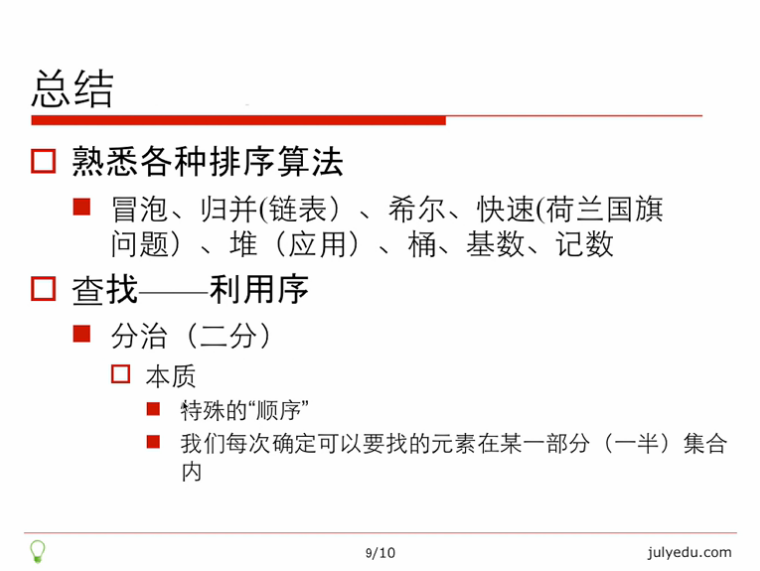
# 排序



1. **综述**

**[按平均时间将排序分为四类]**  
（1）平方阶(O(n2))排序  
    　一般称为简单排序，例如直接插入、直接选择和冒泡排序；  
（2）线性对数阶(O(nlgn))排序  
    　如快速、堆和归并排序；  
（3）O(n1+￡)阶排序  
    　￡是介于0和1之间的常数，即0<￡<1，如希尔排序；  
（4）线性阶(O(n))排序  
    　如桶、箱和基数排序。  
**[各种排序方法比较]**  
     简单排序中直接插入最好，快速排序最快，当文件为正序时，直接插入和冒泡均最佳。  
**[影响排序效果的因素]**  
    　因为不同的排序方法适应不同的应用环境和要求，所以选择合适的排序方法应综合考虑下列因素：  
　　①待排序的记录数目n；  
　　②记录的大小(规模)；  
　　③关键字的结构及其初始状态；  
　　④对稳定性的要求；  
　　⑤语言工具的条件；  
　　⑥存储结构；  
　　⑦时间和辅助空间复杂度等。  
**[不同条件下，排序方法的选择]**  
(1)若n较小(如n≤50)，可采用直接插入或直接选择排序。  
    　当记录规模较小时，直接插入排序较好；否则因为直接选择移动的记录数少于直接插入，应选直接选择排序为宜。  
(2)若文件初始状态基本有序(指正序)，则应选用直接插入、冒泡或随机的快速排序为宜；  
(3)若n较大，则应采用时间复杂度为O(nlgn)的排序方法：快速排序、堆排序或归并排序。  
    　快速排序是目前基于比较的内部排序中被认为是最好的方法，当待排序的关键字是随机分布时，快速排序的平均时间最短；  
    　堆排序所需的辅助空间少于快速排序，并且不会出现快速排序可能出现的最坏情况。这两种排序都是不稳定的。  
    　若要求排序稳定，则可选用归并排序。但本章介绍的从单个记录起进行两两归并的排序算法并不值得提倡，通常可以将它和直接插入排序结合在一起使用。先利用直接插入排序求得较长的有序子文件，然后再两两归并之。因为直接插入排序是稳定的，所以改进后的归并排序仍是稳定的。

**[排序算法的稳定性]**

 1） 稳定的：如果存在多个具有相同排序码的记录，经过排序后，这些记录的相对次序仍然保持不变，则这种排序算法称为稳定的。  
    插入排序、冒泡排序、归并排序、分配排序（桶式、基数）都是稳定的排序算法。  
    2）不稳定的：否则称为不稳定的。  
    直接选择排序、堆排序、shell排序、快速排序都是不稳定的排序算法。

1. **快速排序**

**[快排思想]**

快速排序(quicksort)是分治策略的典型应用，它可以在O(1)时间内由子问题直接得到原文的解，但是将原问题划分为子问题的时间却为O(n)。平均时间复杂度为O(nlgn)。最坏时间复杂度为O(n2)。

策略：随机选取一个数字，以此基准数字将原序列二分(用快速划分函数**partition**)，基准数字左边的数<基准，基准数字右边的数>基准。**递归，**将二分的序列不断划分，当子序列只有一个元素时递归结束(**递归基**)。

**[快速划分Partition]**

* Partition版本1

此版本直观上比较好理解，利用两个指针指向序列的头和尾。以序列头作为基准数字datum，尾指针往前走，直至data[tail]<=datum，然后头指针往后走，直至data[head]>=datum，若此时head<tail则交换data[tail]与data[head]。直至head=tail，return tail。

//数据结构书中的代码：保存基准数字，挖坑法，每走一次填一次坑

int partiton\_1(int data[], int l, int h)

{

if(data == NULL || l<0 || (h-l)<0)

throw exception("Invalide Input!");

int pivot = data[l]; //以首元素作为基准数字

while(l<h)

{

while((l<h) && (data[h]>=pivot) ) //从后往前扫描，直到找到比基准小的数字

h--;

data[l]=data[h]; //填坑

while((l<h) &&(data[l]<=pivot) ) //从前往后扫描，直到找到比基准大的数字

l++;

data[h]=data[l]; //填坑

}

data[h] = pivot; //填坑

return h;

}

* Partition版本2

此版本为算法导论中的代码：设定两个指针同时从头开始走，small=lo-1，index=lo，每次循环中，如果data[index]>pivot则继续向前，small保持不动，当data[index]<=pivot时，small与index之间的数均>pivot，然后small++，指向第一个>pivot的数，然后交换data[index]与data[small]。也就是说small一直指向小于pivot的数据。需要交换的时候再+1。

int partition\_2(int data[], int lo, int hi)

{

if(data == NULL || lo<0 || (hi-lo)<0)

throw exception("Invalide Input!");

int pivot = data[hi];

int small=lo-1;

for(int index = lo;index<hi;++index)

{

if(data[index]<pivot)

{

++small;

if(small!=index)

swap(data[small],data[index]);

}

}

++small;

swap(data[small],data[hi]);

return small;

}

**[快排实现]**

* 递归

void QuickSort(int data[], int lo, int hi)

{

if(lo == hi)

return;

int index = partition\_2(data,lo,hi);

if(index>lo)

QuickSort(data, lo,index-1);

if(index<hi)

QuickSort(data,index+1,hi);

}

* 非递归实现（用栈来实现）

void QuickSort\_2(int data[], int lo, int hi)

{

stack<int> st;

if(lo<hi){

int pivot = partition\_2(data,lo,hi);

st.push(lo);

st.push(pivot-1);

st.push(pivot+1);

st.push(hi);

while(!st.empty()){

hi = st.top();

st.pop();

lo = st.top();

st.pop();

if(lo<hi){

pivot = partition\_2(data,lo,hi);

st.push(lo);

st.push(pivot-1);

st.push(pivot+1);

st.push(hi);

}

}

}

}

**[快排应用扩展]**

**1、K-选取问题：众数、中位数**

* **众数/中位数问题(剑指offerP163)**

[问题描述]

无序向量A中，若有一半以上元素的数值同为m，则称m为A的众数。

[解决思路]

* 1. 利用排序法，时间复杂度为O(nlogn)；
  2. 利用基于Partition的方法，时间复杂度为O(n)，但会改变输入数组；
  3. 利用众数本身的性质：遍历时保存两个值：一个是数组中的一个数字，一个是次数。遍历到下一个数字的时候，若与之前保存的数字相同则次数+1，若不同次数-1。若次数为0，保存下一个数字，把次数置为1。则众数一定是最后一次把次数置为1时对应的数字。

[基于Partition的算法]

//亦可用来求中位数

int FindPlural(int data[],int length, int lo, int hi)

{ //没判断边界条件，自行判断

int middle = length>>1; //此处length作为输入 length=sizeof(data)/sizeof(int)

int index = partition\_2(data, lo, hi);

if(index == middle)

return data[index];

else if(index>middle)

return FindPlural(data,length, lo,index-1);

else

return FindPlural(data,length,index+1,hi);

}

[基于众数本身性质]

int FindPlural\_2(int data[],int length)

{

int num = data[0];

int cnt = 1;

for(int i=1;i<length;++i)

{

if(cnt == 0)

{

num = data[i];

cnt = 1;

}

else if(data[i]==num)

++cnt;

else

--cnt;

}

return num;

}

* **最小的K个数(剑指offerP167)**

[问题描述]

在任意一组可比较大小的元素中，如何找到最小的K个元素？

[解决思路]

1. 利用排序法，时间复杂度为O(nlogn)；
2. 利用基于Partition的方法，时间复杂度为O(n)，但会改变输入数组；
3. 利用数据结构—堆：节省空间，适合处理海量数据：先取出前k个元素，构建最大堆，然后依次遍历剩余元素。当该元素比堆的最大值要小，则删除堆顶，并将该元素加入堆中(下滤)。每次删除的时间为O(1)，加入元素的时间为O(logk)，总时间复杂度为O(nlogk)。在C++STL中set与multiset均是基于排序的的数据容器(用红黑树实现)，其查找、删除、和插入操作都只要O(logk)时间。

[基于Partition的算法—会修改输入数组]

void LeastK(int data[], int k, int lo, int hi)

{

int index = partition\_2(data,lo,hi);

if(index == k)

return;

else if(index>k)

return LeastK(data,k,lo,index-1);

else

return LeastK(data,k,index+1,hi);

}

[基于堆排序的算法—大数据]

typedef multiset<int, greater<int>> intSet; // Dgreater<T> ：#include<functional>

typedef multiset<int, greater<int>>::iterator setIterator; // multiset： #include<set>

void LeastK\_2(int data[],int length, intSet& leastNumbers,int k)

{

leastNumbers.clear();

if(k<1 || length<k)

return;

for(int i=0;i!=length;++i)

{

if(i<k)

leastNumbers.insert(data[i]);

else

{

setIterator interGreatest = leastNumbers.begin();

if(data[i]<\*interGreatest)

{

leastNumbers.erase(interGreatest ); //erase输入为 iterator类型

leastNumbers.insert(data[i]); //insert的输入为<T>类型

}

}

}

}

* **有序向量归并后的中位数**

[问题描述]

任给有序向量s1和s2，如何找出它们归并后的有序向量s中的中位数。

[解决思路]

利用划分策略，分而治之。

[简单的算法实现]

//默认两序列长度都为n，数据项可能重复

int median(vector<int>& s1, int lo1, vector<int>& s2, int lo2)

{

if(n<3) return trivialMedian(s1,lo1,n,s2,lo,n); //递归基，暴力搜索法

int mi1=lo1+n/2; //s1的中位数

mi2=lo2+(n-1)/2; //s2的逆向中位数

if(s1[mi1]<s2[mi2])

return median(s1,mi1,s2,lo2,n+lo1-mi1); //取s1右半，s2左半

else if(s1[mi1]<s2[mi2])

return median(s1,lo1,s2,mi2,n+lo2-mi2); //取s1左半，s2右半

else

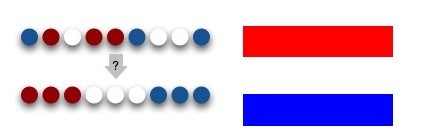
return s1[mi1];

}

* **荷兰国旗问题**

[问题描述]

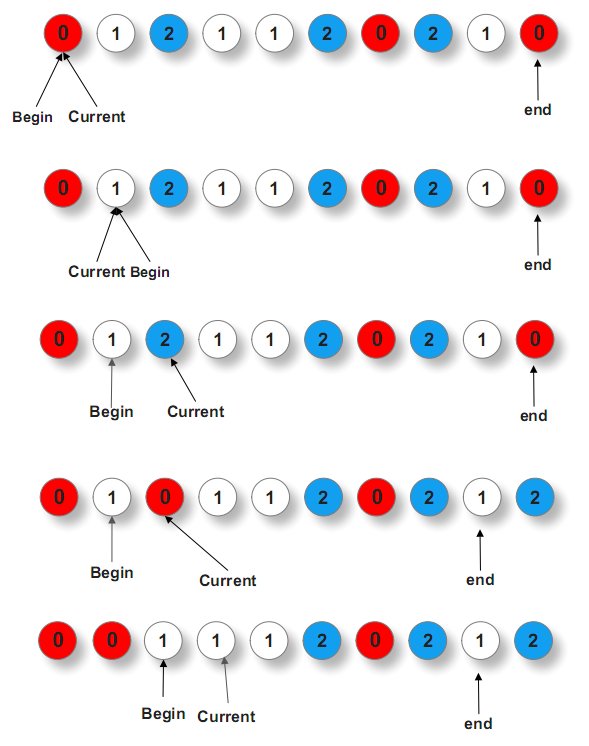
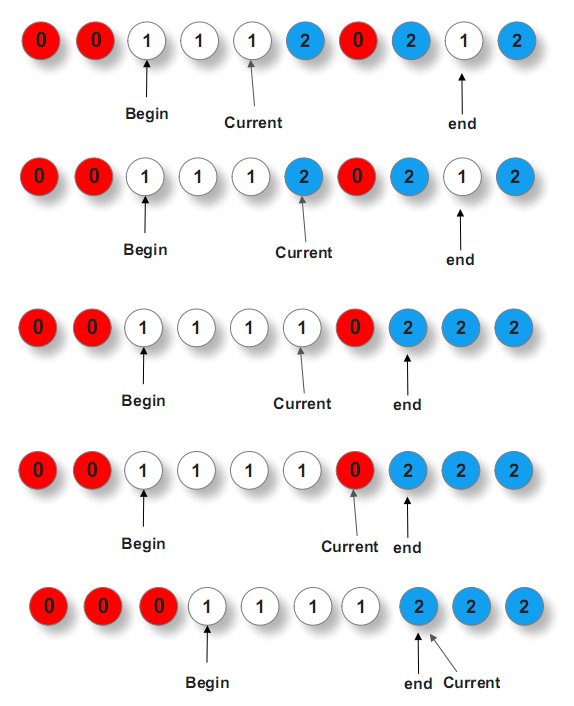
我们将乱序的红白蓝三色小球排列成有序的红白蓝三色的同颜色在一起的小球组。



[解决思路]

这个问题，类似快排中partition过程。不过，要用三个指针，一前begin，一中current，一后end，俩俩交换。0、1、2分别表示三种颜色的球。

1、current遍历整个数组序列，current指1时，current++，  
2、current指0，与begin交换，而后current++，begin++，  
3、current指2，与end交换，而后，current不动，end--。

时间复杂度为O(n)

   else if( array[current] == 1 )                    
      current++;                 
   else //When array[current] =2   
   {               
      swap(array[current],array[end]);              
      end--;            
   }      
}

[算法实现]

while( current<=end )        
{             
  if( array[current] ==0 )             
   {                 
      swap(array[current],array[begin]);                  
      current++;          
      begin++; //得让begin指到1，方便下次交换

 }

* **荷兰国旗问题扩展**

[描述]

假如颜色种类数超过3，将各色球排在一起。

[思路1]

利用分治法，每次排一种，放在中间，然后排两边。

[思路2]

利用STL。每次排一种颜色，放在还未排序的左边。

class Solution **{**

public**:**

void sortColors**(**vector**<**int**>&** nums**)** **{**

const int n **=** nums**.**size**();**

auto A **=** nums**.**begin**();**

auto tmp **=** nums**.**begin**();**

**for(**int i**=**0**;** i**<**2 **&&** tmp**!=**nums**.**end**();** i**++){**

tmp **=** partition**(**tmp**,** A **+** n**,**bind1st**(**equal\_to**<**int**>(),** i**));**

**}**

**}**

**};**

1. **简单排序**

时间复杂度为O(n2): 插入排序、选择排序和冒泡排序

1. **插入排序**

[基本思想]：首先考虑data中前两个元素，交换排序。遍历剩下的元素，根据值的大小插入到合适的位置上。

void insertSort(int data[], int length)

{

for(int i=0; i<length; ++i)

{

int temp = data[i];

int j;

for(j=i;j>0 && data[j-1]>temp;--j) //将所有比data[i]大的元素

data[j] = data[j-1]; //右移一个位置，注意判断条件

data[j] = temp;

}

}

[分析]: 最优复杂度：当输入数组就是排好序的时候，复杂度为O(n)，而快速排序在这种情况下会产生O(n2)的复杂度。内循环每次只需比较1次、移动2次，外循环共n-1次。总移动次数为2(n-1)，比较次数为(n-1)。

最差复杂度：当输入数组为倒序时，复杂度为O(n2)。插入排序比较适合用于“少量元素的数组”。移动次数过多，对于排好序的序列仍需移动。平均时间复杂度仍为O(n2)。内循环需要比较i次，移动i次，总移动次数、比较次数为n(n-1)/2。

平均复杂度：考虑等概率情况下，查找一个数据元素的平均比较次数为(i+1)/2，插入一个数据元素平均需要移动文件中全部数据元素的一半（i/2）。所以平均比较次数为(n+2)(n-1)/4;平均移动次数为：(n+2)(n-1)/4。复杂度为O(n2)。

稳定。

1. **选择排序**

[基本思想]：先找出数组中最小元素，将其与第一个元素换位，然后找到剩余元素中最小的，与第二个元素换位…循环遍历。

void selection(int data[], int n){

for(int i=0;i<n-1;++i){

int minId = i; //minId记录的是最小元素的位置

for(int j=i+1;j<n;++j)

if(data[j]<data[minId])

minId = j;

swap(data[i],data[minId]);

}

}

[分析]: 最好情况时间：O(n2)，最坏情况时间：O(n2)。请求(移动)次数少，但是交换次数为3(n-1)(或者可以通过if比较将交换次数减小，但比较次数增加)。比较次数n(n-1)/2。

不稳定。

1. **冒泡排序**

[基本思想]：遍历数组，如果相邻两个元素逆序，则交换。这样从尾到头一轮扫描之后最小的元素到了数组的第一个位置…然后循环遍历。

void bubblesort(int data[], int n)

{

for(int i=0;i<n-1;++i) //注意边界

for(int j=n-1;j>i;--j)

if(data[j]<data[j-1])

swap(data[j],data[j-1]);

}

[分析]: 时间复杂度O(n2)。缺点是一个元素要和数组中的每一个元素交换位置，而不能像选择排序一样跳过数组中的其他元素。

冒泡排序的比较次数近似于插入排序的两倍，在最好与最坏情况相同，等于n(n-1)/2。移动次数与插入排序相同。对于选择排序，比较次数一样，移动次数冒泡排序要多出n倍。

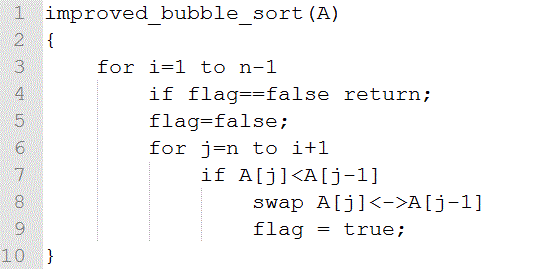
相等则不交换，所以稳定。（稳定的排序算法可以实现同时对多个关键码按照字典序列的排序）。

[思考]：在算法导论思考题2-2中又问了”冒泡排序和插入排序哪个更快“呢？

一般的人回答：“差不多吧，因为渐近时间都是O(n^2)”。

但是事实上不是这样的，插入排序的速度直接是逆序对的个数，而冒泡排序中执行“交换“的次数是逆序对的个数，因此冒泡排序执行的时间至少是逆序对的个数，因此插入排序的执行时间至少比冒泡排序快。

[改进的冒泡排序]：最佳运行时间：O(n), 最坏运行时间：O(n^2)。



[鸡尾酒排序]：

鸡尾酒排序也就是定向冒泡排序, 数组中的数字本是无[规律](http://baike.baidu.com/view/183124.htm)的排放，先找到最小的数字，把他放到第一位，然后找到最大的数字放到最后一位。然后再找到第二小的数字放到第二位，再找到第二大的数字放到倒数第二位。以此类推，直到完成排序。

鸡尾酒排序等于是冒泡排序的轻微变形。不同的地方在于从低到高然后从高到低，而冒泡排序则仅从低到高去比较序列里的每个元素。他可以得到比冒泡排序稍微好一点的效能，原因是冒泡排序只从一个方向进行比对(由低到高)，每次循环只移动一个项目。

以序列(2,3,4,5,1)为例，鸡尾酒排序只需要访问两次（升序降序各一次）次序列就可以完成排序，但如果使用冒泡排序则需要四次。

鸡尾酒排序最糟或是平均所花费的次数都是O(n²)，但如果序列在一开始已经大部分排序过的话，会接近O(n)。

function cocktail\_sort(list, list\_length) // the first element of list has index 0

{

    bottom = 0;

    top = list\_length - 1;

    swapped = true;

    bound = 0; //优化循环次数，记录已经排序的边界，减少循环次数

    while(swapped) {// if no elements have been swapped, then the list is sorted

        swapped = false;

        for(i = bottom; i < top; i = i + 1) {

            if(list[i] > list[i+1]) { // test whether the two elements are in the correct order

                swap(list[i], list[i+1]); // let the two elements change places

                swapped = true;

                bound = i;

            }

        }

        // decreases top the because the element with the largest value in the unsorted

        // part of the list is now on the position top

        //top = top - 1;

        top = bound;

        for(i = top; i > bottom; i = i - 1) {

            if(list[i] < list[i-1])

            {

                swap(list[i], list[i-1]);

                swapped = true;

                bound = i;

            }

        }

        // increases bottom because the element with the smallest value in the unsorted

        // part of the list is now on the position bottom

        //bottom = bottom + 1;

        bottom = bound;

    }

}

1. **线性对数阶排序O(nlogn)**

时间复杂度为O(nlogn)：快速排序、归并排序、堆排序、

**1、快速排序**

int partition (int data[], int lo, int hi)

{

if(data == NULL || lo<0 || (hi-lo)<0)

throw exception("Invalide Input!");

int pivot = data[hi];

int small=lo-1;

for(int index = lo;index<hi;++index)

{

if(data[index]<pivot)

{

++small;

if(small!=index)

swap(data[small],data[index]);

}

}

++small;

swap(data[small],data[hi]);

return small;

}

void QuickSort(int data[], int lo, int hi)

{

if(lo == hi)

return;

int index = partition (data,lo,hi);

if(index>lo)

QuickSort(data, lo,index-1);

if(index<hi)

QuickSort(data,index+1,hi);

}

[分析]:平均时间复杂度为O(nlogn) ，最坏时间复杂度O(n2)，原因是每次划分不平衡导致的。当元素完全相同时复杂度为O(n2)，此时可以用荷兰国旗问题写partition，时间复杂度降为O(n)。

不稳定。

**2、归并排序**

[基本思想]：归并(Merge)的主要过程是将排好序的子序列合并为一个有序序列，然后利用分治思想递归实现。

void merge(int data[], int first, int mid,int last) //此处是以左开右闭区间表示

{

int \* tmp = new int[last-first];

int i=first;

int j=mid;

int cnt=0;

while(i<mid && j<last)

{

if(data[i]<data[j])

tmp[cnt++] = data[i++];

else

tmp[cnt++] = data[j++];

}

while(i<mid)

tmp[cnt++] = data[i++];

while(j<last)

tmp[cnt++] = data[j++];

for(int k=0;k<cnt;++k)

data[first+k] = tmp[k];

delete []tmp;

}

void mergeSort(int data[], int first, int last)

{

if( (last-first)<2 )

return;

int mid = (last+first)>>1;

mergeSort(data,first,mid);

mergeSort(data,mid,last);

merge(data, first,mid, last);

}

int main()

{

int a[]={3,8,7,1,2,5,6,4,10};

int N = sizeof(a)/sizeof(int);

mergeSort(a,0,N);

for(int i=0;i<N;i++)

cout<<a[i]<<endl;

return 0;

}

[分析]: 是第一个可以在最坏情况下依旧保持O(nlogn)复杂度的确定性排序算法。归并排序需要额外的存储空间，这在处理大量数据时难以接受，可以通过链表来解决这个问题。  
　　空间复杂度为 O(n)  
　　比较操作的次数介于(nlogn) / 2和nlogn - n + 1。  
　　赋值操作的次数是(2nlogn)。归并算法的空间复杂度为：0 (n)  
　　归并排序比较占用内存，但却效率高且稳定的算法。

[改进]：Ｒ.ＳｅｄｇｅＷｉｃｋ提出了一种改进的两路归并算法。在把元素序列复制到辅助数组的过程中，把第二个有序表的元素逆序，这样两个待归并的表从两端开始处理，向中间开始处理，向中间归并。两个表的尾端互成“监视哨”，在归并的过程中可以省略检测子序列是否结束的判断，提高程序的执行效率。

void improvedMerge(int \*a,int left,int mid,int right){

    int s1=left;//s1,s2 是检测指针，t是存放指针

    int s2=right;

    int t=left,k;

    int \*b=new int[right];

    for(k= left ; k<=mid; k++)//正向复制

        b[k]=a[k];

    for(k=mid+1;k<=right;k++)//反向复制

        b[right+mid+1-k] = a[k];

    while( t <= right){//归并过程

        if(b[s1] <= b[s2])

a[t++]=b[s1++];

        else

a[t++]=b[s2--];

    }

}

**3、堆排序**

[基本思想]：首先根据Floyd建堆法建堆(时间复杂度为O(n))；通过，顶端最大的元素与最后一个元素不断的交换，交换后又不断重新维持最大堆的性质，最后，一个一个的，从大到小的，把堆中的所有元素都清理掉，也就形成了一个有序的序列。这就是堆排序的全部过程。

#include<iostream>

#include<functional> //greater<T>\less<T>

#include<algorithm> //swap

#include<vector>

using namespace std;

template<typename T>

class Heap

{

public:

Heap(const vector<T>& a\_array) //初始化

{

m\_array.assign(a\_array.begin(), a\_array.end());

}

template<typename Compare> //此处不可放在开头处

void sort(Compare comp);

void printHeap(const vector<T>& m\_array);

private:

vector<T> m\_array;

template<typename Compare>

void creatHeap(Compare comp);

template<typename Compare>

void downElement(int elemId, Compare comp);

};

template<typename T>

template<typename Compare>

void Heap<T>::sort(Compare comp)

{

printHeap(m\_array);

creatHeap(comp);

vector<T> tmp\_array;

for(int i = m\_array.size()-1;i>=0;i--)

{

tmp\_array.push\_back(m\_array[0]);

swap(m\_array[0],m\_array[i]);

m\_array.pop\_back();

downElement(0,comp);

}

printHeap(tmp\_array);

m\_array.assign(tmp\_array.begin(), tmp\_array.end());

}

template<typename T>

template<typename Compare>

void Heap<T>::creatHeap(Compare comp)

{

for(int i =m\_array.size()/2-1; i>=0;i--)

downElement(i, comp);

}

template<typename T>

template<typename Compare>

void Heap<T>::downElement(int elemId, Compare comp)

{

int min;

int index=elemId;

while(index\*2+1<m\_array.size())

{

min = index\*2+1; //记录左孩子和右孩子中最大/小值的位置

if(index\*2+2<m\_array.size())

if(comp(m\_array[index\*2+2],m\_array[min]))

min = index\*2+2;

if(comp(m\_array[index],m\_array[min]))

break;

else

{

swap(m\_array[min],m\_array[index]);

index = min;

}

}

}

template<typename T>

void Heap<T>::printHeap(const vector<T>& m\_array)

{

for(int i=0;i<m\_array.size();++i)

cout << m\_array[i]<<" ";

cout << endl;

}

int main()

{

vector<int> a\_array;

for(int i=10;i<20;i++)

a\_array.push\_back(i);

random\_shuffle(a\_array.begin(),a\_array.end());

Heap<int> heap(a\_array);

heap.sort(less<int>());

heap.sort(greater<int>());

return 0;

}

[分析]: 时间复杂度：O(nlgn)，等同于归并排序。最坏：O(nlgn)，空间复杂度：O(1)。  
不稳定。

**四、O(n1+￡)阶排序**

￡是介于0和1之间的常数，即0<￡<1，如希尔排序；

**1、希尔排序**

[基本思想]：将输入数组data分成不同子组，对每个子组采用简单的排序(如插入排序，直插排序在关键码基本有序的情况下，效率是最好的)。先取一个小于n的整数d1作为第一个增量，把文件的全部记录分成d1个组。所有距离为dl的倍数的记录放在同一个组中。先在各组内进行直接插入排序；然后，取第二个增量d2<d1重复上述的分组和排序，直至所取的增量dt=1(dt<dt-l<…<d2<d1)，即所有记录放在同一组中进行直接插入排序为止。

void shellSort(int data[], int n)

{

for(int gap = n/2; gap>0; gap/=2)

for(int i=0;i<gap;++i)

for(int j=i+gap; j<n; j += gap)

{

int tmp = data[j];

int k = j-gap;

while(k>=0 && data[k]>tmp)

{

data[k+gap] = data[k];

k -= gap;

}

data[k+gap] = tmp;

}

}

**五、线性阶(O(n))排序**  
    　如桶排序和基数排序

1. **桶排序**

[基本思想]：桶排序不是基于比较的排序，最好的时间复杂度可以达到O(n)。桶排序是基于hash的应用。把[a, b]按照某种hash函数划分为n个子区间，每一子区间是一个桶，然后将记录分配到各个桶中。由于同一桶中的记录其关键字不尽相同，所以必须采用关键字比较的排序方法(通常用插入排序)对各个桶进行排序，然后依次将各非空桶中的记录连接(收集)起来即可。对于海量数据而言，特别是当内存一次装不下的时候，桶排序是很有效的。

#include<iostream>

#include <algorithm>

using namespace std;

struct node

{

int data;

node\* next;

};

void bucket(node\* buc[], int data[], int n)

{

for(int i=0;i<n;++i)

{

int num = data[i];

node\* head = buc[num/10]; //以十位为基准分组/桶

node\* tmp = new node; //很多情况下需要将桶号与0,1,2…对应起来

tmp->data = num;

tmp->next = NULL;

if( head->next == NULL)

head->next = tmp;

else

{

while(head->next!= NULL && head->next->data < num)

head = head->next;

tmp->next = head->next;

head->next = tmp;

}

}

}

int main()

{

int\* data = new int[100];

node\*\* buc = new node\*[10]; //此处new了一个指针数组，接下来需要对

for(int i=0; i<100; ++i) //对数组中的每个指针元素继续分配空间

data[i] = i ;

random\_shuffle(&data[0], &data[99]);

for(int i=0; i<10; ++i)

{

buc[i] = new node; //重新分配空间

buc[i]->next = NULL;

}

bucket(buc,data,100);

for(int i=0;i<10;++i)

{

node\* tmp = buc[i]->next;

while( tmp!=NULL )

{

cout<<tmp->data <<endl;

tmp = tmp->next;

}

}

delete []data;

delete []buc;

return 0;

}

[分析] 桶排序是另外一种以O(n)或者接近O(n)的复杂度排序的算法。

Note: 待排序元素越均匀, 桶排序的效率越高. 均匀意味着每个桶在中间过程中容纳的元素个数都差不多,不会出现特别少或者特别多的情况, 这样在排序子程序进行桶内排序的过程中会达到最优效率.

Note: 将元素通过恰当的映射关系(Hash函数)将元素尽量等数量的分到各个桶(值区间)里面, 这个映射关系就是桶排序算法的关键.桶的标记(数组索引Index)的大小也要和值区间有对应关系。

桶排序虽然很强大，但是用错了地方还是不行的，用错地方其复杂度可以达到O(n^2)：对于数据量超级大，但是数据的范围确实在一个不是很大的区间内，那么用桶排序是非常有效的！特别是当均匀分布的时候，更高效。例如：考试成绩0--100分，可以分成10个桶，那么使用桶排序统计每个分数段的处理就是很优雅的！

1. **基数排序**

[基本思想]：基数排序(radixsort)可以理解为带有优先级的多次桶排序。如对数字的排序，第一次桶排序以个位数字为基准，然后依次以十位、百位为基准…多次桶排序之后可以得到最终的排序结果。因涉及到多次排序，每个桶内的多个数字不做排序，而是以队列结构存储，方便下一趟排序时先进先出。

void radixsort(long data[], int n)

{

const int radix = 10; //基数

const int digits = 10; //long数据类型的最大位数

queue<long> queues[radix]; //radix个桶

for(int i=0,factor=1;i<digits;factor\*= radix,++i)

{

for(int j=0;j<n;++j)

queues[ (data[j]/factor)%radix ].push(data[j]); //将data[j]放入合适的桶/

int k=0; //队列中

for(int j=0;j<radix;++j)

while( !queues[j].empty())

{

data[k++] = queues[j].front();

queues[j].pop();

}

}

}

[分析]

基数排序的时间复杂度为 O(t\*(n+m))，其中t为桶排序的次数，m为桶数。



说明：基数排序的最好、最坏和平均时间复杂度都是O(d(n+r))。（上面的基数排序使用的是队列数组）当分配的是指针数组时，基数排序一趟排序需要的辅助空间是r（创建r个队列）但是以后的排序中重复使用这些队列，所以空间复杂度为O(r).

1. **关于排序的题目**

**1、合并两个排序链表（剑指offerP114）**

[基本思想]：给定两个递增排序的链表，合并这两个链表使新链表中结点依旧按照递增排序。分析时注意不能让链表中断、保持链表递增、注意特殊输入。

[方法1]：利用递归实现排序链表的Merge，不设为空的头指针

Node\* Merge(Node\* head1, Node\* head2){

if(head1 == NULL)

return head2;

if(head2 == NULL)

return head1;

Node\* pMerge = NULL;

if(head1->data < head2->data){

pMerge = head1;

pMerge->next = Merge(head1->next, head2);

}

else{

pMerge = head2;

pMerge->next = Merge(head2->next, head1);

}

return pMerge;

}

Node \*MergeSort(Node \*head)

{

if(head == NULL)

return 0;

Node \* r\_head = head;

Node \*head1 = head;

Node\* head2 = head;

while(head2->next != NULL && head2->next ->next!= NULL)

{

head1 = head1->next;

head2 = head2->next->next;

}

if(head1->next == NULL)/\*说明只有一个节点，则返回该节点\*/

return r\_head;

head2 = head1->next;

head1->next = NULL;

head1 = head;

/\*函数MergeList是对两个有序链表进行归并，返回值是归并后的链表的头结点\*/

r\_head = MergeList(MergeSort(head1), MergeSort(head2));

return r\_head;

}

**2、合并两个排序数组，不开额外空间（LeetCode）**

Given two sorted integer arrays A and B, merge B into A as one sorted array.

Note: You may assume that A has enough space to hold additional elements from B. The number of elements initialized in A and B are m and n respectively.

//LeetCode, Merge Sorted Array

// 时间复杂度 O(m+n)，空间复杂度 O(1)

class Solution {

public:

void merge(int A[], int m, int B[], int n) {

int ia = m - 1, ib = n - 1, icur = m + n - 1;

while(ia >= 0 && ib >= 0) {

A[icur--] = A[ia] >= B[ib] ? A[ia--] : B[ib--];

}

while(ib >= 0) {

A[icur--] = B[ib--];

}

}

};

1. **Merge k Sorted Lists（LeetCode）**

//LeetCode, Merge k Sorted Lists

// 时间复杂度 O(n1+n2+...)，空间复杂度 O(1)

class Solution {

public:

ListNode \*mergeKLists(vector<ListNode \*> &lists) {

if (lists.size() == 0) return nullptr;

ListNode \*p = lists[0];

for (int i = 1; i < lists.size(); i++) {

p = mergeTwoLists(p, lists[i]);

}

return p;

}

// Merge Two Sorted Lists

ListNode \*mergeTwoLists(ListNode \*l1, ListNode \*l2) {

ListNode head(-1);

for (ListNode\* p = &head; l1 != nullptr || l2 != nullptr; p = p->next) {

int val1 = l1 == nullptr ? INT\_MAX : l1->val;

int val2 = l2 == nullptr ? INT\_MAX : l2->val;

if (val1 <= val2) {

p->next = l1;

l1 = l1->next;

} else {

p->next = l2;

l2 = l2->next;

}

}

return head.next;

}

};

1. **Insertion Sort List（LeetCode）**

// LeetCode, Insertion Sort List

// 时间复杂度 O(n^2)，空间复杂度 O(1)

class Solution {

public:

ListNode \*insertionSortList(ListNode \*head) {

ListNode dummy(INT\_MIN);

//dummy.next = head;

for (ListNode \*cur = head; cur != nullptr;) {

auto pos = findInsertPos(&dummy, cur->val);

ListNode \*tmp = cur->next;

cur->next = pos->next;

pos->next = cur;

cur = tmp;

}

return dummy.next;

}

ListNode\* findInsertPos(ListNode \*head, int x) {

ListNode \*pre = nullptr;

for (ListNode \*cur = head; cur != nullptr && cur->val <= x;

pre = cur, cur = cur->next);

return pre;

}

};

1. **Sort a linked list in O(nlogn) time using constant space complexity. （LeetCode）**

[分析]：常数空间且 O(nlogn)，单链表适合用归并排序，双向链表适合用快速排序。本题可以复用”Merge Two Sorted Lists” 的代码。

// LeetCode, Sort List

// 归并排序，时间复杂度 O(nlogn)，空间复杂度 O(1)

class Solution {

public:

ListNode \*sortList(ListNode \*head) {

if (head == NULL || head->next == NULL)return head;

// 快慢指针找到中间节点

ListNode \*fast = head, \*slow = head;

while (fast->next != NULL && fast->next->next != NULL) {

fast = fast->next->next;

slow = slow->next;

}

// 断开

fast = slow;

slow = slow->next;

fast->next = NULL;

ListNode \*l1 = sortList(head); // 前半段排序

ListNode \*l2 = sortList(slow); // 后半段排序

return mergeTwoLists(l1, l2);

}

// Merge Two Sorted Lists

ListNode \*mergeTwoLists(ListNode \*l1, ListNode \*l2) {

ListNode dummy(-1);

for (ListNode\* p = &dummy; l1 != nullptr || l2 != nullptr; p = p->next) {

int val1 = l1 == nullptr ? INT\_MAX : l1->val;

int val2 = l2 == nullptr ? INT\_MAX : l2->val;

if (val1 <= val2) {

p->next = l1;

l1 = l1->next;

} else {

p->next = l2;

l2 = l2->next;

}

}

return dummy.next;

}

};

1. **First Missing Positive. （LeetCode）**

Given an unsorted integer array, find the first missing positive integer.

For example, Given [1,2,0] return 3, and [3,4,-1,1] return 2.

Your algorithm should run in O(n) time and uses constant space.

[分析]：本质上是桶排序 (bucket sort)，A[i]应该等于i+1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| i | 0 | 1 | 2 | 3 |
| A[i] | 3 | 4 | -1 | 0 |

应该为

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| i | 0 | 1 | 2 | 3 |
| A[i] | 1 | 2 | 3 | 4 |

所以当A[i]!=i+1时，A[i]里面的数值最终的位置应该放在A[i]-1下标处，所以A[i]!=i+1时，swap(A[i], A[A[i] - 1]);当然在swap前需要加判断条件，以防越界（(A[i] <= 0 || A[i] > n)以及死循环（A[i] == A[A[i] - 1])。

每当 A[i]!= i+1 的时候，将 A[i] 与 A[A[i]-1] 交换，直到无法交换为止，终止条件是 A[i]== A[A[i]-1]，当前这个数已经放在了它应该的位置，进入下次循环。

// LeetCode, First Missing Positive

// 时间复杂度 O(n)，空间复杂度 O(1)

class Solution {

public:

int firstMissingPositive(int A[], int n) {

bucket\_sort(A, n);

for (int i = 0; i < n; ++i)

if (A[i] != (i + 1))

return i + 1;

return n + 1;

}

private:

static void bucket\_sort(int A[], int n) {

for (int i = 0; i < n; i++) {

while (A[i] != i + 1) {

if (A[i] <= 0 || A[i] > n || A[i] == A[A[i] - 1])

break;

swap(A[i], A[A[i] - 1]);

}

}

}

};

同类型的一道百实习生度面试题：

**6.2有N个自然数（1-N），乱序存放在下标为1~n的数组空间中，实现对数组从小到大排序，不能直接使用赋值操作，要求时间复杂度O(n)，空间复杂度O(1)。**

解答：本质上是桶排序

这个条件比较特殊，所以可以实现这个条件比较特殊，所以可以实现O(n)的算法，也可以不交换2个数。如果a[j]!=j，则把a[j]中的元素放到它应在的位置，同时把要被覆盖的元素取出来。不停循环，一直到a[j] = j;

简单写一下代码：

for (i=1; i<=n; i++)

{

j = i;

while (a[j] != j)

{

cache = a[a[j]];

a[a[j]] = a[j];

a[j] = cache;

}

}

虽然for 中嵌套了一个while，但最多循环2次。

1. **Maximum Gap（LeetCode164）**

Given an unsorted array, find the maximum difference between the successive elements in its sorted form.

Try to solve it in linear time/space.

Return 0 if the array contains less than 2 elements.

You may assume all elements in the array are non-negative integers and fit in the 32-bit signed integer range.

Input:**[3,6,9,1]**

Output:**8**

Expected:**3**

分析：// 时间空间都是O(n)

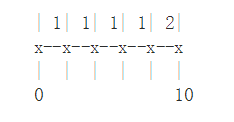
函数的值域是0（所有元素相同）到max−min。

比较排序的时间复杂度是O(nlogn)，而题目要求O(n)。可用的排序算法有桶排序、计数排序、基数排序。

第二，假设数组中最小值是min，最大值是max，数组长度为n，则有

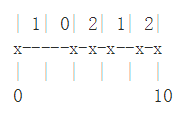
MaxGap⩾（Max−Min）/(n−1)

例如



这里把数组划分为虚拟的n-1组，设每组含头不含尾（除了最后一组），那么每组都划分到1个元素，即最小值（0）和最大值（10）之间的元素是均匀分布的。间隔都是10/(6-1)=2。MaxGap=2。

如果有元素没有平均分布，



那么MaxGap就会大于2。同时可见第二组没有划分到元素，而第三组有2个元素。

把上面的均匀划分的组作为桶，应用桶排序的思想，把数组元素归到桶中。在一个桶内，元素差小于等于block.max-block.min。在两桶之间，元素差等于block2.min-block1.max。

public class Solution **{**

public static int Max**(**int**[]** num**){**

int max **=** Integer**.**MIN\_VALUE**;**

**for(**int i**=**0**;** i**<**num**.**length**;** i**++){**

max **=** Math**.**max**(**max**,** num**[**i**]);**

**}**

**return** max**;**

**}**

public static int Min**(**int**[]** num**){**

int min **=** Integer**.**MAX\_VALUE**;**

**for(**int i**=**0**;** i**<**num**.**length**;** i**++){**

min **=** Math**.**min**(**min**,** num**[**i**]);**

**}**

**return** min**;**

**}**

public int maximumGap**(**int**[]** num**)**

**{**

**if** **(**num **==** null **||** num**.**length **<** 2**)**

**return** 0**;**

//不包含gqqnbig包了

/\*int min = gqqnbig.util.Arrays.min(num);

int max = gqqnbig.util.Arrays.max(num);\*/

int min **=** Min**(**num**);**

int max **=** Max**(**num**);**

// the minimum possible gap, ceiling of the integer division

int gap **=** **(**int**)** Math**.**ceil**((**double**)** **(**max **-** min**)** **/** **(**num**.**length **-** 1**));**

List**<**Integer**>[]** blocks **=** bucketSort**(**num**,** num**.**length **-** 1**);**

// scan the buckets for the max gap

int maxGap **=** Integer**.**MIN\_VALUE**;**

int previousMax **=** min**;**

**for** **(**int i **=** 0**;** i **<** blocks**.**length**;** i**++)**

**{**

**if** **(**blocks**[**i**].**isEmpty**())**

**continue;**

int bucketsMIN **=** Collections**.**min**(**blocks**[**i**]);**

int bucketsMAX **=** Collections**.**max**(**blocks**[**i**]);**

// maxGap = gqqnbig.Math.max(maxGap, bucketsMAX - bucketsMIN, bucketsMIN - previousMax);

maxGap **=** Math**.**max**(**Math**.**max**(**maxGap**,** bucketsMAX **-** bucketsMIN**),** bucketsMIN **-** previousMax**);**

// bucketsMAX - bucketsMIN是为了防止所有数据在一个桶里面

// update previous bucket value

previousMax **=** bucketsMAX**;**

**}**

//maxGap = Math.max(maxGap, max - previousMax);//不用这句

**return** maxGap**;**

**}**

/\*\*

\* 用桶排序算法对arr数组排序，桶的数量由bucketCount指定。

\* <p>

\* 如果bucketCount等于max-min，则返回值退化为int[]； 否则每个桶可能含有多于一个元素。

\* </p>

\* 时间复杂度O(arr.length)，空间复杂度O(arr.length)。

\*

\* **@param** arr

\* **@return**

\*/

// 调用时，bucketCount = n-1

public static List**<**Integer**>[]** bucketSort**(**int**[]** arr**,** int bucketCount**)**

**{**

//定义了bucketCount个桶，每个桶假定只放一个元素，实际上有的桶放了多个元素

List**<**Integer**>[]** buckets **=** **new** List**[**bucketCount**];**

**for** **(**int i **=** 0**;** i **<** buckets**.**length**;** i**++)**

buckets**[**i**]** **=** **new** LinkedList**<**Integer**>();**

// int min = gqqnbig.util.Arrays.min(arr);

// int max = gqqnbig.util.Arrays.max(arr);

int min **=** Min**(**arr**);**

int max **=** Max**(**arr**);**

int gap **=** **(**int**)** Math**.**ceil**((**max **-** min**)** **/** **(**double**)** bucketCount**);**

// 桶的范围是

// [min, min+gap), [min+gap， min+2gap), ... [min+(bucketCount-1)\*gap, min+bucketCount\*gap]

// min+bucketCount\*gap>=max

**for** **(**int i **=** 0**;** i **<** arr**.**length**;** i**++)**

**{**

//算出应该放入桶的下标

int bucketIndex **=** **(**arr**[**i**]** **-** min**)** **/** gap**;**

**if** **(**bucketIndex **==** bucketCount**)** // 桶的范围算头不算尾，但最后一个桶要算尾。最后一个桶是闭合的

bucketIndex**--;**

buckets**[**bucketIndex**].**add**(**arr**[**i**]);**

**}**

**return** buckets**;**

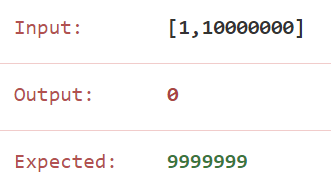
**}**

**}**

**说明：**

maxGap **=** Math**.**max**(**Math**.**max**(**maxGap**,** bucketsMAX **-** bucketsMIN**),** bucketsMIN **-** previousMax**);**

// bucketsMAX - bucketsMIN是为了防止所有数据在一个桶里面，**假如去掉的话下面的测试用例不能通过**



1. **Sort Color. （LeetCode）**

Given an array with n objects colored red, white or blue, sort them so that objects of the same color are adjacent, with the colors in the order red, white and blue.

Here, we will use the integers 0, 1, and 2 to represent the color red, white, and blue respectively.

Note: You are not suppose to use the library’s sort function for this problem.

[分析]：荷兰国旗问题

// LeetCode, Sort Colors

// 双指针，时间复杂度 O(n)，空间复杂度 O(1)

class Solution {

public:

void sortColors(int A[], int n) {

// 一个是 red 的 index，一个是 blue 的 index，两边往中间走

int red = 0, blue = n - 1;

for (int i = 0; i < blue + 1;) {

if (A[i] == 0)

swap(A[i++], A[red++]);

else if (A[i] == 2)

swap(A[i], A[blue--]);

else

i++;

}

}

};