排序——数据结构

排序

假设含有n个记录的序列为{r1,r2,.....,rn}，其相应的关键字分别为{k1,k2,.....,kn}，需要确定一种排列，使其相应的关键字满足非递增或非递减关系，即使得序列成为一个按关键字有序的序列，这样的操作就称为排序。

排序的稳定性

假设ki=kj (1<=i<=n,1<=j<=n,i≠j)，且在排序前的序列中ri领先于rj(即i<j)。如果排序后ri仍领先于rj，则称所用的排序方法是稳定的；反之，若可能使得排序后的序列中rj领先ri，则称所用的排序方法是不稳定的。

排序方法	平均情况	最好情况	最坏情况	辅助空间	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
简单选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
直接插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
希尔排序	$O(n\log n)\sim O(n^2)$	$O(n^{1.3})$	$O(n^2)$	$O(1)$	不稳定
堆排序	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(1)$	不稳定
归并排序	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(n)$	稳定
快速排序	$O(n\log n)$	$O(n\log n)$	$O(n^2)$	$O(\log n)\sim O(n)$	不稳定

直接插入排序

直接插入排序的基本操作是将一个记录插入到已经排好序的有序表中，从而得到一个新的、记录数增1的有序表。

直接插入的实现

```
//直接插入
void InsertSort(int* a, int n)

{
    //多趟排序
    for (int i = 0; i < n-1 ; i++)
    {
        //单趟排序
        //在[0,end]区间中插入tmp，依旧有序
        int end = i;
        int tmp = a[end + 1];
        while (end >= 0)
        {
            if (a[end] > tmp)
            {
                a[end + 1] = a[end];
                end--;
            }
```

```
            else
            {
                break;
            }
        }
        a[end + 1] = tmp;
    }
}
```

直接插入的特点

- 元素集合越接近有序，直接插入排序算法的时间效率越高
- 时间复杂度：O(N^2)
- 空间复杂度：O(1)
- 稳定性：稳定

希尔排序

先选定一个整数gap，把待排序文件中所有记录分组，所有距离为gap的记录分在同一组内，并对每一组内的记录进行排序。然后，重复上述分组和排序的工作。当到达gap=1时，所有记录在统一组内排好序。

希尔排序的实现

```
//希尔排序
void ShellSort(int* a, int n)
{
    //gap>1时，为预排序，接近有序
    //gap=1时，为直接插入排序
    int gap = n;
    while (gap > 1)
    {
        gap = gap / 3 + 1;
        //3个间隔为gap的组同时交叉排序
        for (int i = 0; i < n - gap; i++)
        {
            //间隔为gap的预排序
            int end = i;
            int tmp = a[end + gap];
            while (end >= 0)
            {
                if (a[end] > tmp)
                {
                    a[end + gap] = a[end];
                    end -= gap;
                }
                else
                {
                    break;
                }
            }
            a[end + gap] = tmp;
        }
    }
}
```

希尔排序的特点

- 希尔排序是对直接插入排序的优化。
- 当gap > 1时都是预排序，目的是让数组更接近于有序。当gap == 1时，数组已经接近有序的了，这样就会很快。这样整体而言，可以达到优化的效果。我们实现后可以进行性能测试的对比。
- 希尔排序的时间复杂度不好计算，需要进行推导，推导出来平均时间复杂度： $O(N^{1.3—N}2)$
- 稳定性：不稳定

简单选择排序

每一次从待排序的数据元素中选出最小（或最大）的一个元素，存放在序列的起始位置，直到全部待排序的数据元素排完。

简单选择的实现

```
void Swap(int* a, int* b)
{
    int* tmp = *a;
    *a = *b;
    *b = tmp;
}
//直接选择排序
//一次选择两个数，最小的放前边，最大的放后边
void SelectSort(int* a, int n)
{
    int begin = 0, end = n - 1;
    while (begin < end)
    {
        //[begin,end]区间选出一个最小的一个最大的数的下标
        int min = begin, max = end;
        for (int i = begin; i <= end;  i++)
        {
            if (a[i] > a[max])
            {
                max = i;
            }
            if (a[i] < a[min])
            {
                min = i;
            }
        }
        Swap(&a[begin], &a[min]);
        if (begin == max)
        {
            max = min;
        }
        Swap(&a[end], &a[max]);
        begin++;
        end--;
    }
}
```

简单选择的特点

- 时间复杂度：O(N^2)

- 空间复杂度：O(1)
- 稳定性：不稳定

堆排序

堆排序就是利用堆(假设利用大根堆)进行排序的方法。它的基本思想是，将待排序的序列构造成一个大根堆。此时，整个序列的最大值就是堆顶的根结点。将它移走(其实就是将其与堆数组的末尾元素交换，此时末尾元素就是最大值)，然后将剩余的n-1个序列重新构造成一个堆，这样就会得到n个元素中的次大值，如此反复执行。

堆排序的实现

```
void Swap(int* a, int* b)
{
    int* tmp = *a;
    *a = *b;
    *b = tmp;
}
//向下调整算法
void AdjustDown(int* a, int n, int root)
{
    int parent = root;
    int child = parent * 2 + 1;
    while (child < n)
    {
        //选出左右孩子中大的那一个
        if (child + 1 < n&&a[child + 1] > a[child])
        {
            child++;
        }
        //比较孩子比父亲大，交换，继续向下调整
        if (a[child] > a[parent])
        {
            Swap(&a[child], &a[parent]);
            parent = child;
            child = parent * 2 + 1;
        }
        else
        {
            break;
        }
    }
}
//堆排序
void HeapSort(int* a, int n)
{
    //排升序建大堆 O(n)
    for (int i = (n - 1 - 1) / 2; i >= 0; i--)
    {
        AdjustDown(a, n, i);
    }
    int end = n - 1;
    while (end > 0)
    {
        //把堆顶元素依次换到最后
        Swap(&a[0], &a[end]);
```

```
        //调整选出剩下数中，最大的
        AdjustDown(a, end, 0);
        end--;
    }
}
```

堆排序的特点

- 堆排序使用堆来选数，效率就高了很多
- 时间复杂度：O(N*logN)
- 空间复杂度：O(1)
- 稳定性：不稳定

冒泡排序

冒泡排序是一种交换排序，它的基本思想是：两两比较相邻记录的关键字，入股偶反序则交换，直到没有反序为止。

冒泡排序的实现

```
//冒泡排序
void BubbleSort(int* a, int n)
{
    //控制循环的次数
    for (int i = 0; i < n; i++)
    {
        //flag用来标记，初始化为0
        int flag = 0;
        //n-1是最后一个元素的下标，n-1-1是倒数第二个元素的下标
        for (int j = n - 1 - 1; j >= i; j--)
        {
            if (a[j] > a[j + 1])
            {
                Swap(&a[j], &a[j + 1]);
                //若有数据交换则置flag为1
                flag = 1;
            }
        }
        if (flag == 0)
        {
            break;
        }
    }
}
```

冒泡排序的特点

- 冒泡排序是一种非常容易理解的排序
- 时间复杂度：O(N^2)
- 空间复杂度：O(1)
- 稳定性：稳定

快速排序

快速排序的基本思想是：通过一趟排序将待排记录分割成独立的两部分，其中一部分记录的关键字均比另一部分记录的关键字小，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的目的。

快速排序的实现

快速排序分为两个函数PartSort和QuickSort，PartSort函数就是将要排序序列一分为二，先选取当中的一个关键字作为标准，然后想办法将它放到一个位置，使得它左边的值都比它小，右边得值都比它大；QuickSort函数就是递归实现快速排序。

```
//快速排序
//左右指针法
//选取最右边的为关键字key,分别定义左右两个指针begin和end,begin找大于key的,end找小于key的,找到后交换，直到左右指针相遇，交换相遇位置和最右边的关键字
int PartSort1(int* a, int begin, int end)
{
    //end做key,左边先走；begin做key，右边先走
    int key = a[end];
    int keyindex = end;
    while (begin < end)
    {
        //begin找大于key的
        while (begin < end && a[begin] <= key)
        {
            begin++;
        }
        //end找小于key的
        while (begin < end && a[end] >= key)
        {
            end--;
        }
        Swap(&a[begin], &a[end]);
    }
    Swap(&a[begin], &a[keyindex]);
    return begin;
}
//挖坑法
int PartSort2(int* a, int begin, int end)
{
    //用下标为0的那个位置做坑
    int key = a[begin];
    //从两端交替向中间扫描
    while (begin < end)
    {
        //end找小于key的
        while (begin < end && a[end] >= key)
        {
            end--;
        }
        //将下标为end的元素与坑交换，end做新的坑
        a[begin] = a[end];
        //begin找大于key的
        while (begin < end && a[begin] <= key)
        {
            begin++;
        }
        //将下标为begin的元素与坑交换，begin又做新的坑
        a[end] = a[begin];
```

```
    }
    a[begin] = key;
    return begin;
}
//前后指针法
//选取最右边的为关键字Key,分别定义左右两个指针pre和cur,初始化pre为最左边的前一个位置，cur为最左边的位置，若a[cur]大于key则cur++，若a[cur]小于key，pre++,a[ore]和a[cur]交换，直到cur遇到end就停下来，pre++，交换a[pre]和a[end]。
int PartSort3(int* a, int begin, int end)
{
    int pre = begin - 1;
    int cur = begin;
    int key = a[end];
    //cur指针遇到key的位置就结束
    while (cur < end)pre
    {
        if (a[cur] < key && ++pre != cur)
        {
            Swap(&a[pre], &a[cur]);
        }
        cur++;
    }
    pre++;
    Swap(&a[pre], &a[end]);
    return pre;
}
void QuickSort(int* a, int begin, int end)
{
    if (begin >= end)
    {
        return;
    }
    //[begin,keyindex-1] keyindex [keyindex+1,end]
    //int keyindex = PartSort1(a, begin, end);
    //int keyindex = PartSort2(a, begin, end);
    int keyindex = PartSort3(a, begin, end);
    QuickSort(a, begin, keyindex - 1);
    QuickSort(a, keyindex + 1, end);
}
```

快速排序的特点

- 快速排序整体的综合性能和使用场景都是比较好的，所以才敢叫快速排序
- 时间复杂度：O(N*logN)
- 空间复杂度：O(logN)
- 稳定性：不稳定

归并排序

归并排序的原理是假设初始序列含有n个记录，则可以看成是n个有序的子序列，每个子序列的长度为1，然后两两归并，得到n/2个长度为2或1的有序子序列，再两两归并，如此重复，直至得到一个长度为n的有序序列为止。

归并排序的实现

```
//归并排序
```

```c
void MSort(int* a, int begin, int end, int* tmp)
{
    if (begin >= end)
    {
        return;
    }
    //先划分
    int mid = (begin + end) >> 1;
    MSort(a, begin, mid, tmp);
    MSort(a, mid + 1, end, tmp);
    //归并两段有序区间
    int begin1 = begin, end1 = mid;
    int begin2 = mid + 1, end2 = end;
    int index = begin;
    while (begin1 <= end1 && begin2 <= end2)
    {
        if (a[begin1] < a[begin2])
        {
            tmp[index] = a[begin1];
            index++;
            begin1++;
        }
        else
        {
            tmp[index] = a[begin2];
            index++;
            begin2++;
        }
    }
    if (begin1 <= end1)
    {
        while (begin1 <= end1)
        {
            tmp[index] = a[begin1];
            index++;
            begin1++;
        }
    }
    else
    {
        while (begin2 <= end2)
        {
            tmp[index] = a[begin2];
            index++;
            begin2++;
        }
    }
    memcpy(a+begin, tmp+begin, sizeof(int)*(end-begin+1));
}
void MergingSort(int* a, int n)
{
    int* tmp = (int*)malloc(sizeof(int)*n);
    MSort(a, 0, n - 1, tmp);
    free(tmp);
}
```

归并排序的特点

- 归并的缺点在于需要O(N)的空间复杂度，归并排序的思考更多的是解决在磁盘中的外排序问题。
- 时间复杂度：O(N*logN)
- 空间复杂度：O(N)
- 稳定性：稳定

计数排序

计数排序的实现

计数排序的原理是先统计相同元素出现次数，然后根据统计的结果将序列回收到原来的序列中。

```c
//计数排序
void CountSort(int* a, int n)
{
    int max = a[0], min = a[0];
    //找出最大的和最小的
    for (int i = 0; i < n; i++)
    {
        if (a[i] > max)
        {
            max = a[i];
        }
        if (a[i] < min)
        {
            min = a[i];
        }
    }
    //申请计数空间
    int length = max - min + 1;
    int* countarr = (int*)malloc(sizeof(int)*length);
    //计数数组初始化为0
    memset(countarr, 0, sizeof(int)*length);
    //统计次数
    for (int i = 0; i < n; i++)
    {
        int index = a[i];
        countarr[index - min]++;
    }
    //根据次数进行排序
    int j = 0;
    for (int i = 0; i < length; i++)
    {
        if (countarr[i] == 0)
        {
            break;
        }
        else
        {
            a[j] = i + min;
            j++;
        }
    }
}
```

计数排序的特点

- 计数排序在数据范围集中时，效率很高，但是适用范围及场景有限。
- 时间复杂度：O(MAX(N,范围))
- 空间复杂度：O(范围)
- 稳定性：稳定