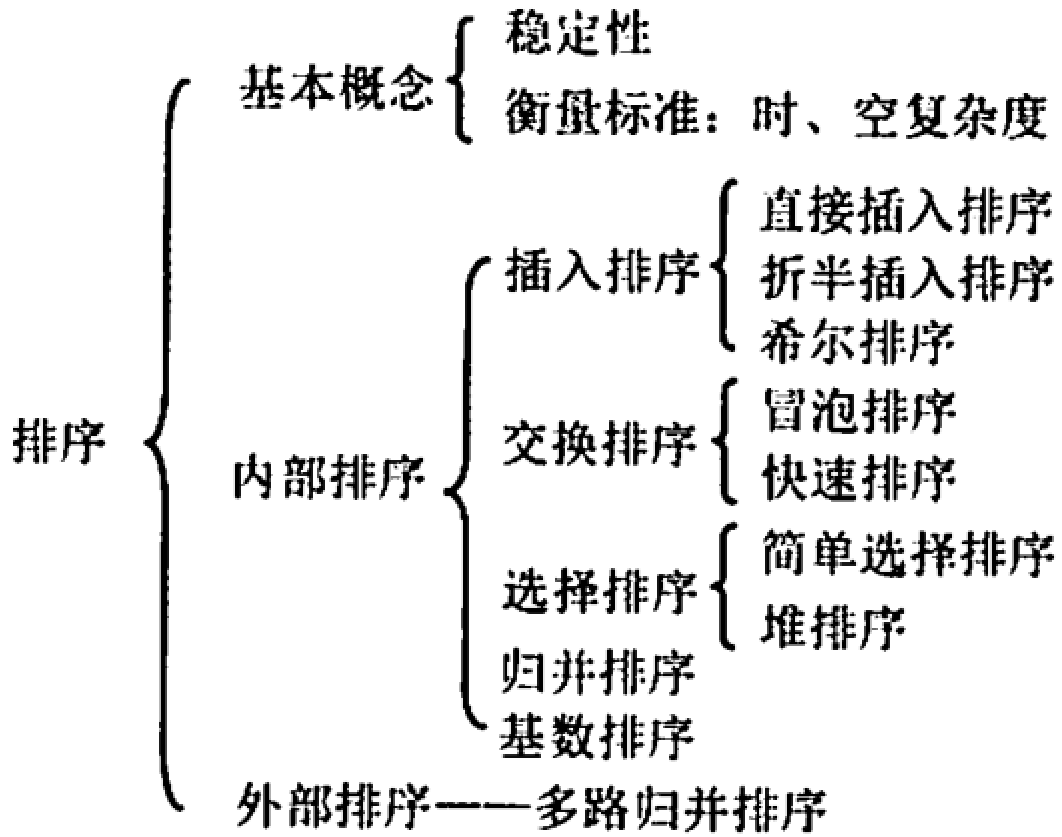
9-排序

C生万物 ● 大道至简 ● 鲍鱼科技+v(15339278619)

1、目标

> 📌 能识别出排序算法属于哪一种排序方法
>
> 掌握排序的分类
>
> 掌握排序的稳定性、时间复杂度、空间复杂度的比较
>
> 了解外部排序的基本概念和方法

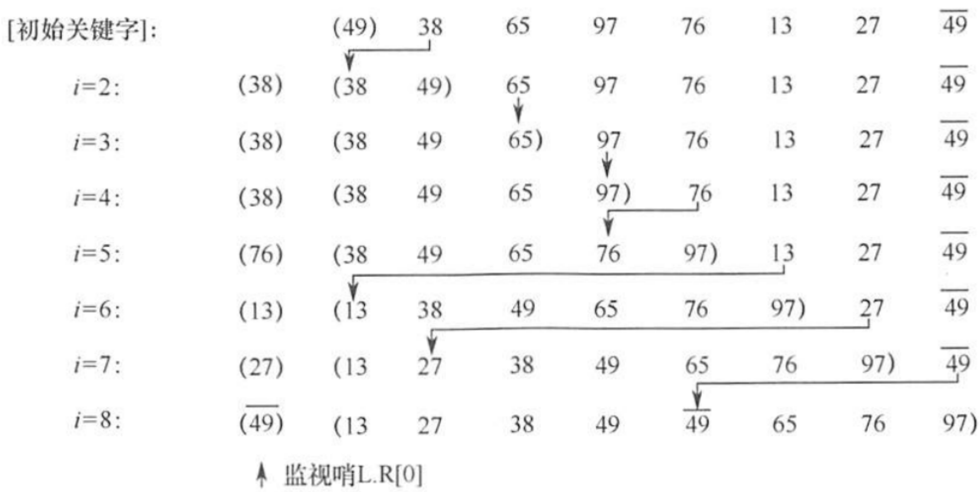


2、排序的分类

> 📌 1、根据数据是否**一次性**全部在内存排序： 内排、外排
>
> 2、根据**重复元素**的位置是否发生变化： 稳定排序、不稳定排序
>
> 3、根据**时间复杂度**是否小于等于O(nlog2n)：简单排序、先进排序
>
> 4、根据实现排序的**方式**： 插入排序、交换排序、选择排序、归并排序、基数排序

3、插入排序

> 📌 将待排元素往已排序好的序列进行插入，叫做插入排序
>
> 重点要理解希尔排序的思想，为什么效率会高?



1、直接插入排序

```c
//直接插入排序--从前往后比较
void InsertSort_1(int *ar, int left, int right)
{
    for(int i=left+1; i<right; ++i)
    {
        int k = left;
        while(ar[i] > ar[k])
            k++;

        int tmp = ar[i];
        for(int j=i; j>k; --j)
            ar[j] = ar[j-1];
        ar[k] = tmp;
    }
}

//直接插入排序--从后往前比较
void InsertSort_2(int *ar, int left, int right)
{
    for(int i=left+1; i<right; ++i)
    {
        int j = i;
```

```c
        while(j>left && ar[j] < ar[j-1])
        {
            Swap(&ar[j], &ar[j-1]);
            j--;
        }
    }
}

//直接插入排序--从后往前比较,不调用交换函数
void InsertSort_3(int *ar, int left, int right)
{
    for(int i=left+1; i<right; ++i)
    {
        int j = i;
        int tmp = ar[i];
        while(j>left && tmp < ar[j-1])
        {
            ar[j] = ar[j-1];
            j--;
        }
        ar[j] = tmp;
    }
}

//直接插入排序--哨兵位
void InsertSort_4(int *ar, int left, int right)
{
    for(int i=left+1; i<right; ++i)
    {
        ar[0] = ar[i]; //哨兵位
        int j = i;
        while(ar[0] < ar[j-1])
        {
            ar[j] = ar[j-1];
            j--;
        }
        ar[j] = ar[0];
    }
}
```

📌 **空间复杂度为O(1)**: 直接插入排序自始至终只是用了一个临时空间

时间复杂度O(n^2):

最好情况，数据有序，只需要比较n-1次，不需要移动数据，所以时间复杂度为O(n)

最坏情况，数据逆序，需要比较n(n-1)/2次，移动数据n(n-1)/2次，所以时间复杂度为O(n^2)

稳定性：稳定排序

无论从前往后比较还是从后往前比较，当数据相等时即停止比较，相等数据的相对位置不变，因此直接插入排序属于稳定排序

2、折半插入排序

```
void BinInsertSort(int *ar, int left, int right)
{
    for(int i=left+1; i<right; ++i)
    {
        int tmp = ar[i];
        int low = left;
        int high = i - 1;
        while(low <= high)
        {
            int mid = (low + high) / 2;
            if(tmp >= ar[mid])
                low = mid + 1;
            if(tmp < ar[mid])
                high = mid - 1;
        }

        for(int j=i; j>low; --j)
            ar[j] = ar[j-1];
        ar[low] = tmp;
    }
}
```

📌 折半插入排序仅减少了比较的次数，但数据的移动次数不变，因此：

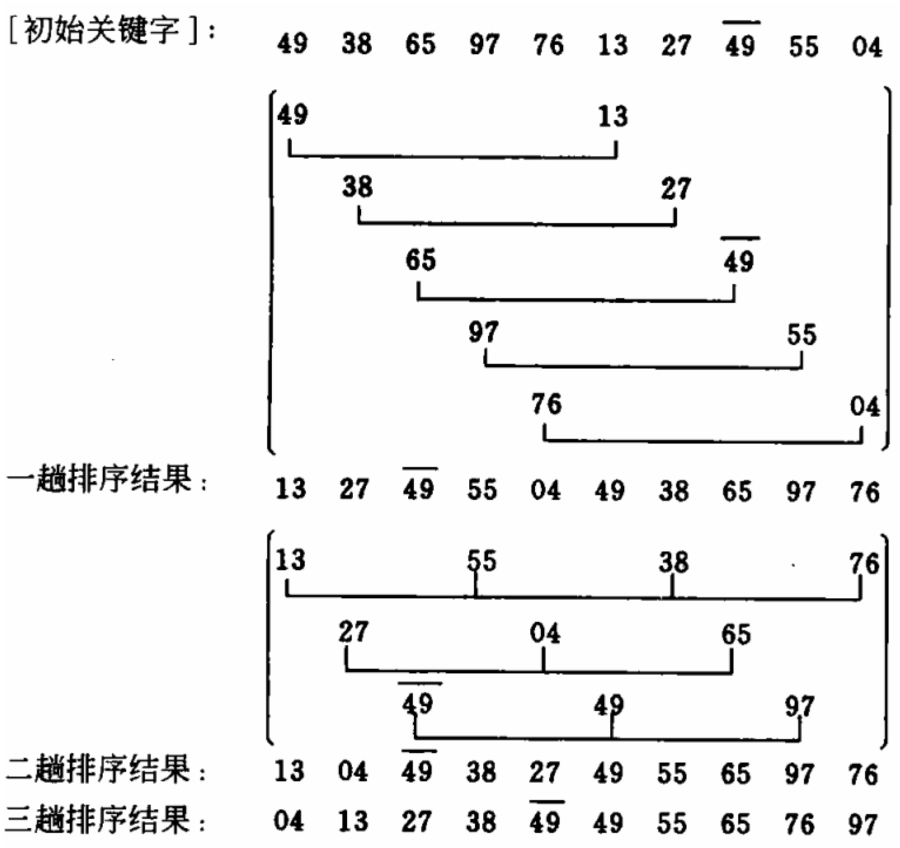
空间复杂度为O(1)、时间复杂度O(n^2)、属于稳定排序

3、希尔排序

📌 希尔排序的效率很高，为什么？毕竟希尔排序也属于插入排序，原因有二：

1、对于插入排序来说，**数据量越小，效率越高**

2、对于插入排序来说，**数据基本有序，效率越高**



- 固定增量

```
void _ShellSort(int *ar, int left, int right, int gap)
{
    for(int i=left+gap; i<right; ++i)
    {
        int tmp = ar[i];
        int j = i;
        while(j>left && tmp<ar[j-gap])
        {
            ar[j] = ar[j-gap];
            j -= gap;
        }
        ar[j] = tmp;
    }
}

int dlta[] = {5, 3, 2, 1}; //素数
void ShellSort(int *ar, int left, int right)
{
    int n = sizeof(dlta) / sizeof(dlta[0]);
    for(int i=0; i<n; ++i)
    {
        _ShellSort(ar, left, right, dlta[i]);
    }
}
```

- 自动增量

```
void ShellSort(int *ar, int left, int right)
{
    int gap = right - left;
    while(gap > 1)
    {
        gap = gap / 3 + 1;   //概数
        for(int i=left+gap; i<right; ++i)
        {
            int tmp = ar[i];
            int j = i;
            while(j>left && tmp<ar[j-gap])
            {
                ar[j] = ar[j-gap];
                j -= gap;
            }
            ar[j] = tmp;
        }
    }
}
```

希尔排序的分析是一个复杂的问题，因为它的时间是所取"增量"序列的函数，这涉及一些数学上尚未解决的难题。因此，到目前为止尚未有人求得一种最好的增量序列，但大量的研究已得出一些局部的结论。如有人指出，当增量序列为 $dlta[k]=2^{t-k+1}-1$ 时，希尔排序的时间复杂度为 $O(n^{3/2})$，其中 t 为排序趟数，$1\leqslant k\leqslant t\leqslant\lfloor\log_2(n+1)\rfloor$。还有人在大量的实验基础上推出：当 n 在某个特定范围内，希尔排序所需的比较和移动次数约为 $n^{1.3}$，当 $n\rightarrow\infty$ 时，可减少到 $n(\log_2 n)^2$[2]。增量序列可以有各种取法①，但需注意：应使增量序列中的值没有除 1 之外的公因子，并且最后一个增量值必须等于 1。

📌 **空间复杂度为O(1)**: 希尔排序只是用了一个临时空间

时间复杂度O(n^2):

当n在某个范围时，约为O(n^1.3), 最坏情况下为O(n^2)

稳定性：不稳定

当相同的关键字划分到不同的子序列时，可能会改变原来的相对位置，因此希尔排序为不稳定排序

4、交换排序

📌 顾名思义，就是通过比较交换顺序而达到的排序，就叫做交换排序

交换排序需要重点掌握快速排序的实现和改进

1、冒泡排序

```
//简单冒泡排序
void BubbleSort_1(int *ar, int left, int right)
{
    for(int i=left; i<right-1; ++i)
    {
        for(int j=left; j<right-1-i; ++j)
        {
            if(ar[j] > ar[j+1])
            {
                Swap(&ar[j], &ar[j+1]);
            }
        }
    }
}

//改进冒泡排序
void BubbleSort_2(int *ar, int left, int right)
{
    for(int i=left; i<right-1; ++i)
    {
        bool is_swap = false;
        for(int j=left; j<right-1-i; ++j)
        {
            if(ar[j] > ar[j+1])
            {
                Swap(&ar[j], &ar[j+1]);
                is_swap = true;
            }
        }
        if(!is_swap)
            break;
    }
}
```

📌 **空间复杂度为O(1)**: 冒泡排序只需要在交换数据的时候使用一个临时空间

时间复杂度O(n^2):

最好情况：数据有序，比较n-1次，时间复杂度为O(n)

最坏情况：数据逆序，比较n(n-1)/2,时间复杂度为O(n^2)

稳定性：稳定

由于ar[j]跟ar[j+1]比较的过程中，如果ar[j]等于ar[j+1]不交换，所以冒泡排序为稳定排序

2、快速排序

📌 快速排序从名字上看，就知道效率不低，要不也不能称为快排，不是金刚钻，也不敢揽瓷器活

快排是基于分治策略实现的，所以代码使用递归进行处理

一趟快排就是排序好一个数据，并以此数据为分割点进行快速排序

快排有两个改进：

1、为了避免取到极值，可以采取三数取中法

2、如果数据量小于阈值，可以直接采取直接插入排序

5	1	3	9	8	2	6	4	7

```
//简单快排
int _Partition_1(int *ar, int left, int right)
{
    int low = left;
    int high = right - 1;

    int key = ar[low];

    while(low < high)
    {
        while(high>low && ar[high] > key)
                high--;
        Swap(&ar[high], &ar[low]);

        while(low<high && ar[low] <= key)
                low++;
        Swap(&ar[low], &ar[high]);
    }
    return  low;
}
```

```c

//改进快排
int _Partition_2(int *ar, int left, int right)
{
    int low = left;
    int high = right - 1;

    int key = ar[low];

    while(low < high)
    {
        while(high>low && ar[high] > key)
            high--;
        ar[low] = ar[high];

        while(low<high && ar[low] <= key)
            low++;
        ar[high] = ar[low];
    }
    ar[low] = key;
    return  low;
}

//三数取中
int GetMidIndex(int *ar, int left, int right)
{
    int mid = (right-1 + left) / 2;
    if(ar[left]<ar[mid] && ar[mid]<ar[right-1])
            return mid;
    if(ar[mid]<ar[left] && ar[left]<ar[right-1])
            return left;
    return right-1;
}

//双指针法
int _Partition_3(int *ar, int left, int right)
{
    //三数取中
    int mid_index = GetMidIndex(ar, left, right);
    if(mid_index == left)
        Swap(&ar[left], &ar[mid_index]);

    //////////////////////////////////////////////////////////
    int key = ar[left];
    int prev = left;

    for(int cur=left+1; cur<right; ++cur)
```

```
    {
        if(ar[cur] < key)
        {
            prev++;
            if(prev != cur)
            {
                Swap(&ar[prev], &ar[cur]);
            }
        }
    }

    Swap(&ar[left], &ar[prev]);
    return prev;
}

#define M 5
void QuickSort(int *ar, int left, int right)
{
    if(left >= right)
        return;

    if(right - left <= M)
        InsertSort_3(ar, left, right); //小于阈值，直接插入排序
    else
    {
        int pos = _Partition_3(ar, left, right); //寻找曲轴点
        QuickSort(ar, left, pos);      //左区间
        QuickSort(ar, pos + 1, right); //右区间
    }
}
```

📌 **空间复杂度为O(log2(n))**:

最好情况：每趟快排的曲轴点能够均分数据，因此将数据形成二叉树，高度为log2(n),所以递归调用需要的栈空间为log2(n),则空间复杂度为O(log2(n))

最坏情况：每趟排序的曲轴点都 是最大或者最小数据，导致树的高度为n，所以递归调用需要的栈空间为n-1,因此空间复杂度为O(n)

时间复杂度O(nlog2(n)):

看下面的推导：

稳定性：不稳定

再划分区间时，如果右端有两个相同的关键字，且值均小于基准值，则在交换到左端时，他们的相对位置会发生变化，因此快排是一种不稳定排序。

- 快排时间复杂度的推导

> 📌 根据代码知道，每一层的递归操作次数为该次递归所传入的元素个数（忽略每次的曲轴元素）即：
>
> 第1层是n次，
>
> 第2层有2次递归，每次n/2次，共n次操作，
>
> 第3层有4次递归，每次n/4次，共n次操作，
>
> ……
>
> （最后一层）第k层有k次递归，每次n/2^(k-1)次，共n次操作
>
> 由于递归结束的条件是只有一个元素，所以这里的n/2^(k-1)=1 => k=log2(n)+1
>
> 即递归树的深度为log2(n)
>
> 时间复杂度=每层的操作次数*树的深度=nlog2(n) 即：O(nlog2(n));

时间复杂度分析:

经过上述一趟快速排序，我们只确定了一个元素的最终位置，我们最终需要经过n趟快速排序才能将一个含有 n 个数据元素的序列排好序，下面我们来分析其时间复杂度.

设 n 为待排序数组中的元素个数， $T(n)$ 为算法需要的时间复杂度，则

$$T(n)=\begin{cases} D(1) & n\le 1 \\ D(n)+T(I_1)+T(I_2) & n>1 \end{cases}$$

其中 $D(n)=n-1$,是一趟快排需要的比较次数，一趟快排结束后将数组分成两部分 I_1 和 I_2

最好时间复杂度:

核心点:最好情况下，每一次划分都正好将数组分成长度相等的两半

$$T(n)=\begin{cases} D(1) & n\le 1 \\ D(n)+T(\frac{n}{2})+T(\frac{n}{2}) & n>1 \end{cases}$$

$$\begin{aligned} \text{所以}: T(n) &= D(n)+2T(n/2) \\ &= D(n)+2D(n/2)+4T(n/4) \\ &\vdots \\ &= D(n)+2D(n/2)+\cdots 2^kD(n/2^k) \\ &= n-1+2(\frac{n}{2}-1)+\cdots 2^k(\frac{n}{2^k}-1) \end{aligned}$$

$$
\begin{aligned}
&= n-1+n-2+\cdots n-2^k \\
&\because k = log_2 n \\
&\therefore \text{原式} = nlog_2 n - 2n + 1 \in O(nlog_2 n)
\end{aligned}
$$

最坏时间复杂度:

核心点:最坏情况下，每一次划分都将数组分成了0和n-1两部分

$$
T(n) = \begin{cases} D(1) & n \leq 1 \\ D(n) + T(0) + T(n-1) & n > 1 \end{cases}
$$

$$
\begin{aligned}
\text{所以}: T(n) &= D(n) + T(n-1) \\
&= D(n) + D(n-1) + T(n-2) \\
&\vdots \\
&= D(n) + D(n-1) + \cdots + D(2) + D(1) \\
&= n-1+n-2+\cdots+1+0 \\
&= \frac{n(n-1)}{2} \in O(n^2)
\end{aligned}
$$

5、选择排序

📌 顾名思义，每次通过选择剩下元素的极值（最大值或最小值），往已排序好的序列之后存放，通过选择元素而达到的排序叫做选择排序

重点把握堆排序，堆排是一种通过堆结构进行数据选择，从而避免不必要的元素比较，效率很高

1、简单选择排序

```
int _GetMinIndex(int *ar, int left, int right)
{
    int min_val = ar[left];
    int min_index = left;
    for(int i=left+1; i<right; ++i)
    {
        if(ar[i] < min_val)
        {
            min_val = ar[i];
```

```
            min_index = i;
        }
    }
    return min_index;
}
void SelectSort(int *ar, int left, int right)
{
    for(int i=left; i<right-1; ++i)
    {
        int min_index = _GetMinIndex(ar, i, right);
        if(min_index != i)
            Swap(&ar[i], &ar[min_index]);
    }
}
```

📌 **空间复杂度为O(1):** 仅使用一个临时空间作为交换使用

时间复杂度O(n^2):

选择最值的比较次数为n(n-1)/2

稳定性：不稳定

比如数据为{3,3,1}, 经过一趟排序之后变成{1,3,3},很明显相同数据已经发生了位置交换，因为简单选择排序为不稳定排序

2、堆排序

📌 堆排序分为两个步骤：建堆+排序

升序排序：建大堆

降序排序：建小堆

```
void _AdjustDown(int *ar, int left, int right, int start)
{
    int n = right - left;
    int i = start;
    int j = 2*i + 1;

    int tmp = ar[i];

    while(j < n)
    {
        if(j+1<n && ar[j]<ar[j+1])
```

```
            j++;

        if(tmp < ar[j])
        {
            ar[i] = ar[j];
            i = j;
            j = 2*i + 1;
        }
        else
            break;
    }
    ar[i] = tmp;
}
void HeapSort(int *ar, int left, int right)
{
    //构建大堆
    int n = right - left;
    int adj_pos = n/2-1 + left; //最后一个分支
    while(adj_pos >= left)
    {
        _AdjustDown(ar, left, right, adj_pos);
        adj_pos--;
    }

    //排序
    int end = right - 1;
    while(end > left)
    {
        Swap(&ar[left], &ar[end]);
        _AdjustDown(ar, left, end, left);
        end--;
    }
}
```

📌 **空间复杂度为O(1):** 仅使用一个临时空间作为调整和交换

时间复杂度O(nlog2(n)):

参考下面的推导:

稳定性：不稳定

比如数据为{3,3,1}, 经过一趟排序之后变成{3,1,3},很明显相同数据已经发生了位置交换，所以堆排序为不稳定排序

- 堆排序的时间复杂度推导:

堆排序的时间复杂度:
由于堆排序是由两部分(堆调整 + 堆排序)完成的，所以时间复杂度也应该是两部分之和。

首先堆调整，堆调整的时间复杂度为O(n)
假设堆高度为K,从倒数第二层开始每个节点都需要进行与子节点的比较，也就是要进行堆调整，所以计算如下

2^(i-1)*(k - i)其中i表示第几层，2^(i-1)表示第i层有几个节点，k 表示堆的深度，k-i表示需要向下调整几次
i 的取值范围为k -1,到1
所以总时间就是
s=2^(k - 2)*1 + 2^(k - 3)*2+......+2*(k-2) + 2^0*(k - 1)
化简得：s=2^k - k - 1;
因为二叉树的深度时间复杂度为O(log n)
所以s = n - log n - 1; 最后为O(n)

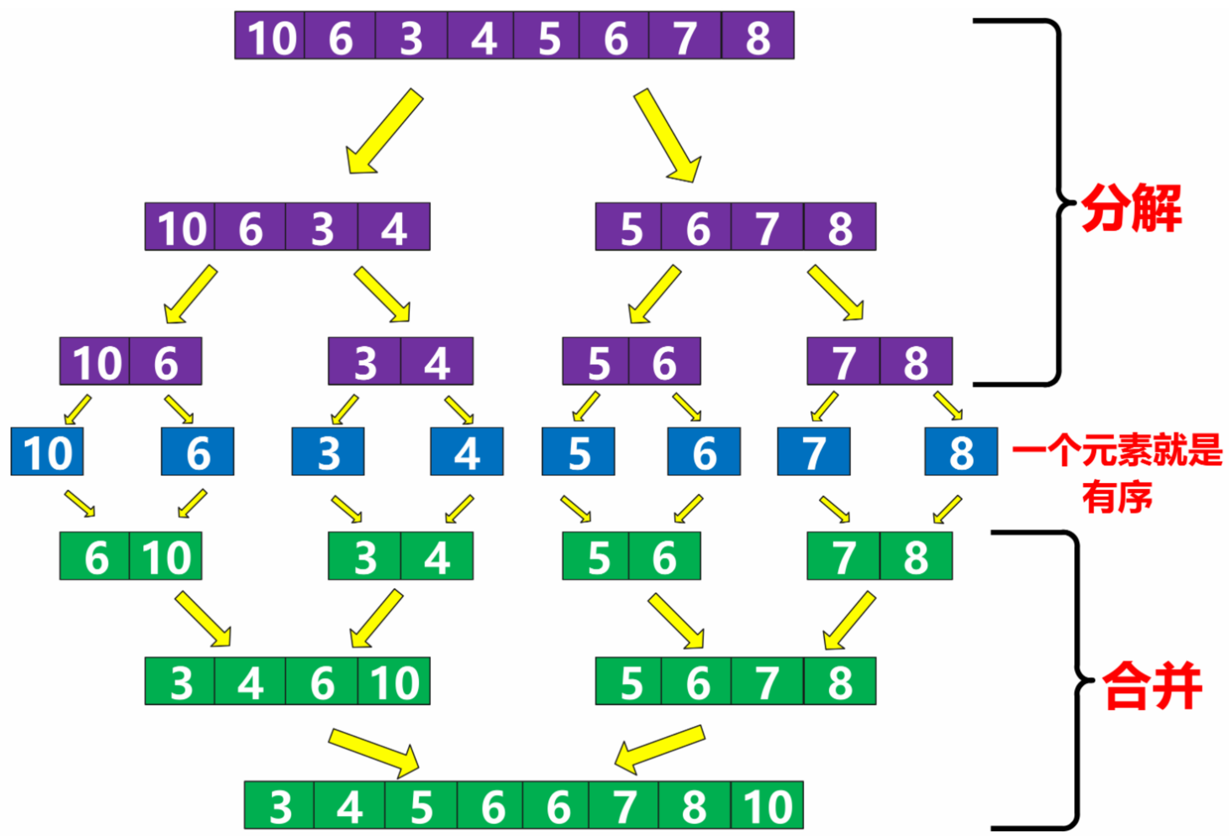
堆排序的时间复杂度为O(nlogn)
假设节点数为n,所以需要进行n - 1次调换，也就是需要n-1次堆调整，每次堆调整的时间复杂度为O(logn)，那么总的时间复杂度就是(n - 1)O(logn) = O(nlogn)

最后堆排序的时间复杂度为:
O(n) + O(nlogn) = O(nlogn)

6、归并排序

📌 通过将两个或两个以上的**有序**表进行合并，从而达到排序的过程就叫归并排序

归并排序由两个步骤组成：分解+合并



```c
void _MergeSort(int *ar, int left, int right, int *tmp)
{
    //分解
    if(left >= right)
        return;
    int mid = (left+right) / 2;
    _MergeSort(ar, left, mid, tmp);
    _MergeSort(ar, mid+1, right, tmp);

    //归并
    int begin1, end1, begin2, end2;
    begin1 = left, end1 = mid;
    begin2 = mid+1, end2 = right;

    int k = left;
    while(begin1<=end1 && begin2<=end2)
    {
        if(ar[begin1] < ar[begin2])
            tmp[k++] = ar[begin1++];
        else
            tmp[k++] = ar[begin2++];
    }

    while(begin1 <= end1)
        tmp[k++] = ar[begin1++];
    while(begin2 <= end2)
        tmp[k++] = ar[begin2++];

    memcpy(ar+left, tmp+left, sizeof(int)*(right-left+1));
}

void MergeSort(int *ar, int left, int right)
{
    int n = right - left;
    int *tmp = (int*)malloc(sizeof(int) * n);
    assert(tmp != NULL);

    //归并
    _MergeSort(ar, left, right-1, tmp);

    free(tmp);
}
```

📌 **空间复杂度为O(n):** 需要n个辅助空间

时间复杂度O(nlog2(n)):

参考下面的推导：

稳定性：稳定

合并过程不会更改相等数据的相对位置，所以归并排序为稳定排序

- 归并排序时间复杂度推导：

时间复杂度计算

1、首先可知

$$f(n) = 2f(\frac{n}{2}) + n$$

其中：

$$f(n)\text{表示对}n\text{个数进行归并排序}$$

$$2f(\frac{n}{2})\text{表示将}n\text{个数分成两部分分别进行归并排序}$$

$$n\text{表示对两个子过程结束之后合并的过程}$$

2、推导

$$f(\frac{n}{2}) = 2f(\frac{n}{4}) + \frac{n}{2} \qquad \text{当}n = \frac{n}{2}\text{时}$$

$$f(\frac{n}{4}) = 2f(\frac{n}{8}) + \frac{n}{4} \qquad \text{当}n = \frac{n}{4}\text{时}$$

$$\cdots\cdots$$

$$f(\frac{n}{2^{m-1}}) = 2f(\frac{n}{2^m}) + \frac{n}{2^{m-1}} \qquad \text{当}n = \frac{n}{2^{m-1}}\text{时}$$

3、由此可得：

$$f(n) = 2f(\frac{n}{2}) + n$$

$$= 2 \times \left(2f(\frac{n}{4}) + \frac{n}{2}\right) + n$$

$$= 2^2 f(\frac{n}{2^2}) + 2n$$

$$= 2^2 \times \left(2f(\frac{n}{8}) + \frac{n}{4}\right) + 2n$$

$$=2^3f(\frac{n}{2^3})+3n$$

$$\cdots\cdots$$

$$=2^mf(\frac{n}{2^m})+mn$$

当 m 足够大时(仅剩一个数字时)，可使得

$$\frac{n}{2^m}=1$$

求出

$$m=\log_2 n$$

$$代入\ \ f(n)=2^mf(\frac{n}{2^m})+mn\ \ 中可得$$

$$f(n)=2^{(\log_2 n)}f(1)+n\cdot\log_2 n$$

$$其中\ \ f(1)=0$$

$$所以最终\ \ f(n)=n\cdot\log_2 n$$

7、基数排序

📌 基数排序是唯一不通过比较和移动达到目的，而是基于多关键字的单逻辑排序的方法，主要通过分发和回收两个过程完成排序

```
//基数排序
#include"queue.h"
#define K 3
#define RADIX 10
LinkQueue Q[RADIX];

int GetKey(int value, int k)
{
    int key;
    while(k >= 0)
    {
```

```
        key = value % 10;
        value /= 10;
        k--;
    }
    return key;
}

//{278, 109, 63, 930, 589, 184, 505, 269, 8, 83};
void Distribute(int *ar, int left, int right, int k)
{
    for(int i=left; i<right; ++i)
    {
        int key = GetKey(ar[i], k);
        LinkQueuePush(&Q[key], ar[i]);
    }
}

void Collect(int *ar)
{
    int k = 0;
    for(int i=0; i<RADIX; ++i)
    {
        while(!LinkQueueEmpty(&Q[i]))
        {
            int data = LinkQueueFront(&Q[i]);
            LinkQueuePop(&Q[i]);
            ar[k++] = data;
        }
    }
}

void RadixSort(int *ar, int left, int right)
{
    //初始化基数
    for(int i=0; i<RADIX; ++i)
        LinkQueueInit(&Q[i]);

    for(int i=0; i<K; ++i)
    {
        //分发
        Distribute(ar, left, right, i);

        //回收
        Collect(ar);
    }
}
```

📌 **空间复杂度为O(r):**

一趟排序需要辅助空间为r个队列

时间复杂度O(d(n+r)):

基数排序需要进行d趟分配和收集，一趟分配需要O(n), 一趟收集需要O(r),与数据初始状态无关

稳定性：稳定

分配过程和回收过程不会对相等数据的位置产生影响，因此基数排序为稳定排序

8、排序比价

表 8.1　各种排序算法的性质

算法种类	时间复杂度			空间复杂度	是否稳定
	最好情况	平均情况	最坏情况		
直接插入排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	是
冒泡排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	是
简单选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	否
希尔排序				$O(1)$	否
快速排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n^2)$	$O(\log_2 n)$	否
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	否
2 路归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	是
基数排序	$O(d(n+r))$	$O(d(n+r))$	$O(d(n+r))$	$O(r)$	是

📌 时间复杂度:

先进排序是对数，基数排序最特殊，简单排序是n方

空间复杂度:

快排对数归并n, 基数排序是r,其余都是常数1

是否稳定:

先进简排不稳定，其余都是稳定排