2.2.1

> 从顺序表中删除具有最小值的元素（假设唯一）并由函数返回被删元素的值。空出的位置由最后一个元素填补，若顺序表为空，则显示出错信息并退出运行。

```
bool Delete_MinElem(SqList &L,ElemType &e)
{
    if(/*Empty(L)==1*/L.length==0)//空表显示出错信息,并退出运行
    {
        cout << "空表,无法删除!" << endl;
        return false;
    }

    int min_pos=0;//寻找最小值的位置
    for(int i=1;i<L.length;i++)
        if(L.data[i]<L.data[min_pos])
            min_pos=i;

    e=L.data[min_pos];//引用传回最小值
    L.data[min_pos]=L.data[L.length-1];//更改最小位置的值为最后一个元素

    L.length--;//注意length需要减掉

    return true;
}

```

2.2.2

> 设计一个高效算法，将顺序表 L的所有元素逆置，要求算法的空间复杂度为 $O(1)$.
>
> - 第一种,直接交换第 $i(0 \leq i < length/2)$个元素和第 $length-1-i$个元素;
> - 第二种,类似于**双指针**的思想.

```
void ListReverse(SqList &L)
{
    //第一种
    for(int i=0;i<L.length/2;i++)
    {
        //swap(L.data[i],L.data[L.length-1-i]);
        ElemType temp=L.data[i];
        L.data[i]=L.data[L.length-1-i];
        L.data[L.length-1-i]=temp;
    }
    //第二种
    /*
    int l=0,r=L.length-1;
    while(l<r)
    {
        swap(L.data[l],L.data[r]);
```

```
        l++,r--;
    }
    */
}
```

2.2.3

> 对长度为 n的顺序表 L，编写一个时间复杂度为 $O(n)$、空间复杂度为 $O(1)$的算法，该算法删除线性表中所有值为 x的数据元素.

- **方法一**: 用 cnt记录不为 x的元素的个数,扫描时将不等于 x的元素移到下标为 cnt的位置,并更新 cnt的值,扫描结束后修改 L的长度(cnt).

```
void Delete_Elem(SqList &L,ElemType e)
{
    int cnt=0;

    for(int i=0;i<L.length;i++)
        if(L.data[i]!=e)
            L.data[cnt++]=L.data[i];
    L.length=cnt;
}
```

- **方法二**: 用 cnt记录等于 x的元素的个数, 扫描时,遇到 x更新 cnt,反之遇到不等于 x的元素应前移 cnt位,扫描结束后修改 L的长度($length-cnt$).

```
void Delete_Elem(SqList &L,ElemType e)
{
    int cnt=0;

    for(int i=0;i<L.length;i++)
    {
        if(L.data[i]==e)
            cnt++;
        else L.data[i-cnt]=L.data[i];
    }
    L.length-=cnt;
}
```

2.2.4

> 从**有序顺序表**中删除其值在给定值 s与 t之间（要求 $s<t$)的所有元素，若 s或 t不合理或顺序表为空，则显示出错信息并退出运行。

```
//不考虑其有序,参考2.2.3
bool Delete_Elem(SqList &L,ElemType s,ElemType t)
{
    if(/*Empty(L)==1*/L.length==0)//空表显示出错信息,并退出运行
    {
        cout << "空表,无法删除!" << endl;
```

```
        return false;
    }
    if(s>=t)
    {
        cout << "左区间应小于右区间" << endl;
        return false;
    }

    int cnt=0;

    for(int i=0;i<L.length;i++)
        if(L.data[i]<s||L.data[i]>t)
            L.data[cnt++]=L.data[i];
    L.length=cnt;

    return true;
}

bool Delete_Elem(SqList &L,ElemType s,ElemType t)
{
    if(/*Empty(L)==1*/L.length==0)//空表显示出错信息,并退出运行
    {
        cout << "空表,无法删除!" << endl;
        return false;
    }
    if(s>=t)
    {
        cout << "左区间应小于右区间" << endl;
        return false;
    }
    int cnt=0;

    for(int i=0;i<L.length;i++)
    {
        if(L.data[i]>=s&&L.data[i]<=t)
            cnt++;
        else L.data[i-cnt]=L.data[i];
    }
    L.length-=cnt;
    return true;
}
```

- **官方做法**:寻找第一个**大于等于** s的位置和**大于** t的位置,将后面剩余元素移到前面来.

```
bool Delete_Elem(SqList &L,ElemType s,ElemType t)
{
    if(/*Empty(L)==1*/L.length==0)//空表显示出错信息,并退出运行
    {
        cout << "空表,无法删除!" << endl;
        return false;
    }
```

```
    if(s>=t)
    {
        cout << "左区间应小于右区间" << endl;
        return false;
    }

    int l=0;
    while(l<L.length&&L.data[l]<s)
        l++;
    // if(l>=L.length)
    //     return false;

    int r=l;
    while(r<L.length&&L.data[r]<=t)
        r++;

    while(r<L.length)
    {
        L.data[l]=L.data[r];
        l++,r++;
    }

    L.length=l;
    return true;
}
```

2.2.5

> 从**顺序表**中删除其值在给定值 s与 t之间（包含 s和 t，要求 $s < t$)的所有元素，若 s或 t不合理或顺序表为空，则显示出错信息并退出运行。

```
//可参考2.2.4
bool Delete_Elem(SqList &L,ElemType s,ElemType t)
{
    if(/*Empty(L)==1*/L.length==0)//空表显示出错信息,并退出运行
    {
        cout << "空表,无法删除!" << endl;
        return false;
    }
    if(s>=t)
    {
        cout << "左区间应小于右区间" << endl;
        return false;
    }

    int cnt=0;

    for(int i=0;i<L.length;i++)
        if(L.data[i]<s||L.data[i]>t)
            L.data[cnt++]=L.data[i];
    L.length=cnt;
```

```
21
22      return true;
23  }
24
25  bool Delete_Elem(SqList &L,ElemType s,ElemType t)
26  {
27      if(/*Empty(L)==1*/L.length==0)//空表显示出错信息,并退出运行
28      {
29          cout << "空表,无法删除!" << endl;
30          return false;
31      }
32      if(s>=t)
33      {
34          cout << "左区间应小于右区间" << endl;
35          return false;
36      }
37      int cnt=0;
38
39      for(int i=0;i<L.length;i++)
40      {
41          if(L.data[i]>=s&&L.data[i]<=t)
42              cnt++;
43          else L.data[i-cnt]=L.data[i];
44      }
45      L.length-=cnt;
46      return true;
47  }
```

2.2.6

> 从**有序顺序表**中删除所有其值重复的元素，使表中所有元素的值均不同。

```
1   bool Delete_Same(SqList &L)
2   {
3       if(L.length==0)
4           return false;
5       int cnt=0;
6
7       for(int i=0;i<L.length;i++)
8       {
9           int j=i;
10          while(j<L.length&&L.data[j]==L.data[i])
11              j++;
12          L.data[cnt++]=L.data[i];
13          i=j-1;
14      }
15
16      L.length=cnt;
17
18      return true;
19  }
```

- **官方做法**

```
bool Delete_Same(SqList &L)
{
    if(L.length==0) return false;
    int i,j; //i存储第一个不相同的元素，j为工作指针
    for(i=0,j=1;j<L.length;j++)
        if(L.data[i]!=L.data[j]) //查找下一个与上个元素值不同的元素
           L.data[++i]=L.data[j];  //找到后就将元素前移
    L.length = i+1; //因为i是从0开始的
    return true;
}

//自我做法
bool Delete_Same(SqList &L)
{
    if(L.length==0) return false;

    int cnt=0;

    L.data[cnt++]=L.data[0];

    for(int i=1;i<L.length;i++)//注意i从1开始枚举
        if(L.data[i]!=L.data[cnt-1])//注意是cnt-1(和官方有部分差异)
        {
            L.data[cnt++]=L.data[i];
            //PrintList(L);
        }
    L.length=cnt;

    return true;
}
```

- **扩展**:假设数均为正数,且将**有序表**改为**无序表**,使用`散列表`(类似于**标记数组**),保证时间复杂度为 $O(n)$;若存在负数,标记数组设为有**偏移量**的标记数组即可

```
#define MaxNumSize 1000010

bool Delete_Same(SqList &L)
{
    if(L.length==0)
        return false;
    int vis[MaxNumSize]={0};
    int cnt=0;

    for(int i=0;i<L.length;i++)
        if(vis[L.data[i]]==0)
        {
            vis[L.data[i]]=1;
            L.data[cnt++]=L.data[i];
        }
```

```
    L.length=cnt;

    return true;
}
```

2.2.7

> 将两个有序顺序表合并为一个新的有序顺序表，并由函数返回结果顺序表。

时间复杂度为 $O(n)$

```
bool MergeList(SeqList A,SeqList B,SeqList &C)
{
    if(A.length+B.length>C.MaxSize)
        return false;

    int i=0,j=0,cnt=0;

    while(i<A.length&&j<B.length)
    {
        if(A.data[i]<=B.data[j])
            C.data[cnt++]=A.data[i++];
        else C.data[cnt++]=B.data[j++];
    }

    while(i<A.length)
        C.data[cnt++]=A.data[i++];

    while(j<B.length)
        C.data[cnt++]=B.data[j++];

    C.length=A.length+B.length;

    return true;
}
```

2.2.8

> 已知在一维数组 $A[m+n]$中依次存放两个线性表($a_1,a_2,a_3,\cdots,a_m$)和($b_1,b_2,b_3,\cdots,b_n$)。试编写一个函数，将数组中两个顺序表的位置互换，即将($b_1,b_2,b_3,\cdots,b_n$)放在($a_1,a_2,a_3,\cdots,a_m$)的前面。

- 前 m个元素,即右移 n个元素;后 n个元素,前移 m个元素.时间复杂度为 $O(n+m)$,空间复杂度为 $O(n+m)$.

```
bool ChangeList(SeqList &L,int m,int n)
{
    if(L.length==0||m<=0||n<=0)
        return false;

    ElemType *ans=(ElemType *)malloc((m+n)*sizeof(ElemType));
```

```
    for(int i=0;i<m;i++)
        ans[i+n]=L.data[i];

    for(int i=0;i<n;i++)
        ans[i]=L.data[i+m];

    for(int i=0;i<m+n;i++)
        L.data[i]=ans[i];

    free(ans);
    return true;
}
```

- 先逆置前 m个元素,再逆置后 n个元素,最后整体逆置,即可得到最终结果,时间复杂度为 $O(n+m)$,空间复杂度为 $O(1)$.

```
void ReverseList(SeqList &L,int start,int length)
{
    for(int i=0;i<length/2;i++)
        swap(L.data[start+i],L.data[start+length-1-i]);
}

bool ChangeList(SeqList &L,int m,int n)////m为左半部分,n为右半部分
{
    if(L.length==0||m<=0||n<=0)
        return false;

    ReverseList(L, 0, m);
    ReverseList(L, m, n);
    ReverseList(L, 0, m+n);
    return true;
}
```

2.2.9

> 线性表$(a_1,a_2,a_3,\cdots,a_n)$中的元素递增有序且按顺序存储于计算机内。要求设计一个算法，完成用最少时间在表中查找数值为 x的元素，若找到，则将其与后继元素位置相交换，若找不到，则将其插入表中并使表中元素仍递增有序.

二分查找时间复杂度为 $O(log_2n)$.

```
bool Binary_Search(SqList &L,ElemType e)
{
    if(L.length==0)
        return false;

    int l=0,r=L.length-1,mid=0;

```

```
    while(l<=r)//相等需判断--二分查找
    {
        mid=(l+r)/2;
        if(L.data[mid]==e)
            break;
        if(L.data[mid]<e)
            l=mid+1;
        else r=mid-1;
    }

    if(L.data[mid]==e)//找到对应元素
    {
        if(mid<L.length-1)
            swap(L.data[mid],L.data[mid+1]);
    }
    else//未找到对应元素
    {
        int i;
        for(i=L.length-1;i>=l;i--)//后面元素后移
            L.data[i+1]=L.data[i];
        L.data[i+1]=e;
    }
    return true;
}
```

2.2.10

> **2010统考真题**：设将 $n(n>1)$个整数存放到一维数组 R中。设计一个在时间和空间两方面都尽可能高效的算法。将 R中保存的序列循环左移 $p(0<p<n)$个位置，即将 R中的数据由 $(X_0,X_1,\cdots,X_{n-1})$变换为 $(X_p,X_{p+1},\cdots,X_{n-1},X_0,X_1,\cdots,X_{p-1})$。要求：
>
> 1)给出算法的基本设计思想。
>
> 2)根据设计思想，采用C或C++或Java语言描述算法，关键之处给出注释。
>
> 3)说明你所设计算法的时间复杂度和空间复杂度。

(1)算法的基本设计思想:可将这个问题视为把数组 ab转换成数组 ba(a代表数组的前 p个元素， b代表数组中余下的 $n-p$个元素)，先将 a逆置得到 $a^{-1}b$，再将 b逆置得到 $a^{-1}b^{-1}$，最后将整个 $a^{-1}b^{-1}$逆置得到 $a^{-1}b^{-1}=ba$。设 $ReverseList$函数执行将数组元素逆置的操作，对 $abcdefgh$向左循环移动 3 ($p=3$) 个位置的过程如下:

- $ReverseList(0,p-1)$得到 $cbadefgh$;
- $ReverseList(p,n-1)$得到 $cbahgfed$;
- $ReverseList(0,n-1)$得到 $defghabc$;

注: $ReverseList$ 中，两个参数分别表示**数组中待转换元素的始末位置。**给出的代码是**起始位置和长度**

(2)代码如下

```
同2.2.8
void ReverseList(SeqList &L,int start,int length)
```

```
{
    for(int i=0;i<length/2;i++)
        swap(L.data[start+i],L.data[start+length-1-i]);
}

bool ChangeList(SeqList &L,int m,int n)//m为左半部分,n为右半部分
{
    if(L.length==0||m<=0||n<=0)
        return false;

    ReverseList(L, 0, m);
    ReverseList(L, m, n);
    ReverseList(L, 0, m+n);
    return true;
}

int len=11,m=5;
int n=len-m;
ChangeList(L, m, n);//左边部分长度,右边部分长度
```

(3) 每个 $ReverseList$函数的时间复杂度分别为 $O(m/2), O(n/2), O((n+m)/2)$[实际应用中为 $O(p/2), O((n-p)/2), O(n/2)$],总的时间复杂度为 $O(n+m)$[实际应用中为 $O(n)$],空间复杂度为 $O(1)$.

2.2.11

> **2011统考真题**：一个长度为 $L(L \geq 1)$的升序序列 S，处在第 $\lceil L/2 \rceil$个位置的数称为 S的中位数。例如，若序列 $S_1 = (11, 13, 15, 17, 19)$。则 S的中位数是 15，两个序列的中位数是含它们所有元素的升序序列的中位数。例如，若 $S_2 = (2, 4, 6, 8, 20)$，则 S_1和 S_2的中位数是 11。现在有两个等长升序序列 A和 B，试设计一个在时间和空间两方面都尽可能高效的算法，找出两个序列 A和 B的中位数。要求:
>
> 1)给出算法的基本设计思想。
>
> 2)根据设计思想，采用`C或C++或Java语言`描述算法，关键之处给出注释。
>
> 3)说明你所设计算法的时间复杂度和空间复杂度。

- **方法一:归并排序**的思想,设 A和 B的长度均为 n,**合并两个数组**放入 S,找中位数 $S[n-1]$(注意下标从 0 开始),时间复杂度为 $O(n)$,空间复杂度为 $O(n)$.

```
bool Search_Median(SeqList L1,SeqList L2,int n,ElemType &e)//n为每个线性表的长度
{
    if(L1.length==0||L2.length==0||L1.length!=L2.length)
        return false;

    ElemType *ans=(ElemType *)malloc(2*n*sizeof(ElemType));

    int i=0,j=0,k=0;

    while(i<n&&j<n)
    {
```

```
        if(L1.data[i]<=L2.data[j])
            ans[k++]=L1.data[i++];
        else ans[k++]=L2.data[j++];
    }
    while(i<n)
        ans[k++]=L1.data[i++];
    while(j<n)
        ans[k++]=L2.data[j++];

    e=ans[n-1];

    return true;
}
```

- **方法二**:可发现方法一只需要扫 n个(**一半**)就可以结束.时间复杂度为 $O(n)$,空间复杂度为 $O(n)$,虽然只优化了部分空间,并没有降低太多时间复杂度.

```
bool Search_Median(SeqList L1,SeqList L2,int n,ElemType &e)//n为每个线性表的长度
{
    if(L1.length==0||L2.length==0||L1.length!=L2.length)
        return false;

    ElemType *ans=(ElemType *)malloc(n*sizeof(ElemType));

    int i=0,j=0,k=0;

    while(i<n&&j<n&&k<n)
    {
        if(L1.data[i]<=L2.data[j])
            ans[k++]=L1.data[i++];
        else ans[k++]=L2.data[j++];
    }
    while(i<n&&k<n)
        ans[k++]=L1.data[i++];
    while(j<n&&k<n)
        ans[k++]=L2.data[j++];

    e=ans[n-1];

    return true;
}
```

- **方法三**:不用数组存储,直接在**方法二**每次更新它的值即可,时间复杂度为 $O(n)$,空间复杂度为 $O(1)$.

```
bool Search_Median(SeqList L1,SeqList L2,int n,ElemType &e)//n为每个线性表的长度
{
    if(L1.length==0||L2.length==0||L1.length!=L2.length)
        return false;

    int i=0,j=0,k=0;

```

```
    while(i<n&&j<n&&k<n)
    {
        if(L1.data[i]<=L2.data[j])
            e=L1.data[i++];
        else e=L2.data[j++];
        k++;
    }
    while(i<n&&k<n)
        e=L1.data[i++],k++;
    while(j<n&&k<n)
        e=L2.data[j++],k++;

    return true;
}
```

- **方法4**(官方做法):思想类似于**二分**,时间复杂度为 $O(log_2n)$,空间复杂度为 $O(1)$.

分别求两个升序序列 A、 B的中位数，设为 a和 b，求序列 A、 B的中位数过程如下:

- 若 $a=b$，则 a或 b即为所求中位数，算法结束。
- 若 $a<b$，则舍弃序列 A中**较小**的一半，同时舍弃序列 B中**较大**的一半，要求两次舍弃的长度相等。
- 若 $a>b$，则舍弃序列 A中**较大**的一半，同时舍弃序列 B中**较小**的--半，要求两次舍弃的长度相等。

在保留的两个升序序列中，重复上述步骤，直到两个序列中均**只含一个元素时**为止，较小者即为所求的中位数。

```
bool Search_Median(SeqList L1,SeqList L2,int n,ElemType &e)//n为每个线性表的长度
{
    if(L1.length==0||L2.length==0||L1.length!=L2.length)
        return false;

    int l1=0,r1=n-1;//线性表L1的左右边界
    int l2=0,r2=n-1;//线性表L2的左右边界

    while(r1>l1)
    {
        int mid1=(l1+r1)/2;
        int mid2=(l2+r2)/2;
        //cout << l1 << " " << r1 << " " << mid1 << endl;
        //cout << l2 << " " << r2 << " " << mid2 << endl;
        if(L1.data[mid1]==L2.data[mid2])//相等,两者均为中位数
        {
            e=L1.data[mid1];
            return true;
        }
        else if(L1.data[mid1]<L2.data[mid2])//x<y,把小于x和大于y的排除
        {
            if((r1-l1+1)%2==1)//奇数
            {
                l1=mid1;
                r2=mid2;
```

```
            }
            else//偶数
            {
                l1=mid1+1;//当前mid不可能成为中位数
                r2=mid2;
            }
        }
        else//x>y,把大于x和小于y的排除
        {
            if((r1-l1+1)%2==1)//奇数
            {
                r1=mid1;
                l2=mid2;
            }
            else//偶数
            {
                r1=mid1;
                l2=mid2+1;//当前mid不可能成为中位数
            }
        }
    }

    e=min(L1.data[l1],L2.data[l2]);

    return true;
}
```

2.2.12

> **2013统考真题**：已知一个整数序列 $A=(a_0,a_1,\cdots,a_{n-1})$，其中 $0\leq a_i<n\,(0\leq i<n)$。若存在 $a_{p1}=a_{p2}=\cdots=a_{pm}=x$且 $m>n/2\,(0\leq p_k<n,1\leq k\leq m)$，则称 x为 A的主元素。例如 $A=(0,5,5,3,5,7,5,5)$，则 5为主元素;又如 $A=(0,5,5,3,5,1,5,7)$、则 A中没有主元素。假设 A中的 n个元素保存在一个一维数组中，请设计一个尽可能高效的算法，找出 A的主元素。若存在主元素，则输出该元素;否则输出 -1.要求:
>
> 1)给出算法的基本设计思想。
>
> 2)根据设计思想，采用`C或C++或Java语言`描述算法，关键之处给出注释。
>
> 3)说明你所设计算法的时间复杂度和空间复杂度。

- **方法一**:维护一个**计数数组**进行保存每个数的个数,若存在某个数的个数大于 $n/2$,则返回该数,否则返回 -1.时间复杂度为 $O(n)$,空间复杂度为 $O(n)$.

```
int Search_Mode(SeqList L,int n)
{
    int *cnt=(int *)malloc(n*sizeof(int));

    for(int i=0;i<n;i++)//初始化计数数组为0
        cnt[i]=0;

    for(int i=0;i<n;i++)
```

```
        cnt[L.data[i]]++;

    int ans=-1;
    for(int i=0;i<n;i++)
        if(cnt[i]>n/2)
        {
            ans=i;
            break;
        }

    free(cnt);
    return ans;
}
```

- **方法二**:

选取候选的主元素。依次扫描所给数组中的每个整数，将第一个遇到的整数 Num保存到 c中，记录 Num 的出现次数为 1;若遇到的下一个整数仍等于 Num，则计数加 1，否则计数减 1:当计数减到 0时，将遇到的下一个整数保存到 c中，计数重新记为 1，开始新一轮计数，即从当前位置开始重复上述过程，直到扫描完全部数组元素。

判断 c中元素是否是真正的主元素。再次扫描该数组，统计 c中元素出现的次数，若大于 $n/2$，则为主元素;否则，序列中不存在主元素。

时间复杂度为 $O(n)$,空间复杂度为 $O(1)$.

```
int Search_Mode(SeqList L,int n)
{
    int temp=L.data[0];//存储候选主元素,并设置L.data[0]为候选主元素
    int cnt=1;//存储个数
    for(int i=1;i<n;i++)//查找候选主元素
    {
        if(L.data[i]==temp)
            cnt++;//对A中的候选主元素计数
        else
        {
            if(cnt>0)//处理不是候选主元素的情况
                cnt--;
            else//更新候选主元素,重新计数
            {
                temp=L.data[i];
                cnt=1;
            }
        }
    }

    int tot=0;//统计候选主元素个数
    for(int i=0;i<n;i++)
        if(L.data[i]==temp)
            tot++;

    if(tot>n/2)
```

```
        return temp;
    else return -1;
}
```

2.2.13

> **2018统考真题**：给定一个含 $n(n \geq 1)$个整数的数组，请设计一个在时间上尽可能高效的算法，找出数组中未出现的最小正整数。例如，数组 $-5,3,2,3$中未出现的最小正整数是 1;数组 $1,2,3$中未出现的最小正整数是 4。要求:
>
> 1)给出算法的基本设计思想。
>
> 2)根据设计思想，采用`C或C++或Java语言`描述算法，关键之处给出注释。
>
> 3)说明你所设计算法的时间复杂度和空间复杂度。

用**标记数组**记录对应是否在数组中存在,最后从头遍历正整数即可,时间复杂度为 $O(n)$,空间复杂度为 $O(n)$.

要求在时间上尽可能高效，因此采用空间换时间的办法。分配一个用于标记的数组 $B[n]$,用来记录 A中是否出现了 $1 \sim n$中的正整数， $B[0]$对应正整数 1， $B[n-1]$对应正整数 n,初始化 B中全部为 0。由于 A中含有 n个整数，因此可能返回的值是 $1 \sim n+1$，当 A中 n个数恰好为 $1 \sim n$时返回 $n+1$。当数组 A中出现了小于等于 0或大于 n的值时,会导致 $1 \sim n$中出现空余位置，返回结果必然在 $1 \sim n$中，因此对于 A中出现了小于等于 0或大于 n的值，可以不采取任何操作。(数组标记代码采用的是从 $1 \sim n$开始标记的,而不是从 $0 \sim n-1$开始的,此处单纯复制官方的思路)

```
//思想同2.2.12的方法一
int Search_Min_Positive_Integer(SeqList L,int n)
{
    int *flag=(int *)malloc((n+1)*sizeof(int));

    for(int i=1;i<=n;i++)
        flag[i]=0;

    for(int i=0;i<n;i++)
        if(L.data[i]>=1&&L.data[i]<=n)
            flag[L.data[i]]=1;
    int ans=n+1;
    for(int i=1;i<=n;i++)
        if(flag[i]==0)
        {
            ans=i;
            break;
        }

    free(flag);
    return ans;
}
```

2.2.14

> **2020统考真题**：定义三元组 (a,b,c)(a、 b、 c均为正数）的距离 $D=|a-b|+|b-c|+|c-a|$ 。给定 3个非空整数集合 S_1、 S_2和 S_3,按升序分别存储在 3个数组中。请设计一个尽可能高效的算法，计算并输出所有可能的三元组 (a,b,c)($a\in S_1,b\in S_2,C\in S_3$）中的最小距离。例如 $S_1=\{-1,0,9\}$, $S_2=\{-25,-10,10,11\}$, $S_3=\{2,9,17,30,41\}$，则最小距离为 2，相应的三元组为 $(9,10,9)$。要求:
>
> 1)给出算法的基本设计思想。
>
> 2)根据设计思想，采用`C或C++或Java语言`描述算法，关键之处给出注释。
>
> 3)说明你所设计算法的时间复杂度和空间复杂度。

- **方法一**:循环枚举每个数组的元素,构成三元组,找出最小值,时间复杂度为 $O(n^3)$,空间复杂度为 $O(1)$.

```
int calc_dist(int x,int y,int z)
{
    return abs(x-y)+abs(y-z)+abs(z-x);
}

int Calc_Min_Dist(SeqList A,SeqList B,SeqList C)
{
    if(A.length==0||B.length==0||C.length==0)
        return -1;

    int ans=calc_dist(A.data[0], B.data[0], C.data[0]);//初始值置为某一个值

    for(int i=0;i<A.length;i++)
        for(int j=0;j<B.length;j++)
            for(int k=0;k<C.length;k++)
                ans=min(ans,calc_dist(A.data[i], B.data[j], C.data[k]));
    return ans;
}
```

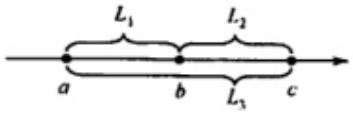
- **方法二**: **最小值的下标右移**会影响到最终的结果,时间复杂度为 $O(n)$,空间复杂度为 $O(1)$.

[讲解链接]

分析。由 $D=|a-b|+|b-c|+|c-a|\geq 0$ 有如下结论。

① 当 $a=b=c$ 时，距离最小。

② 其余情况。不失一般性，假设 $a\leq b\leq c$，观察下面的数轴：



$L_1=|a-b|$

$L_2=|b-c|$

$L_3=|c-a|$

$D=|a-b|+|b-c|+|c-a|=L_1+L_2+L_3=2L_3$

由 D 的表达式可知，事实上决定 D 大小的关键是 a 和 c 之间的距离，于是问题就可以简化为每次固定 c 找一个 a，使得 $L_3=|c-a|$最小。

1）算法的基本设计思想

① 使用 D_{min} 记录所有已处理的三元组的最小距离，初值为一个足够大的整数。

② 集合 S_1、S_2 和 S_3 分别保存在数组 A、B、C 中。数组的下标变量 $i=j=k=0$，当 $i<|S_1|$、$j<|S_2|$且 $k<|S_3|$时（$|S|$表示集合 S 中的元素个数），循环执行下面的 a）～c）。

a）计算(A[i], B[j], C[k])的距离 D；　（计算 D）

b）若 $D<D_{min}$，则 $D_{min}=D$；　（更新 D）

c）将 A[i]、B[j]、C[k]中的最小值的下标+1；　（对照分析：最小值为 a，最大值为 c，这里 c 不变而更新 a，试图寻找更小的距离 D）

③ 输出 D_{min}，结束。

```
int calc_dist(int x,int y,int z)
{
    return abs(x-y)+abs(y-z)+abs(z-x);
}

bool check_min(int a,int b,int c)
{
    return (a<=b&&a<=c);
}

int Calc_Min_Dist(SeqList A,SeqList B,SeqList C)
{
    if(A.length==0||B.length==0||C.length==0)
        return -1;
    int ans=calc_dist(A.data[0], B.data[0], C.data[0]);//初始值置为某一个值

    int i=0,j=0,k=0;

    while(i<A.length&&j<B.length&&k<C.length)
    {
        ans=min(ans,calc_dist(A.data[i], B.data[j], C.data[k]));

        if(check_min(A.data[i], B.data[j], C.data[k]))
            i++;
        else if(check_min(B.data[j], A.data[i], C.data[k]))
            j++;
        else k++;
```

```
    }
    return ans;
}
```