1. 顺序表

1.1. 设计一个算法，使得整个顺序表有序 (重要)

设顺序表用数组 A[]表示，表中元素存储在数组下标 1~m+n 的范围内，**前 m 个元素递增有序，后 n 个元素递增有序**，设计一个算法，使得**整个顺序表有序**。(P47)

1. 给出算法的基 设计思想。
2. 根据设计思想，采用C或C++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。

> 算法思路:
> 取表 R 中的第一个元素 A[m+1]存入辅助变量 temp 中，让 temp 逐个与 A[m],A[m-1],...,A[1]进行比较，当 temp<Aj时，将 A[j]后移一位; 否则将 temp 存入 A[j+1]中。

```
void resort(int A[], int m, int n){
    int i,j,temp;
    for(i=m; i<m+n; i++){
        temp = A[i];

        // 比 temp 大的数后移
        for(j=i-1; j>0 && A[j]>temp; j--){
            A[j+1] = A[j];
        }

        // A[j] 是第一个比 temp 小的数
```

```
        A[j+1] = temp;
    }
}
```

1.2. 设计一个算法，将顺序表中的所有元素逆置

设计一个算法，将顺序表中的所有元素逆置。

> 思路:
> 两个变量 i，j 指示顺序表的第一个元素和最后一个元素，交换 i，j 所指元素，然后 i 向后移一个位置，j 向前移动一个位置，如此循环， 直到 i 与 j 相遇时结束，此时顺序表 L 中的元素已经逆置。

```
void reverse02(SqList &A){
    int i, j, tmp;
    // 交换两端的元素
    for(i=0, j=A.length-1; i<j; i++, j--){
        // 交换数组元素
        tmp = A.data[i];
        A.data[i] = A.data[j];
        A.data[j] = tmp;
    }
}
```

1.3. 设计一个算法删除顺序表 L 中 i 到 j之间的所有元素

设计一个算法，从一给定的顺序表 L 中删除下标 i 到 j(i≤j，包括 i，j)之间的所有元素，
假定 i,j 都是合法的。

> 思路:
> 从第 j+1 个元素开始到最后一个元素为止，用这之间的每个元素去覆盖从这个元素开始往前数第 j-i+1 个元素

```
void delRange(SqList &sqList, int i, int j){
    int m;
    int len = j-i+1;
    for(m=j+1; m<sqList.length; m++){
        sqList.data[m-len] = sqList.data[m];
    }
    sqList.length -= len;
}
```

1.4. 设计一个算法，将 L 中所有小于表头元素的... (重要)

有一个顺序表 L，其元素为整型数据，设计一个算法，将 L 中所有小于表头元素的整数放在前半部分，大于的整数放在后半部分，数组从下表 1 开始存储。

> 思路:
> 1) i 和 j 是轮流移动的，即当 i 找到比 2 大的元素时，将 i 所指元素放入 j 所指位置，i停在当前位置不动，j 开始移动。j 找到比 2 小的元素，将 j 所指元素放在 i 所指位置，j 停在当前位置不动，i 开始移动，如此交替直到 i==j。

> 2) 每次元素覆盖(比如执行L.data[i]=L.data[j];)不会造成元素丢失，因为在这之前被覆盖位置的元素已经存入其他位置

```
void move(SqList &L){
    int i=0, j=L.length-1;
    int tmp = L.data[i];
    while(i<j){
        // 从右向左 找到第一个比表头元素小的数组下标
        while(i<j && L.data[j]>tmp) j--;
        if(i<j){
            L.data[i] = L.data[j];
            i++;
        }

        // 从左向右 找到第一个比表头元素大的数组下标
        while(i<j && L.data[i]<tmp) i++;
        if(i<j){
            L.data[j] = L.data[i];
            j--;
        }
    }
    L.data[i]=tmp;
}
```

1.5 在一个有 n 个整数的顺序表 A 中找出最大和最小值

试编写一个函数，用不多于3n/2的平均比较次数，在一个有 n 个整数的顺序表 A 中找出最大和最小值

> 思路:
> 最好情况: $2(n-1)$
> 最坏情况: $n-1$
> 期望: $E=\frac{2(n-1)+(n-1)}{2}=\frac{3(n-1)}{2}<\frac{3n}{2}$

```
void findMaxMin(int A[], int n, int &max, int &min){
    max = A[0];
    min = A[0];
    for(int i=1; i<n; i++){
        if(A[i] > max){
            max = A[i];
        }else if(A[i] < min){
            min = A[i];
        }
    }
}
```

1.6. 试写一个比较A、B大小的算法

设A＝（a1，…，am）和B＝（b1，…，bn）均为顺序表，A’和B’分别为A和B中除去最大共同前缀后的子表（例如，A＝（x，y，y，z，x，z），B＝（x，y，y，z，y，x，x，z），则两者中最大的共同前缀为（x，y，y，z），在两表中除去最大共同前缀后的子表分别为A’＝（x，z）和B’＝（y，x，x，z））。若A’＝B’＝空表，则A＝B；若A’＝空表，而B’≠空表，或者两者均不为空，且A’的首元小于B’的首元，则AB。试写一个

比较A、B大小的算法。（请注意：在算法中，不要破坏原表A和B，并且也不一定先求得A'和B'才进行比较）

```
char* compare02(char A[], int m, char B[], int n){
    int i=0;
    // 找到A、B不同元素的下标
    while(A[i]==B[i] && i<m && i<n) i++;

    if(i>=m && i>=n){
        return  (char*)"A == B";
    }else if ((i<m && i>=n) || (i<m && i<n && A[i]>B[i])){
        return  (char*)"A > B";
    }else{
        return  (char*)"A < B";
    }
}
```

1.7. 将$(x_0,\ldots,x_{n-1})$变换为$(x_p,\ldots,x_{n-1},x_0,\ldots,x_{p-1})$ (重要)

设将$n(n>1)$个整数存放到一维数组R中，试设计一个在时间和空间两方面尽可能有效的算法，将R中保有的序列循环左移P（0＜P＜n）个位置，即将R中的数据由
$(x_0,\ldots,x_{n-1})$变换为$(x_p,\ldots,x_{n-1},x_0,\ldots,x_{p-1})$
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语言表述算法，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思路:要实现R中序列循环左移P个位置，只需先将R中前P个元素逆置，再将剩下的元素逆置，最后将R中所有的元素再整体做一次逆置操作即可

```
void reverse(int R[], int start, int end){ // 逆置
    if(start > end){
        return;
    }
    int i=start, j=end, tmp;
    while(i<j){
        tmp = R[i];
        R[i] = R[j];
        R[j] = tmp;
        i++;
        j--;
    }
}

void movep(int R[],  int n, int p){
    if(p<0 || p>=n){
        return;
    }
    reverse(R, 0, p-1);
    reverse(R, p, n-1);
    reverse(R, 0, n-1);
}
```

1.8. 请设计一个尽可能高效的算法，找出A的主元素 (重要)

已知一个整数序列A=(a0，a1，...,an-1)，其中0≤ai<n(0≤in/2(0≤pk<n，1≤k≤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。

```
int mostEle(int A[], int n){
    int main=A[0], count=1, i;
    // 找出候选主元素
    for(i=1; i<n; i++){
        if(A[i] == main){
            count ++;
        }else{
            if(count > 0){
                count--;
            }else{
                count=1;
                main = A[i];
            }
        }
    }
    // 确认真正的主元素
    if(count > 0){
        count = 0;
        for(i=0; i<n; i++){
            if(A[i] == main){
                count++;
            }
        }
    }
    if(count > n/2){
        return main;
    }else{
        return -1;
    }
}
```

2. 链表

2.1. 请设计算法以求出两个集合 A 和 B 的差集 A-B (重要)

已知**递增有序**的单链表 A,B(A,B 中元素个数分别为 m,n 且 A,B 都带有头结点) 分别存储了一个集合，请设计算法以求出两个集合 A 和 B 的差集 A-B(即仅由在 A 中出现 而不在 B 中出现的元素所构成的集合)。将差集保存在单链表 A 中，并**保持元素的递增有序性**。 (P47)

1. 给出算法的基 设计思想。
2. 根据设计思想，采用C或C++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。

> 算法思路:
> 设置两个指针 p,q 开始时分别指向 A 和 B 的开始结点，同时遍历链表 A B;
> 1) 若 B < A， 则 q 指针后移;
> 2) 若 A < B， 则 p 指针后移;
> 3) 若 A = B， 则 删除节点。
> 最后 p 与 q 任一指针为 NULL 的时候算法结束。

```
void difference(LNode *&A, LNode *&B){
    if(A==NULL || B==NULL){
        return;
    }

    LNode *p = A->next;
    LNode *pre = A; //pre 为 A 中 p 所指结点的前驱结点的指针。删除必须知道前驱节点的指针!!!!!
    LNode *q = B->next;

    while(p!=NULL && q!=NULL){
        if(p->data > q->data){
            q = q->next;
        } else if(p->data < q->data){
            pre = p;
            p = p->next;
        } else{
            pre->next = p->next;
            free(p);
            p = pre->next;
        }
    }
}
```

2.2. 有一个递增非空单链表，设计一个算法删除值域重复的结点 (较重要)

有一个递增非空单链表，设计一个算法删除值域重复的结点。
比如 {1,1,2,3,3,3,4,4,7,7,7,9,9,9} 经过删除后变成{1,2,3,4,7,9}。

> 思路:
> 定义指针 p 指向起始结点。将 p 所指当前结点值域和其直接后继结点值域比较。
> 如果当前结点值域等于后继结点值域，删除后继结点;否则 p 指向后继结点，重复以上过程，直到 p 的后继结点为空。

```
void unique(LNode *&L){
    LNode *pre,*p,*tmp;
    pre = L->next;
    p = pre->next;
    while(p!=NULL){
        if(pre->data == p->data){
            tmp = p;
            pre->next = p->next;
            free(tmp);
            p = p->next;
```

```
        }else{
            pre = p;
            p = p->next;
        }
    }
}
```

2.3. 设计一个算法删除单链表L中的一个最小值结点 (较重要)

设计一个算法删除单链表 L(有头结点)中的一个最小值结点。

> 思路:
> 用 p 从头至尾扫描链表，pre 指向p 结点的前驱，用 minp 保存值最小的结点指针，minpre 指向 minp 的前驱。一边扫描，一边比较，将最小值结点放到 minp 中。

```
void delMin(LNode *&L){
    if(L==NULL){
        return;
    }
    LNode *pre, *p, *minPre, *minP;
    pre = L;
    p = L->next;
    minPre = pre;
    minP = p;

    while(p!=NULL){
        if(p->data < minP->data){
            minP = p;
            minPre = pre;
        }
        pre = p;
        p = p->next;
    }
    //删除*minp结点。
    minPre->next = minP->next;
    free(minP);
}
```

2.4. 设计一个算法将其逆置，要求不能建立新结点

有一个线性表，采用带头结点的单链表 L 来存储。设计一个算法将其逆置。要求不能建立新结点，只能通过表中已有结点的重新组合来完成。

> 思路:
> 在前边讲过的算法基础中，提到过关与逆序的问题，那就是链表建立的头插法。

```
void reverseLNode(LNode *&L){
    LNode *p, *q;
    p = L->next;
    L->next = NULL;

    while(p!=NULL){
        q = p->next;
```

```
        p->next = L->next;
        L->next = p;
        p=q;
    }
}
```

2.5. 设计一个算法将一个头结点为 A 的单链表分解成两个单链表

设计一个算法将一个头结点为 A 的单链表(其数据域为整数)分解成两个单链表 A 和 B，使得 A 链表只含有原来链表中data域为奇数的结点，而B链表只含有原链表中data域为偶数的结点，且保持原来相对顺序。

> 思路:
> 用指针p从头至尾扫描A链表，当发现结点data域为偶数的结点则取下，插入链表B中。

```
void split(LNode *&A, LNode *&B){
    LNode *pre, *p, *tmp, *r;

    pre = A;
    p = pre->next;

    B = (LNode *)malloc(sizeof(LNode *));
    B->next = NULL;
    r = B;

    while(p!=NULL){
        if(p->data %2 !=0) {
            // 奇数节点pre和p向右移动
            pre = p;
            p = p->next;
        }else{
            // 将偶数节点断开，p 向右移动
            pre->next = p->next;
            tmp = p;
            p = p->next;

            // 尾插法，增加偶数节点
            r->next = tmp;
            tmp->next = NULL;
            r = tmp;
        }
    }
}
```

2.6. 写一个函数，逆序打印单链表中的数据 (重要)

写一个函数，逆序打印单链表中的数据，假设指针 L 指向了单链表的开始结点。

> 思路:
> 递归调用即可逆序打印

```
void reprint(LNode *L){
    LNode *p = L->next;

```

```
    // 递归出口
    if(p==NULL){
        return;
    }

    // 递归
    reprint(p);
    printf((char*) " %d ", p->data);
}
```

2.7. 查找链表中倒数第k个位置上的结点 (重要)

已知一个带有头结点的单链表，该链表只给出了头指针head，在不改变链表的前提下，请设计一个尽可能高效的算法，查找链表中倒数第k个位置上的结点（k为正整数）。若查找成功，算法输出该结点的data值，并返回1；否则，只返回0。

> 思路
> 遍历链表，遍历过程中，当当前节点遍历到第k个时，pk随着每次遍历也向后移动

```
int findK(LNode *&head, int k){
    LNode *p = head->next;
    LNode *pk = head;
    // 当前遍历到第i个
    int i = 1;
    while(p!=NULL){
        // 当当前节点遍历到第k个时，pk随着每次遍历也向后移动
        if(i >= k){
            pk = pk->next;
        }
        p = p->next;
        i++;

        // 标准答案
        // p = p->next;
        // index ++;
        // if(index>k)
        //     pk = pk->next;
    }
    if(head == pk){
        return 0;
    }else{
        printf("%d", pk->data);
        return 1;
    }
}
```