



电子信息学院



算法与数据结构

(基于现代C++的方法及实践)

ALGORITHM & DATA STRUCTURE  
IN MODERN C++

第5章 排序算法

王文伟 Wang Wenwei, Dr.-Ing.


Tel: 189-71562600

Email: wwwang@aliyun.com

课程QQ群: 珞珈EIS数据结构与算法, 668792335

电子信息学院

Table of Contents



第1章 绪论

第2章 C++编程基础

第3章 遍历、迭代与递归

第4章 字符串

第5章 排序算法

第6章 线性表

第7章 栈与队列

第8章 数组和广义表

第9章 树和二叉树

第10章 图

第11章 查找算法

本章位置

本章介绍排序的基本概念，讨论多种经典排序算法，包括插入、交换、选择和归并等排序算法，并比较各种排序算法的运行效率。

IPL

第5章 排序算法

2

电子信息学院

Table of Contents



5.0 简介

5.1 数据序列及其排序

5.2 插入排序

5.3 交换排序

5.4 选择排序

5.5 归并排序

IPL

第5章 排序算法

3

5.0 Introduction

- 有序的数据便于处理，例如提高查找的效率。
- 排序(sort)**：排序操作是将某种数据结构按数据元素的**关键字值**的大小以递增或递减的次序排列的过程，它是计算机数据处理中的重要基本操作，有着广泛的应用。
- 本章介绍排序的基本概念，讨论多种经典排序算法，包括**插入、交换、选择、归并**等算法，并比较各种排序算法的运行效率。

IPL

第5章 排序算法

4

5.1 数据序列及其排序

5.1.1 排序操作相关基本概念

1. 数据序列、**关键字**和**排序**

2. 内排序与外排序

3. 排序算法的**性能评价**

4. 排序算法的**稳定性**

5.1.2 C++标准库中的排序算法

IPL

第5章 排序算法

5

5.1.1 排序操作相关基本概念

- 数据序列**：数据序列（data series）是特定数据结构中的一系列元素，是待加工处理的数据元素的有限集合。
- 排序是将**数据序列或数据结构**按数据元素的关键字的值以递增或递减的次序排列的过程。
- 数据序列的**排序**建立在元素间的**比较**操作基础之上。以数据元素某个**数据项**作为**比较和排序**依据，则该数据项称为**排序关键字**（sort key）。
- 如果某一关键字能唯一地标识一个数据元素，则称这样的关键字为**主关键字**(primary key)。用主关键字进行排序会得到唯一确定的结果。依据**非主关键字**排序的结果可能不是唯一的。

IPL

第5章 排序算法

6

## 内排序与外排序

- ◆ 根据被处理的数据规模大小，排序过程中涉及的存储器类型可能不同。
- ◆ **内排序**：如果待排序的数据序列的数据元素个数较少，在整个排序过程中，所有的数据元素可以同时保留在内存中。
- ◆ **外排序**：待排序的数据元素非常多，它们必须存储在磁盘等外部存储介质上，在整个排序过程中，需要多次访问外存逐步完成数据的排序。

内排序是基础，外排序建立在内排序的基础之上，但增加了一些复杂性。

TPL

第5章 排序算法

7

## 排序算法的性能评价

- ◆ **排序算法的时间复杂度**：数据排序的基本操作是数据元素的**比较与移动**，分析某个排序算法的时间复杂度，就是要确定该算法执行中的数据元素比较次数或数据元素移动次数与待排序数据序列的长度之间的关系。
- ◆ **排序算法的空间复杂度**：数据的排序过程需要一定的内存空间才能完成，这包括待排序数据序列本身所占用的内存空间，以及其他**附加的内存空间**。分析某个排序算法的空间复杂度，就是要确定该算法执行中，所需**附加内存空间**与待排序数据序列的长度之间的关系。

TPL

第5章 排序算法

8

## 排序算法的稳定性

- ◆ 用**主关键字**进行排序会得到唯一的结果，而用**非主关键字**进行排序，结果不是唯一的。
- ◆ 在数据序列中，如果有两个数据元素 $r_i$ 和 $r_j$ ，它们的关键字（**非主关键字**） $k_i$ 等于 $k_j$ ，且在未排序时， $r_i$ 位于 $r_j$ 之前。如果排序后，元素 $r_i$ 仍在 $r_j$ 之前，则称这样的排序算法是稳定的（**stable**），否则是不稳定的排序算法。

TPL

第5章 排序算法

9

## 各种排序算法的定义

- ◆ 本章仅讨论**内排序问题**：待排序数据保存在一个**数组**中；一般是按关键字值**非递减**的次序对数据进行排序。
- ◆ 关键字为某种**可比较**的类型：如int、double、string等，及已对运算符“<”进行了重载的复合类型（C++）。C#中实现IComparable接口的类型。或用Lambda表达式定义比较规则。
- ◆ 各排序算法定义在SortAlgorithms模块中，以全局函数模板的形式提供。

```
template<typename T> void InsertSort(T* items, int cnt);
template<typename T> void ShellSort(T* items, int cnt);
template<typename T> void QuickSort(T* items, int cnt, int nLower, int nUpper);
template<typename T> void SelectSort(T* items, int cnt);
template<typename T> void HeapSort(T* items, int cnt);
template<typename T> void MergeSort(T* items, int cnt);
```

TPL

第5章 排序算法

10

### 5.1.2 C++标准库中的排序算法

- ◆ C++标准库的algorithm模块以具有多种重载形式的sort函数提供排序功能。函数sort()应用QuickSort算法，时间复杂度为 $O(n\log 2n)$ ，但属于不稳定排序，亦即，如果两元素相等，则其原顺序在排序后可能会发生改变。
- ◆ algorithm模块中还包括有stable\_sort()函数，应用MergeSort算法进行排序，时间复杂度为 $O(n\log 2n)$ ，而且是稳定的排序，亦即，如果两元素相等，则其原顺序在排序后保持不变。

TPL

第5章 排序算法

11

### C++标准库中的排序算法（II）

```
template<class Iterator>
void sort(Iterator first, Iterator last);
对[first, last)范围内的元素进行排序。序列的元素类型需是可比较的，即该类型通过对运算符“<”进行重载来定义元素间的比较协议，据此对整个序列进行排序。
```

```
template<class Iterator, class Compare>
void sort((Iterator first, Iterator last,
          Compare pred);
```

参数pred表示排序所依据的比较规则，常用Lambda表达式定义的匿名函数，在语义上表达出元素之间的比较规则，sort函数使用指定的“比较”规则进行排序。

TPL

第5章 排序算法

12

例：对数组进行排序

```
const int CNT = 20;
int a[CNT];
RandomizeData(a, CNT);
cout << "sort by natural order: ";
sort(a, a+CNT); Show(a, a + CNT);
cout << "sort by abs: ";
sort(a, a+CNT, [](int x, int y)
    {return abs(x) < abs(y); });
Show(a, a + CNT);
```

[例5.1] 学生信息表的定义与排序演示

```
struct Student {
    int id; string name; float age; double score;
    Student():name("no name") {id=0;age=0.0; score=-1.0;}
    Student(int iid, const char* na, float age, double sc):
        name(na) {id = iid; this->age; score = sc; }
    Student(const Student& s):name(s.name) { //copy
        id=s.id; age=s.age; score=s.score; } //constructor
    const Student& operator=(const Student& rhs) {
        if (this != &rhs) { //copy assignment operator
            id = rhs.id; age = rhs.age;
            score = rhs.score; _name = rhs.name;
        }
        return *this;
    }
    bool operator<(const Student& y) const {
        return id < y.id;
    }
};
```

定义所需的比较规则: Comparison型函数对象

```
enum class CompareKey {ID, Name, Score, IDD, NameD, ScoreD };
function<bool>(Student&, Student&)>
    ComparisonBy(CompareKey k = CompareKey::ID) {
    function<bool>(Student&, Student&)> cmp;
    switch (k) {
        case CompareKey::Name:
            cmp = [](Student& x, Student& y)
                {return x._name < y._name; }; break;
        case CompareKey::IDD:
            cmp = [](Student& x, Student& y)
                {return y._studentID < x._studentID; }; break;
        ..... default:
            cmp = [](Student& x, Student& y)
                {return x._studentID < y._studentID; }; break;
    }
    return cmp; };
```

按不同关键字进行排序的实例

```
vector<Student> items; SetData(items); Show(items);
cout << "按学号排序:" << endl;
sort(items.begin(), items.end()); Show(items);
sort(begin(items), end(items), ComparisonBy(CompareKey::Score));
cout << "按成绩排序:" << endl; Show(items);
sort(items.begin(), items.end(), ComparisonBy(CompareKey::Name));
cout << "按姓名排序:" << endl; Show(items);
sort(items.begin(), items.end(), ComparisonBy(CompareKey::IDD));
cout << "按学号倒排序:" << endl; Show(items);
sort(items.begin(), items.end(),
    ComparisonBy(CompareKey::ScoreD));
cout << "按成绩倒排序:" << endl; Show(items);
sort(items.begin(), items.end(),
    ComparisonBy(CompareKey::NameD));
cout << "按姓名倒排序:" << endl; Show(items);
```

## 5.2 插入排序

- ◆ **insertion sort**: 基于简单的基本思想, 将待排序的数据依次有序地插入成一个有序的数据序列。该算法将整个数据序列视为由两个子序列组成: 处于前面的**已排序子序列**和处于后面的**待排序子序列**; 分趟将一个待排序元素, 按关键字大小, 插入到**已排序的数据序列**中, 从而得到一个新的、元素个数增1的有序序列, 重复该过程直到全部元素插入完毕。

### 5.2.1 直接插入排序

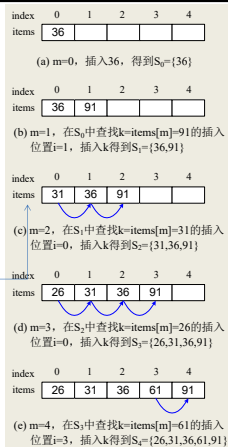
### 5.2.2 希尔排序算法

### 5.2.1 直接插入排序straight insertion sort

- ◆ **分趟**将待排序的数据依次**有序地插入**成一个有序的数据序列。
- ◆ 核心思想: 在**第 $m$ 趟**插入第 $m$ 个数据元素 $k$ 时, 前 $m-1$ 个数据元素已组成有序数据序列 $S_{m-1}$ , 将 $k$ 与 $S_{m-1}$ 中各数据元素依次进行比较并插入到适当位置, 得到新的序列 $S_m$ 仍是有序的。

$$S_{m-1} + k = S_m$$

1. 初始化：以 $\text{items}[0]=36$ 建立有序子序列 $S_0=\{36\}$ ,  $m=1$ 。
2. 在第 $m$ 趟，欲插入元素值 $k = \text{items}[m]$ ，在 $S_{m-1}$ 中进行顺序查找，找到 $k$ 值应插入的位置 $i$ ；从序列 $S_{m-1}$ 末尾开始到 $i$ 位置的元素依次向后移动一位，空出位置 $i$ ；将 $k$ 置入 $\text{items}[i]$ ，得到有序子序列 $S_m$ ,  $m++$ 。例如，当 $m=1$ 时， $k=91$ ,  $i=1$ ,  $S_1=\{36, 91\}$ 。当 $m=2$ 时， $k=31$ ,  $i=0$ ,  $S_2=\{31, 36, 91\}$
3. 重复步骤2，依次将其其他数据元素插入到已排序的子序列中。



```
template <typename T>
void InsertSort(T* items, int cnt) {
    T k; int m, n = cnt; int i, j;
    for (m = 1; m < n; m++) {
        k = items[m];
        for (i = 0; i < m; i++) {
            if (k < items[i]) {
                for (j=m-1; j>=i; j--) items[j+1]= items[j];
                items[i] = k;
                break;}
        }
    }
    show(m, items, cnt);
}
```

数据序列:	83	63	78	72	73	17	46	59
第1趟排序后:	63	83	78	72	73	17	46	59
第2趟排序后:	63	78	83	72	73	17	46	59
第3趟排序后:	63	72	78	83	73	17	46	59
第4趟排序后:	63	72	73	78	83	17	46	59
第5趟排序后:	17	63	72	73	78	83	46	59
第6趟排序后:	17	46	63	72	73	78	83	59
第7趟排序后:	17	46	59	63	72	73	78	83

◆ 数据的排序过程包含的基本操作是数据的比较与移动。

平均比较次数  $C = \sum_{m=1}^{n-1} \frac{m+1}{2} = \frac{1}{4}n^2 + \frac{1}{4}n - \frac{1}{2} \approx \frac{n^2}{4}$

平均移动次数  $M = \sum_{m=1}^{n-1} \frac{m}{2} = \frac{n(n-1)}{4} \approx \frac{n^2}{4}$

直接插入排序算法的时间复杂度为 $O(n^2)$

空间复杂度为 $O(1)$

直接插入排序算法是稳定的

思考题：可以用**二分查找**算法代替**顺序查找**算法完成在有序子表中查找一个数据元素的工作，这样可以降低平均比较次数，但不能减少移动次数。

```
template <typename T>
void InsertSortBS(T* items, int cnt) {
    T k; int i, j, m, n = cnt;
    for (m = 1; m < n; m++) {
        k = items[m];
        i = BinarySearch(k, items, cnt, 0, m);
        if (i < 0) i = ~i;
        for (j = m-1; j >= i; j--)
            items[j+1]=items[j];
        items[i] =k;
        show(m, items, cnt);
    }
}
```

- ◆ 直接插入排序每次比较的是相邻的数据元素，一趟排序后数据元素最多移动一个位置。
- ◆ Shell sort 又称**缩小增量排序**（diminishing increment sort），其基本思想是：先将整个序列分割成若干子序列分别进行排序，待整个序列基本有序时，再进行全序列的直接插入排序，这样可使排序过程加快。
- ◆ 希尔排序算法在排序之初，将相隔较远的若干元素归为一个子序列，因而进行比较的是相隔较远的元素，使得数据元素移动时能够跨越多个位置；然后逐渐减少被比较数据元素间的距离（缩小增量），直至距离为1时，各数据元素都已按序排好。

排序items={36, 91, 31, 26, 61, 37, 97, 1, 93, 71}

1. jump=n/2=5, j从第0个位置元素开始, 将相隔jump的元素items[j]与items[j+jump]进行比较。如果反序, 则交换, 依次重复进行完一趟排序, 得到序列{36, 91, 1, 26, 61, 37, 97, 31, 93, 71}。
2. jump=2, 相隔jump的元素组成子序列{36, 1, 61, 97, 93}和子序列{91, 26, 37, 31, 71}。在子序列内比较元素items[j]与items[j+jump], 如果反序, 则交换, 依次重复。得到序列{1, 26, 36, 31, 61, 37, 93, 71, 97, 91}。
3. jump=1, 在全序列内比较元素items[j]与items[j+jump], 如果反序, 则交换, 得到序列{1, 26, 31, 36, 37, 61, 71, 91, 93, 97}。

IPL

第5章 排序算法

25

## 程序运行结果与算法分析

数据序列: 36 91 31 26 61 37 97 1 93 71  
 jump=4 第1趟排序后: 36 36 18 21 52 71 52 54  
 jump=2 第2趟排序后: 18 21 36 36 52 52 54 71  
 jump=1 第3趟排序后: 18 21 36 36 52 52 54 71

- 若增量的取值比较合理, 希尔排序算法的时间复杂度为约 $O(n(\log_2 n)^2)$ 。
- 希尔排序算法的空间复杂度为 $O(1)$ 。
- 希尔排序算法是一种不稳定的排序算法。

IPL

第5章 排序算法

26

## 5.3 交换排序

- ◆ 基于交换的排序算法有两种: 冒泡排序(bubble sort)和快速排序(quick sort)。

5.3.1 冒泡排序: 经典的交换排序算法

5.3.2 快速排序: 平均性能较好的一种排序算法, C++标准库算法模块中的sort()函数采用quick sort算法进行排序。

IPL

第5章 排序算法

27

### 5.3.1 冒泡排序

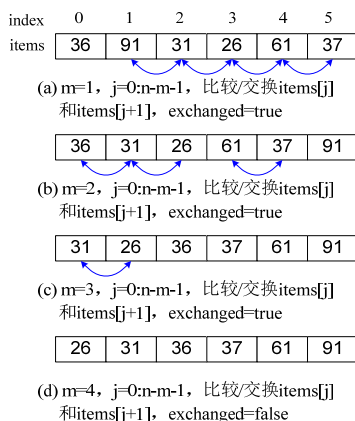
- ◆ 基本思想: 依次比较相邻的两个数据元素, 反序则交换位置。经过一趟排序后, 最大值元素移到最后位置, 值较小的数据元素向最终位置移动一位(一趟起泡)。
- ◆ 对于有n个数据元素的数据序列, 最多需n-1趟排序, 第m趟对从位置0到位置n-m-1的数据元素与其后一位的元素进行比较、交换, 因此冒泡排序算法用二重循环实现。

IPL

第5章 排序算法

28

排序  
36,  
91,  
31,  
26,  
61,  
37



### 冒泡排序算法

```
void BubbleSort(T* items, int cnt) {
    T t; bool exchanged=false; int n = cnt;
    for (int m = 1; m < n; m++) {
        exchanged = false;
        for (int j = 0; j < n - m; j++) {
            if (items[j + 1] < items[j]) {
                t=items[j]; items[j]= items[j+1];
                items[j+1]=t; exchanged=true;
            }
        }
        show(m, items, cnt);
        if (!exchanged)break;
    }
}
```



## 冒泡排序算法分析

- ◆ **时间复杂度**: 用两重循环分趟实现交换排序算法，外循环控制排序趟数，内循环查找倒序元素与交换。
- ◆ **在最好的情况下**，如果序列已排序，只需一趟排序即可，进行比较操作的次数为 $n-1$ ，移动次数为0，算法的时间复杂度为 $O(n)$ ；
- ◆ **最坏的情况**是：序列已按反序排列，需要 $n-1$ 趟排序，每趟过程中比较和移动的次数均为 $n-m$ ，算法的时间复杂度为 $O(n^2)$ 。
- ◆ 平均而言，冒泡排序算法的时间复杂度为 $O(n^2)$ 。

IPL

第5章 排序算法

31

## 冒泡排序算法的空间复杂度与稳定性

- ◆ **空间复杂度**: 冒泡排序中，因交换两个数据元素需要一个辅助空间，这与序列的长度无关，故空间复杂度为 $O(1)$ 。
- ◆ **稳定性**: 从交换的过程易看出，对于关键字相同的元素，排序不会改变它们原有的次序，故冒泡排序是稳定的。

IPL

第5章 排序算法

32

## 5.3.2 快速排序

- ◆ 基本思想：将长序列以其中的某值为基准（这个值称作枢纽pivot）分成两个独立的子序列，第一个子序列的元素均比pivot小，第二个子序列则比它大；分别对两个子序列继续进行排序，直到整个序列有序。



每趟排序过程中，将找到基准值pivot在序列中的最终排序位置，并据此将原序列分成两个小序列。

IPL

第5章 排序算法

33

## 具体方法

- ◆ 在待排序的数据序列中任意选择一个元素（如第一个元素）作为基准值pivot，由序列的两端交替地向中间进行比较、交换，使得所有比pivot小的元素都交换到序列的左端，所有比pivot大的元素都交换到序列的右端，这样序列就被划分成三部分：左子序列，pivot和右子序列。再对两个子序列分别进行同样的操作，直到子序列的长度为1。
- ◆ 每趟排序过程中，将找到pivot在最终排好序的序列中的应有位置，并据此将原序列分成两个小序列。



IPL

第5章 排序算法

34

## 快速排序的算法

```
void QuickSort(T* items, int cnt, int nLower=0, int nUpper=-1) {
    if (nUpper == -1) nUpper = cnt - 1;
    if (nLower < nUpper) { // Split and sort partitions
        int nSplit = Partition(items, cnt, nLower, nUpper);
        cout << "left=" << nLower << "right=" << nUpper << "Pivot=" << nSplit << "\t";
        show(0, items, cnt);
        QuickSort(items, cnt, nLower, nSplit-1);
        QuickSort(items, cnt, nSplit+1, nUpper);
    }
}
```

在Partition方法中，选取第一个元素为pivot，分别从序列的最左、右端向中间扫描。在左端发现大于pivot或右端发现小于pivot的元素，则交换到另一端，并收缩两端的范围，最终确定pivot应该具有的位置。最后将pivot交换到该位置，并将该位置值作为方法结果返回。

IPL

第5章 排序算法

35

index	0	1	2	3	4	5
items	36	91	31	26	61	37
(a) 原序列						
items	31	26	36	91	61	37
(b) left=0, right=5, pivot=36, nSplit=2, 进入左分支						
items	26	31	36	91	61	37
(c) left=0, right=1, pivot=31, nSplit=1, 该部停止						
items	26	31	36	37	61	91
(d) left=3, right=5, pivot=91, nSplit=5, 进入右分支						
items	26	31	36	37	61	91
(e) left=3, right=4, pivot=37, nSplit=3, 该部停止						
items	26	31	36	37	61	91

经一趟排序后，原序列分为两个子序列，分别为[nLower, nSplit-1]和[nSplit+1, nUpper]。

left=0 right=5 Pivot=2  
数据序列: 31 26 36 91 61 37  
left=0 right=1 Pivot=1  
数据序列: 26 31 36 91 61 37  
left=3 right=5 Pivot=5  
数据序列: 26 31 36 37 61 91  
left=3 right=4 Pivot=3  
数据序列: 26 31 36 37 61 91  
排序后数据序列: 26 31 36 37 61 91

## 快速排序算法分析

- 快速排序的效率与序列的**初始排列**及基准值的选取有关。
- 最坏情况**是：当序列已排序时，如{1, 2, 3, 4, 5, 6, 7, 8}，如选取序列的第一个值作为基准，那么所分的两个子序列将分别是{1}和 {2, 3, 4, 5, 6, 7, 8}，仍然是已排序的；必须经过 $n-1$ 趟才能完成最终的排序。时间复杂度为 $O(n^2)$ ，排序速度已退化，比冒泡法还慢。
- 较坏情况**是：一般而言，对于接近已排序的数据序列，快速排序算法的时间效率不理想。
- 最好情况**是，每趟排序将序列分成两个长度相同的子序列。

## 快速排序算法分析 (II)

- 研究证明，当 $n$ 较大时，对平均情况而言，快速排序名符其实，其**时间复杂度**为 $O(n \log_2 n)$ 。但当 $n$ 很小时，或基准值选取不适当时，会使快速排序的时间复杂度**退化**为 $O(n^2)$ 。
- 在算法实现中，常常以随机方法在待排序的数据序列中选择一个元素，而不是固定选第一个元素，作为初始基准值。
- 快速排序是递归过程，需要在系统栈中传递递归函数的参数及返回地址，算法的**空间复杂度**为 $O(\log_2 n)$ 。
- 快速排序算法是**不稳定**排序算法。

## 5.4 选择排序

- 选择排序算法常用的有两种：直接选择排序 (straight select sort) 和堆排序 (heap sort)。
- 5.4.1 直接选择排序
- 5.4.2 堆排序
- 直接选择排序的基本思想是依次选择出待排序数据中的最小者将其有序排列。
- 具体过程：对于有 $n$ 个元素的待排序数据序列，第1趟排序，比较 $n$ 个元素，找到最小的元素 $items[\min]$ ，将其交换到序列的首位置 $items[0]$ ；第2趟排序，在余下的 $n-1$ 个元素中选取最小的元素，交换到序列的 $items[1]$ ；这样经过 $n-1$ 趟排序，完成 $n$ 个元素的排序。

## 选择排序算法

```
void SelectSort(T* items, int cnt) {
    T t; int minIdx, n = cnt;
    for (int m = 1; m < n; m++) {
        minIdx = m - 1;
        for (int j = m; j < n; j++) {
            if (items[j] < items[minIdx]) minIdx = j;
        }
        if (minIdx != m - 1) {
            t = items[m-1]; items[m-1] = items[minIdx];
            items[minIdx] = t;
        }
        cout << "minIdx= " << minIdx << " ";
        show(m, items, cnt);
    }
}
```

排序

index	0	1	2	3	4
items	26	91	31	36	61

(a) 第 $m=1$ 趟,  $\min=3$ , 交换 $m-1$ 和 $\min$ 项

items	26	31	91	36	61
-------	----	----	----	----	----

(b) 第 $m=2$ 趟,  $\min=2$ , 交换 $m-1$ 和 $\min$ 项

items	26	31	36	91	61
-------	----	----	----	----	----

(c) 第 $m=3$ 趟,  $\min=3$ , 交换 $m-1$ 和 $\min$ 项

items	26	31	36	61	91
-------	----	----	----	----	----

(d) 第 $m=4$ 趟,  $\min=4$ , 交换 $m-1$ 和 $\min$ 项

二重循环实现直接选择排序：

1) 外层for循环控制 $m=1:n-1$ 分趟排序，每趟排序找到一个最小值置于 $items[m-1]$ ；

2) 内层for循环控制 $j=m:n-1$ 在序列剩余元素中查找到最小的元 $items[\min]$ ，然后与 $items[m-1]$ 交换。

## 选择排序算法分析

- 直接选择排序的比较次数与数据序列的初始排列**无关**。对于有 $n$ 个数据元素的待排序数据序列，在第 $m$ 趟排序中，查找最小值所需的比较次数是 $n-m$ 次。所以，直接选择算法总的**比较次数**为：

$$C = \sum_{m=1}^{n-1} (n-m) = \frac{1}{2} n(n-1) \approx \frac{n^2}{2}$$

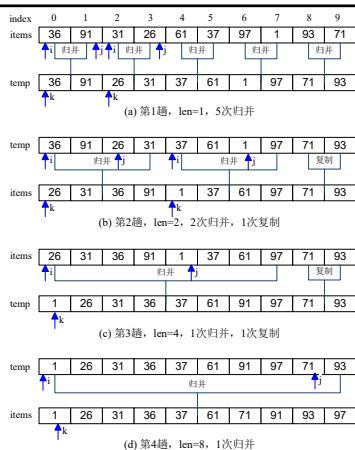
## 选择排序算法分析(II)

- ◆ 数据元素的移动次数与序列的初始排列有关。最好的情况是，数据序列初始已排序，数据移动的次數最少， $M = 0$ 。最坏情况是，每一趟排序都要进行交换，总的數據元素移动次数为  $M = 3 \times (n-1)$ 。
- ◆ 所以，直接选择排序算法的时间复杂度为  $O(n^2)$ 。
- ◆ 它的空间复杂度为  $O(1)$ 。
- ◆ 直接选择排序算法是不稳定的。对于关键字相同的元素，排序会改变它们原有的次序。

## 5.5 归并排序

- ◆ 有序的数据便于处理，如果待排序序列内已存在某种有序性，排序算法利用上这种内在的有序性，那么将加快排序操作的运行。
- ◆ 将两个有序子序列合并，形成一个大的有序序列的过程称为归并（merge），又称两路归并。
- ◆ 对于有  $n$  个元素的待排序数据序列，两路归并排序算法的过程如下：
  1. 将待排序序列看成是  $n$  个长度为1的已排序子序列。
  2. 依次将两个相邻子序列合并成一个大的有序序列。
  3. 重复第2步，合并更大的有序子序列，直到完成整个序列的排序。

排序：  
36,  
91,  
31,  
26,  
61,  
37,  
97,  
1,  
93,  
71



## 归并排序算法实现

```
void MergeSort(T* items, int cnt) {
    int len = 1; // 已排序的序列长度, 初始值为1
    T* temp = new T[cnt];
    do {
        MergePass(items, temp, cnt, len);
        // 将items中元素归并到temp中
        show(0, temp, cnt); len *= 2;
        MergePass(temp, items, cnt, len);
        // 将temp中元素归并到items中
        show(0, items, cnt); len *= 2;
    } while (len < cnt);
    delete [] temp;
}
```

## 测试归并排序算法的程序运行结果

36 91 31 26 61 37 97 1 93 71

len=1 数据序列: 36 91 26 31 37 61 1 97 71 93  
 len=2 数据序列: 26 31 36 91 1 37 61 97 71 93  
 len=4 数据序列: 1 26 31 36 37 61 91 97 71 93  
 len=8 数据序列: 1 26 31 36 37 61 71 91 93 97  
 排序后序列: 1 26 31 36 37 61 71 91 93 97

## 归并排序算法分析

- ◆ Merge方法完成两个有序子序列的归并，需要进行  $O(\text{len})$  次比较。
- ◆ MergePass方法完成一趟归并排序，需要调用Merge方法  $O(n/\text{len})$  次。
- ◆ MergeSort方法实现归并排序算法，需要调用MergePass方法  $O(\log_2 n)$  次。所以，归并算法的时间复杂度为：

$$O(n \log_2 n)$$



## 归并排序算法分析 (II)

- ◆ 归并排序算法在运行过程中需要与存储数据序列的空间相等的辅助空间，所以它的空间复杂度为 $O(n)$ 。
- ◆ 归并排序算法是稳定的，对于关键字相同的元素，排序不会改变它们原有的次序。

## 本章学习要点

1. 了解排序的定义和各种排序方法的特点。熟悉各种方法的排序过程及其依据的原则。
2. 掌握各种排序方法的时间复杂度的分析方法。能分析排序算法的最坏情况/最好情况和平均情况的时间性能。
3. 按平均时间复杂度划分，内部排序可分为： $O(n^2)$ 的简单排序方法， $O(n\log_2 n)$ 的高效排序方法。
4. 理解排序方法“稳定”或“不稳定”的含义，弄清楚在什么情况下要求应用的排序方法必须是稳定的。