sort algorithm

> 问题描述：给定一个乱序数组，将其排序成为有序数组
>
> 输入：无序数组和数组长度
>
> 输出：有序数组(下面的全部是按照升序)

常见的sort算法主要是以下几类，这些算法都是在不同的角度定义：有序，然后才长得千奇百怪。

> - bubble_sort $O(n^2)$
> - insertion_sort $O(n^2)$
> - mergesort O(nlogn)
> - combine_sort O(nlogn)
> - quick_sort O(nlogn)
> - heap_sort O(nlogn)
> - hash_sort 以后再加

一些联想：

> 目前的排序算法，能够达到nlogn的复杂度就算是好算法了，但是一直有一个疑问困扰着我，当所有的数组元素都已经知道了，难道它们之间的大小关系还不能够知道吗？为什么在知道所有元素的瞬间不能够马上知道数组元素正确的次序？要说原因也能够勉强说出一个：比如一个数组的信息不单单是所有元素是什么，而且还有元素之间的关系；
>
> 之所以产生这种疑问，是因为在人的思维之中有一种排序的好办法，数形结合，画一根数轴，然后把读取的数字依次给放到数轴上正确的位置，那么最后似乎可以直接很直观的看出元素之间的大小关系，读完数字的时候就是结果出来的时候。这是一个美好的愿景，但是或许并不能实现。考虑具体的场景，当给定的数字足够大的时候，当初画的短短的数轴就不能用了，要把数轴给延长，然后才能容纳下全部的数字，而且数字之间过于接近的时候，也要把数轴某一段给拉长，才能够较为清晰的显示出数字之间的关系。所以，这种方法似乎没有看起来的那么好，也有代价。

> 假设存在理想的无限可分的数轴，那么这种方法似乎可以在在N的时间之内解决排序问题。但是在计算机的世界之中不存在无限可分的东西，精度越高存储空间消耗的越大，范围越广存储空间消耗也越大，要求越高代价越大。不过最接近数轴思维的排序就是hash_sort，借助哈希映射，找到合适的位置。
>
> 实际上可以通过证明得出所有基于比较的排序算法，也就是CBA算法，nlogn就是它的复杂度下界，不可能比nlogn更低。

algorithm

1.bubble_sort

- 排序思路:每一次循环依次比较相邻元素，然后让较大元素后移。它基于这样的视角：那就是有序数组，就是从末尾，慢慢变小的数组。所以每一次的目标都是为了让当前无序数组之中最大的元素归位，然后下一次只考虑除去末尾的其余部分。可以很直观的发现，由于依次让最大元素归位，因此它的算法正确性是可以得到保证的。

代码:

```
void swap(int& a,int& b){
    int t=a;
    a=b;
    b=t;
}

void bubble_Sort(int* a,int length){
    while(true){
        int sig=1;//这是有序标志
        for(int i=0;i<length-1;i++){
            if(a[i]>a[i+1]){
                swap(a[i],a[i+1]);
                sig=0;
            }
        }
        /*if() break;
        这个地方值得一提，大循环跳出去的方式有两种:
        1.由于每次都有一个元素归位，那么小循环执行length-1次后一定有序
        2.提前有序的判断，如果某一次没有发生swap()，那么该次就一定已经有序了，可以跳出
        */
        if(sig==1)break;
    }
}
```

其实基于每一次使得最大元素归位的写法还有，比如直接找出最大的元素，然后把它和末尾的元素交换，然后再找出次大的元素，让他和倒数第二位交换，一直到换无可换。

```
void mypractice_sort(int*a,int length){
    int k=length-1;
    while(true){
        int imax=0;
        int sig=1;
        for(int i=0;i<=k;i++){
            if(a[imax]<a[i]){
                imax=i;
                sig++;
            }
```

```
        }
        swap(a[imax],a[k]);k--;
        /*if() break;
        这个地方值得一提，大循环也有办法提前跳出去:
        1.每次归位最大元素，那么循环length-1次，一定可以全部归位
        2.提前跳出，某次循环，sig达到该次的数组长度k+1，说明已经有序
        (因为sig是从1计数的，所以实际上sig==k+2)
        在这种策略之下，实际上也可以判断是否逆序，当sig++一次都没执行过，就说明逆序了
        */
        if(sig==k+2 || k==0) break;
    }
}
```

应该说，所有的迭代，换一个角度来看也可以变成是递归，使用递归的视角来看在每次归位最大元素的方法，那么就是：

1.divide:把数组分作是末尾元素和末尾之前的小数组

2.conquer:递归基是只有两个元素，然后把大的元素放右边，小的元素放左边

3.combine:每次找到合适的位置，然后把末尾元素插入进去

```
void mypractice_sort(int* a,int first,int last){
    if(last==first+1){
        if(a[first]>a[last]) swap(a[first],a[last]);
    }else{
        mypractice_sort(a,first,last-1);
        insert_into(a,first,last);
    }
}
void insert_into(int* a,int first,int last){
    int key=a[last];
    for(int i=last-1;i>=first;i--){
        if(a[i]>a[i+1]){
           swap(a[i],a[i+1]);
        } else break;
    }
}
/*
其实这种方法并没有起到分而治之的作用，递归次数太多，每次都仅仅把问题规模减小了一个元素，所以只是一种递归写法
*/
```

在这种方法之下，其实很容易发现，一种新的思路，把数组看做是已经排好序的左边小数组，以及最右边的元素，然后找到合适的位置把最右边的元素放进去，然后再把右边的元素依次后移（想可以这么想，但是这样去写代码就错了）。只不过上面是采用递归的视角去考虑问题，当从迭代的角度来看的时候，其实就是有序数组不断增长的过程，于是就引入了第二种经典排序，insertion_sort。

2.insertion_sort

- 排序思路:从只有两个元素开始，把数组看做是{已经排序的小数组}+{中间元素}+{右边暂未涉及的数组}

于是，在每一次执行之中，把中间元素在左边数组之中找到一个合适的位置，此位置之后直到中间元素之前的位置的所有元素依次后移，把中间元素放到这个让出来的空位之中。在此次之行之中，可见已经排序的小数组扩大规模了，未排序部分规模减小，知道未排序部分称为空集，排序就完成了。

显而易见，每一次执行扩大一个有序规模，于是算法的正确性是被保证的。

代码：

```
void insertion_sort(int* a,int length){
    _insertion_sort(a,0,length-1);
}
void _insertion_sort(int* a,int first,int last){
    for(int i=first+1;i<=last;i++){
        int temp=a[i];//这个地方必须暂存a[i]
        int p=i-1;
       for(;p>=first;p--){
           if(a[p]>temp){//不建议把这个条件写到循环体，因为可能在其他情形之中会越界，不是好习惯
               a[p+1]=a[p];
           }else break;
       }
        a[p+1]=temp;
        /*这是一种写法，也能够把元素插到正确的位置上去
        长得很像bubble_sort，其实说是bubble_sort也差不多了
        for(int p=i-1;p>=first;p--){
            if(a[p]>a[p+1]){
                swap(a[p],a[p+1]);
            }else break;
        }
        那么这种写法比上面坏在哪里呢，主要是因为swap内部有三个操作，开销更大

        */
        /*
        还有别的写法，先找到第一次比temp小的位置，然后把这个位置右边的元素依次右移，
        然后再把元素插入该位置
        int p=i-1;
        for(;p>=first;p--){
            if(a[p]<=temp) break;
        }
        for(int k=i-1;k>=p+1;k--){
            a[k+1]=a[k];
        }
        a[p+1]=temp;
        显而易见的是，这种写法，与最后的写法意义是一模一样的，然后再看两个for循环，
        可以发现循环体内部的条件是一模一样的，也就是说，最上面的写法可以看做是合并
        同类项。容易证明，在代码之中，这种合并同类项最后的代码也是正确的。
        */
    }
}
```

可以发现，insertion_sort和bubble_sort有所不同，bubble_sort其实瞄准的是最大或者最小元素，然后把它优先归位，但是insertion_sort是每次把有序部分最右边的元素归位扩大有序部分，是一种生长的算法。但是bubble_sort也是可以用insertion的视角进行改造的，因为最大元素也是普通元素的一种特例。上个部分已经写过了。

3.mergesort

- 排序思路：分而治之，把元素等分为两部分，然后假设这两部分已经有序，接下来要做的就是将两个有序数组组合在一起成为大数组。递归基是单个元素，单个元素必然有序，满足有序条件；组合办法是从数组头开始依次比较两个小数组，把较小的元素放到大数组的正确位置，每次比较必定可以放回一个元素，最终不超过n次比较，就可以全部归位。

  算法正确性收到递归基和组合方法的保证，递归程度最多到logn，而每一次组合都能够在n步操作内完成，那么可以保证，nlogn内程序一定是可以完成的。递归基可以保证n=1时算法是正确的，然后组合方法可以保证在假设n=k时算法正确的情况下，n=k+1一定正确，就是数学归纳法。

代码：

```
void mergesort(int* a,int length){
    _mergesort(a,0,length-1);
}
/*
mergesort的语句很容易在语义上理解：
1.如果只有一个元素，立即返回
2.选取中点，然后解决左边部分，再解决右边部分（这个时候算法都没写完，但是就已经假定算法必定可以正确返回结果
3.组合返回结果

完全可以看做是：当n=1时成立，假设n=k时成立，然后保证下面n=k+1时成立，和数学归纳法的思维很相似
*/
void _mergesort(int* a,int first,int last){
    if(first>=last) return;//先写递归基是一个好习惯
    int p=(first+last)/2;
    _mergesort(a,first,p);
    _mergesort(a,p+1,last);
    merge(a,first,p,last);
}
void merge(int* a,int first,int mid,int last){
    //要注意原数组是从first到last，而a1 a2都是从0开始，然后到n1-1,n2-1结束，不要搞混
    //完全可以开1个数组，但是对思维的消耗更大，可能更容易出bug
    int n1=mid-first+1;
    int n2=last-mid;
    int* a1=new int[n1];
    int* a2=new int[n2];
    //为了避免数组[]内部的表达式，当然可以用指针进行定位，但是没有必要，还更容易出bug
    for(int i=0;i<n1;i++){
        a1[i]=a[i+first];
    }
    for(int i=0;i<n2;i++){
        a2[i]=a[i+mid+1];
    }
    for(int k=first,i=0,j=0;k<=last;k++){//k指示大数组 i指示a1 j指示a2
        if(i>=n1) a[k]=a2[j++];
        else if(j>=n2) a[k]=a1[i++];
        else if(a1[i]<a2[j]) a[k]=a1[i++];
        else if(a2[j]<=a1[i]) a[k]=a2[j++];
    }
    delete[] a1;delete[] a2;//不删除有风险
    /*组合a1与a2的方法还有一种在清华书上出现
    先理清楚逻辑：
    循环条件——i或者j指针没有全部走完 (i<n1 || j<n2)
    放置a1的条件——a1比较小或者j指针走完 (j>=n2 || (a1[i]<=a2[j] && i<n2))
```

```
    放置a2的条件--a2比较小或者i指针走完 (i>=n1 || (a2[j]<a1[i] && j<n1))

    放置a1:语义上的意思是(j指针走完 或者 i指针还没有越界的情况下，左边小于右边)
    一个括号要包含两类条件：指针情况与数组大小比较，数组大小比较是互斥的，但是指针
    情况的搭配总共有3种:
    1.i j不越界       这种不用考虑，天然的没问题
    2.i越界 j不越界
    3.i不越界 j越界
    对于(j>=n2 || (a1[i]<=a2[j] && i<n2)) 左边的含义是:j越界 i不越界 而右边是j不越界 i越界        的情况
    想清楚了不难，想不清楚bug理都理不过来
    int k=first;
    int i=0;
    int j=0;
    while(i<n1 || j<n2){
        if (j >= n2 || (a1[i] <= a2[j] && i<n1)) {
            a[k]=a1[i];k++;i++;
        }
        else if (i >= n1 || (a2[j] < a1[i] && j<n2)) {
            a[k]=a2[j];k++;j++;
        }
    }

    第一种写法，把所有的情况拆开来写，所有更加的清晰且不容易出bug


    要小心一点，想一行代码解决过多的逻辑问题往往是会出问题的
    如果把上面的代码拆分掉，第一部分执行比较功能，第二部分把剩下的数组填充
    那么就有:
    int k=first;
    int i=0;
    int j=0;
    int sig=0;
    while(true){
        if(a1[i]<=a2[j]){
            a[k]=a1[i];k++;i++;
            if(i>=n1){
                sig=1;
                break;
            }
        }
        else if(a2[j]<a1[i]){
            a[k]=a2[j];k++;j++;
            if(j>=n2){
                 sig=2;
                 break;
            }
        }
    }
    if(sig==1){
        while(j<n2) a[k++]=a2[j++];
    }else if(sig==2){
        while(i<n1) a[k++]=a1[i++];
    }
    这种写法相比前一种，逻辑更加清晰，但是代码行数也显著增加，逻辑直接，但是书写繁琐。
    相较而言，目前第一种写法是逻辑清晰，兼有书写也少的代码，更可取
    然而我看网络的代码，同样的思路，写法是:
    int k=first,i=0,j=0;
```

```
    while(i<n1 && j<n2){
        if(a1[i]<=a2[j]) a[k++]=a1[i++];
        else a[k++]=a2[j++];
    }
    while(i<n1) a[k++]=a1[i++];
    while(j<n2) a[k++]=a2[j++];
    这种写法相当优美，逻辑清晰，书写也轻松，可以和第一种方法媲美
    */

}

```

稍作归纳，这是一个典型的divide-conquer-combine的过程:

1.divide:把大数组分作是两个小数组，每个数组继续往下分直到递归基一个元素

2.conquer:递归基是自然有序的

3.combine:就是merge，通过依次比较把元素放到合适的位置上

4.combine_sort

- 排序思路:insertion_sort在数据规模小的时候，速度比mergesort还要快，所以一个思路就是mergesort往下分的时候并不分到尽头，而是当数组规模很小的时候直接调用insertion_sort，而不是继续划分一直分到单个元素才返回。
- 划分到k=10个元素的时候就调用insertion_sort效果接近最好，测试过程在另一篇笔记之中。
- 最后会给出几种代码的测试比较。

```
void combine_sort(int* a,int length,int k){
    _combine_sort(a,0,length-1,k);
}
void _combine_sort(int* a,int first,int last,int k){
    if(last-first+1<=k){//先写递归基是一种好习惯
        _insertion_sort(a,first,last);
        return;
    }
    else{
        int p=(first+last)/2;
        _combine_sort(a,first,p,k);
        _combine_sort(a,p+1,last,k);
        merge(a,first,p,last);
    }
}
```

5.quick_sort

- 排序思路:这又是一种对有序的新视角，有序就是任意一个元素，它左边的元素全部小于它，右边的元素全部大于它。所以，如果每次能够选定一个元素，然后把它排到最终位置上去，保证{左边部分}<{此元素}<{右边元素}，然后下一次再继续分别处理左边和右边，以此类推，就可以保证每一次都有一个元素归位，那么不超过n次递归，就可以使得整个数组有序。

代码:

```
void quick_sort(int*a,int length){
    _quick_sort(a,0,length-1);
}

```

```
void _quick_sort(int*a,int first,int last){
    if(first>=last) return;//先写递归基
    int p=partition(a,first,last);//partition默认将数组末尾元素归位
    _quick_sort(a,first,p-1);
    _quick_sort(a,p+1,last);
}
int partition(int*a,int first,int last){
    int temp=a[last];
    int i=first,j=last-1;//i指示小于区，j指示大于等于区
    while(true){
        while(a[i]<temp){//每一次停在大于等于temp的地方
            i++;
            if(i==last) break;
        }
        while(a[j]>=temp){//每一次停在小于temp的地方
            j--;
            if(j==first) break;
        }
        if(i>=j) break;//先判断，以免错误交换
        swap(a[i],a[j]);
    }
    /*
    最终的i必定停在第一个大于等于temp的位置，j停留在第一个小于temp的位置，
    所以交换的是a[i]与a[last]，若是交换了a[i-1]与a[last],会把一个小于
    temp的元素交换到最右边去
    */
    swap(a[i],a[last]);
    return i;//返回最终a[last]值最终应该在的位置
}
/*
partition有别的写法，上面是双指针同时往中间移动，最后交汇，其实双指针可以
同向移动，还有单指针的写法
*/
```

- 小结:quick_sort是一个不太典型的分而治之算法，没有combine的过程，因为每次都归位一个元素直接到达最终位置，所以不用combine

5.1 random_quicksort

- 排序思路：上面的quick_sort有一个风险，那就是如果原数组已经有序了，那么每一次取末尾元素，然后因为他已经归位了，造成的格局是{左边元素}<{末尾元素}，没有右边的元素，每一次只能把数据规模减小一个元素，那么递归深度将到达n，时间复杂度成为n^2。于是，不是每次默认取得末尾元素，而是随机抽取一个元素与末尾交换，避免这种情况
- 扩展来想，光是为了避免有序的情况，也完全可以在代码之中插入一段检测有序的代码，一旦局部数组有序，直接返回也不排序了

代码1：

```
void random_quicksort(int* a,int length){
    _random_quicksort(a,0,length-1);
}
void _random_quicksort(int* a,int first,int last){
    if(first>=last) return;
    else{
        int r=rand()%(last-first+1)+first;
        swap(a[r],a[last]);
        int p=partition(a,first,last);
```

```
10
11          _random_quicksort(a,first,p-1);
12          _random_quicksort(a,p+1,last);
13      }
14  }
```

代码2：

```
1   void check_quicksort(int*a,int length){
2       _check_quicksort(a,0,length-1);
3   }
4   void _check_quicksort(int* a,int first,int last){
5       if(check(a,first,last)) return;
6       else{
7           int p=partition(a,first,last);
8           _check_quicksort(a,first,p-1);
9           _check_quicksort(a,p+1,last);
10      }
11  }
12  bool check(int*a,int first,int last){
13      bool p=true;
14      if(first>=last) p=true;
15      else{
16          for(int i=first;i<=last-1;i++){
17              if(a[i]>a[i+1]) p=false;
18          }
19      }
20      return p;
21  }
```

其实通过代码也能够看出来了，这个方法并不是一个好方法，因为：

1.check每一次都执行，对于有序数组部分可能确实能够节省一点时间，但是对于乱序部分，开销太大，不值得这样做

2.例如：1 2 3 5 4 7 8 9这样大部分已经有序，只有个别无序，而且最右端任然是最大值的数组，check的辅助作用极其有限

综上，这种策略不如random，不是一个好策略

5.2 determinstic_quicksort

- 排序思路：quick_sort递归调用的过程之中，往往分离出来的左边数组和右边数组是数量不平衡的，在这种情况下，会导致复杂度O(nlogn)的常数项比较大。而均匀的分割可以减小常数项，那么是否可以通过选取数组的中位数为主元的方法来减小常数项获得优化呢？答案不一定，因为选取中位数也是有开销的，结果要试过才知道

代码：

```
1   void determinstic_quicksort(int* a,int length){
2       _determinstic_quicksort(a,0,length-1);
3   }
4   void _determinstic_quicksort(int* a,int first,int last){
5       if(last-first+1<=10) _insertion_sort(a,first,last);//递归基，小于10个
    元素直接插入排序
6       else{
7           select_pivot(a,first,last);//完成找到中位数并且交换到末尾的功能
```

```
        int p=partition(a,first,last);
        _determinstic_quicksort(a,first,p-1);
        _determinstic_quicksort(a,p+1,last);
    }
}
/*
select_pivot必须在nlogn甚至更小的时间内找到中位数否则代价就是不可承受的
*/
void select_pivot(int *a,int first,int last){
    int num=ceil((last-first+1)/5.0);//5个元素一组
    int* b=new int[num];//存储每个组的中位数
    int k=0;
    /*
    这个for循环在3.2n的时间完成
    */
    for(int i=first;i<=last;i+=5){
        int j=i+4;
        if(j<=last){
            _insertion_sort(a,i,j);
            b[k]=a[i+2];k++;
        }else{
            _insertion_sort(a,i,last);
            b[k]=a[(i+last)/2];k++;
        }
    }
    int* c=new int[num];
    for(int i=0;i<num;i++) c[i]=b[i];
    int p=0;
    //如果相信determinstic真的更快，这个地方就要使用它
    _determinstic_quicksort(b,0,num-1);
    for(int i=0;i<num;i++){
        if(c[i]==b[(num-1)/2]){
            p=i;
            break;
        }
    }
    p=first+p*5;//定位到这一组在原数组之中的头部
    if(p+4<=last) swap(a[p+2],a[last]);
    else swap(a[(p+last)/2],a[last]);
}
```

对于它的时间，我也很好奇。

6.heap_sort

- 排序思路：借用最大堆的概念，整颗树，任何一个位置的根节点一定大于子节点，这是一个偏序关系。于是，排序的过程就可以总结为，拿走根节点得到最大值（实际做的其实是，根节点与尾节点交换，然后取出最大值，再调用建堆函数，通过下滤把树恢复为最大堆），接着运行。
- 算法正确性得益于每一次能够取出一个最大值，n步操作就可以从大到小取出所有的元素。

代码：

```
#define lc(x) 2*x
#define rc(x) lc(x)+1
#define parent(x) x/2 //x除以2一定能够得到父节点
#define maxn 100000
void heapify(int *a,int root,int last){
```

```
    for(int i=lc(root);i<=last;i=lc(i)){
        if(a[i+1]>a[i] &&  i<last) i+=1;
        if(a[i]>a[parent(i)]) swap(a[i],a[parent(i)]);
        else break;
    }
}
/*
试一下下滤的递归写法:
//1.寻找子节点中的最大值,可能只有一个子节点
//2.与父节点比较
//3.如果大就互换继续往下调用，小就返回
//合并同类项:分作直接返回和继续调用两种可能
*/
void heapify(int *a,int root,int last){
    if(lc(root)>last) return;
    else{
        int p=lc(root);//p节点指示下一次递归的位置
        //这个地方就应该先假设一个最大值，以减小比较次数，精简代码
        if(a[p+1]>a[p] && p+1<=last) p=p+1;
        if(a[p]>a[root]) swap(a[p],a[root]);
        else return;
        heapify(a,p,last);
    }
}
void heap_sort(int *a,int n){
    int* b=new int[n+1];b[0]=maxn;//哨兵节点
    for(int i=0;i<n;i++){
        b[i+1]=a[i];//因为a对齐的是b从1到n
    }
    for(int i=parent(n);i>=1;i--){
        heapify(b,i,n);//自下而上的下滤，是floyd算法 nlogn时间内建堆
    }
    for(int i=n;i>=1;i--){//从n到1比起从1到n要好理解一点
        swap(b[1],b[i]);//先把最大值交换到末尾
        a[i-1]=b[i];//然后把末尾的值赋给a的末尾 a与b对应的下标永远少1
        heapify(b,1,i-1);//每次拿掉一个元素都会减小一个规模
    }
    delete[] b;
}
```

debug记录

> 尽管算法我写了很多很多次，但是每次写都要出Bug，太惨了，实在太惨了
>
> 虽然不是什么很严重的Bug，但是希望借此留下一个印象，不要再犯这种低级错误

1.bubble_sort

```
while (true) {
    int sig = 1;//这是有序标志
    for (int i = 0; i < length; i++) {
        if (a[i] > a[i + 1]) {
            swap(a[i], a[i + 1]);
            sig = 0;
        }
    }
```

- 由于swap交换的是a[i]与a[i+1]，所以i的范围不能到length-1，否则i+1会越界

```
void mypractice_sort(int* a, int length) {
    int k = length - 1;
    while (true) {
        int imax = 0;
        int sig = 1;
        for (int i = 0; i <= k; i++) {
            if (a[imax] < a[i]) {
                imax = i;
                sig++;
            }
        }
        swap(a[imax], a[k]); k--;
        /*if() break;
        这个地方值得一提，大循环也有办法提前跳出去：
        1.每次归位最大元素，那么循环length-1次，一定可以全部归位
        2.提前跳出，某次循环，sig达到该次的数组长度k，说明已经有序
        在这种策略之下，实际上也可以判断是否逆序，当sig++一次都没执行过，就说明逆序了
        */
        if (sig == k || k == 0) break;
    }
}
```

- 该次数组的长度是k+1，其次由于sig初值为1，所以提前跳出的条件是sig==k+2

```
void mypractice_sort2(int* a, int first, int last) {
    if (last == first + 1) {
        if (a[first] > a[last]) swap(a[first], a[last]);
    }
    mypractice_sort2(a, first, last - 1);
    insert_into(a, first, last);
}
```

- 递归基执行完之后就应该返回了，要么加一个return，要么下面的内容在else之中执行

2.insertion_sort

```
void _insertion_sort(int* a, int first, int last) {
    for (int i = first + 1; i <= last; i++) {
        int temp = a[i];
        int p = i - 1;
            for (; p >= first; p--) {
                int temp = a[i];//这个地方必须暂存a[i]
                if (a[p] > temp) {//不建议把这个条件写到循环
                    a[p + 1] = a[p];
                }
                else break;
            }
        a[p + 1] = temp;
```

- 不知道是不是精神不好，但是为什么int temp=a[i]写了两次?

3.mergesort

```
    int k=first;
    int i=0;
    int j=0;
    while(i<n1 || j<n2){
        if (j >= n2 || (a1[i] <= a2[j] && j < n2)) {
            a[k]=a1[i];k++;i++;
        }
        else if (i >= n1 || (a2[j] < a1[i] && i<n1)) {
            a[k]=a2[j];k++;j++;
        }
    }
```

- 第一个地方是:两个if里面 或 右边的条件的j<n2和 i<n1都是错的，搞错了逻辑关系
- 第二个地方是：第二个if必须是else if，没有else，最后一次执行的时候,j和i越界，但是第二个if进去了

  从另一个思路来想，每一次执行都必须有一个元素放进去，要么是a1的，要么是a2的，所以必须是else关系

4.heap_sort

主要就是下滤的过程写的太复杂

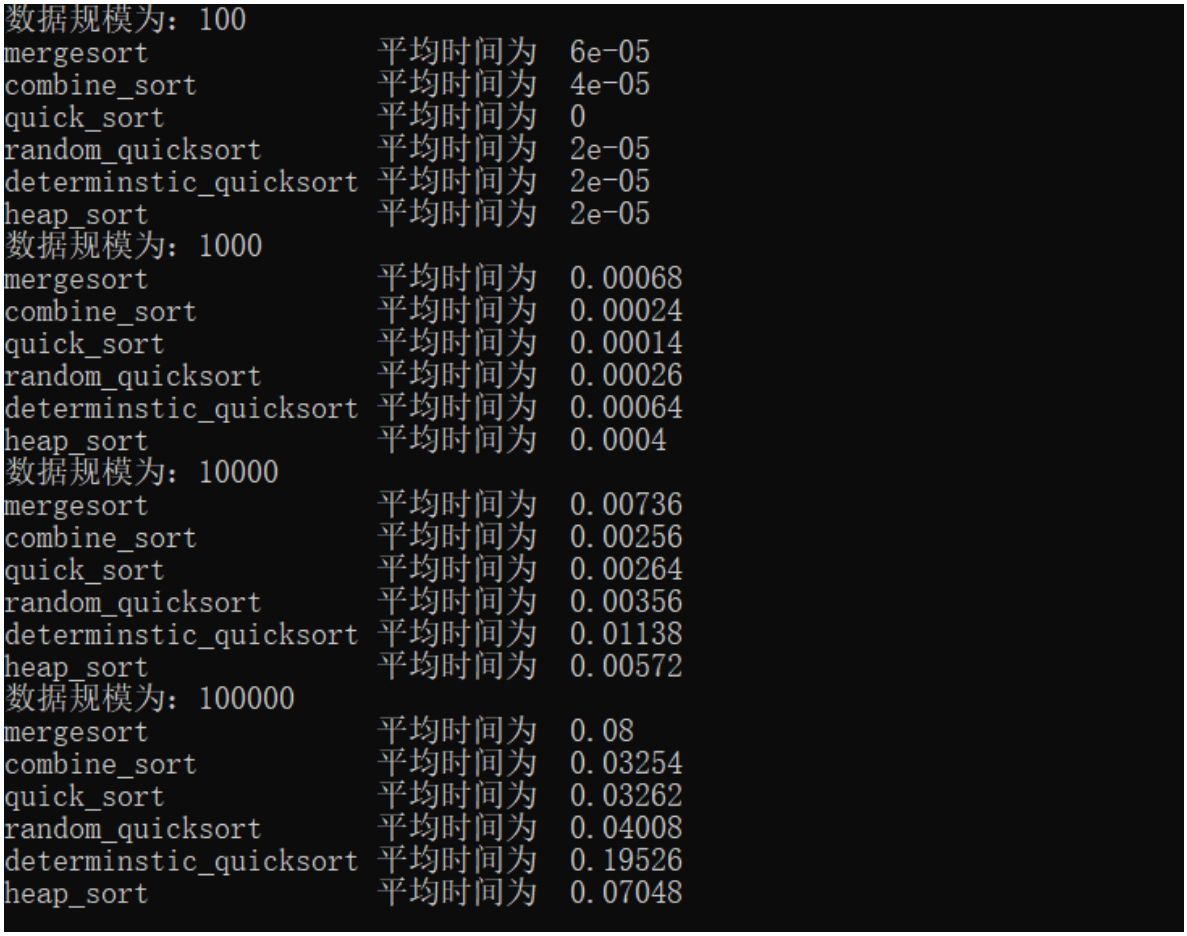
```
//1.寻找子节点中的最大值,可能只有一个子节点
//2.与父节点比较
//3.如果大就互换继续往下调用，小就返回
if (rc(root) > last) {
    if (a[lc(root)] > a[root]) swap(a[root], a[lc(root)]);
    else return;
}
else {
    if (a[lc(root)] > a[rc(root)]) {
        if (a[lc(root)] > a[root]) {
            swap(a[root], a[lc(root)]);
            HeapAdjust(a, lc(root), last);

        }
        else return;
    }
    else {
        if (a[rc(root) > a[root]]) {

            swap(a[root], a[rc(root)]);
            HeapAdjust(a, rc(root), last);
        }
        else return;
    }
}
```

- 没有事先假设的意识，以至于4行代码能够解决的问题写那么长

模拟测试

仅仅测试nlogn的速度比较，也就是：

> 1.mergesort
>
> 2.combine_sort
>
> 3.quick_sort
>
> 4.random_quicksort
>
> 5.determinstic_quicksort
>
> 6.heap_sort



结论：

- 整体而言:d_quicksort>mergesort>heap_sort>r_quicksort≈quick_sort≈combine_sort
- quick_sort和简单的random_quicksort以及combine_sort表现都不错，只不过determinstic quicksort比mergesort慢稍微有点出乎意料，也就是说,为了选中位数的花销，比采取这种手段的好处要更大，导致速度减慢很多