数据结构重点(from 吴昊)

1.进出栈操作,可能或不可能出栈的序列

1.进出栈操作

进栈 (Push) 和出栈 (Pop) 操作是指在栈这种数据结构中的两种基本操作。栈是一种具有后进先出 (LIFO) 特性的数据结构，类似于我们平时堆叠的一叠书。

进栈操作将元素放入栈的顶部，而出栈操作将栈顶元素移除。

下面是进出栈操作的示例：

1. 进栈 (Push) 操作：
    - 将元素放入栈顶
    - 栈顶指针上移
2. 出栈 (Pop) 操作：
    - 移除栈顶元素
    - 栈顶指针下移

进栈和出栈操作可以用于构建各种算法和数据结构，如函数调用堆栈、括号匹配、表达式求值等。

2.可能或不可能出栈的序列

1.理论分析与结论

对于一个给定的序列，判断是否能够通过进栈和出栈操作得到该序列是一个常见的问题。以下是一个判断序列是否可能出栈的例子：

给定一个序列 [a, b, c, d, e]，以下是可能的进栈和出栈序列：

- 进栈序列：a, b, c, d, e
- 出栈序列：e, d, c, b, a

在此序列中，我们按照 a, b, c, d, e 的顺序进栈，并按照 e, d, c, b, a 的顺序出栈。这是一种可能的进出栈序列。

这里有一规律可记

任何出栈的元素后面出栈的元素必须满足以下三点：

1、在原序列中相对位置比它小的，必须是逆序；

2、在原序列中相对位置比它大的，顺序没有要求；

3、以上两点可以间插进行。

- 结论:后入栈的,出栈后比他先入栈且还未出栈的一定**倒序**排在后面(对每一个元素都适用!)
- 因此可以有一下思路:
  1. 选择一个元素(可以从第一个开始依次判断)
  2. 根据该元素判断已入栈元素(例如下面例子中选择出栈元素D则可判断出ABCD肯定都已入栈)
  3. 查看该元素在出栈序列前的元素,判断已出栈元素
  4. 判断已入未出是否满足结论要求

2.案例分析

技术之瞳 阿里巴巴技术笔试心得习题2.65：

一个栈的入栈序列为ABCDEF，则不可能的出栈序列是（D）

A、DEFCBA　　B、DCEFBA　　C、FEDCBA

D、FECDBA　　E、ABCDEF　　F、ADCBFE

分析:此处考察栈的先进后出特性,且此处的**入栈和出栈顺序是未知的**,**有可能入栈中穿插出栈**:

- 例如可以先入栈ABCD,后出栈D，然后入栈E，出栈E，入栈F，出栈F，然后CBA依次出栈，也就是A选项的情况。
- B选项并未出现结论的情况,顺序可以为:ABCD入栈,D,C出栈,E入栈E出栈,F入栈F出栈,BA出栈.
- C选项并未出现结论的情况,顺序可以为:ABCDEF入栈然后顺序出栈
- E选项虽然直观上完全不符合先进后出,但是可以出栈是穿插在入栈中的,因此即进即出即可满足,且也满足结论要求
- F选项并未出现结论的情况,顺序可以为:A入A出,BCD入DCB出,EF入FE出
- D选项的错误出在"CD"上,E之前F出栈则ABCD均为出栈,因此都应该是倒序,而CD是正序,则与结论相悖,答案错误

2.矩阵中求某个位置的地址,或给出地址求对应位置

3.特殊矩阵的用一维存储的映射函数

首先回顾有关于矩阵的几种数据类型

1.数组

1.1一维数组

```
int n;
ElemType a[n];//长度为n的一维数组,数据类型是ElemType
```

一维数组起始地址设置为Loc,数组中每个元素大小都相同,且存储时是连续存放的.

数组元素a[i]的存放地址=Loc+i*sizeof(ElemType),其中0≤i≤(n-1)

1.2二维数组

二维数组有两种存储方式，分别是行优先存储和列优先存储

对于M行N列的二维数组b [M] [N]，设二维数组b的初始地址为Loc，则：

（1）若按行优先存储，则b[i][j]的存储地址=Loc+(*i*N+j)*sizeof(ElemType)。

（2）若按列优先存储，则b[i][j]的存储地址=Loc+(*j*M+i)*sizeof(ElemType)。

2.矩阵

2.1普通矩阵

学过线性代数就不必多说了,可以按照二维数组存储方式

注意:描述矩阵元素时，行、列号通常从1开始；而描述数组时通常下标从0开始。

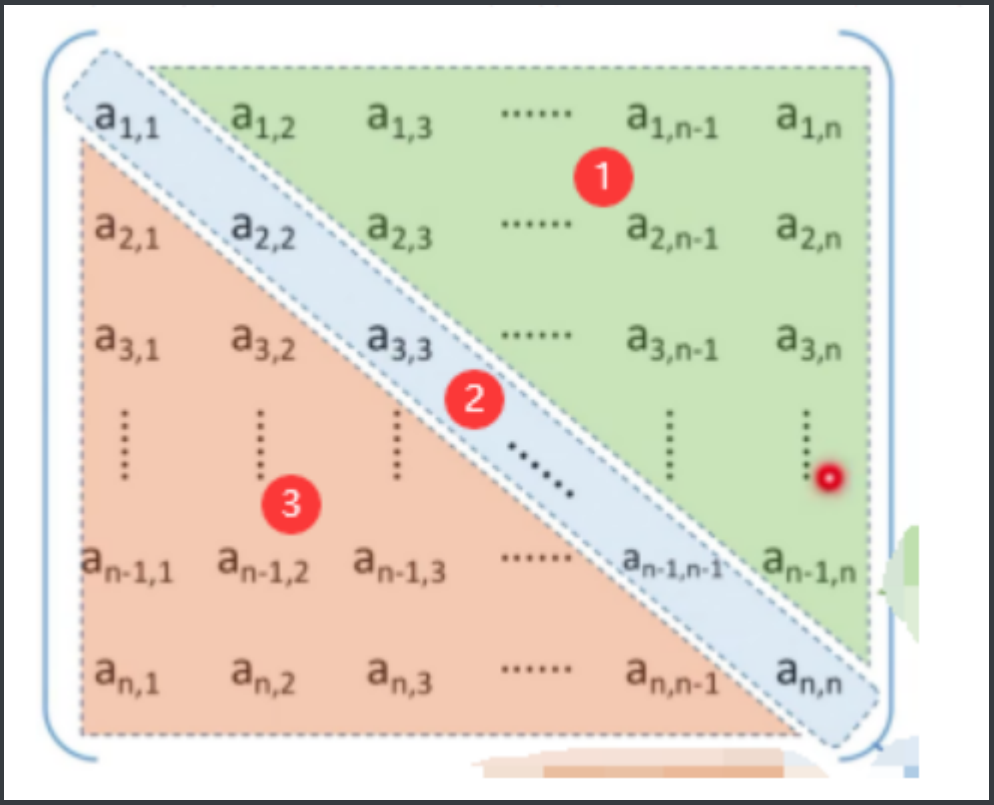
2.2特殊矩阵

注意,本节及一下讨论的都是方阵

2.2.1对称矩阵

若n阶方阵中任意一个元素aij都有aij=aji，则该矩阵为对称矩阵。按普通的存储方式即二维数组存储就浪费存储空间了。

对称矩阵的压缩存储策略：只存储主对角线和下三角区或者主对角线和上三角区。



2.2.1.1 策略1以及对应映射函数

存储策略:只存储主对角线+下三角区的元素,并且按**行优先**原则将各元素存入**一维数组**,因此需要存储(n+1)*n/2个元素.存储是为了使用，因此存储方式要使得访问矩阵中的某个元素时能很快找到它.可以实现映射函数:将矩阵元素a[i] [j]的下标映射为一维数组的b[k]的下标,其中i,j从1开始到n,k从0到(n+1)*n/2 - 1.因此矩阵的 `a[i][j]` 元素对应于一维数组的：

1. i≥j时是 `b[i(i-1)/2+j-1]`；
2. i<j时是 `b[j(j-1)/2+i-1]` 。

2.2.1.2 策略二以及对应映射函数

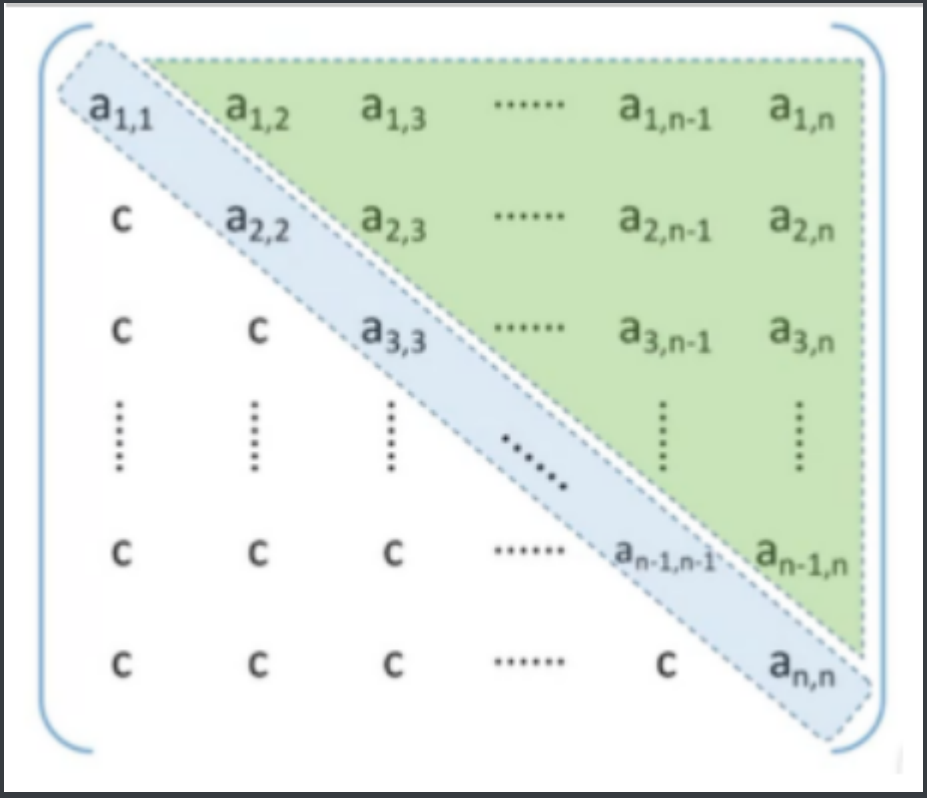
存储策略：只存储主对角线+下三角区的元素，并且按**列优先**原则将各元素存入**一维数组**中，因此需要存储(n+1)*n/2个元素。存储是为了使用，因此存储方式要使得访问矩阵中的某个元素时能很快找到它。可以实现一个映射函数——将矩阵元素a[i] [j]的下标映射为一维数组b[k]的下标，其中i、j从1开始到n，k从0到(1+n)*n/2-1。因此矩阵的a[i][j]元素对应于一维数组的：

1. i≥j时是 `b[(2n-j+2)(j-1)/2+i-j]`；
2. i<j时是 `b[(2n-i+2)(i-1)/2+j-i]`。

2.2.2三角矩阵

2.2.2.1 上三角矩阵

上三角矩阵：除了主对角线和上三角区，其余的元素都相同，如下：

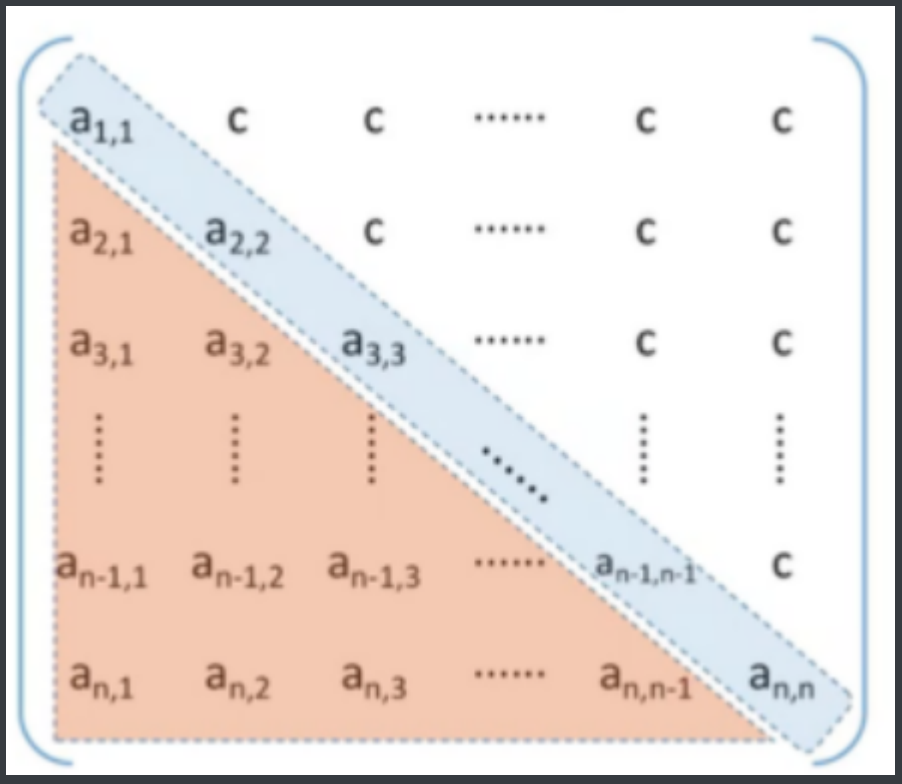


存储策略：按行优先原则将两个阴影部分合成的三角形区域的元素存入一维数组中，并在最后一个位置存储常量c，因此需要存储(1+n)*n/2+1个元素。可以实现一个映射函数——将矩阵元素a[i] [j]的下标映射为一维数组b[k]的下标，其中i、j从1开始到n，k从0到(1+n)*n/2。因此矩阵的a[i][j]元素对应于一维数组的：

1. i≤j时是 `b[(2n-i+2)(i-1)/2+j-i]` (列优先);
2. i<j时是 `b[(1+n)*n/2]`。

2.2.2.2 下三角矩阵

下三角矩阵：除了主对角线和下三角区，其余的元素都相同，如下：

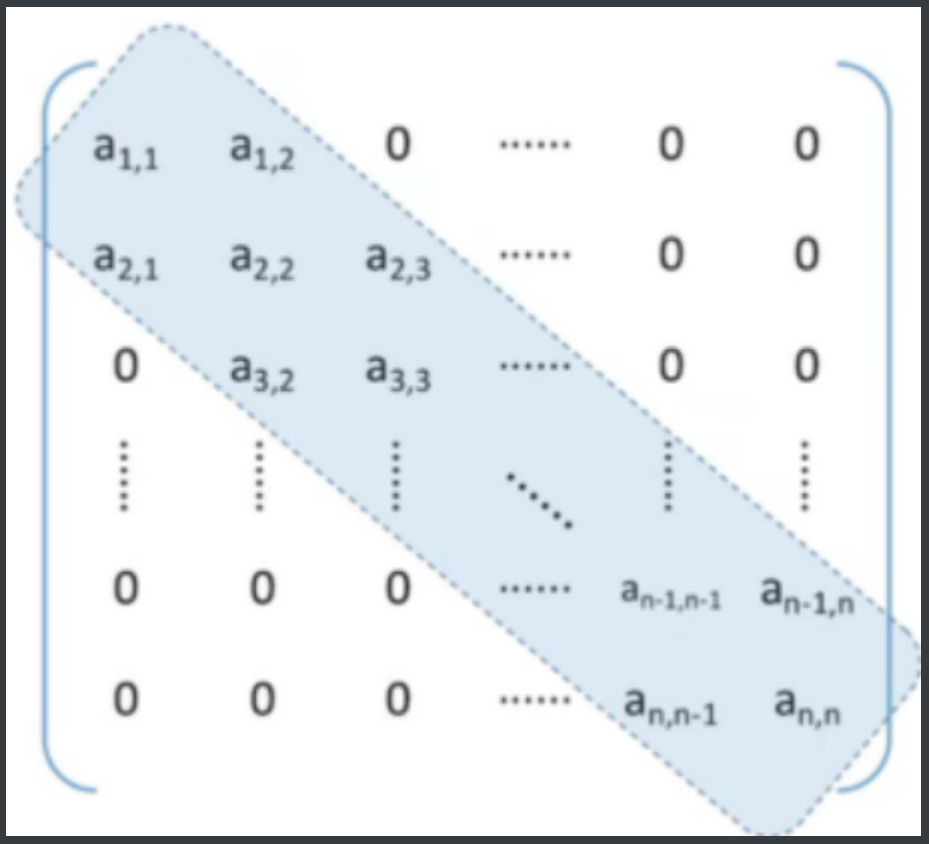


存储策略：按行优先原则将两个阴影部分合成的三角形区域的元素存入一维数组中，并在最后一个位置存储常量c，因此需要存储(1+n)*n/2+1个元素。可以实现一个映射函数——将矩阵元素a[i] [j]的下标映射为一维数组b[k]的下标，其中i、j从1开始到n，k从0到(1+n)*n/2。因此矩阵的a[i][j]元素对应于一维数组的：

1. i≥j时是 `b[i(i-1)/2+j-1]` (行优先)；
2. i<j时是 `b[(1+n)*n/2]` 。

2.2.3 三对角矩阵

三对角矩阵又称带状矩阵：当|i-j|>1时，有 `a[i][j]=0` ，其中1≤i,j≤n。



存储策略：按行优先原则，只存储带状部分，因此需要存储3n-2个元素。可以实现一个映射函数——将矩阵元素a[i][j]的下标映射为一维数组b[k]的下标，其中i、j从1开始到n，k从0到3n-3。因此矩阵的a[i][j]元素对应于：

1. |i-j|>1时是0；
2. |i-j|≤1时是一维数组的b[2i+j-3]。
3. 有一个问题，若已知数组下标k，如何得到矩阵的i,j？b[k]是数组中第k+1个元素，对应于矩阵的第 `i=⌈(k+2)/3⌉` 行第 `j=k-2i+3` 列的元素，[m]是对m向上取整。

2.2.4 稀疏矩阵

稀疏矩阵是非0元素远少于矩阵元素个数的矩阵,0很多

存储策略1:使用顺序存储三元组<行,列,值>

存储策略2:使用链式存储三元组<行,列,值>,即**十字链表法**,其中非0数据的节点如下:

全部节点如下

上图中：绿色为向右域right，指向第i行的第一个元素，红色为向下域down，指向第j列的第一个元素。

4.七种排序的算法思想,伪代码或第二轮or第三轮的排序结果

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	In-place	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
希尔排序	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$	In-place	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Out-place	稳定
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	In-place	不稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	In-place	不稳定
计数排序	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	Out-place	稳定
桶排序	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$	Out-place	稳定
基数排序	$O(n \times k)$	$O(n \times k)$	$O(n \times k)$	$O(n + k)$	Out-place	稳定

先放一张图镇楼

1.冒泡排序(交换排序之一)

1.原理:

在无序区间中，将两两相邻的元素进行比较，每一轮比较出最大的一个元素，交换到有序区间。

2.步骤:

1.冒泡排序的主要思想是两两比较，因此先确定要比较多少次，因为前面经过比较只剩最后一个元素的时候，最后一个元素必定已经有序，因此，如果有n个元素，只需要比较n-1次。

2.进行两两比较的过程，从第一个元素开始，把它与它两两相邻的元素比较，把大的换到右边，一直比较，直到把数组里最大的元素换到最右边，因为只需要比较到倒数第二个元素的时候，它与倒数第一个元素比较已经可以换到最右边，因此下标只需要到数组的倒数第二个就行。

3.实现

```
import java.util.Arrays;
public class BubbleSort {
   //实现数组内两元素交换
    public static void swap(int[]array,int i,int j){
        int temp=array[i];
        array[i]=array[j];
        array[j]=temp;
    }
    //冒泡排序
    public static void bubbleSort(int[]array){
        int size=array.length;
      //一开始控制进行两两比较的次数，最后一轮必定有序，所有比较次数为size-1次
        for(int i=0;i<size-1;i++){
            //一开始假设已经有序
            boolean sort=true;
            //两两比较过程
            for(int j=0;j<size-1;j++){
                if(array[j]>array[j+1]){
                    swap(array,j,j+1);
                    //若发生交换则说明已经还未有序
                    sort=false;
                }
            }
            if(sort==true){
                //一轮比较后未发生交换，说明已经有序
                return ;
            }
        }
    }
    public static void main(String[] args) {
        int[]arr={1,3,5,0,8,2,6};
        bubbleSort(arr);
        System.out.println("冒泡排序："+Arrays.toString(arr));
    }
}
```

2.快速排序(交换排序之二)

1.原理

从待排序的区间取一个基准值，比基准值大的放到基准值的右边，比基准值小的放到基准值的左边，对于左右两边，再次充分这样的步骤。

快速排序是冒泡排序的改进算法，它采用了分治的思想，将原问题划分为若干个规模更小的子问题，子问题的依旧与原问题是相似的，递归解决所有的子问题，也就解决了原问题。

2.步骤

1.从待排序区间取一个数作为基准值

2.遍历待排序区间，把所有比基准值小的数放到基准值的左边，比基准值大的放到基准值的右边，这个过程使用专业的术语叫做parttion.

3.对于基准值的左右两边重复以上过程，直到整个序列变得有序。

3.实现

```
/* 对顺序表L做快速排序 */
void QuickSort (SqList *L){
    QSort(L,1,L->length);
}
void QSort(SqList *L,int low,int high){
    int pivot;
    if(low<high){
        pivot = Partition(L,low,high);/* 将L->r[]一分为二,并算出枢轴量pivot */

        QSort(L,low,pivot-1);/* 对低子表做递归 */
        QSort(L,pivot+1,high);/* 对高子表做递归 */
    }
}
int Partiton(SqList *L,int low,int high){
    int pivotKey;
    pivotKey=L->r[low];
    while(low<high){
        while(low<high && L->r[high] >= pivotKey){
```

```
            high--;
        }
        swap(L,low,high);   /* 将比枢轴量小的记录交换到低端 */
        while(low<high && L->[low] <= pivotKey){
            low++
        }
        swap(L,low,high);   /* 将比枢轴量大的记录交换到高端 */
    }
    return low;             /* 返回枢轴量所在位置 */
}
```

3.直接选择排序(选择排序之一)

1.原理：

每一次从无序区间选出最大（或者最小）的元素，放在有序区间的最后（或者无序区间的最前），直到所有的元素有序。

2.步骤：（此步骤针对无序区间在前，有序区间在后）

1.找到到无序区间的最大值元素的下标

2.将无序区间最大值元素的下标交换到有序区间的最后一个

3.重复此过程，直到数组有序

3.实现

```
import java.util.Arrays;
public class SelectSort {
    private static void swap(int[]arr,int i,int j){

        int temp=arr[i];
        arr[i]=arr[j];
        arr[j]=temp;
    }
   //直接选择排序（无序区间在前，有序区间在后的写法）
    public static void selectSort(int[]arr,int size) {
        //外层需要选择size-1次
```

```
        for (int i = 0; i < size - 1; i++) {
            //无序区间 [0,size-i)

            //有序区间 [size-i,size)
            //要找到一个最大值下标，可以先假设最大值下标，再拿其他与它比较
            int maxIdx = 0;
            //遍历整个无序区间
            int j=0;//已经取了第一个元素为最大
            //遍历时从第二个元素开始遍历
            for( j=1;j<size-i;j++){
                if(arr[maxIdx]<arr[j]){
                    //找到无序区间最大元素的下标
                   maxIdx=j;

                }
            }
           //每一次选择到最大下标之后 将它交换到无序区间的最后一个
            //无序区间的最后一个下标 size-i-1
            swap(arr,maxIdx,size-i-1);
        }
    }
    //直接选择排序（有序区间在前，无序区间在后）
    public static void selectSort2(int[]arr,int size){
        for(int i=0;i<size-1;i++){
            //有序区间 [0,i)
            //无序区间 [i,size)
            int minIdx=i;
            int j=i;
            for( j=i+1;j<size;j++){
                if(arr[minIdx]>arr[j]){
                    minIdx=j;
                }
            }
            //找到无序区间的最小值以后，交换到无序区间的第一个元素
            swap(arr,minIdx,i);
        }

    }
```

```
    //直接选择排序（左边为有序区间，中间为无序区间，右边为无序区间）
    //每次找出最大与最小，最大往右边有序区间移，最小往左边有序区间移

    public static void selectSort3(int[]arr,int size){



    }
//简单测试：
    public static void main(String[] args) {
        int[]arr={9,8,5,9,2,3,4};
        selectSort2(arr, arr.length);
        System.out.println(Arrays.toString(arr));
    }

}
```

4.堆排序(选择排序之二)

1.原理：

堆排序也是选择排序中的一种，找到无序区间中的最大值（或者最小值），将它交换到有序区间的最后（或者无序区间的最前）。与直接选择排序最大的不同在于，它不在使用遍历的方式来寻找无序区间的最大值（或最小值），而是通过**创建堆**的方式来创建最大值（或者最小值）。

将待排序序列构造称成一个大顶堆,此时整个序列最大值就是堆顶根节点,将其移走(其实是于堆数组末位元素交换,此时末位元素到达根节点位置),然后将剩余的序列重新构造为大顶堆.如此反复执行便可

2.步骤：

1.创建堆，要创建堆，先实现向下调整。所谓向下调整,便是遍历所有的非叶子节点,从下往上,从右到左将每个非叶子节点作为根节点,将其和其子树调整为大顶堆

2.创建堆以后，堆顶元素就是最大值，找到了最大值就将它交换到最后，放到无序区间的最后

3.放到无序区间最后以后，再从堆顶进行向下调整，调整长度减掉有序区间的长度

3.实现

先来看主方法代码:

```
/* 对顺序表L进行堆排序 */
void HeapSort(SqList *L){
    int i;
    for(i = L->length/2;i>0;i--){
        HeapAdjust(L,i,L_>length);
    }

    for(i=L->length;i>1;i--){
        swap(L,1,i);/* 将堆顶记录和当前未经排序子序列的最后一个记录交换 */
        HeapAdjust(L,1,i-1);/* L->r[1...i-1]重新调整为大顶堆 */
    }
}
```

再来看看关键函数HeapAdjust(堆调整)函数是如何实现的:

```
/* 已知L->r[s..m]中记录的关键字除L->r[s]之外均满足堆的定义 */
/* 本函数调整L->r[s]的关键字,使L->r[s....m]成为一个大顶堆 */
void HeapAdjust(SqList *L,int s,int m){
    int temp,j;
    temp = L->r[s];
    for(j = 2*s;j<=m;j*=2){ /* 沿关键字较大的孩子节点向下筛选 */
        if(j<m && L->r[j] < L->[j+1]){
            ++j;
        }
        if(temp >= L->r[j]){
            break;          /* rc应插入在位置s上 */
        }
        L->r[s]=L->r[j];
        s=j;
    }
    L->r[s]=temp;           /* 插入 */
}
```

总之是比较目标节点和其孩子节点,谁最大谁做根.

5.直接插入排序(插入排序之一)

1.原理：

每次在无序区间选择无序区间的第一个元素，在有序区间找到合理的位置插入。可以将它想象成平时打牌我们对于扑克牌的排序。

2.步骤：

1.遍历整个无序区间，循环选择无序区间的第一个元素

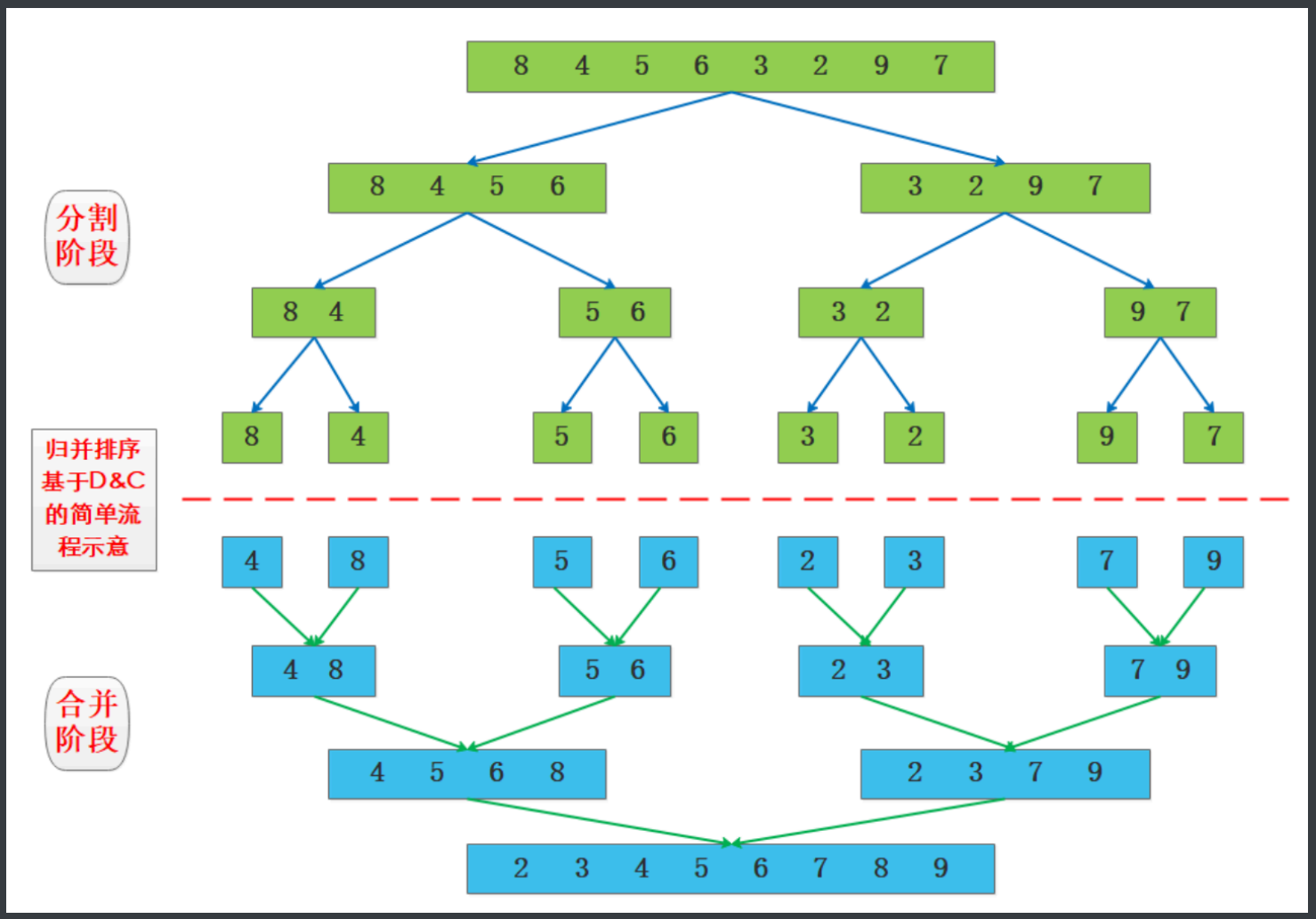
2.在有序区间找到合适的位置，进行插入即可。（在插入时，要提前把不合适的位置往后搬）

3.实现:

```
/* 对顺序表L做直接插入排序 */
void InsertSort(SqList *L)
{
    int i,j;
    for(i=2;i<=L->length;i++)
    {
        if(L->r[i] < L->r[i-1])     /* 需要将L->r[i]插入有序子表 */
        {
            L->r[0]=L->r[i];        /* 设置哨兵 */
            for(j=i-1;L->r[j] > L->r[0];j--)
            {
                L->r[j+1]=L->r[j];  /* 记录后移 */
            }
            L->r[j+1]=L->r[0]       /* 插入到正确位置 */
        }
    }
}
```

其中r[0]是空位,初始可以设置为0,作为整体移动时用于比较大小的哨兵

6.归并排序



1.原理:

归并排序的原理是原序列先分割成一个一个的小的子序列，先使每个子序列有序，再将子序列合并，得到一个完整的有序序列，这就是归并排序。

2.步骤（我这里采用二路归并）:

1.先将原来的无序序列分割成若干个子序列，当分割到一个序列里只有一个元素时说明分割完毕。

2.再将分割后的子序列不断归并，归并的原理是合并两个有序的子序列，一直归并，直到归并得到一个完整的序列。

3.实现

先看主程序:

```
void Mergesort(SqList *L){
    MSort(L->r,L->r,1,L->length);
}
```

主程序就一行代码,是对核心函数的一层包装,.因为核心函数使用了递归调用.

再看核心函数MSort的实现:

```
/* 将SR[s..t]归并排序为TR1[s..t] */
void MSort(int SR[],int TR1[],int s,int t){
    int m;
    int TR2[MAXSIZE+1];
    if(s==t)
        TR1[s]=SR[s];
    else
    {
        m = (s+t)/2;/* 将SR[s...t]平分为SR[s..m]和SR[m+1..t] */
        MSort(SR,TR2,s,m);/* 递归将SR[s..m]归并为有序TR2[s..m] */
        MSort(SR,TR2,m+1,t);/* 递归将SR[m+1..t]归并为有序TR2[m+1..t] */
        Merge(TR2,TR1,s,m,t);/* 将TR2[s..m]和TR2[m+1..t]归并到TR1[] */
    }
}
```

现在来看看Merge函数的代码实现

```
/* 将有序的SR[i..m]和SR[m+1..n]归并为有序的TR[i..n] */
void Merge(int SR[],int TR[],int i,int m,int n){
    int j,k,l;
    for(j = m+1,k=i;i<=m && j<=n;k++){  /* 将SR中记录由小到大归并入TR */
        if(SR[i]<SR[j]){
            TR[k]=SR[i++];
        }else{
            TR[k]=SR[j++];
        }
        if(i<=m){
            for(l=0;l<=m-i;l++){
                TR[k+1]=SR[i+1];         /* 将剩余的SR[i..m]复制到TR */
```

```
                }
                if(j<=n){
                    for(l=0;l<=n-j;l++){
                        TR[k+1]=SR[j+1];    /* 将剩余的SR[j..n]复制到TR */
                    }
                }
            }
        }
    }
```

7.桶/箱排序

桶排序(或称箱排序)并不是一个具体的排序，而是一个逻辑概念。

之所以叫桶，是因为他会根据数据状况设计出一个容器，里面每个index将相当于一个桶。在遍历数据的时候将根据划分将数据一次放入每个桶中，遍历结束后将桶依次倒出。在每个桶内部，数据会被处理成有序结构。具体操作可以参考记数排序。

核心思想就是**大问题化小**,按照一定规则将待排序数据放入数个桶中,并用其他排序方法分别对每个桶的数据进行排序,最后按照一定要求输出

- 桶排序的特点:
  - 非基于比较的排序，与被排序的样本的实际数据状况很有关系。
    并不能作为一个通解被应用在普遍场景下，所以实际中并不经常使用。
  - 时间复杂度O(N)，额外空间复杂度O(N)
  - 稳定的排序

1.计数排序

为何把计数排序放在桶排序之下呢,其实计数排序的开辟新空间做映射的思想与桶排序属于同一类思想,因此作为桶排序思想的一种实现在此出现.

1.1算法思想

1. 计算开辟空间:开辟一块可以容纳待排序数据的空间,如数据 [4,4,6,8,9,3,3,0] ,最大为9,最小为0,因此我们开辟十个空间用于容纳这一堆数据.但是并不是必须从0开始,而是恰好容纳即可,因此只需要(Max-Min+1)块空间.
2. 统计数据**出现次数**:依旧是上述例子