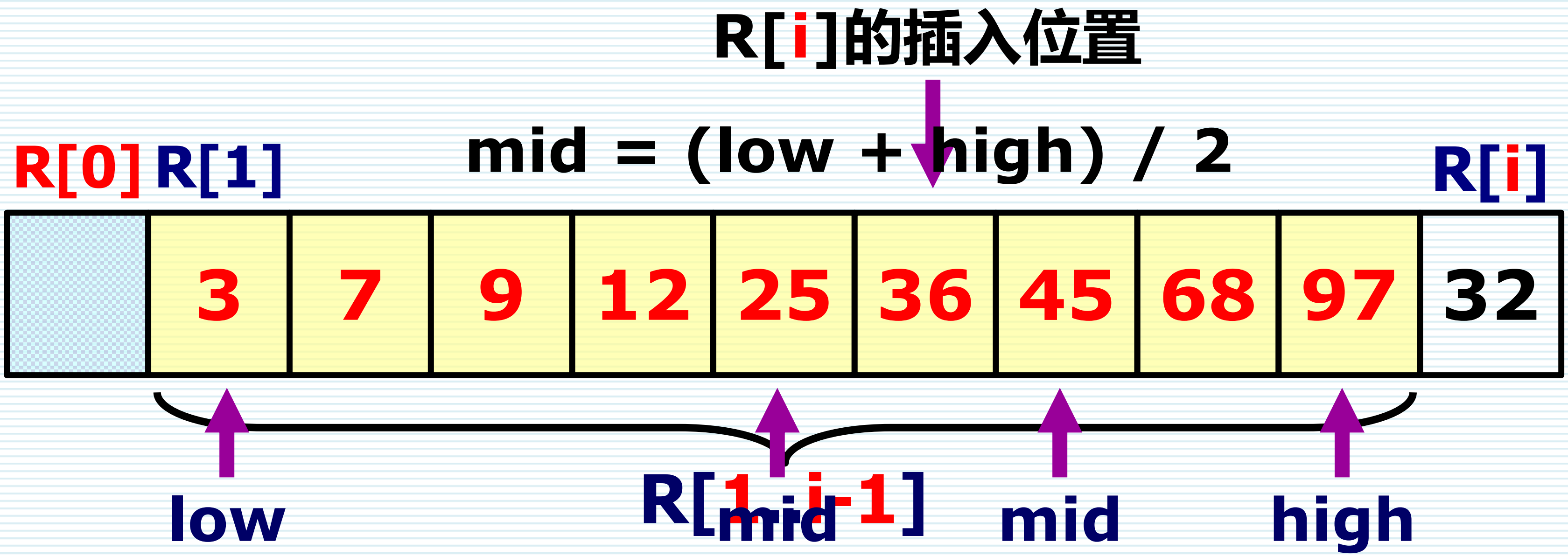
- 时间性能分析(续)　**T(n)=O(n²)**

- 最好的情况：记录序列中关键字按顺序有序
  - 元素比较的次数： $\sum_{i=2}^{n} 1 = n-1$
  - 元素移动的次数：0

- 最坏的情况：记录序列中关键字按逆序有序
  - 元素比较的次数： $\sum_{i=2}^{n} (i-1) = \frac{n(n-1)}{2}$
  - 元素移动的次数： $\sum_{i=2}^{n} (i+1) = \frac{(n+4)(n-1)}{2}$

折半插入排序

- 利用**折半查找**实现：在R[1..i-1]中查找R[i]的插入位置



low > high 时，查找结束

元素右移，完成插入

若：R[i] > R[mid] 则：low = mid + 1

若：R[i] ≤ R[mid] 则：high = mid - 1

折半插入排序

```
// R为顺序表，len为表长（R[0]闲置)
void binary_insert_sort(int *R, int len){
    int i;
    for(i = 2; i <= len; i++){
        binsert (R, i);  // 将R[i]插入到R[1..i]合适的位置上
    }
}
```

折半插入排序

```
// R[1..n-1]为有序表，n为表长
void binsert(int *R, int n){
    int i, mid, low = 1, high = n-1; R[0] = R[n];
    while(low <= high) {   // 查找待插入位置
        mid = (low + high)/2;
        if(R[0] < R[mid])  high = mid - 1;
        else  low = mid + 1;
    }
    for(i = n; i > low; --i){
        R[i] = R[i-1];  // 元素移动
    }
    R[low] = R[0];     // 将R[n]插入到合适的位置
}
```

折半插入排序

- 时间性能分析 **T(n)=O(n²)**
  - 最好的情况：记录序列中关键字按顺序有序
    - 元素比较的次数：**$n\log_2 n$**
    - 元素移动的次数：0
  - 最坏的情况：记录序列中关键字按逆序有序
    - 元素比较的次数：**$n\log_2 n$** **(比较次数得到了改善)**
    - 元素移动的次数：$\sum_{i=2}^{n}(i+1)=\frac{(n+4)(n-1)}{2}$

排序算法的四种设计思路

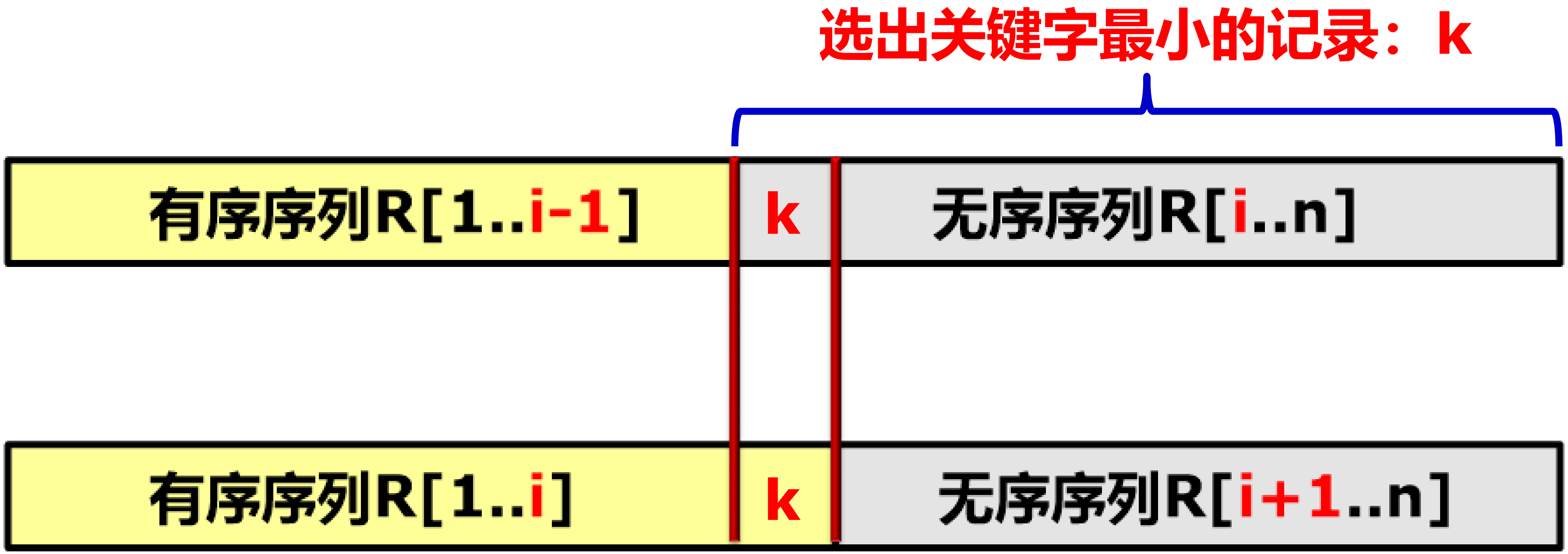
交换、插入、选择、归并

选择法：简单选择排序（回顾）

简单选择排序（Selection Sort）

- 算法基本思想
  - 从无序子序列中选择关键字最小或最大的记录
  - 将其加入到有序子序列中（子序列初始长度为零）
  - 逐步增加有序子序列的长度直至长度等于原始序列
- 排序过程
  - 首先通过n-1次关键字比较，从n个记录中找出关键字最小的记录，将它与第一个记录交换
  - 再通过n-2次比较，从剩余的n-1个记录中找出关键字次小的记录，将它与第二个记录交换
  - 重复上述操作，共进行n-1趟排序后，排序结束

简单选择排序（Selection Sort）



- 选择排序思路：排序过程中
  - 设：第 i-1 趟直接选择排序之后待排记录序列的状态为
  - 则：第 i 趟直接选择排序之后待排记录序列的状态为

k 跟踪最小值

一趟： i=1 [13 38 65 97 76 49 27]

二趟： i=2 13 [27 65 97 76 49 38]

三趟： 13 27 [65 97 76 49 38]

四趟： 13 27 38 [97 76 49 65]

五趟： 13 27 38 49 [76 97 65]

六趟： 13 27 38 49 65 [97 76]

七趟： 13 27 38 49 65 76 [97]

结束： 13 27 38 49 65 76 97

对序列R[1..n]按升序进行简单选择排列

```
void select_sort( int* R, int n ){
    int i, j, k;        // k跟踪每一轮的最小值
    for(i = 1; i < n; ++i){   // n-1轮选择
        k = i;
        for(j = i+1; j <= n; ++j ){ // 找出最小值
            if( R[j] < R[k] )  k = j;
        }
        if( i != k ){   // 元素交换
            R[0] = R[k];  R[k] = R[i];  R[i] = R[0];
        }
    }
}
```

简单选择排序性能分析

- 算法的时间复杂度　**T(n)=O(n^2)**
  - 记录移动次数
    - 最好情况下（正序）：　**0 次**
    - 最坏情况下（逆序）：　**3(n-1)次**
  - 记录比较次数：$\sum_{i=1}^{n-1}(n-i)=\frac{1}{2}(n^2-n)$
- 算法的空间复杂度
  - 只需要一个辅助存储单元：　S(n) = O(1)

排序算法的四种设计思路

交换、插入、选择、归并

归并法：二路归并排序（回顾）

二路归并排序

- **基本思想：**
  - 通过划分子序列，降低排序问题的复杂度
  - 通过合并有序的子序列，得到有序的序列
  - 核心思想：逐步增加记录有序序列的长度
- **2-路归并排序算法**
  - 每次归并操作仅处理两个位置相邻的有序子序列
  - 例如：给定如下关键字序列

6　15　45　23　9　78　35　38　18　27　20

归并排序

分解	6	15	45	23	9	78	35	38	18	27	20
归并	6	15	23	45	9	78	35	38	18	27	20
归并	6	15	23	45	9	35	38	78	18	20	27
归并	6	9	15	23	35	38	45	78	18	20	27
归并	6	9	15	18	20	23	27	35	38	45	78

二路归并排序

```
void merge_sort(int R[], int start, int end){
    int mid;
    if (start < end){
        mid = (start + end) / 2;
        merge_sort(R, start, mid);  ................... T(n/2)
        merge_sort(R, mid+1, end);  ................ T(n/2)
        // 合并相邻的有序子序列
        merge(R, start, mid, end);  .....................Θ(n)
    }
}
```

$$T(n) = 2T(n/2) + \Theta(n)$$

二路归并排序

```
// Ra[h..t]的两部分Ra[h..s]和Ra[s+1..t]已按关键字有序
// Ra[h..s]和Ra[s+1..t]合并成有序表（Rb[s..t]为辅助表)
void merge(int Ra[], int Rb[], int h, int s, int t){
    int i = h, k = h; j = s + 1;
    while( i <= s && j <= t ){
        if(Ra[i] < Ra[j]) Rb[k++] = Ra[i++];
        else  Rb[k++] = Ra[j++];
    }
    while (i <= s) Rb[k++] = Ra[i++];
    while (j <= t) Rb[k++] = Ra[j++];
    for (i = h; i <= t; i++) Ra[i] = Rb[i];
}
```

2-路归并排序递归算法性能分析

- 空间复杂度
  - 2-路归并排序需要一个与原始序列等长的临时数组
  - 因此空间复杂度为O(n)
- 时间复杂度
  - 每一趟归并的时间复杂度为O(n)
  - 总共需要执行的归并次数为：$\log_2 n$
  - 因而，总的时间复杂度为$O(n\log_2 n)$

排序算法的改进

希尔排序、快速排序

插入法：希尔排序

希尔排序（Shell Sort）

算法描述

- 将待排序序列分割成若干个较小的子序列
  - 对各个子序列分别执行直接插入排序
- 当序列达到基本有序时，对其执行一次直接插入排序

算法基本思想

- 对待排记录序列先作宏观调整，再作微观调整
  - 宏观调整：分段执行插入排序
  - 微观调整：对全序列执行一次直接插入排序

希尔排序

{ R[1]，R[1+d]，R[1+2d]，…，R[1+kd] }

{ R[2]，R[2+d]，R[2+2d]，…，R[2+kd] }

… … … … … … … … … … … … …

{ R[d]，R[d+d]，R[d+2d]，…，R[(k+1)d] }

- 例如：将 n 个记录分成 **d** 个子序列：
  - 其中正整数 d 称为增量
    - 它的值在排序过程中从大到小逐渐递减
    - 直至最后一趟排序减为 1

16	25	12	30	47	11	23	36	9	18	31

第一趟希尔排序，设置增量 **d=5**，分为5个子序列

11	23	12	9	18	16	25	36	30	47	31

第二趟希尔排序，设置增量 **d=3**，分为3个子序列

9	18	12	11	23	16	25	31	30	47	36

第三趟希尔排序，设置增量 **d=1**，对整个序列进行排序

9	11	12	16	18	23	25	30	31	36	47

希尔排序

```
// R[]为待排数组；n为待排数组长度（R[0]闲置）
// D[]为增量数组；n为增量数组长度
void shell_sort(int* R, int n, int* D, int m){
    int i, d;
    // 根据增量数组执行m轮分组排序
    for(i = 0; i < m; ++i){
        d = D[i];
        stepwise(R, d, n);
    }
}
```

希尔排序

```
// R[]为待排数组，n为待排数组长度，d为步长
void stepwise(int* R, int d, int n){
    int i, j, k;
    for(i = 1; i <= d; ++i){  // 数据被分成d组
        for(j = i + d; j < n; j += d){
            R[0] = R[j]; k = j;
            while( (k-d) > 0 && (R[0] < R[k-d])) {
                R[k] = R[k-d]; k = k - d;
            }
            R[k] = R[0];
        }
    }
}
```

希尔排序

```
// 修订书上的代码
void stepwise(int* R, int d, int n){
    int i, j, k;
    for(i = 1; i <= d; ++i){ // 数据被分成d组
        for(j = i + d; j < n; j += d){
            R[0] = R[j]; k = j;
            for(k = j; k-d > 0 ; k = k - d){
                if(R[0] < R[k-d]) R[k] = R[k-d];
                else break;
            }
            R[k] = R[0];
        }
    }
}
```

希尔排序

- 算法设计依据
  - 进行插入排序时：若待排序序列“基本有序”
    - 即：序列中具有如下特性的记录数较少

$$R[i] < \max\{ R[k] :\ 1 \le k < i \}$$

    - 则序列中大多数记录都不需要进行插入和元素移动
  - 当序列基本有序时，直接插入排序的效率可大幅提高
    - 空间复杂度为：**O（1）**
    - 时间复杂度接近：**O（n）**

希尔排序

- 为什么希尔排序可提高排序速度?
  - 分组内采用直接插入排序，元素比较次数近似为：$O(n^2)$
    - 从元素比较次数来看：分组后n值减小，n^2更小
    - 从总体上看：元素比较次数大幅减少 ($n^2<<N^2$)
  - 将相隔某个增量的记录组成一个子序列
    - 关键字较小的记录跳跃式前移
    - 最后一轮增量为1的插入排序时，数组已基本有序
    - 所以从总体上看元素移动次数减少
  - 希尔排序的时间复杂度近似等于：**$O(n^{1.3})$**

希尔排序性能分析

- 希尔排序时需要一个存储单元的辅助空间：S(n)=O(1)
- 希尔排序的时间性能与增量因子d_i 有直接关系
  - 取不同步长的时间复杂度不一样（目前尚无最佳取法）
  - Shell建议：$\mathbf{d_1=\lfloor n/2\rfloor}$，$\mathbf{d_{i+1} = \lfloor d_i/2\rfloor}$
  - 但必须满足：最后一个步长一定为1
- 希尔排序是一种不稳定的排序方法
  - 例如：对序列**{4，7，2，9，5，3，<u>2</u>}** 执行希尔排序
  - 结果：**{ <u>2</u>，2，3，4，5，7}**

排序算法的改进

希尔排序、快速排序

插入法：快速排序

快速排序（Quick Sort）

算法基本思想

- 在数组中确定一个记录（的关键字）作为“划分元”
- 将数组中关键字小于划分元的记录均移动至该记录之前
- 将数组中关键字大于划分元的记录均移动至该记录之后
- 由此：一趟排序之后，序列R[s...t]将分割成两部分
  - R[s … i-1] 和 R[i+1 … t]
  - 且满足：R[s … i-1]≤ R[i]≤ R[i+1…t]
  - 其中：R[i] 为选定的“划分元”
- 对各部分重复上述过程，直到每一部分仅剩一个记录为止

快速排序（Quick Sort）

- 首先对无序的记录序列进行一次划分
- 之后分别对分割所得两个子序列“递归”进行快速排序

划分元

无序的记录序列

36	9	12	25	39	45	97	7	68	32

根据选定的划分元（36）进行一次划分

32	9	12	25	7	36	97	45	68	39

无序记录子序列(1)　　　　**无序记录子序列(2)**

对子序列1进行快速排序　　　　对子序列2进行快速排序

快速排序算法流程

36	32	9	12	25	7	36	97	45	68	39

R[0] low high

- 首先：设 R[s]=36 为划分元，将其暂存到R[0]
- 比较 R[high] 和划分元的大小，要求：R[high] ≥ 划分元
- 比较 R[low] 和划分元的大小，要求：R[low] ≤ 划分元
- 若条件不满足，则交换元素，并在low-high之间进行切换
- 一轮划分后得到：（32,9,12,25,7）36（97,45,68,39）

快速排序算法特点

- 时间复杂度：最好情况
  - $T(n)=O(n \log n)$ （每次总是选到中间值作划分元）
- 时间复杂度：最坏情况
  - $T(n)=O(n^2)$ （每次总是选到最小或最大元素作划分元）
  - 解决方案：三者取中，或者随机选取划分元
    - 三者取中：设首记录为R[h]、尾记录为R[t]
    - 取： R[h]、R[t]、R[(h+t)/2] 的中间值为划分元
- 快速排序算法的平均时间复杂度为：O(nlogn)
- 快速排序算法是不稳定的
  - 例如待排序序列： **49** 49 38 65
  - 快速排序结果为： 38 49 **49** 65

快速排序（Quick Sort）

```
void quicksort ( int R[], int low, int high) {
    int idx;
    if(low < high){
        // 调用划分过程将R一分为二，以idx保存“划分元”的位置
        idx = partition(R, low, high);
        quicksort (R, low, idx-1);     // 对低端序列递归
        quicksort (R, idx+1, high);  // 对高端序列递归
    }
}
```

快速排序（Quick Sort）

```
int partition(int R[], int low, int high){
    R[0] = R[low];    // 暂存划分元
    while(low < high){
        while( (low < high) && ( R[high] >= R[0] ) ) high--;
        if( low < high ){
            R[low] = R[high]; low++;
        }
        while( (low < high) && (R[low] <= R[0]))  low++;
        if( low < high ) {
            R[high] = R[low]; high--;
        }
    }
    R[low] = R[0];
    return low;
}
```

快速递归算法性能分析

- 空间复杂度
  - 思考：快速排序是否需要一个与原始序列等长的临时数组?
  - 答案：不需要，因此空间复杂度为O(1)
- 时间复杂度
  - 递归每一层的元素划分时间复杂度为O(n)
  - 总共需要执行的递归次数为：$\log_2 n$
  - 因而，总的时间复杂度为$O(n\log_2 n)$

电子科技大学

UESTC

1956