根据排序的原则,内排序可以分为:

- 插入排序
- 交换排序
- 选择排序
- 归并排序

预备知识:

1.等差数列之和:S=n*(a1+an)/2

等比数列之和:S=a1(1-q^n)/(1-q)

2.使用哨兵提高效率

比如基本的顺序查找我们可以这样做:

```
int search(int a[],int n,int key){
    for(int i=0;i<n;i++)
        if(a[i]==key)
            return n+1;          //返回第几个,而不是返回下标
    return 0;                    //返回 0 说明没有找到
}
```

注意到每次 for 循环都对边界进行检查(i<n),使用哨兵就不需要进行边界检查.

```
int search(int a[],int n,int key){
    a[0]=key;
    for(int i=n;a[i]!=key;i--);
    return i;
}
```

但是使用哨兵的前提是在数组中 a[1]--a[n]存储的是实际的元素,a[0]是拿来做哨兵的,即 a 的长度是 n+1.

3.time()返回从 1970 年 1 月 1 日到现在的秒数,是实际时间。

clock 返回开启进程和调用 clock()之间的的 CPU 时钟计时单元(clock tick)数，不包括显式调用 sleep()的时间,常用来测试任务执行的速度。

插入排序

简单插入排序

非常的简单，想想你玩牌的时候一边起牌,一边就把牌排好序了,那就是插入排序.

时间复杂度：O(N^2)，1+2+...+(N-1)=N^2/2。这是最坏的情况，其实大致上说插入排序的平均情况和最坏情况一样差。

空间上来讲，任一时刻最多只有一个数字在存储数组之外，属于原地排序,O(1)。

稳定的排序.

```
template <typename Comparable>
void InsertSort(vector<Comparable> &vec,int begin,int end){
    for(int i=begin+1;i<=end;i++){
        if(vec[i]<vec[i-1]){
            vec[0]=vec[i];              //把 vec[i]放入哨兵
            int k;
            for(k=i-1;vec[0]<vec[k];k--)        //从 vec[0]<vec[k]看出是稳定排序
                vec[k+1]=vec[k];
            vec[k+1]=vec[0];
        }
    }
}
template <typename Comparable>
void InsertSort(vector<Comparable> &vec){
    InsertSort(vec,1,vec.size()-1);
}
```

希尔排序

希尔排序利用利用了插入排序的两个特点:

- 基本有序时直接插入排序最快
- 对于数据很少的无序数列,直接插入也很快

谢尔排序的时间复杂度在 O(nlogn)和 O(n^2)之间,空间复杂度为 O(1).

为了使集合基本有序,而不是局部有序,不能简单地逐段分割,而应将相距为某个”增量”的元素组成一个子序列.通常取增量为 d1=n/2,di+1=di/2.

```
template <typename Comparable>
void ShellSort(vector<Comparable> &vec){
    int n=vec.size()-1;
    for(int inc=n/2;inc>=1;inc/=2){          //以 inc 为增量,vec.size()-1 才是 vec 里面存储的有效元素的个数
        for(int i=inc+1;i<=n;i++){              //一趟希尔排序
            if(vec[i]<vec[i-inc]){
                vec[0]=vec[i];
                int k;
                for(k=i-inc;k>0&&vec[0]<vec[k];k-=inc)
                    vec[k+inc]=vec[k];
```

```
                vec[k+inc]=vec[0];
            }
        }
    }
}
```

冒泡排序

把待排序的序列分为有序区和无序区,每次把无序区最大的数放到无序区的最后面,这样无序区的末元素成为有序区的首元素.

时间复杂度为 O(n^2),空间复杂度 O(1).

```
template <typename Comparable>
void BubbleSort(vector<Comparable> &vec){
    int pos=vec.size()-1;          //用 pos 标记无序区域的最后一个元素.大数往后排,所以无序区域在前,有序区域在后
    while(pos!=0){                 //当 pos=0 时说明 1--n 都已经有序了
        int bound=pos;             //本次操作前无序区域的边界
        pos=0;
        for(int i=1;i<bound;i++){
            if(vec[i]>vec[i+1]){
                vec[0]=vec[i];
                vec[i]=vec[i+1];
                vec[i+1]=vec[0];
                pos=i;             //只要发生交换就更改 pos
            }
        }
    }
}
```

快速排序

快速排序是对冒泡排序的改进,由于冒泡排序是不断比较相邻元素然后进行交换,需要比较移动多次才能到达最终位置.而快速排序的比较和移动是从两端向中间进行,因而元素移动的距离较远.

初始主轴的选取采用三元取中法.经过 N 趟排序,当元素已基本有序后采用直接插入排序法.

```
template <typename Comparable>
void median3(vector<Comparable> & a,int left,int right){
        int center=(left+right)/2;
        if(a[center]<a[left])
            swap(a[center],a[left]);
        if(a[right]<a[left])
            swap(a[left],a[right]);
        if(a[center]>a[right])
            swap(a[center],a[right]);

        //place pivot at position left
        swap(a[center],a[left]);
}
template <typename Comparable>
const int Partion(vector<Comparable> & a,int left,int right){
       a[0]=a[left];                    //选取基准存放在 a[0]中
       while(left<right){
            while((left<right)&&(a[right]>=a[0]))          //右侧扫描
                right--;
            a[left]=a[right];
            while((left<right)&&(a[left]<=a[0]))           //左侧扫描
                left++;
            a[right]=a[left];
       }
       a[left]=a[0];
       return left;
}
template <typename Comparable>
void QuickSort(vector<Comparable> &vec,int begin,int end){
       median3(vec,begin,end);
```

```
    if(begin+10<end){
        int pivotloc=Partion(vec,begin,end);
        QuickSort(vec,begin,pivotloc-1);
        QuickSort(vec,pivotloc+1,end);
    }
    else{
        InsertSort(vec,begin,end);
    }
}
```

理想情况下,每次划分左右两侧的序列长度是相同的,长度为 n 的序列可划分为 logn 层.定位一个元素要对整个序列扫描一遍,所需时间为 O(n),总的时间复杂度为 O(nlogn).

最坏情况下待排序列完全有序(正序或逆序),每次划分只得到比上一次少一个的子序列,总的比较次数为 1+2+3+...+(n-1)=O(n^2).

平均来说快速排序的时间复杂度为 O(nlogn),栈的深度为 O(logn).

快速排序是一种不稳定的排序算法.

选择排序

简单选择排序

简单选择排序和冒泡排序很像,每趟排序把把无序序列中最小的元素放到有序列的最后面,有序序列在前,无序序列在后.但有一个重要的区别:冒泡排序在一趟排序中边比较,边交换；而简单选择排序在一趟排序中只作一次交换.

简单选择排序是不稳定的,时间复杂度为 O(n^2),空间复杂度为 O(1).

```
template <typename Comparable>
void SelectSort(vector<Comparable> &vec){
    int n=vec.size()-1;
    for(int begin=1;begin<n;begin++){
        int tmp=begin;              //记录本趟排序的起点
        for(int i=tmp+1;i<=n;i++)
            if(vec[i]<vec[tmp])
                tmp=i;              //tmp 跟踪最小的元素
        if(tmp!=begin){
            vec[0]=vec[tmp];
```

```
            vec[tmp]=vec[begin];
            vec[begin]=vec[0];
        }
    }
}
```

堆排序

堆排序是对简单选择排序的改进,在简单选择排序中前一次的比较结果没有保存下来,后一趟排序中反复对以前已经做过的比较重新做了一遍.

时间复杂度为 O(nlogn),不稳定的排序.

```
template <typename Comparable>
void percolate(vector<Comparable> & a,int k,int n){      //"渗透",k 是我们当前关注的节点,我们要保证它比其左右孩子都要小
    int i=k;            //i 是要筛选的节点
    int j=2*i;              //j 是 i 的左孩子
    while(j<=n){
        if(j<n && a[j]>a[j+1])
            j++;            //确保 a[j]是左右孩子中的较小者
        if(a[i]<a[j])           //满足小根堆的条件,结束
            break;
        else{
            swap(a[i],a[j]);
            i=j;
            j=2*i;
        }
    }
}
template <typename Comparable>
void HeapSort(vector<Comparable> & a){
    int n=a.size()-1;
    for(int i=n/2;i>=1;i--)             //n/2 是第 1 个有孩子的节点.从下往上渗透,建堆
```

```
        percolate(a,i,n);
    for(int j=1;j<n;j++){
        cout<<a[1]<<"    ";              //输出堆顶元素
        swap(a[1],a[n-j+1]);       //C++内置 swap()函数
        percolate(a,1,n-j);              //重新建堆
    }
}
```

归并排序

二路归并排序

```
template<typename Comparable>
void Merge(vector<Comparable> & a,vector<Comparable> & tmpArray,int leftPos,int rightPos,int rightEnd){
    int leftEnd=rightPos-1;
    int tmpPos=leftPos;
    int numElements=rightEnd-leftPos+1;

    while(leftPos<=leftEnd && rightPos<=rightEnd)
        if(a[leftPos]<=a[rightPos])
            tmpArray[tmpPos++]=a[leftPos++];
        else
            tmpArray[tmpPos++]=a[rightPos++];
    while(leftPos<=leftEnd)
        tmpArray[tmpPos++]=a[leftPos++];
    while(rightPos<=rightEnd)
        tmpArray[tmpPos++]=a[rightPos++];
    for(int i=0;i<numElements;i++,rightEnd--)
        a[rightEnd]=tmpArray[rightEnd];
}
template<typename Comparable>
void MergeSort(vector<Comparable> & a,vector<Comparable> & tmpArray,int left,int right){
```

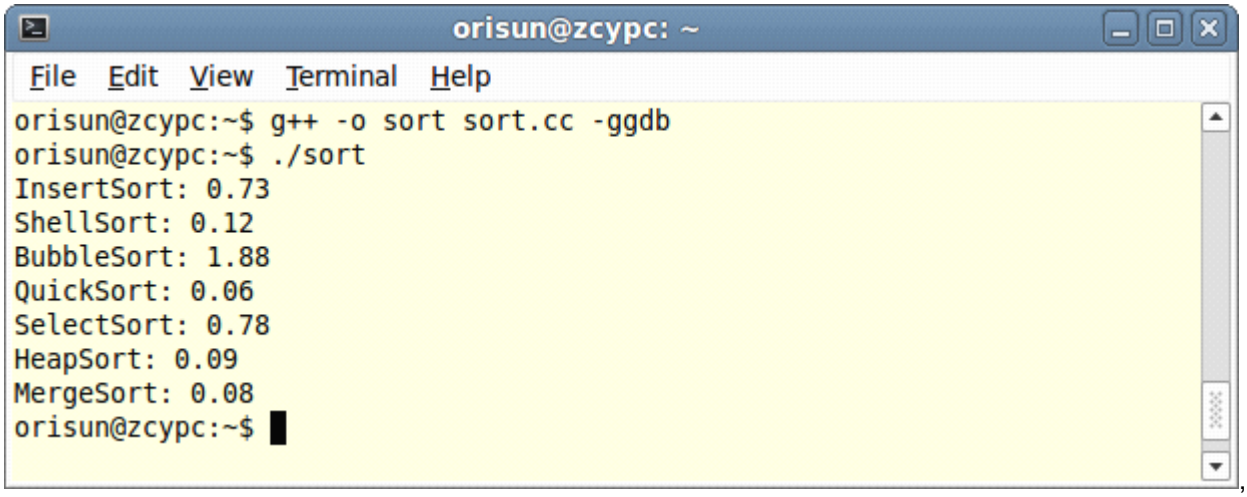
```
        if(left<right){
            int center=(left+right)/2;
            MergeSort(a,tmpArray,left,center);
            MergeSort(a,tmpArray,center+1,right);
            Merge(a,tmpArray,left,center+1,right);
        }
}
```

总结

对以上种算法进行一次测试,比一比哪个快.

测试的完整代码见附录.

我们来排个序



,直接插入,冒泡和简单选择排序都是对 1 万条数据排序

直接插入:0.73 秒

简单选择:0.78 秒

冒泡:1.88 秒

以下算法是对 10 万条数据进行排序

快速排序:0.06 秒

归并排序:0.08 秒

堆排序:0.09 秒

希尔:0.12 秒

排序方法	平均情况	最好情况	最坏情况	辅助空间	稳定性
直接插入	O(n^2)	O(n)	O(n^2)	O(1)	是

希尔	O(nlogn)~O(n^2)	O(n^1.3)	O(n^2)	O(1)	否
冒泡	O(n^2)	O(n)	O(n^2)	O(1)	是
快速	O(nlogn)	O(nlogn)	O(n^2)	O(nlogn)~O(n)	否
简单选择	O(n^2)	O(n^2)	O(n^2)	O(1)	是
堆排序	O(nlogn)	O(nlogn)	O(nlogn)	O(1)	否
归并	O(nlogn)	O(nlogn)	O(nlogn)	O(n)	是

注:冒泡排序采用 pos 来标记已有序的序列位置后,最好情况才是 O(n),如果没有采用此改进算法,最好情况也是 O(n^2).我们的快速排序每次都把主轴放在 vec[0]中,没用另外使用单独的变量,所以辅助空间为 O(1),否则就是 O(nlogn)~O(n).

STL 排序

STL 中的所有排序算法都需要输入一个范围,[begin,end).如果要自己定义比较函数,可以把定义好的仿函数(functor)作为参数传入.

函数名	功能描述
sort	指定区间排序
stable_sort	指定区间稳定排序
partial_sort	对指定区间所有元素部分排序
partial_sort_copy	对指定区间复制并排序
nth_element	找到指给定区间某个位置上对应的元素
is_sorted	判断给定区间是否已排序好
partition	使得符合条件的元素放在前面
stable_partion	相对稳定地使得符合条件的元素放在前面

```
#include<iostream>
#include<algorithm>
#include<functional>
#include<vector>
using namespace std;
class myclass{
public:
    myclass(int a,int b):first(a),second(b){}
    int first;
```

```
    int second;
    bool operator < (const myclass &m)const{
        return first<m.first;
    }
};
//自定义仿函数
bool less_second(const myclass &m1,const myclass &m2){
    return m1.second<m2.second;
}
int main(){
    vector<myclass> vec;
    for(int i=0;i<10;i++){
        myclass my(10-i,i*3);
        vec.push_back(my);
    }
    //原始序列
    for(int i=0;i<10;i++)
        cout<<vec[i].first<<"    "<<vec[i].second<<endl;

    sort(vec.begin(),vec.end());
    cout<<"按照第 1 个数从小到大:"<<endl;
    for(int i=0;i<10;i++)
        cout<<vec[i].first<<"    "<<vec[i].second<<endl;

    sort(vec.begin(),vec.end(),less_second);
    cout<<"按照第 2 个数从小到大:"<<endl;
    for(int i=0;i<10;i++)
        cout<<vec[i].first<<"    "<<vec[i].second<<endl;

    return 0;
}
```

stable_sort 是稳定的排序,或许你会问既然相等又何必在乎先后顺序呢?这里的相等是指提供的比较函数认为两个元素相等,并不是指两个元素真的一模五一样.

sort 采用的是”成熟”的快速排序(结合了内插排序算法),保证很好的平均性能,时间复杂度为 O(nlogn).stable_sort 采用的是归并排序,分派内存足够时,其时间复杂度为 O(nlogn),否则为 O(nlognlogn).

附录

```
#include<iostream>
#include<ctime>                //time()
#include<vector>
#include<cstdlib>        //srand()和 rand()
using namespace std;
/*插入排序*/
template <typename Comparable>
void InsertSort(vector<Comparable> &vec,int begin,int end){
      for(int i=begin+1;i<=end;i++){
            if(vec[i]<vec[i-1]){
                  vec[0]=vec[i];              //把 vec[i]放入哨兵
                  int k;
                  for(k=i-1;vec[0]<vec[k];k--)        //从 vec[0]<vec[k]看出是稳定排序
                        vec[k+1]=vec[k];
                  vec[k+1]=vec[0];
            }
      }
}
template <typename Comparable>
void InsertSort(vector<Comparable> &vec){
      InsertSort(vec,1,vec.size()-1);
}
/*希尔排序*/
template <typename Comparable>
void ShellSort(vector<Comparable> &vec){
      int n=vec.size()-1;
      for(int inc=n/2;inc>=1;inc/=2){          //以 inc 为增量,vec.size()-1 才是 vec 里面存储的有效元素的个数
            for(int i=inc+1;i<=n;i++){                    //一趟希尔排序
                  if(vec[i]<vec[i-inc]){
                        vec[0]=vec[i];
```

```
                int k;
                for(k=i-inc;k>0&&vec[0]<vec[k];k-=inc)
                    vec[k+inc]=vec[k];
                vec[k+inc]=vec[0];
            }
        }
    }
}
/*冒泡排序*/
template <typename Comparable>
void BubbleSort(vector<Comparable> &vec){
    int pos=vec.size()-1;          //用 pos 标记无序区域的最后一个元素.大数往后排,所以无序区域在前,有序区域在后
    while(pos!=0){                 //当 pos=0 时说明 1--n 都已经有序了
        int bound=pos;             //本次操作前无序区域的边界
        pos=0;
        for(int i=1;i<bound;i++){
            if(vec[i]>vec[i+1]){
                vec[0]=vec[i];
                vec[i]=vec[i+1];
                vec[i+1]=vec[0];
                pos=i;             //只要发生交换就更改 pos
            }
        }
    }
}
/*快速排序*/
template <typename Comparable>
void median3(vector<Comparable> & a,int left,int right){
    int center=(left+right)/2;
    if(a[center]<a[left])
        swap(a[center],a[left]);
    if(a[right]<a[left])
```

```
        swap(a[left],a[right]);
    if(a[center]>a[right])
        swap(a[center],a[right]);


    //place pivot at position left
    swap(a[center],a[left]);
}
template <typename Comparable>
const int Partion(vector<Comparable> & a,int left,int right){
    a[0]=a[left];                   //选取基准存放在 a[0]中
    while(left<right){
        while((left<right)&&(a[right]>=a[0]))       //右侧扫描
            right--;
        a[left]=a[right];
        while((left<right)&&(a[left]<=a[0]))        //左侧扫描
            left++;
        a[right]=a[left];
    }
    a[left]=a[0];
    return left;
}
template <typename Comparable>
void QuickSort(vector<Comparable> &vec,int begin,int end){
    median3(vec,begin,end);
    if(begin+10<end){
        int pivotloc=Partion(vec,begin,end);
        QuickSort(vec,begin,pivotloc-1);
        QuickSort(vec,pivotloc+1,end);
    }
    else{
        InsertSort(vec,begin,end);
    }
}
```

```
/*简单选择排序*/
template <typename Comparable>
void SelectSort(vector<Comparable> &vec){
    int n=vec.size()-1;
    for(int begin=1;begin<n;begin++){
        int tmp=begin;              //记录本趟排序的起点
        for(int i=tmp+1;i<=n;i++)
            if(vec[i]<vec[tmp])
                tmp=i;              //tmp 跟踪最小的元素
        if(tmp!=begin){
            vec[0]=vec[tmp];
            vec[tmp]=vec[begin];
            vec[begin]=vec[0];
        }
    }
}
/*堆排序*/
template <typename Comparable>
void percolate(vector<Comparable> & a,int k,int n){        //"渗透",k 是我们当前关注的节点,我们要保证它比其左右孩子都要小
    int i=k;            //i 是要筛选的节点
    int j=2*i;          //j 是 i 的左孩子
    while(j<=n){
        if(j<n && a[j]>a[j+1])
            j++;                //确保 a[j]是左右孩子中的较小者
        if(a[i]<a[j])           //满足小根堆的条件,结束
            break;
        else{
            swap(a[i],a[j]);
            i=j;
            j=2*i;
        }
    }
```

```
}
template <typename Comparable>
void HeapSort(vector<Comparable> & a){
    int n=a.size()-1;
    for(int i=n/2;i>=1;i--)              //n/2 是第 1 个有孩子的节点.从下往上渗透,建堆
        percolate(a,i,n);
    for(int j=1;j<n;j++){
        //cout<<a[1]<<"   ";                 //输出堆顶元素
        swap(a[1],a[n-j+1]);          //C++内置 swap()函数
        percolate(a,1,n-j);           //重新建堆
    }
}
/*归并排序*/
template<typename Comparable>
void Merge(vector<Comparable> & a,vector<Comparable> & tmpArray,int leftPos,int rightPos,int
rightEnd){
    int leftEnd=rightPos-1;
    int tmpPos=leftPos;
    int numElements=rightEnd-leftPos+1;

    while(leftPos<=leftEnd && rightPos<=rightEnd)
        if(a[leftPos]<=a[rightPos])
            tmpArray[tmpPos++]=a[leftPos++];
        else
            tmpArray[tmpPos++]=a[rightPos++];
    while(leftPos<=leftEnd)
        tmpArray[tmpPos++]=a[leftPos++];
    while(rightPos<=rightEnd)
        tmpArray[tmpPos++]=a[rightPos++];
    for(int i=0;i<numElements;i++,rightEnd--)
        a[rightEnd]=tmpArray[rightEnd];
}
template<typename Comparable>
```

```
void MergeSort(vector<Comparable> & a,vector<Comparable> & tmpArray,int left,int right){
    if(left<right){
        int center=(left+right)/2;
        MergeSort(a,tmpArray,left,center);
        MergeSort(a,tmpArray,center+1,right);
        Merge(a,tmpArray,left,center+1,right);
    }
}
int main(){
    const int N=100000;                 //对十万个随机数进行排序
    clock_t begin;
    clock_t end;
    srand((unsigned)time(NULL));
    vector<int> v;
    v.push_back(0);                 //数组第 1 个元素存放哨兵,而不是参与排序的数据

    for(int i=0;i<N/10;i++)
        v.push_back(rand());
    begin=clock();
    InsertSort(v);
    end=clock();
    cout<<"InsertSort: "<<(double)(end-begin)/CLOCKS_PER_SEC<<endl;

    v.clear();
    v.push_back(0);
    for(int i=0;i<N;i++)
        v.push_back(rand());
    begin=clock();
    ShellSort(v);
    end=clock();
    cout<<"ShellSort: "<<(double)(end-begin)/CLOCKS_PER_SEC<<endl;
```

```
v.clear();
v.push_back(0);
for(int i=0;i<N/10;i++)
	v.push_back(rand());
begin=clock();
BubbleSort(v);
end=clock();
cout<<"BubbleSort: "<<(double)(end-begin)/CLOCKS_PER_SEC<<endl;

v.clear();
v.push_back(0);
for(int i=0;i<N;i++)
	v.push_back(rand());
begin=clock();
QuickSort(v,1,N);
end=clock();
cout<<"QuickSort: "<<(double)(end-begin)/CLOCKS_PER_SEC<<endl;

v.clear();
v.push_back(0);
for(int i=0;i<N/10;i++)
	v.push_back(rand());
begin=clock();
SelectSort(v);
end=clock();
cout<<"SelectSort: "<<(double)(end-begin)/CLOCKS_PER_SEC<<endl;

v.clear();
v.push_back(0);
for(int i=0;i<N;i++)
	v.push_back(rand());
begin=clock();
```

```
	HeapSort(v);
	end=clock();
	cout<<"HeapSort: "<<(double)(end-begin)/CLOCKS_PER_SEC<<endl;

	v.clear();
	v.push_back(0);
	for(int i=0;i<N;i++)
		v.push_back(rand());
	vector<int> vv(v.size());
	begin=clock();
	MergeSort(v,vv,1,v.size()-1);
	end=clock();
	cout<<"MergeSort: "<<(double)(end-begin)/CLOCKS_PER_SEC<<endl;

	return 0;
}
```