排序 1

#sort函数的cmp函数编写

cmp函数如果返回真值，那么sort就不会对其做出改变
如果cmp返回假值，那么sort就会交换两个元素的位置。

cmp函数要求当a=a时，cmp返回假值。

#冒泡排序

~~不必多言~~

#插入排序

插入排序原理为先对数组进行遍历，当遇到顺序与要求不符的部分时，将数字依次向前移动，直到遇到了相应的部分，然后交换数字。

```
# include<bits/stdc++.h>
# include<algorithm>
typedef long long ll;
using namespace std;
void insert_sort(int arr[],int n)
{
        int j;
        int temp;
        for(int i=1;i<n;i++)
        {
                if(arr[i]<arr[i-1])
                temp=arr[i];
                for( j=i-1;j>=0&&arr[j]>temp;j--)    //这里容易误写成j=i注意
                {
                        arr[j+1]=arr[j];
                }
                arr[j+1]=temp;
        }
}
int main (void)
{
        int arr[10];
        for(int i=0;i<10;i++)
        arr[i]=rand()%(114514-i+1)+1;

        insert_sort(arr,10);
```

```
        for(int i=0;i<10;i++)
        cout<<arr[i]<<" ";
        return 0;

}
```

#快速排序

这部分不做赘述，通常只用stl，数据结构应付一下考试即可

```
# include<bits/stdc++.h>
typedef long long ll;
using namespace std;
void quick_sort(int arr[],int l,int r)
{
        if(l>=r)    //注意这里的大小关系非常容易弄混
        return;
        int i=l-1;
        int j=r+1;
        int x=arr[(l+r)/2];
        while(i<j)
        {
                do i++;while(arr[i]<x);
                do j--;while(arr[j]>x);
                if(i<j)
                swap(arr[j],arr[i]);
        }
        quick_sort(arr,l,j);
        quick_sort(arr,j+1,r);

}
int main (void)
{
        int arr[1155];
        for(int i=0;i<15;i++)
        arr[i]=rand()%(114514-i+1)+1;
        for(int i=0;i<15;i++)
        cout<<arr[i]<<" ";
        cout<<endl;

        quick_sort(arr,0,14);
        for(int i=0;i<15;i++)
        cout<<arr[i]<<" ";
        return 0;
```

```
}
```

#归并排序

思想与快速排序很类似。本质还是分治。
归并排序以数组中间为断点，分别将左边和右边排序好之后合并成一个新的有序堆。
复杂度是o(nlog(n))；
空间复杂度是o(n)；
归并是一种稳定排序算法。

```
# include<bits/stdc++.h>
using namespace std;
typedef long long  ll;
ll temp[114514];
void merge_sort(ll arr[],ll l,ll r)
{
        if(l>=r)
        return;
        ll mid=l+r>>1;
        merge_sort(arr,l,mid);
        merge_sort(arr,mid+1,r);
        ll k=0,i=l,j=mid+1;
        while(i<=mid&&j<=r)
        {
                if(arr[i]<arr[j]) temp[k++]=arr[i++];
                else temp[k++]=arr[j++];
        }
        while(i<=mid)  temp[k++]=arr[i++];
        while(j<=r) temp[k++]=arr[j++];
        for(int i=l,j=0;i<=r;i++,j++)
        arr[i]=temp[j];
}
int main (void)
{
        ll arr[114514];
        for(int i=0;i<=17;i++)
        arr[i]=rand()%(114514-i+1)+1;
        merge_sort(arr,0,17);
        for(int i=0;i<=17;i++)
        cout<<arr[i]<<endl;
```

```
}
```

#堆排序

这里的堆不是stl里面的堆，而是手写模拟的堆。堆的本质就是完全二叉树。

（1）堆排序需要满足几种要求：
1.求集合当中最小值/最大值
2.插入数据
3.删除最小值/最大值
4.删除任何一个元素
5.修改任何一个元素
堆是一颗完全二叉树，除了最后一层节点之外所有节点的叶子节点都是满的。
主要有小根堆和大根堆两种，下面主要介绍小跟堆。
小根堆意为任何一个节点的值都比他的叶子结点小，因此该树的根节点对应的值是最小值。

（2）堆的存储：（优先队列）
以数组存储，任何节点i的左儿子为2 *i ;右儿子是2* i+1；
手动堆所要满足的操作主要有两个：
up与down
up操作：对堆中的所有元素，查看是否小于其父节点，如果是，则交换位置；
down: 对堆中的所有元素，查看是否小于其叶子节点，如果是，则交换位置；
如此我们可以得到之前提到的操作的实现方法。

```
        1.求集合当中最小值/最大值    heap[1]
        2.插入数据                  heap[++size]=x;up(size);
        3.删除最小值/最大值          heap[1]=heap[size];size--;down(1);
        4.删除任何一个元素           与上述操作相同，heap[k]=heap[size];size-
-;up(k);down(k);
        5.修改任何一个元素

==注意下标的起点是1，因为一旦起点是0那么左儿子和右儿子的标签值将会一样
```

```
# include<bits/stdc++.h>
# include<algorithm>
typedef long long ll;
using namespace std;
ll h[114514];
ll size;
void down(ll u)
{
```

```
        ll t=u;
        if(u*2<=size&&h[2*u]<h[t]) t=2*u;
        if(u*2+1<=size&&h[t]>h[2*u+1]) t=2*u+1;
        if(t!=u)
        {
        swap(h[t],h[u]);
        down(t);
        }                    //注意这里带着括号
}
int main (void)
{
        cin.tie(0);
        cout.tie(0);
        for(int i=1;i<=16;i++)
        h[i]=rand()%(114514-i+1)+1;
        size=16;
        for(int i=16/2;i;i--)
        down(i);
        ll now=size;
        while(now--)
        {
                cout<<h[1]<<endl;
                h[1]=h[size];
                size--;
                down(1);
        }

        return 0;

}
```

堆排序的复杂度是0(nlog(n))；在这里的建堆方式复杂度是0(n)的，对堆中的元素进行排序的复杂度是o(log(n))；

#希尔排序

按照一定的序号差值依次取出若干个元素进行排序，排完再插回原数组，不断缩小/扩大序号gap，最后得到完整序列。

（貌似不太可能考手搓代码，不会写只找到一份java版）

```
  // 希尔排序

                public static void shellSort(inta[]) {
```

```
                        intd= a.length;//gap的值
                        while (true){
                                d = d/ 2;//每次都将gap的值减半
                                for (int x = 0; x< d; x++) {//对于gap所分的每一个
组
                                        for (int i = x+ d; i < a.length; i= i + d)
{       //进行插入排序
                                                int temp= a[i];
                                                intj;
                                                for (j= i - d; j>= 0 && a[j] >
temp;j = j - d){
                                                        a[j+ d] = a[j];
                                                }
                                                a[j+ d] = temp;
                                        }
                                }
                                if (d== 1) {//gap==1，跳出循环
                                        break;
                                }
                        }
        }
```

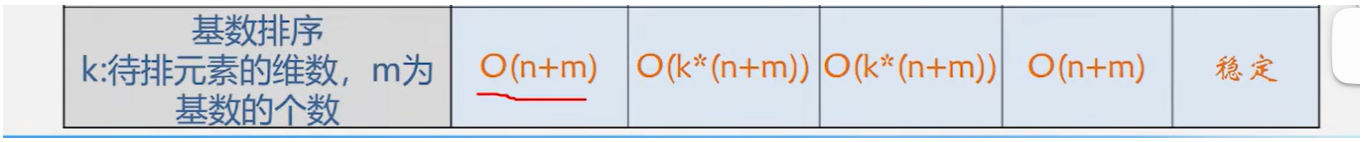
#基数排序/桶排序

复杂度：o（k*(n+m)）（k是提取的关键字个数，n是待排序的数字个数，M是关键字的个数）

> 例如对九个三位数进行排序，那么k就是三（个十百），N就是9，M是10

空间复杂度为o(n+m).因此在排序时如果允许按照关键字进行查找，基数排序的速度非常快。如果元素的关键字取值范围不一定那么用不了基数排序。

排序名称 最好时间复杂度 最坏情况 -平均情况 空间复杂度 稳定性




```
int maxbit(int data[], int n) //辅助函数，求数据的最大位数
{
    int maxData = data[0];      ///< 最大数
    /// 先求出最大数，再求其位数，这样有原先依次每个数判断其位数，稍微优化点。
    for (int i = 1; i < n; ++i)
    {
        if (maxData < data[i])
            maxData = data[i];
    }
    int d = 1;
    int p = 10;
    while (maxData >= p)
    {
        //p *= 10; // Maybe overflow
        maxData /= 10;
        ++d;
    }
    return d;
/*    int d = 1; //保存最大的位数
    int p = 10;
    for(int i = 0; i < n; ++i)
    {
        while(data[i] >= p)
        {
            p *= 10;
            ++d;
        }
    }
    return d;*/
```

```
}
void radixsort(int data[], int n) //基数排序
{
    int d = maxbit(data, n);
    int *tmp = new int[n];
    int *count = new int[10]; //计数器
    int i, j, k;
    int radix = 1;
    for(i = 1; i <= d; i++) //进行d次排序
    {
        for(j = 0; j < 10; j++)
            count[j] = 0; //每次分配前清空计数器
        for(j = 0; j < n; j++)
        {
            k = (data[j] / radix) % 10; //统计每个桶中的记录数
            count[k]++;
        }
        for(j = 1; j < 10; j++)
            count[j] = count[j - 1] + count[j]; //将tmp中的位置依次分配给每个桶
        for(j = n - 1; j >= 0; j--) //将所有桶中记录依次收集到tmp中
        {
            k = (data[j] / radix) % 10;
            tmp[count[k] - 1] = data[j];
            count[k]--;
        }
        for(j = 0; j < n; j++) //将临时数组的内容复制到data中
            data[j] = tmp[j];
        radix = radix * 10;
    }
    delete []tmp;
    delete []count;
}
```

这份代码来自知乎

#排序的稳定性、复杂度

各种排序算法如果在排序完成之后，大小相同的元素的先后顺序不变，则称这种排序算法是稳定的。
各种常见排序算法的稳定性和复杂度见下：

类别 最好情况 最坏情况 平均 空间复杂度 是否稳定

插入排序	直接插入排序	O(n)	$O(n^2)$	$O(n^2)$	O(1)	稳定
	希尔排序	O(n)	$O(n^2)$	$\sim O(n^{1.3})$	O(1)	不稳定
交换排序	冒泡排序	O(n)	$O(n^2)$	$O(n^2)$	O(1)	稳定
	快速排序	O(nlogn)	$O(n^2)$	O(nlogn)	O(nlogn)	不稳定
选择排序	直接选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	不稳定
	堆排序	O(nlogn)	O(nlogn)	O(nlogn)	$O(n^2)$	不稳定
归并排序		O(nlogn)	O(nlogn)	O(nlogn)	O(n)	稳定
基数排序 k:待排元素的维数，m为基数的个数		O(n+m)	O(k*(n+m))	O(k*(n+m))	O(n+m)	稳定

值得注意的是, stl中的sort可以修改bool函数改成稳定的。