

# 十大经典排序算法

## 一、 排序算法说明

### 1. 排序的定义

对一序列对象根据某个关键字进行排序。

### 2. 术语说明

- 稳定：如果 a 原本在 b 前面，而  $a=b$ ，排序之后 a 仍然在 b 的前面；
- 不稳定：如果 a 原本在 b 的前面，而  $a=b$ ，排序之后 a 可能会出现在 b 的后面；
- 内排序：所有排序操作都在内存中完成；
- 外排序：由于数据太大，因此把数据放在磁盘中，而排序通过磁盘和内存的数据传输才能进行；
- 时间复杂度：一个算法执行所耗费的时间；
- 空间复杂度：运行完一个程序所需内存大小。

### 3. 算法总结

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	In-place	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
希尔排序	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$	In-place	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Out-place	稳定
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	In-place	不稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	In-place	不稳定
计数排序	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	Out-place	稳定
桶排序	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$	Out-place	稳定
基数排序	$O(n \times k)$	$O(n \times k)$	$O(n \times k)$	$O(n + k)$	Out-place	稳定

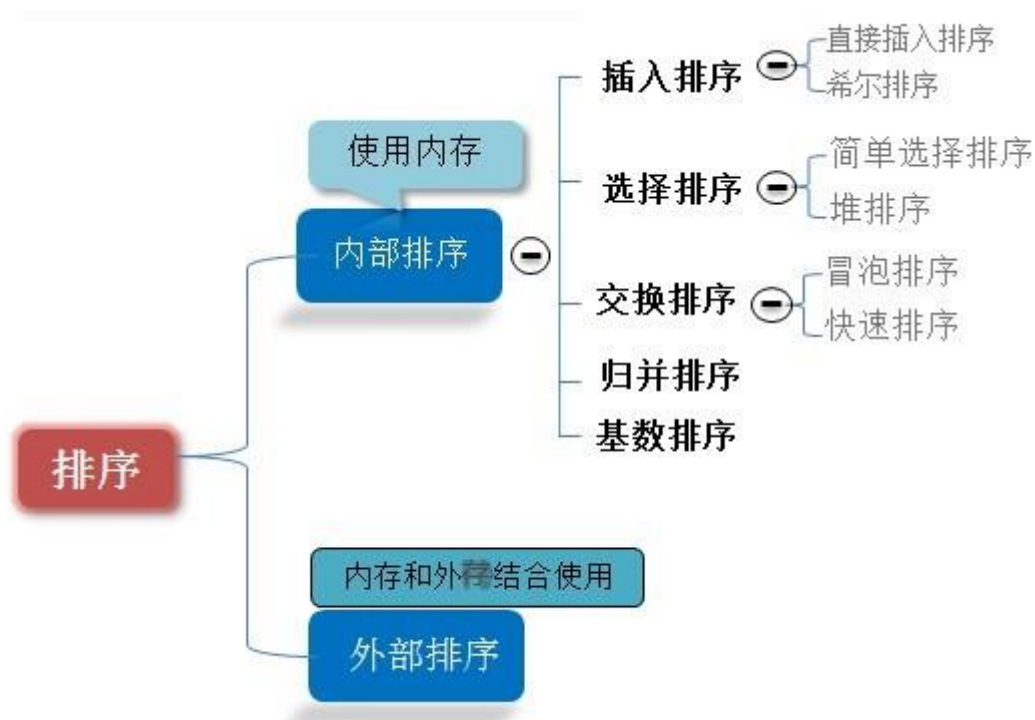
图片名称解释：（错误很多）

- n：数据规模
- k：“桶”的个数
- In-place：占用常数内存，不占用额外内存
- out-place：占用额外内存

类别	排序方法	时间复杂度			空间复杂度	稳定性
		平均情况	最好情况	最坏情况	辅助存储	
插入排序	直接插入	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	Shell排序	$O(n^{1.3})$	$O(n)$	$O(n^2)$	$O(1)$	不稳定
选择排序	直接选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
	堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	不稳定
交换排序	冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	快速排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n^2)$	$O(n\log_2 n)$	不稳定
归并排序		$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	稳定
基数排序		$O(d(r+n))$	$O(d(n+rd))$	$O(d(r+n))$	$O(rd+n)$	稳定

注：基数排序的复杂度中， $r$ 代表关键字的基数， $d$ 代表长度， $n$ 代表关键字的个数。

#### 4. 算法分类



## 二、冒泡排序法

冒泡排序是一种简单的排序算法。它重复地走访过要排序的数列，一次比较两个元素，如果它们的顺序错误就把它们交换过来。走访数列的工作是重复地进行直到没有再需要交换，也就是说该数列已经排序完成。第一趟排序完之后最大的那个数就会落在最后一位。第二趟排序完之后第二大的数会落在倒数第二位，就这样下去每一趟需要排的数都比上一趟少一个。这个算法的名字由来是因为越小的元素会经由交换慢慢“浮”到数列的顶端。

### 1. 算法描述

- 比较相邻的元素。如果第一个比第二个大，就交换它们两个；
- 对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对，这样在最后的元素应该会是最大的数；
- 针对所有的元素重复以上的操作，除了最后一个；
- 重复步骤 1~3，知道排序完成。

### 2. 动画演示

参见文件夹《十大算法动画演示》下的冒泡排序法.gif。

### 3. 代码实现

```
// sort.cpp : 定义控制台应用程序的入口点。

#include "stdafx.h"
#include <iostream>
#include <stdlib.h>
using namespace std;
// BubbleSort: 冒泡排序法; 输入: 指向数组的指针、数组的长度
void BubbleSort(int *arr, int len) { // 通过指针接收数组
    for (int i = 0; i < len; i++) { // 排序的趟数
        for (int j = 0; j < len - i - 1; j++) // 每趟排序中的操作
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j + 1];
                arr[j + 1] = arr[j];
                arr[j] = temp;
            }
    }
}

int main(int argc, char* argv[]) {
    int my_arr[] = {1, 2, 3, 4, 5, 621, 21, 32, 31, 54};
    int len = (sizeof(my_arr) / sizeof(my_arr[0])); // 计算数组的长度
    BubbleSort(my_arr, len);
    for (int m = 0; m < len; m++) {
        std::cout << my_arr[m] << std::endl;
    }
    return 0;
}
```

### 4. 算法分析

---

最佳情况:  $T(n)=O(n)$  最差情况:  $T(n)=O(n^2)$

### 三、 选择排序 (Selection Sort)

表现最稳定的排序算法之一，因为无论什么数据进去都是  $O(n^2)$  的时间复杂度，所以用到它的时候，数据规模越小越好。唯一的好处可能就是不占用额外的内存空间了吧。理论上讲，选择排序可能也是平时排序一般人想到的最多的排序方法了吧。

选择排序(Selection-sort)是一种简单直观的排序算法。它的工作原理：首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置，然后，再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。

#### 1. 算法描述

$n$  个记录的直接选择排序可经过  $n-1$  趟直接选择排序得到有序结果。具体算法描述如下：

- 初始状态：无序区为  $R[1\cdots n]$ ，有序区为空；
- 第  $i$  趟排序 ( $i=1, 2, 3, \cdots, n-1$ ) 开始时，当前有序区和无序区分别为  $R[1, \cdots i-1]$  和  $R[i\cdots n]$ 。该趟排序从当前无序区中选出关键字最小的记录  $R[k]$ ，将它与无序区的第 1 个记录  $R$  交换，使  $R[1\cdots i]$  和  $R[i+1\cdots n]$  分别变为记录个数增加 1 个的新有序区和记录个数减少 1 个的新无序区；
- $n-1$  趟结束，数组有序化了。

#### 2. 动画演示

参见文件夹《十大算法动画演示》下的[选择排序法.gif](#)。

#### 3. 代码实现

```
// SelectionSort.cpp : 定义控制台应用程序的入口点。
#include "stdafx.h"
#include <iostream>
#include <stdlib.h>

using namespace std;

// SelectionSort: 选择排序法; 输入: 指向数组的指针、数组的长度
void SelectionSort(int *arr, int len) {
    int temp;
    for (int i = 0; i < len; i++) {
        temp = *(arr+i); // 记录下本趟排序第一个数的值
```

```

        int index = i; // 初始化最小值的下标为i
        for (int j = i+1; j < len; j++) { // 寻找最小值
            if ((*arr+j) < temp) { // 发现更小的值
                temp = *(arr+j); // 将其作为最小值
                index = j; // 记录下最小值的下标
            }
        }
        *(arr+index) = *(arr+i); // 将最小值与本趟第一个数进行互换
        *(arr+i) = temp;
    }
}

int main(int argc, char* argv[])
{
    int my_arr[] = {1, 5, 4, 4, 5, 621, 21, 32, 31, 54};
    int len = (sizeof(my_arr)/sizeof(my_arr[0])); // 计算数组的长度
    SelectionSort(my_arr, len);
    for (int m = 0; m < len; m++) { // 打印出排序完的数组
        std::cout << my_arr[m] << std::endl;
    }

    return 0;
}

```

#### 4. 算法分析

最佳情况： $T(n)=O(n^2)$  最差情况： $T(n^2)$  平均情况： $T(n) = O(n^2)$

## 四、 插入排序（Insertion Sort）

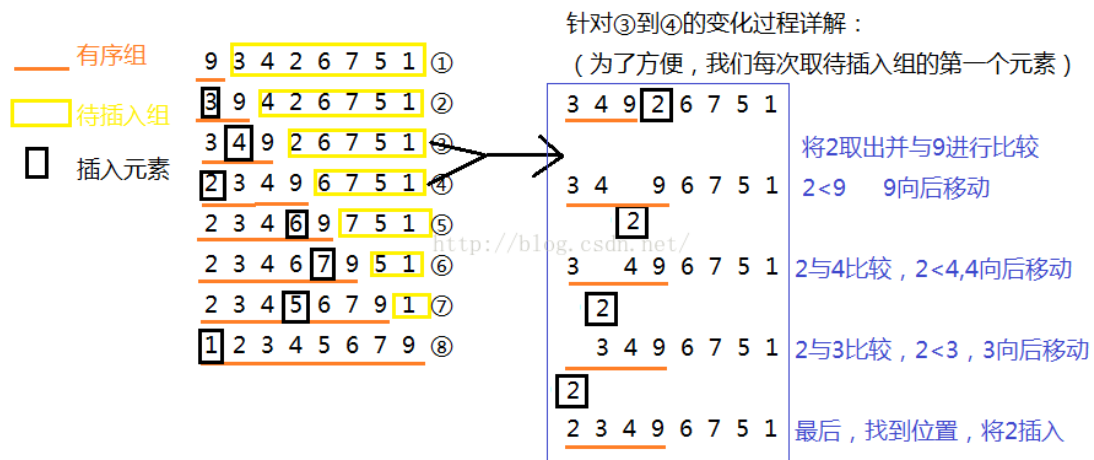
插入排序（Insertion-Sort）的算法描述是一种简单直观的排序算法。它的工作原理是**通过构建有序序列**，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入。插入排序在实现上，通常采用 in-place 排序（即只需用到  $O(1)$  的额外空间的排序），因而在从后向前扫描过程中，需要反复把已排序元素逐步向后挪位，为最新元素提供插入空间。

#### 1. 算法描述

一般来说，插入排序都采用 in-place 在数组上实现。具体算法描述如下：

- 从第一个元素开始，该元素可以认为已经被排序；
- 取出下一个元素，在已经排序的元素序列中从后向前扫描；
- 如果该元素（已排序）大于新元素，将该元素移到下一个位置；
- 重复步骤 3，直到找到已排序的元素小于或者等于新元素的位置；

- 将新元素插入到该位置后；
- 重复步骤 2~5。



## 2. 动画演示

参见文件夹《十大算法动画演示》下的插入排序法.gif。

## 3. 代码实现

```
// InsertSort.cpp : 定义控制台应用程序的入口点。
#include "stdafx.h"
#include <iostream>
#include <stdlib.h>

using namespace std;

// SelectionSort: 插入排序法; 输入: 指向数组的指针、数组的长度
void InsertSort(int *arr, int len) {
    int temp;
    for (int i = 1; i < len; i++) { // 待插入组
        temp = *(arr+i); // 从待插入组取出第一个元素
        int j = i - 1; // 有序组最后一个元素 (与待插元素相邻) 的下标
        while ((j>=0) && (temp<*(arr+j))) { // 注意判断为两个, 一个是j>0进行
            // 边界限制; 另一个是插入的判断条件
            *(arr+j+1) = *(arr+j);
            j--;
        }
        *(arr+j+1) = temp; // 找到合适的位置插入待排元素
    }
}
```

```
int main(int argc, char* argv[])
{
    int my_arr[] = {1, 5, 4, 4, 5, 62, 32, 21, 31, 54};
    int len = (sizeof(my_arr)/sizeof(my_arr[0])); // 计算数组的长度
    InsertSort(my_arr, len);
    for (int m = 0; m < len; m++) { // 打印出排序完的数组
        std::cout << my_arr[m] << std::endl;
    }

    return 0;
}
```

#### 4. 算法分析

最佳情况:  $T(n) = O(n)$  最坏情况:  $T(n) = O(n^2)$  平均情况:  $T(n) = O(n^2)$

## 五、 希尔排序 (Shell Sort)

1959 年 Shell 发明, 第一个突破  $O(n^2)$  的排序算法, 是简单插入排序的改进版。它与插入排序的不同之处在于, 它会优先比较距离较远的元素。希尔排序又叫缩小增量排序。

希尔排序的核心在于间隔序列的设定。既可以提前设定好间隔序列, 也可以动态的定义间隔序列。动态定义间隔序列的算法是《算法 (第 4 版)》的合著者 Robert Sedgewick 提出的。

#### 1. 算法描述

先将整个待排序的记录序列分割成为若干子序列分别进行直接插入排序, 具体算法描述:

- 选择一个增量序列  $t_1, t_2, \dots, t_k$ , 其中  $t_i > t_j, t_k = 1$
- 按增量序列个数  $k$ , 对序列进行  $k$  趟排序;
- 每趟排序, 根据对应的增量  $t_i$ , 将待排序列分割成若干长度为  $m$  的子序列, 分别对各子表进行直接插入排序。仅增量因子为 1 时, 整个序列作为一个表来处理, 表长度即为整个序列的长度。

以  $n=10$  的一个数组 49, 38, 65, 97, 26, 13, 27, 49, 55, 4 为例:

第一次  $gap = 10/2 = 5$

49 38 65 97 26 13 27 49 55 4

1A

1B

2A

2B

3A

3B

4A

4B

5A

5B

1A, 1B, 2A, 2B 等为分组标记，数字同样的表示在同一组，大写字母表示是该组的第几个元素，每次对同一组的数据进行直接插入排序。

即分成了五组 (49, 13) (38, 27) (65, 49) (97, 55) (26, 4) 这样每组排序后就变成了 (13, 49) (27, 38) (49, 65) (55, 97) (4, 26)，下同。

第二次  $\text{gap} = 5/2 = 2$

排序后：

13 27 49 55 4 49 38 65 97 26

1A

1B

1C

1D

1E

2A

2B

2C

2D

2E

第三次  $\text{gap} = 2/2 = 1$

4 26 13 27 38 49 49 55 97 65

1A

1B

1C

1D

1E

1F

1G

1H

1I

1J

第四次  $\text{gap} = 1/2 = 0$

4 13 26 27 38 49 49 55 65 97

## 2. 动画演示

参见文件夹《十大算法动画演示》下的[希尔排序法.swf](#)。

## 3. 代码实现

```
// ShellSort.cpp : 定义控制台应用程序的入口点。
// example:
//5, 2, 6, 0, 3, 9, 1, 7, 4, 8
//3,      4,      5,
// 2,      8,      9
//  1      6
//    0      7

//3, 2, 1, 0, 4, 8, 6, 7, 5, 9
//1,  3,  4,  5,  6,
// 0,  2,  7,  8,  9

//1, 0, 3, 2, 4, 7, 5, 8, 6, 9
```



```

//0, 1, 2, 3, 4, 5, 6, 7, 8, 9

#include "stdafx.h"
#include <iostream>
#include <stdlib.h>

using namespace std;

// ShellSort: Shell排序法; 输入: 指向数组的指针、数组的长度
void ShellSort(int *arr, int len) {
    int i, j, k, gap;
    int temp;
    for (gap = len/2; gap>0; gap/=2) { // 间隔
        for (k = 0; k<gap; k++) { // 第k组直接插入排序仿照插入排序进行写
            for (i=1 ; i<len/gap; i++) { // 第k组的待插入组
                temp = *(arr+i*gap);
                j = i - 1; // 有序组最后一个元素（与待插元素相邻）的下标
                while ((j>=0) && (temp < *(arr+j*gap))) { //注意判断为两个，一个
                    // 是j>0进行边界限制；另一个是插入的判断条件
                    *(arr+(j+1)*gap) = *(arr+j*gap);
                    j--;
                }
                *(arr+(j+1)*gap) = temp; // 找到合适的位置插入待排元素
            }
        }
    }
}

int main(int argc, char* argv[])
{
    int my_arr[] =
{1, 5, 4, 4, 5, 62, 32, 21, 31, 54, 13, 92, 123, 87, 63, 43, 21, 91, 11, 32, 22, 3345, 321, 1, 21, 21, 3211};
    int len = (sizeof(my_arr)/sizeof(my_arr[0])); // 计算数组的长度
    ShellSort(my_arr, len);
    for (int m = 0; m < len; m++) { // 打印出排序完的数组
        std::cout << my_arr[m] << std::endl;
    }

    return 0;
}

```

#### 4. 算法分析

最佳情况:  $T(n) = O(n^{1.3})$     最坏情况:  $T(n) = O(n^2)$     平均情况:  $T(n) = O(n^2)$

---

## 六、 归并排序（Merge Sort）

### 1. 算法描述

和选择排序一样，归并排序的性能不受输入数据的影响，但表现比选择排序好的多，因为始终都是  $O(n \log n)$  的时间复杂度。代价是需要额外的内存空间。

归并排序是建立在归并操作上的一种有效的排序算法。该算法是采用分治法（Divide and Conquer）的一个非常典型的应用。归并排序是一种稳定的排序方法。将已有序的子序列合并，得到完全有序的序列；即先使每个子序列有序，再使子序列段间有序。若将两个有序表合并成一个有序表，称为 2-路归并。

### 2. 动画演示

参见文件夹《十大算法动画演示》下的归并排序法.gif。

### 3. 代码实现

```
// MergeSort.cpp : 定义控制台应用程序的入口点。

#include "stdafx.h"
#include <iostream>
#include <stdlib.h>

using namespace std;

// 将数组a[low,mid]与a[mid,high]合并（归并）
void Merge(int *a, int low, int mid, int high, int *temp)
{
    int i, j, k;
    i = low;
    j = mid + 1; // 避免重复比较a[mid]
    k = 0;
    while ((i <= mid) && (j <= high)) { // 数组a[low,mid] 与数组 (mid,high]均没有完全归入数组temp
        if (a[i] <= a[j]) // 如果a[i]小于等于a[j]
            temp[k++] = a[i++]; // 则将a[i]的值赋给temp[k]，之后i,k各加一，表示后移一位
        else
            temp[k++] = a[j++];
    }

    while (i <= mid) { // 表示a (mid, high]已经全部归入temp中去了，而数组a[low,mid]还有剩余
```

```

        temp[k++] = a[i++]; // 将a[low,mid]剩余的逐一归并到temp中
    }

    while (j<=mid) { // 表示a[low,mid]已经全部归入temp中去了，而数组
a[mid,high]还有剩余
        temp[k++] = a[j++]; // 将a[mid,high]剩余的逐一归并到temp中
    }

    for (i=0;i<k;i++) { // 归并后的数组逐一赋值给数组a[low,high]
        a[low+i] = temp[i];
    }
}

// MergeSort: 二路Merge排序法（递归实现）
void MergeSort(int *arr, int low, int high, int *temp) {
    if (low < high) {
        int mid = (low + high)/2;
        MergeSort(arr, low, mid, temp);
        MergeSort(arr, mid+1, high, temp);
        Merge(arr, low, mid, high, temp);
    }
}

int main(int argc, char* argv[])
{
    int my_arr[] =
{1, 5, 4, 4, 5, 62, 32, 21, 31, 54, 13, 92, 123, 87, 63, 43, 21, 91, 11, 32, 22, 3345, 321, 1, 21, 21,
3211};
    int len = (sizeof(my_arr)/sizeof(my_arr[0])); // 计算数组的长度
    int *temp = new int[len]; // 用来辅助归并
    MergeSort(my_arr, 0, len-1, temp);
    for (int m = 0; m < len; m++) { // 打印出排序完的数组
        std::cout << my_arr[m] << std::endl;
    }
    return 0;
}

```

#### 4. 算法分析

最佳情况： $T(n) = O(n)$  最差情况： $T(n) = O(n \log n)$  平均情况： $T(n) = O(n \log n)$

## 七、快速排序（Quick Sort）

快速排序的基本思想：通过一趟排序将待排记录分隔成独立的两部分，其中一部分记录的关键字均比另一部分的关键字小，则可分别对这两部分记录继续进行排序，以达到整个序列有序。

### 1. 算法描述

快速排序使用分治法来把一个串（list）分为两个子串（sub-lists）。具体算法如下：

- 从数列中挑出一个元素，称为“基准”（pivot）；
- 重新排序数列，所有元素比基准值小的摆在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区退出之后，该基准就处于数列的中间位置。这个称为分区（partition）操作；
- 递归地（recursive）把小于基准值元素的子数列和大于基准值的子序列排序。

## 算法概述

什么是快速排序算法的最好情况？

### ■ 分而治之

每次正好中分  $\rightarrow T(N) = O(N \log N)$

```
void Quicksort( ElementType A[], int N )
{
    ? pivot = 从A[]中选一个主元;
    ? 将S = { A[] \ pivot } 分成2个独立子集:
        A1={ a∈S | a ≤ pivot } 和
        A2={ a∈S | a ≥ pivot };
    A[] = Quicksort(A1,N1) ∪
          {pivot} ∪
          Quicksort(A2,N2);
}
```

## 选主元

- 令 **pivot** = **A[0]** ?

① 2 3 4 5 6 ..... N-1 N  
② 3 4 5 6 ..... N-1 N  
3 4 5 6 ..... N-1 N



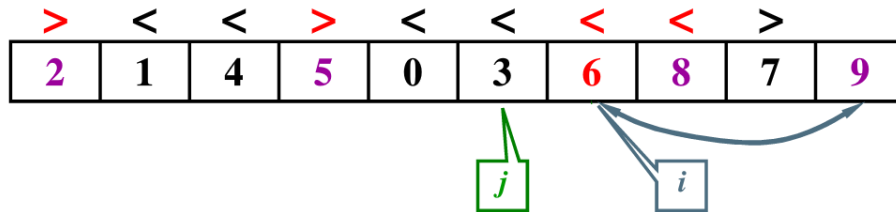
$$\begin{aligned} T(N) &= O(N) + T(N-1) \\ &= O(N) + O(N-1) + T(N-2) \\ &= O(N) + O(N-1) + \dots + O(1) \\ &= O(N^2) \end{aligned}$$

## 选主元

- 随机取 **pivot**? **rand()** 函数不便宜啊!
- 取头、中、尾的中位数
  - 例如 8、12、3的中位数就是8
  - 测试一下**pivot**不同的取法对运行速度有多大影响?

```
ElementType Median3( ElementType A[], int Left, int Right )
{
    int Center = ( Left + Right ) / 2;
    if ( A[ Left ] > A[ Center ] )
        Swap( &A[ Left ], &A[ Center ] );
    if ( A[ Left ] > A[ Right ] )
        Swap( &A[ Left ], &A[ Right ] );
    if ( A[ Center ] > A[ Right ] )
        Swap( &A[ Center ], &A[ Right ] );
    /* A[ Left ] <= A[ Center ] <= A[ Right ] */
    Swap( &A[ Center ], &A[ Right-1 ] ); /* 将pivot藏到右边 */
    /* 只需要考虑 A[ Left+1 ] ... A[ Right-2 ] */
    return A[ Right-1 ]; /* 返回 pivot */
}
```

## 子集划分



- 如果有元素正好等于**pivot**怎么办?
  - 停下来交换? ✓
  - 不理它, 继续移动指针?

## 小规模数据的处理

- 快速排序的问题
  - 用递归.....
  - 对小规模的数据 (例如**N**不到**100**) 可能还不如插入排序快
- 解决方案
  - 当递归的数据规模充分小, 则停止递归, 直接调用简单排序 (例如插入排序)
  - 在程序中定义一个**Cutoff**的阈值 — 课后去实践一下, 比较不同的**Cutoff**对效率的影响

# 算法实现

```
void Quicksort( ElementType A[], int Left, int Right )
{ if ( Cutoff <= Right-Left ) {
    Pivot = Median3( A, Left, Right );
    i = Left; j = Right - 1;
    for( ; ; ) {
        while ( A[ ++i ] < Pivot ) { }
        while ( A[ --j ] > Pivot ) { }
        if ( i < j )
            Swap( &A[i], &A[j] );
        else break;
    }
    Swap( &A[i], &A[ Right-1 ] );
    Quicksort( A, Left, i-1 );
    Quicksort( A, i+1, Right );
}
else
    Insertion_Sort( A+Left, Right-Left+1 );
}
```

```
void Quick_Sort(ElementType A[],int N)
{
    Quicksort( A, 0, N-1 );
}
```

## 2. 动图演示

参见文件夹《十大算法动画演示》下的快速排序法.gif。

## 3. 代码实现

```
// QuickSort.cpp : 定义控制台应用程序的入口点。
//
#include "stdafx.h"
#include <stdlib.h>
#include <iostream>
typedef int ElementType;
#define Cutoff 5 // 停止递归的阈值

// 交换两个地址内的数
void Swap(ElementType* a, ElementType* b) {
    ElementType temp = *a;
    *a = *b;
    *b = temp;
}

// 以头、中、尾的中位数形式取主元
ElementType Median3(ElementType A[], int Left, int Right) {
    int Center = (Left + Right)/2;
    if (A[Left] > A[Center])
```

```

        Swap(&A[Left], &A[Center]);
    if (A[Left] > A[Right])
        Swap(&A[Left], &A[Right]);
    if (A[Center] > A[Right])
        Swap(&A[Center], &A[Right]);
    /* A[Left] <= A[Center] <= A[Right] */
    Swap(&A[Center], &A[Right-1]); /* 将pivot藏到右边 */
    /* 只需要考虑A[Left+1] ...A[Right-2] */
    return A[Right-1]; /*返回Pivot*/
}

// 插入排序：当待排序的数组长度小于Cutoff阈值时，采用插入排序，停止递归
// SelectionSort：插入排序法；输入：指向数组的指针、数组的长度
void InsertSort(int arr[], int len) {
    int temp;
    for (int i = 1; i < len; i++) { // 待插入组
        temp = arr[i]; // 从待插入组取出第一个元素
        int j = i - 1; // 有序组最后一个元素（与待插元素相邻）的下标
        while ((j >= 0) && (temp < arr[j])) { // 注意判断为两个，一个是j>0进行边界限制；另一个是插入的判断条件
            arr[j+1] = arr[j];
            j--;
        }
        arr[j+1] = temp; // 找到合适的位置插入待排元素
    }
}

// 快速排序的核心函数
void Qsort(ElementType arr[], int Left, int Right) {
    int Pivot; // 主元
    int i;
    int j;
    if (Cutoff <= Right - Left) { // 如果序列元素充分多，进入快排
        Pivot = Median3(arr, Left, Right); // 选主元
        i = Left; j = Right - 1;
        for ( ; ; ) { // 将序列中比基准小的移到基准左边，大的移到右边
            while (arr[++i] < Pivot) { }
            while (arr[--j] > Pivot) { }
            if ( i < j )
                Swap(&arr[i], &arr[j]);
            else
                break;
        }
    }
}

```



```

        Swap(&arr[i], &arr[Right-1]);
        Qsort(arr, Left, i-1);
        Qsort(arr, i+1, Right);
    }
    else
        InsertSort(arr+Left, Right-Left+1); // 元素太少的话就用插入排序
}

// 统一接口
void QuickSort(ElementType A[], int len) {
    Qsort(A, 0, len-1);
}

int main(int argc, char *argv[]) {
    int my_arr[] = {1, 5, 4, 8, 6, 9, 3, 6};
    int len = (sizeof(my_arr)/sizeof(my_arr[0])); // 计算数组的长度

    std::cout << "Before Sort: " << std::endl;
    for (int m = 0; m < len; m++) { // 打印出排序前的数组
        std::cout << my_arr[m] << " ";
    }
    std::cout << std::endl;

    QuickSort(my_arr, len); // 调用快速排序算法
    std::cout << "After Sort: " << std::endl;
    for (int m = 0; m < len; m++) { // 打印出排序完的数组
        std::cout << my_arr[m] << " ";
    }
    std::cout << std::endl;

    return 0;
}

```

#### 4. 算法分析

最佳情况： $T(n) = O(n \log n)$  最差情况： $T(n) = O(n^2)$  平均情况： $T(n) = O(n \log n)$

## 八、堆排序（Heap Sort）（调整为最大堆那一块还没能完全理解）

堆排序（Heapsort）是指利用堆这种数据结构所设计的一种排序算法。堆积是一个近似完全二叉树的结构，并同时满足堆积的性质：即子结点的键值或索引

---

总是小于（或者大于）它的父节点。堆排序是对选择排序的一种改进，主要是在找最小元的过程中，采用堆的方法

### 1. 算法描述

- 将初始待排序关键字序列  $(R_1, R_2, \dots, R_n)$  构建成大顶堆，此堆为初始无序区；
- 将堆顶元素  $R[1]$  与最后一个元素  $R[n]$  交换，，此时得到新的无序区  $(R_1, R_2, \dots, R_{n-1})$  和新的有序区  $(R_n)$ ，且满足  $R[1, 2, 3, \dots, n-1] \leq R[n]$ ；
- 由于交换后新的堆顶  $R[1]$  可能违反堆的性质，因此需要对当前无序区  $(R_1, R_2, \dots, R_{n-1})$  调整为新堆，然后再次将  $R[1]$  与无序区最后一个元素交换，得到新的无序区  $(R_1, R_2, \dots, R_{n-2})$  和新的有序区  $(R_{n-1}, R_n)$ 。不断重复此过程直到有序区的元素个数为  $n-1$ ，则整个排序过程完成。

## 堆排序

### ■ 算法1

```
void Heap_Sort ( ElementType A[], int N )
{   BuildHeap(A);   /* O(N) */
    for ( i=0; i<N; i++ )
        TmpA[i] = DeleteMin(A); /* O(logN) */
    for ( i=0; i<N; i++ ) /* O(N) */
        A[i] = TmpA[i];
}
```

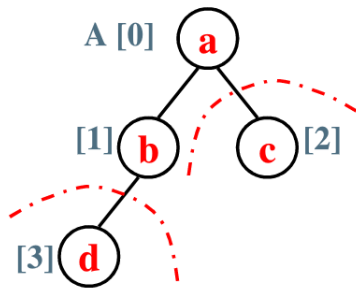
$$T(N) = O(N \log N)$$



需要额外 $O(N)$ 空间，并且复制元素需要时间。

# 堆排序

## ■ 算法2



```
void Heap_Sort ( ElementType A[], int N )
{   for ( i=N/2-1; i>=0; i-- ) /* BuildHeap */
        PercDown( A, i, N );
    for ( i=N-1; i>0; i-- ) {
        Swap( &A[0], &A[i] ); /* DeleteMax */
        PercDown( A, 0, i );
    }
}
```

- 定理：堆排序处理 $N$ 个不同元素的随机排列的平均比较次数是  $2N \log N - O(N \log \log N)$ 。
- 虽然堆排序给出最佳平均时间复杂度，但实际效果不如用 **Sedgewick**增量序列的希尔排序。

### 2. 动画演示

参见文件夹《十大算法动画演示》下的[堆排序法.gif](#)。

### 3. 代码实现

```
// MaxHeap.cpp : 定义控制台应用程序的入口点。
//

#include "stdafx.h"
#include <stdlib.h>
#include <iostream>
typedef int ElementType;
typedef struct HeapStruct *MaxHeap;
#define MaxData 100000000;

void Swap( ElementType *a, ElementType *b )
{
    ElementType t = *a; *a = *b; *b = t;
}

void PercDown( ElementType A[], int p, int N )
{ /* 改编代码.24的PercDown( MaxHeap H, int p ) */
    /* 将N个元素的数组中以A[p]为根的子堆调整为最大堆 */
    int Parent, Child;
```

```

ElementType X;

X = A[p]; /* 取出根结点存放的值*/
for( Parent=p; (Parent*2+1)<N; Parent=Child ) {
    Child = Parent * 2 + 1;
    if( (Child!=N-1) && (A[Child]<A[Child+1]) )
        Child++; /* Child指向左右子结点的较大者*/
    if( X >= A[Child] ) break; /* 找到了合适位置*/
    else /* 下滤X */
        A[Parent] = A[Child];
}
A[Parent] = X;
}

void HeapSort( ElementType A[], int N )
{ /* 堆排序*/
    int i;

    for ( i=N/2-1; i>=0; i-- ) /* 建立最大堆*/
        PercDown( A, i, N );

    for ( i=N-1; i>0; i-- ) {
        /* 删除最大堆顶*/
        Swap( &A[0], &A[i] ); /* 见代码.1 */
        PercDown( A, 0, i );
    }
}

int main(int argc, char *argv[]) {
    int my_arr[] = {1,5,4,8,6,9,3,6};
    int len = (sizeof(my_arr)/sizeof(my_arr[0])); // 计算数组的长度

    std::cout << "Before Sort: " << std::endl;
    for (int m = 0; m < len; m++) { // 打印出排序前的数组
        std::cout << my_arr[m] << " ";
    }
    std::cout << std::endl;

    HeapSort(my_arr, len); // 调用快速排序算法
    std::cout << "After Sort: " << std::endl;
    for (int m = 0; m < len; m++) { // 打印出排序完的数组
        std::cout << my_arr[m] << " ";
    }
    std::cout << std::endl;
}

```

```
    return 0;
}
```

#### 4. 算法分析

最佳情况： $T(n) = O(n \log n)$  最差情况： $T(n) = O(n \log n)$  平均情况： $T(n) = O(n \log n)$ 。

## 九、 计数排序（Counting Sort）

计数排序的核心在于将输入的数据值转化为键存储在额外开辟的数组空间中。作为一种线性时间复杂度的排序，计数排序要求输入的数据必须是有确定范围的整数。

计数排序(Counting sort)是一种稳定的排序算法。计数排序使用一个额外的数组 C，其中第 i 个元素是待排序数组 A 中值等于 i 的元素的个数。然后根据数组 C 来将 A 中的元素排到正确的位置。它只能对**整数**进行排序。

#### 1. 算法描述

- 找出待排序的数组中最大和最小的元素；
- 统计数组中每个值为 i 的元素出现的次数，存入数组 C 的第 i 项；
- 对所有的计数累加（从 C 中的第一个元素开始，每一项和前一项相加）；
- 反向填充目标数组：将每个元素 i 仿真新数组的第 C(i) 项，没放一个元素就将 C(i) 减去 1。

#### 2. 动画演示

参见文件夹《十大算法动画演示》下的**计数排序法.gif**。

#### 3. 代码实现

```
// CountSort.cpp : 定义控制台应用程序的入口点。

#include "stdafx.h"
#include <iostream>
#include <stdlib.h>
using namespace std;

#define MAX(x, y) (((x) > (y)) ? (x) : (y)) // 比较大小

int FindMax(int arr[], int arrSize) { // 找出数组中的最大值
    int max = arr[0];
    for(int i=0; i<arrSize; i++) {
```

```

        max = MAX(arr[i], max);
    }
    return max;
}

// CountSort: 计数排序法; 输入: 指向数组的指针、数组的长度、数组中最大值
void CountSort(int *arr, int len, int MaxValue) {

    int *bucket = (int *)malloc((MaxValue+1)*sizeof(int));
    int sortedIndex = 0; // 写回arr时的下标
    int bucketLen = MaxValue + 1;

    if(bucket == NULL) { // 判断内存是否申请成功
        printf("内存申请失败! \n");
        return;
    }
    else {
        memset(bucket, 0, sizeof(int)*(MaxValue+1)); // 对bucket进行初始化
        for (int i = 0; i < len; i++) // bucket 统计每个值的出现次数
            bucket[arr[i]]++;

        for (int j = 0; j < bucketLen; j++) { // 遍历bucket 写回arr
            while(bucket[j]>0) {
                arr[sortedIndex++] = j;
                bucket[j]--;
            }
        }
        free(bucket);
    }
}

int main(int argc, char* argv[])
{
    int my_arr[] =
{1, 5, 4, 4, 5, 62, 32, 21, 31, 54, 78, 32, 43, 21, 34, 5, 63, 21, 23, 32, 5, 32, 1, 1, 322, 1, 31, 1233
, 25, 324, 432234};

    int len = (sizeof(my_arr)/sizeof(my_arr[0])); // 计算数组的长度
    int MaxValue = FindMax(my_arr, len);
    CountSort(my_arr, len, MaxValue);
    for (int m = 0; m < len; m++) { // 打印出排序完的数组
        std::cout << my_arr[m] << std::endl;
    }
    return 0;
}

```

#### 4. 算法分析

当输入的元素是  $n$  个  $0$  到  $k$  之间的整数时，它的运行时间是  $O(n+k)$ 。计数排序不是比较排序，排序的速度快于任何比较算法。由于用来计数的数组  $C$  的长度取决于待排序数组中数据的范围（等于待排序数组的最大值与最小值的差加 1），这使得**计数排序对于数据范围很大的数组，需要大量时间和内存。**

最佳情况： $T(n)=O(n+k)$  最差情况  $T(n)=O(n+k)$  平均情况  $T(n)=O(n+k)$

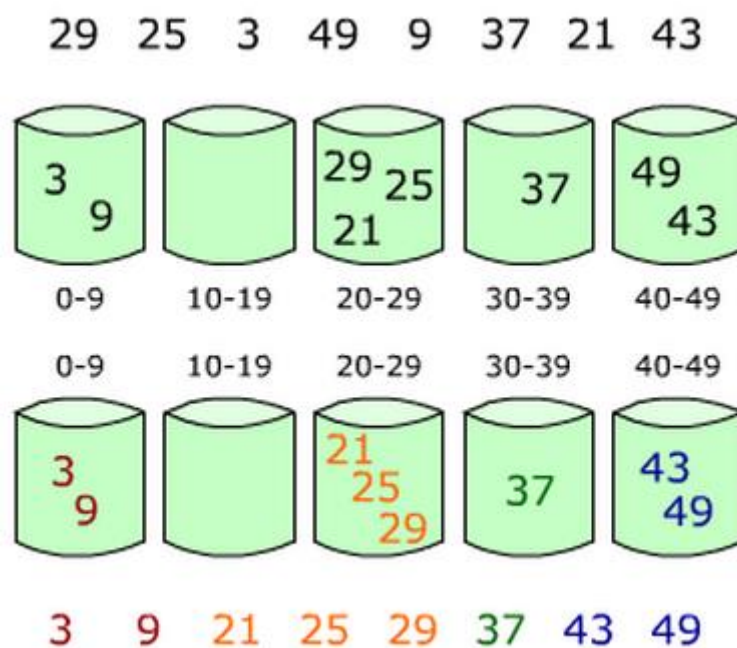
## 十、桶排序（Bucket Sort）

桶排序是计数排序的升级版。它是利用了函数的映射关系，高效与否的关键在于这个映射函数的确定。

桶排序（Bucket Sort）的工作原理：假设输入数据服从均匀分布，将数据分到有限数量的桶里，每个桶再分别排序（有可能再使用别的排序算法或者是以递归方式继续使用桶排序进行排序）。

#### 1. 算法描述

- 设置一个定量的数组作为空桶；
- 遍历输入数据，并且把数据一个一个地放在对应的桶里去；
- 对每个不是空的桶进行排序；
- 从不是空的桶里面把排好序的数据拼接起来。



#### 2. 动画演示

---

参见文件夹《十大算法动画演示》下的桶排序法.png。

### 3. 代码实现

```
#include "stdafx.h"
#include <iostream>
#include <vector>
#include <stdio.h>
#include <algorithm>

using namespace std;

void BucketSort(int arr[], int len, int bucket_num = 5) {
    if (len == 0) { // 数组为空
        return;
    }

    int i;
    int MinValue = arr[0];
    int MaxValue = arr[0];
    for (i = 1; i < len; i++) {
        if (arr[i] < MinValue) {
            MinValue = arr[i]; // 输入数据的最小值
        }
        else if (arr[i] > MaxValue) {
            MaxValue = arr[i]; // 输入数据的最大值
        }
    }

    std::vector<int> *v = new std::vector<int>[bucket_num]; // 定义
    bucket_num个桶

    int basis; // 桶的编号
    int concat_index = 0;
    int step = (MaxValue - MinValue) / bucket_num + 1;

    // Insert array element to buckets.
    for (i = 0; i < len; i++) { // 把每个元素分配到相应的桶里
        basis = (arr[i] - MinValue) / step;
        v[basis].push_back(arr[i]);
    }

    // Sort individual bucket
    for (i = 0; i < bucket_num; i++) { // 调用vector自带的sort函数学了数据
```



结构后可以自己写一个链表的排序

```
        sort(v[i].begin(), v[i].end());

    }

    // Concatenate all elements in buckets.
    for (i = 0; i < bucket_num; i++) { // 遍历每个桶
        for (unsigned int j = 0; j < v[i].size(); j++) { // 遍历每个桶里的每个元素
            arr[concat_index] = v[i][j];
            concat_index += 1;
        }
    }

    delete[] v;
}

int main(int argc, char* argv[]) {
    int my_arr[] = {29, 25, 3, 49, 9, 37, 21, 43, 11, 23, 45, 21, 67, 89, 88, 99, 77, 55, 66, 77, 55, 33, 22, 11};
    int len = (sizeof(my_arr)/sizeof(my_arr[0])); // 计算数组的长度
    int bucket_num = 3;
    cout << "Unsorted array is " << endl;
    for (int i = 0; i < len; i++)
        cout << my_arr[i] << " ";
    cout << endl;

    BucketSort(my_arr, len);

    cout << "Sorted array is " << endl;
    for (int i = 0; i < len; i++)
        cout << my_arr[i] << " ";
    cout << endl;
    return 0;
}
```

#### 4. 算法分析

桶排序最好的情况下线性时间是  $O(n)$ ，桶排序的时间复杂度，取决于对每个桶之间数据进行排序的时间复杂度，因为其他部分的时间复杂度都是  $O(n)$ 。很显然，桶划分得越小，各个桶之间的数据越少，排序所用的时间也会越少，但相应的空间消耗也会增大。

最佳情况： $T(n)=O(n+k)$  最差情况： $T(n)=O(n^2)$  平均情况： $T(n)=O(n+k)$

## 十一、 基数排序（Radix Sort）

基数排序也是非比较的排序算法，对每一位进行排序，从最低位开始排序，复杂度为  $O(kn)$ ，为数组长度， $k$  为数组中的数的最大的位数；

基数排序是按照**低位先排序，然后收集；再按照高位排序，然后再收集；依次类推，直到最高位**。有时候有些属性是有优先级顺序的，先按低优先级排序，再按高优先级排序。最后的次序就是高优先级高的在前，高优先级相同的低优先级高的在前。基数排序基于分别排序，分别收集，所以是稳定的。

### 1. 算法描述

- 取得数组中的最大数，并取得位数；
- $arr$  为原始数组，从最低位开始取每个位数组成  $radix$  数组；
- 对  $radix$  进行计数排序（利用计数排序适合于小范围数的特点）；

## 基数排序



假设我们有  $N = 10$  个整数，每个整数的值在 **0** 到 **999** 之间（于是有  $M = 1000$  个不同的值）。还有可能在**线性时间**内排序吗？

输入序列：64, 8, 216, 512, 27, 729, 0, 1, 343, 125  
用“次位优先”（Least Significant Digit）

$$T=O(P(N+B))$$

Bucket	0	1	2	3	4	5	6	7	8	9
Pass 1	0	1	512	343	64	125	216	27	8	729
Pass 2	0	512	125		343		64			
	1	216	27							
		8		729						
Pass 3	0	125	216	343		512		729		
	1									
	8									
	27									
	64									

## 多关键字的排序



一副扑克牌是按**2**种关键字排序的

$K^0$  [花色]

♣ < ♦ < ♥ < ♠

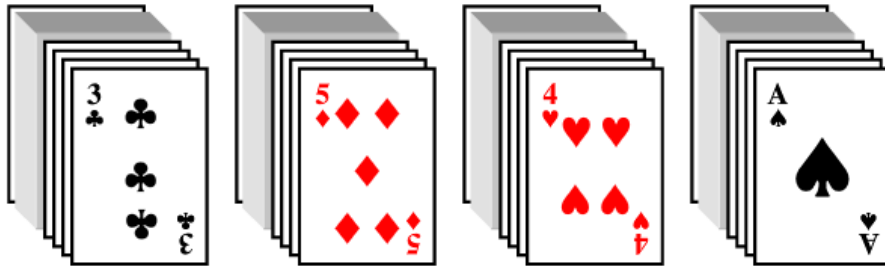
$K^1$  [面值]

2 < 3 < 4 < 5 < 6 < 7 < 8 < 9 < 10 < J < Q < K < A

有序结果:

2♣ ... A♣ 2♦ ... A♦ 2♥ ... A♥ 2♠ ... A♠

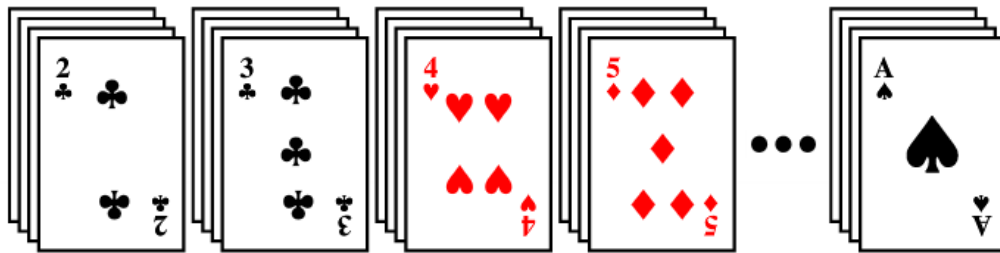
☞ 用“主位优先” ( **Most Significant Digit** ) 排序: 为花色建4个桶



在每个桶内分别排序, 最后合并结果。

## 多关键字的排序

■ 用“次位优先” ( **Least Significant Digit** ) 排序: 为面值建13个桶



■ 将结果合并, 然后再为花色建4个桶

■ 问题: **LSD**任何时候都比**MSD**快吗?

2. 动画演示

参见文件夹《十大算法动画演示》下的[基数排序法.png](#)。

3. 代码实现 (代码还没搞明白, 等学完链表之后再看)

```
// RadixSort.cpp : 定义控制台应用程序的入口点。
// * 基数排序- 次位优先*/

#include "stdafx.h"
#include <iostream>
#include <stdlib.h>

/* 基数排序- 次位优先*/

/* 假设元素最多有MaxDigit个关键字，基数全是同样的Radix */
#define MaxDigit 4
#define Radix 10
typedef int ElementType;

/* 桶元素结点*/
typedef struct Node *PtrToNode;
struct Node {
    int key;
    PtrToNode next;
};

/* 桶头结点*/
struct HeadNode {
    PtrToNode head, tail;
};
typedef struct HeadNode Bucket[Radix];

int GetDigit ( int X, int D ) // X = 189 D = 1, 返回; X = 189 D = 2, 返回; X =
189 D = 3, 返回;
{ /* 默认次位D=1, 主位D<=MaxDigit */
    int d, i;

    for (i=1; i<=D; i++) {
        d = X % Radix;
        X /= Radix;
    }
    return d;
}

void LSDRadixSort( ElementType A[], int N )
{ /* 基数排序- 次位优先*/
    int D, Di, i;
    Bucket B;
    PtrToNode tmp, p, List = NULL;
```

```

for (i=0; i<Radix; i++) /* 初始化每个桶为空链表*/
    B[i].head = B[i].tail = NULL;
for (i=0; i<N; i++) { /* 将原始序列逆序存入初始链表List */
    tmp = (PtrToNode)malloc(sizeof(struct Node));
    tmp->key = A[i];
    tmp->next = List;
    List = tmp;
}
/* 下面开始排序*/
for (D=1; D<=MaxDigit; D++) { /* 对数据的每一位循环处理*/
    /* 下面是分配的过程*/
    p = List;
    while (p) {
        Di = GetDigit(p->key, D); /* 获得当前元素的当前位数字*/
        /* 从List中摘除*/
        tmp = p; p = p->next;
        /* 插入B[Di]号桶尾*/
        tmp->next = NULL;
        if (B[Di].head == NULL)
            B[Di].head = B[Di].tail = tmp;
        else {
            B[Di].tail->next = tmp;
            B[Di].tail = tmp;
        }
    }
    /* 下面是收集的过程*/
    List = NULL;
    for (Di=Radix-1; Di>=0; Di--) { /* 将每个桶的元素顺序收集入List */
        if (B[Di].head) { /* 如果桶不为空*/
            /* 整桶插入List表头*/
            B[Di].tail->next = List;
            List = B[Di].head;
            B[Di].head = B[Di].tail = NULL; /* 清空桶*/
        }
    }
}
/* 将List倒入A[]并释放空间*/
for (i=0; i<N; i++) {
    tmp = List;
    List = List->next;
    A[i] = tmp->key;
    free(tmp);
}

```

```

}

using namespace std;
int main(int argc, char* argv[])
{
    int my_arr[] =
{29, 25, 3, 49, 9, 37, 21, 43, 11, 23, 45, 21, 67, 89, 88, 99, 77, 55, 66, 77, 55, 33, 22, 11};
    int len = (sizeof(my_arr)/sizeof(my_arr[0])); // 计算数组的长度
    int bucket_num = 3;
    cout << "Unsorted array is " << endl;
    for (int i = 0; i < len; i++)
        cout << my_arr[i] << " ";
    cout << endl;

    LSDRadixSort(my_arr, len);

    cout << "Sorted array is " << endl;;
    for (int i = 0; i < len; i++)
        cout << my_arr[i] << " ";
    cout << endl;

    return 0;
}

```

#### 4. 算法分析

最佳情况： $T(n) = O(n + k)$  最差情况： $T(n) = O(n + k)$  平均情况： $T(n) = O(n + k)$


## 十二、 表排序（Table Sort）

### 1. 算法描述


什么情况下用到表排序呢？表排序是应用于这么一种情形：待排的元素不是一个简简单单的整数而已，每一个待排的元素都是一个庞大的结构，比如说是一本书。表排序不需要移动原始数据，所需要移动的只是指向这些数据位置的指针。

#### ● 间接排序

- 定义一个指针作为“表”（table）



A	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
key	f	d	c	a	g	b	h	e
table	0	1	2	3	4	5	6	7




A	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
key	f	d	c	a	g	b	h	e
table	3	5	2	1	7	0	4	6

如果仅要求按顺序输出，则输出：

$A[table[0]], A[table[1]], \dots, A[table[N-1]]$

## ● 物理排序



A	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
key	f	d	c	a	g	b	h	e
table	3	5	2	1	7	0	4	6

N 个数字的排序由若干个独立的环组成

用 temp 记录初始值，每次换位置修改 table 值，用  $if(table[i]==i)$  判断一个环的结束

## 2. 动画演示

看浙大版《数据结构》10.2 节表排序。

---

### 3. 代码实现

略

### 4. 算法分析

最好情况：初始即有序

最坏情况：

- 有  $\lceil N/2 \rceil$  个环，每个环包含 2 个元素
- 需要  $\lceil 3N/2 \rceil$  次元素移动

$T = O(mN)$ ， $m$  是每个 A 元素的复制时间。