顺序表算法 16个

```
void del_min(sqList &L,int &value){
    //删除顺序表L中最小值元素结点，并通过引用型参数value返回其值
    //若删除成功返回true,否则返回false
    if( L.length==0 )
        return false;
    value = L.data[0];
    int pos = 0;
    for(int i=1;i<L.length;i++){
        if( L.data[i]<value ){
            value=L.data[i];
            pos=i;
        }
    }
    L.data[pos]=L.data[L.length-1];
    L.length--;
    return true;
}

/* 顺序表L所有元素逆置，要求算法空间复杂度为O(1)*/
void Reverse(SqList &L){
    Elemtype temp;  //辅助变量
    for ( i=0;i<L.length;i++ ){
        temp = L.data[i];    //交换L.data[i]与L.data[L.length-i-1]
        L.data[i] = L.data[L.length-i-1];
        L.data[L.length-i-1]=temp;
    }
}
```

```
/* 顺序表长度 n,删除该表中值为x的数据元素,要求：时间复杂度:O(n),空间复杂度O(1) */
/**
 *解法一:
 *用k记录L中不等于x的数据元素个数（即需要保存的元素个数），
 *边扫描L边统计K,并将不等于x的元素向前移动k个位置，最后修改L的长度.
*/
void del_x_1(Sqlist &L,Elemtype x){
    int k=0;
    for(i=0;i<L.length;i++){
        if(L.data[i] != x){
            L.data[k] = L.data[i];
            k++;
        }
    }
    L.length=k;
}
/**
 * 解法二:
 * 用k记录L中等于x的元素，边扫描边统计k，并将不等于x的元素前移k个位置，
 * 最后修改l的长度
*/
void del_x_2(Sqlist &L, Elemtype x){
    int k=0;
    int i=0;
    while(i<L.length){
        if(L.data[i]==x){
            k++;
        }else{
            L.data[i-k] = L.data[i]; //当前元素前移 k 个位置
            i++;
        }
    }
    L.length = L.length-k;
}
```

```
/* 删除 x,  s< x <t */
//同上题算法一
void del_s_t_1( SqList *l, int s, int t ){
    int k=0,i=0;
    while(i<l->length){
        if(l->data[i]>s && l->data[i]<t){    //如果条件加上'=', 则s, t也会被删除
            k++;
        }else{
            l->data[i-k] = l->data[i];
        }
        i++;
    }
    l->length = l->length-k;
}
/* 删除 x ,  s<x<t 或 s<=x<=t*/
// 同上题算法二
void del_s_t_2( SqList *L, int s, int t ){
    int k=0;
    for(int i=0;i<L->length;i++){
        if(L->data[i] <= s || L->data[i] >= t){ //将等号去掉, 则s, t也将被删除
            L->data[k] = L->data[i];
            k++;
        }
    }
    L->length = k;
}
//删除 s<=x<=t
bool Del_s_t2(SqList &L,ElemType s, ElemType t){
    int i,j;
    if(s>=t || L.length==0)
        return false;
    for(i=0;i<L.length && L.data[i]<s;i++);  //寻找大于等于s的第一个元素
    if(i>=L.length)
        return false;
    for(j=i;j<L.length&&L.data[j]<=t;j++);  //寻找值大于t的第一个元素
    for( ; j<L.length; i++, j++ )
        L.data[i] = L.data[j];
    L.length = i;
    return true;
}
```

```
/**
 * 题目描述：有序表删除所有重复的元素，使表中所有元素的值均不同
 * 算法思想：初始时将第一个元素看作是没有重复的顺序表，依次遍历L后序元素，
 * 遇到不同，插入，遇到相同，跳过，继续遍历。最后更新表的长度
*/
bool Del_same(SqList &L){
    if(L.length==0)
        return false;
    int i,j;
    for(i=0,j=1;j<L.length;j++){
        if(L.data[i]!=L.data[j])
            L.data[++i] = L.data[j];
        L.length=i+1;
        return true;
    }
}

/**
 * 题目描述：合并有序顺序表，新表保持有序
*/
bool Merge(SqList A,SqList B,SqList &C){
    if(A.length+B.length>C.MaxSize)
        return false;
    int i=0, j=0, k=0;
    while(i<A.length && j<L.length){
        if(A.data[i]<=B.data[j])
            C.data[k++]=A.data[i++];
        else
            C.data[k++]=B.data[j++];
    }
    while(i<A.length)
        C.data[k++]=A.data[i++];
    while(j<B.length)
        C.data[k++]=B.data[j];
    C.length=k;
    return true;
}
```

```
/**
 * 题目描述:A[m+n]中存放 a1~am,b1~bn,使用算法将其转为b1~bn,a1~am
 * 算法思想:现将整个顺序表逆置为 bn~b1,am~a1
 * 再将A的前n项和后m项分别逆置。
*/
typedef int DataType;
void Reverse ( DataType A[], int left, int right, int arraySize ){
    if( left>=right || arraySize==0 )
        return;
    int min=( left+right )/2;
    for( int i=0; i<mid; i++ ){
        DataType tmp = A[left+i];
        A[left+i] = A[right-i];
        A[right-i] = tmp;
    }
}

void Exchange(DataType A[], int m, int n, int arrarySize){
    Reverse(A, 0,m+n-1,arraySize);
    Reverse(A,0,n-1,arraySize);
    Reverse(A,n,m+n-1,arraySize);
}
```

```
/**
 * 题目描述：递增顺序表，查找x，若找到，与其后继元素交换位置，
 * 若没找到，则插入
 * 算法思想：二分查找
*/
void SearchExchangeInsert(ElemType A[],ElemType x){
    int low=0,high=A.size()-1,mid;
    while( low <= high ){
        mid = (low+high)/2;
        if(A[mid]<x){
            low = mid+1;
        }else{
            high = mid-1;
        }
    }
    if(A[mid]==x && mid!=n-1){  //若和最后一个元素相等，则不存在与其后继交换的操作
        ElemType tmp = A[mid];
        A[mid] = A[mid+1];
        A[mid+1] = tmp;
    }
    if(low>high){       //查找失败
        for(int i=A.size()-1;i>high;i--) A[i+1] = A[i];
        A[i+1] = x;
    }
}

/**
 * 题目描述：数组左移p个单位
 * 算法思想：将问题视为把数组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
 * 现将a逆置为a^-1,将b转为b^-1,再将整体转为ba
*/
void Reverse(int R[], int from, int to){
    int i, temp;
    for(i=0;i<(from+to)/2;i++){
        temp = R[from+i];
        R[from+i] = R[to-i];
        R[to-i] = temp;
    }//Reverse
    void Converse(int R[], int n, int p){
        Reverse(R,0,p-1);
        Reverse(R,p,n-1);
        Reverse(R,0,n-1);
    }
}
```

```
/**
 * eg: S1= (11, 13, 15, 17, 19) ; s2(2,4,6,8,20),给出定义 中位数（长度）L/2 向上取整。eg中s1和s2的中位数是 11
 * 算法思想：二路归并排序，排到第(L+1)/2个停止，即为s1和s2的中位数
*/
int sear_mid(SqList &s1,SqList &s2){
    int count=0;
    int i,j;
    while( i<s1.size() && j<s2.size() && count < (s1.size()+s2.size()+1)/2 ){
        if(s1.data[i]<s2.data[j]){
            i++;
            count++;
        }else{
            j++;
            count++;
        }
    }
    if(s1.data[i]<s2.data[j])
        return s1.data[i];
    else
        return s2.data[j];
}

int Majority(vector<int> nums){
unordered_map<int ,int> m;
    for(auto& v : nums){
        m[v]++;
    }
    int key = m.begin()->first;
    int value = m.begin()->second;
    for(auto it=m.begin();it!=m.end();it++){
        //cout << it->first <<":"<<it->second << endl;
        if(it->second>value){
            key = it->first;
            value = it->second;
        }
    }
    if(value > nums.size()/2)
        return value;
    else
        return << -1;
}
```

链表算法 24个

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
/** 迭代 删除值为val的结点 */
class Solution {
public:
    ListNode* deleteNode(ListNode* head, int val) {
        ListNode* dummy = new ListNode(-1);
        dummy->next = head; //建立虚拟头节点
        ListNode* cur = dummy;
        while(cur->next)
            if(cur->next->val == val) cur->next = cur->next->next;
            else cur = cur->next;
        return dummy->next;
    }
};

// 递归 删除 值为val的节点
class Solution {
public:
        ListNode* removeElements(ListNode* head, int val) {
        //1、递归边界
        if(!head)return nullptr;
        //2、递去：直到到达链表尾部才开始删除重复元素
        head->next=removeElements(head->next,val);
        //3、递归式：相等就是删除head，不相等就不用删除
        return head->val==val?head->next:head;
    }
};
```

```
/* 用delete删除被删节点 */
class Solution {
  public:
  ListNode* removeElements(ListNode* head, int val) {
    ListNode* sentinel = new ListNode(0);
    sentinel->next = head;

    ListNode *prev = sentinel, *curr = head, *toDelete = nullptr;
    while (curr != nullptr) {
      if (curr->val == val) {
        prev->next = curr->next;
        toDelete = curr;
      } else prev = curr;

      curr = curr->next;

      if (toDelete != nullptr) {
        delete toDelete;
        toDelete = nullptr;
      }
    }
    ListNode *ret = sentinel->next;
    delete sentinel;
    return ret;
  }
};
```

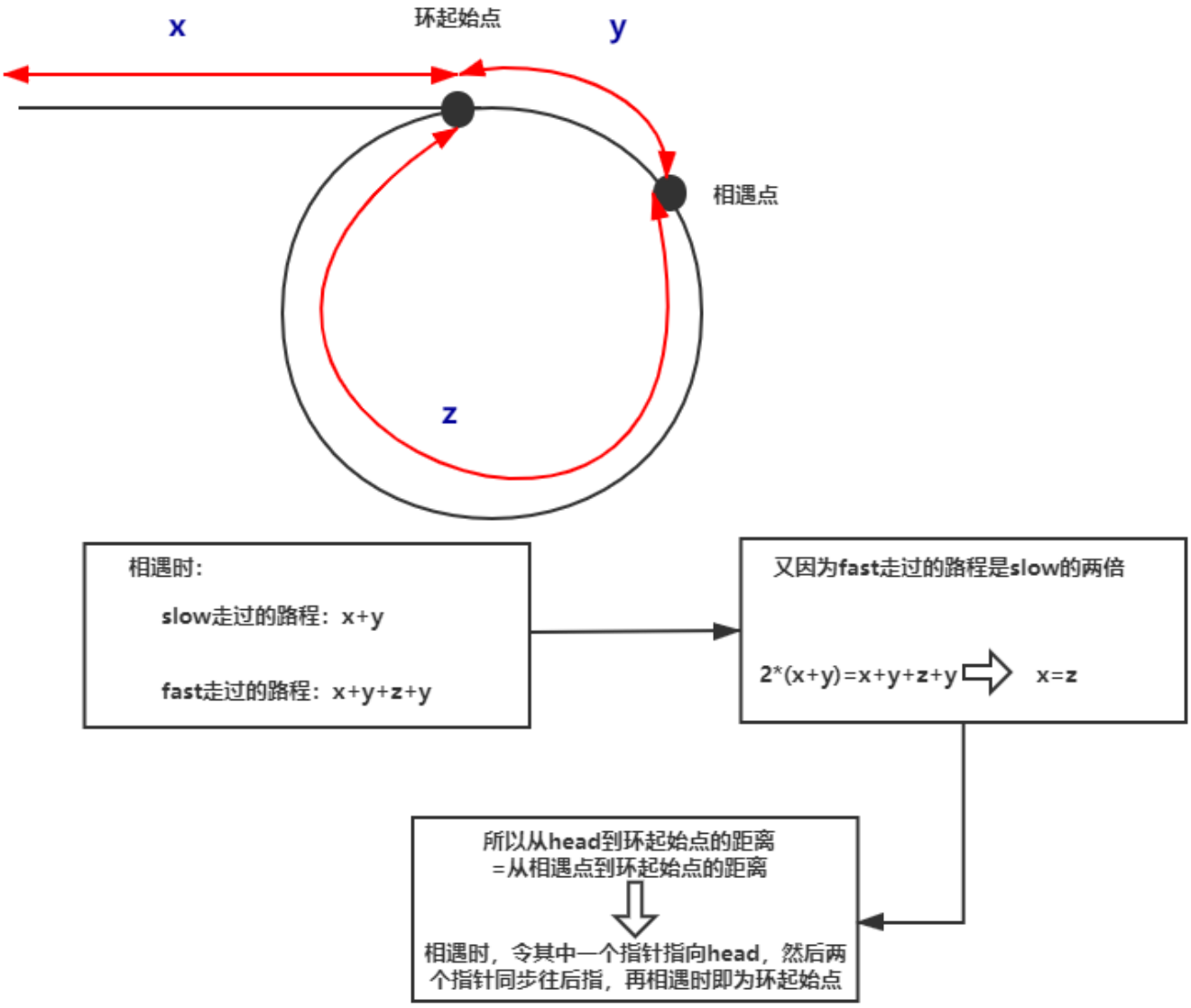
```
//反转链表
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        ListNode *cur = nullptr;
        ListNode *pre = head;
        while(pre){
            ListNode *node = pre->next;
            pre->next = cur;
            cur = pre;
            pre = node;
        }
        return cur;
    }
};
// 递归解（反转链表）
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        // 递归终止条件是当前为空，或者下一个为空
        if(!head || !head -> next) return head;
        // 递归调用来反转每一个结点
        ListNode *curr = reverseList(head -> next);
        // 每一个结点都是这样反转的
        head -> next -> next = head;
        // 防止链表循环，需要将head->next置为空
        head -> next = nullptr;
        // 每层递归函数都返回cur，也就是最后一个结点
        return curr;
    }
};
```

```
/**
 * 判断链表中是否有环
 */
class Solution {
public:
    //哈希表
    bool hasCycle(ListNode *head) {
        set<ListNode*> s;
        while(head != nullptr){
            if(s.insert(head).second==false){
                return true;
            }else{
                s.insert(head);
            }
            head = head->next;
        }
        return false;
    }

    // 快慢指针
    bool hasCycle(ListNode *head) {
        if(!head || !head->next) return false;
        ListNode *slow = head;
        ListNode *fast = head;
        while(fast != nullptr && fast->next != nullptr){
            slow = slow->next;
            fast = fast->next->next;
            if(fast == slow){
                return true;
            }
        }
        return false;
    }
};

/**
*给定一个链表，返回链表开始入环的第一个节点。 如果链表无环，则返回 null。
*为了表示给定链表中的环,我们使用整数pos来表示链表尾连接到链表中的位置(索引从 0 开始)如果 pos 是 -1,则在该链表中没有环。
*说明：不允许修改给定的链表。
*示例 1:
*输入：head = [3,2,0,-4], pos = 1
*输出：tail connects to node index 1
*解释：链表中有一个环，其尾部连接到第二个节点。

*算法思想：快慢指针相遇的时候，此时从node1从head出发，node2从fast出发，相遇即为入口
*/
```


x
环起始点
y
相遇点
z
相遇时：
slow走过的路程：x+y
fast走过的路程：x+y+z+y
又因为fast走过的路程是slow的两倍
2*(x+y)=x+y+z+y
x=z
所以从head到环起始点的距离
=从相遇点到环起始点的距离
相遇时，令其中一个指针指向head，然后两个指针同步往后指，再相遇时即为环起始点

```
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
       ListNode* slow = head;
       ListNode* fast = head;
       while(fast != nullptr && fast->next != nullptr){
           slow = slow->next;
           fast = fast->next->next;
           if(slow == fast){
            ListNode* node1 = fast;
            ListNode* node2 = head;
            while(node1 != node2){
                node1 = node1->next;
                node2 = node2->next;
            }
            return node2;
           }
       }
       return NULL;
    }
};

/* 编写一个程序，找到两个单链表相交的起始节点。 */
//算法思想：为了和你相遇，即使我走过了我所有的路，我也愿意走一遍你走过的路。
class Solution {
public:
    ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
        ListNode *boy = headA;
        ListNode *girl = headB;
        while(boy != girl){
            boy = boy==nullptr ? headB : boy->next;
            girl = girl==nullptr ? headA : girl->next;
        }
        return girl;
    }
};
```

```
/* 给定一个链表，删除链表的倒数第 n 个节点，并且返回链表的头结点。
 *示例:
 *给定一个链表: 1->2->3->4->5, 和 n = 2.
 *当删除了倒数第二个节点后，链表变为 1->2->3->5.
 */
 //算法思想：快慢指针，相距 n个单位,当fast到达表尾时,slow->next 即为要删除的点
class Solution {
public:
    ListNode* removeNthFromEnd(ListNode* head, int n) {
        ListNode *dummyHead = new ListNode(0);
        dummyHead->next = head;
        ListNode *slow = dummyHead;
        ListNode *fast = dummyHead;
        for(int i=0; i<n+1 ; i++){
            fast = fast->next;
        }
        while(fast){
            fast = fast->next;
            slow = slow->next;
        }
        ListNode *delNode = slow->next;
        slow->next = delNode->next;
        delete delNode;

        ListNode *newHead = dummyHead->next;
        delete dummyHead;
        return newHead;
    }
};
```

```
/* 合并两个有序链表 */
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        ListNode *dummyHead = new ListNode(-1);
        ListNode *p = dummyHead;

        while( l1 != nullptr && l2 != nullptr ){
            if(l1->val < l2->val)
            {
                p->next = l1;
                l1=l1->next;
            }else{
                p->next = l2;
                l2=l2->next;
            }
            p=p->next;
        }
        p->next = l1==nullptr ? l2 : l1;
        ListNode *newHead = dummyHead->next;
        delete(dummyHead);
        return newHead;
    }
};
```

```
/*
给定一个链表和一个特定值 x，对链表进行分隔，使得所有小于 x 的节点都在大于或等于 x 的节点之前。
你应当保留两个分区中每个节点的初始相对位置。
示例：
输入：head = 1->4->3->2->5->2, x = 3
输出：1->2->2->4->3->5
*/
class Solution {
public:
    ListNode* partition(ListNode* head, int x) {
        ListNode *lowdummyhead = new ListNode(-1);  // 小于x的结点的虚拟头结点
        ListNode *plow = lowdummyhead;
        ListNode *highdummyhead = new ListNode(-1); // 大于等于x的结点的虚拟头结点
        ListNode *phigh = highdummyhead;
        while(head){
            ListNode *next = head->next;     // 记录head->next 防止链表断裂
            head->next = nullptr;
            if(head->val < x){               // 小于x的值连接到lowdummyhead
                plow->next = head;
                plow = plow->next;
            }else{
                phigh->next = head;          // 大于等于x的值链接到highdummyhead
                phigh = phigh->next;
            }
            head = next;                     // head 向前走一步
        }
        plow->next = highdummyhead->next;
        ListNode *newhead = lowdummyhead->next;
        delete lowdummyhead;
        delete highdummyhead;
        return newhead;
    }
};
```

```
/*
 * 给定一个带有头结点 head 的非空单链表，返回链表的中间结点。
 * 如果有两个中间结点，则返回第二个中间结点。
*/
class Solution {
public:
    ListNode* middleNode(ListNode* head) {
        if(!head) return head;
        ListNode *slow = head;
        ListNode *fast = head;
        while(fast->next && fast->next->next){
            slow = slow->next;
            fast = fast->next->next;
        }
        return fast->next==nullptr ? slow : slow->next; //奇偶个数分别处理
    }
};

/*
 *给定一个链表，两两交换其中相邻的节点，并返回交换后的链表。
 *你不能只是单纯的改变节点内部的值，而是需要实际的进行节点交换。
 */
class Solution {
public:
    ListNode* swapPairs(ListNode* head) {
        if(!head) return nullptr;
        ListNode *dummyhead = new ListNode(-1);
        dummyhead->next = head;
        ListNode *pre = dummyhead;
        ListNode *p = head;
        ListNode *after = head->next;
        while(p && p->next){
            pre->next = p->next;
            p->next = after->next;
            after->next = p;
            pre = p;
            p = p->next;
            if(p) after = p->next;
        }
        ListNode *newHead = dummyhead->next;
        delete dummyhead;
        return newHead;
    }
};
```

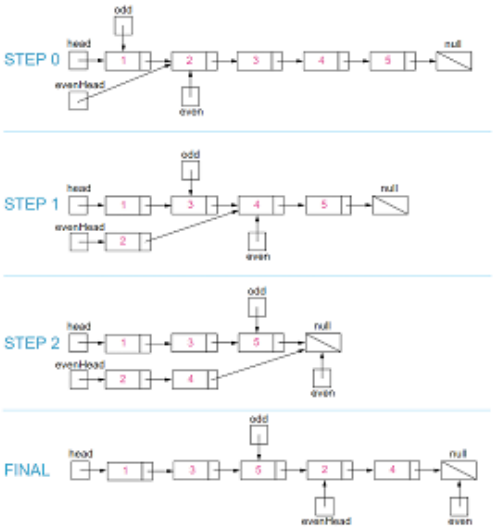
```
/*
 *给你两个 非空 链表来代表两个非负整数。数字最高位位于链表开始位置。
 *它们的每个节点只存储一位数字。将这两数相加会返回一个新的链表。
 *你可以假设除了数字 0 之外，这两个数字都不会以零开头。
 *复杂度分析
 *时间复杂度：O(max(m, n))O(max(m,n))，其中 mm 与 nn 分别为两个链表的长度。我们需要遍历每个链表。
 *空间复杂度：O(m + n)O(m+n)，其中 mm 与 nn 分别为两个链表的长度。这是我们把链表内容放入栈中所用的空间。
*/

class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        stack <int> s1,s2;
        while(l1){          // 入栈
            s1.push(l1->val);
            l1=l1->next;
        }
        while(l2){
            s2.push(l2->val);
            l2=l2->next;
        }
        int carry=0;
        ListNode *ans=nullptr;
        while( !s1.empty() or !s2.empty() or carry > 0 ){
            int x = s1.empty() ? 0 : s1.top();
            int y = s2.empty() ? 0 : s2.top();
            if(!s1.empty())s1.pop();
            if(!s2.empty())s2.pop();
            int curr = x + y +carry;
            carry = curr/10;
            curr%=10;
            ListNode *temp = new ListNode(curr);
            temp->next = ans;
            ans = temp;
        }
        return ans;
    }
};

class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        ListNode *p1 = l1;
        ListNode *p2 = l2;
        ListNode *dummyhead = new ListNode(-1);
        ListNode *cur = dummyhead;
        int carry = 0;
        while(p1 || p2){                            // 不论长短 一起遍历
```

```
            int x = p1 != nullptr ? p1->val : 0;     // 若p1到尾部，按照加法规则，从低位对齐，高位视为0
            int y = p2 != nullptr ? p2->val : 0;
            int sum = x + y + carry;
            carry = sum / 10;
            cur->next = new ListNode(sum%10);        // sum % 10 十进制，满十进位，剩余保留
            cur = cur->next;                         // 结果链表指针移到当前尾部
            if(p1) p1 = p1->next;                    // 如果p1不空,p1前进一步。防止空指针报错，因为本循环无视长度
            if(p2) p2 = p2->next;
        }
        if(carry > 0){                               // 如果最高位满十，则进位，逻辑同其他
            cur->next = new ListNode(carry);
        }
        ListNode *newhead = dummyhead->next;
        delete dummyhead;
        return newhead;
    }
};

/*
    给定一个单链表，把所有的奇数节点和偶数节点分别排在一起。
    请注意，这里的奇数节点和偶数节点指的是节点编号的奇偶性，而不是节点的值的奇偶性。
*/
```



```
class Solution {
public:
    ListNode* oddEvenList(ListNode* head) {
        if(!head) return nullptr;
        // odd是奇链表的当前节点，先初始化为head（初始化为奇链表头）
        ListNode *odd = head;
        // even是偶链表的当前节点，初始化为第二个节点也就是head.next
        ListNode *even = head->next;
        // evenHead是偶链表的头节点，初始化为链表第二个节点（初始化为奇链表头的下一个节点）
        ListNode *evenhead = even;
        while(even && even->next){  //涉及到链表长度为奇或偶，用双指针，条件好像都是这个
            odd->next = even->next;
            odd = odd->next;
            even->next = odd->next;
            even = even->next;
        }
        odd->next = evenhead;    // 奇(拼接)偶
        return head;
    }
};

/* 删除 有序 链表中的重复项 */
void Del_Same(LinkList &L){
    LNode* p = L->next, *q;
    if(p==null) return;
    while(p->next){
        q = p->next;
        if(p->data == q->data){
            p->next = q->next;
            free(q);
        }else{
            p = p->next;
        }
    }
}
```

```
/* 请判断一个链表是否为回文链表。(栈) */
class Solution {
public:
    bool isPalindrome(ListNode* head) {
        stack<int> st;
        ListNode *cur = head;
        int length = 1;
        while(cur){
            st.push(cur->val);
            cur = cur->next;
            length++;
        }
        int cnt = length/2;
        cur = head;
        while(cnt--){
            if(cur->val != st.top()) return false;
            st.pop();
            cur = cur->next;
        }
        return true;
    }
};
```

```
/* 顺序表法 */
class Solution {
public:
    bool isPalindrome(ListNode* head) {
        vector<int> res;
        while(head){
            res.push_back(head->val);
            head = head->next;
        }
        int i = 0; int j = res.size() - 1;
        while(i<j){ // 啊啊啊啊 边界值真的是佛了啊
            if(res[i++]!=res[j--]) return false;
        }
        return true;
    }
/*  双指针，反转链表 */
    public:
        bool isPalindrome(ListNode *head){
            ListNode *cur = head;
            ListNode *pre = nullptr;
            ListNode *fast = head;
            while(fast && fast->next){
                fast = fast->next->next; //位置不可以变，因为链表在后面断了
                ListNode *temp = cur->next;
                cur->next = pre;
                pre = cur;
                cur = temp;
            }
            if(fast) cur = cur->next;
            while(pre){
                if(cur->val != pre->val) return false;
                cur = cur->next;
                pre = pre->next;
            }
            return true;
        }

};
```

```
/* 给定一个链表，旋转链表，将链表每个节点向右移动 k 个位置，其中 k 是非负数。 */
class Solution {
public:
    ListNode* rotateRight(ListNode* head, int k) {
        if(!head || !head->next || k==0) return head;
        ListNode *phead = head;
        int length = 1;
        while(phead->next){length++;phead = phead->next;}
        int move = k % length;
        if(move == 0) return head;
        ListNode *cur = head;
        for(int i=0;i<length-move-1;i++) cur = cur->next;
        ListNode *tmp = cur->next;
        cur->next = nullptr;
        phead->next = head;
        return tmp;
    }
};
```

```
/** 给定一个只包括 '(', ')', '{', '}', '[', ']' 的字符串，判断字符串是否有效。
    有效字符串需满足：
    左括号必须用相同类型的右括号闭合。左括号必须以正确的顺序闭合。
 */
class Solution{
public:
    bool isValid(string s){
        int n = s.size();
        if(n % 2 == 1){return false;}
        unordered_map<char,char> pairs = {
            {')','('},
            {']','['},
            {'}','{'}
        };
        stack<char> stk;
        for(char ch : s){
            if(pairs.count(ch)){
                if(stk.empty() || stk.top() != pairs[ch]){
                    return false;
                }else{
                    stk.pop();
                }
            }else{
                stk.push(ch);
            }
        }
        return stk.empty();
    }
};

// I,O表示入栈出栈，栈的始态和终态均为空，判断序列是否合法
bool judge(vector<char> vec){
    if(vec[0] == 'O') return false;
    int j = 0, k = 0;
    for(int i = 0; i < vec.size(); ++i){
        vec[i] == 'I' ? ++j : ++k;
        if(k > j) return false;
        i++;
    }
    return j != k ? false : true;
}
```

二叉树算法 29个

```
//前序遍历
void preorder(bitree *t){
    bitree *temp;
    temp = t;
    while(temp || !empty()){
        while(temp){
            printf("%4d",temp->data);
            push(temp);
            temp = temp->lchild;
        }
        if(s.top!=0){
            temp = pop();
            temp = temp->rchild;
        }
    }
    printf("\n");
}

//中序遍历
void inorder(bitree *t){
    bitree *temp = t;
    while(temp || !empty()){
        if(temp){
            push(temp);
            temp = temp->lchild;
        }else{
            temp = pop();
            printf("%4d",temp->data);
            temp = temp->rchild;
        }
    }
    printf("\n");
}
```

```
//后序遍历
void postorder01(bitree *t){
    bitree *temp = t;
    bitree *r=NULL;
    while(temp || !empty()){
        if(temp){
            push(temp);
            temp = temp->lchild;
        }else{
            temp = getTop();
            //printf("栈顶元素:%4d\n",temp->data);
            if(temp->rchild && temp->rchild!=r){
                temp = temp->rchild;
                push(temp);
                temp = temp->lchild;
            }else{
                temp = pop();
                printf("%4d",temp->data);
                r = temp;
                temp = NULL;
            }
        }
    }
    printf("\n");
}
```

```
/**
*    二叉树的前序遍历非递归(统一写法)
*/
class Solution{
public:
    vector<int> preorderTraversal(TreeNode* root) {
        vector<int> result;
        stack<TreeNode*> st;
        if(root != NULL) st.push(root);
        while( !st.empty() ){
            TreeNode *node = st.top();
            if(node != NULL){
                st.pop();
                if(node->right) st.push(node->right);   //右
                if(node->left) st.push(node->left);     //左
                st.push(node);                          //中
                st.push(NULL);
            }else{
                st.pop();
                node = st.top();
                st.pop();
                result.push_back(node->val);
            }
        }
        return result;
    }
};
```

```
/**
*    二叉树的层序遍历
*/
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        vector<vector<int>> result;
        queue<TreeNode*> que;
        if(root != NULL) que.push(root);
        while(!que.empty()){
            int size = que.size();
            vector<int> vec;
            for(int i=0; i<size; i++){
                TreeNode *node = que.front();
                q.pop();
                vec.push_back(node->val);
                if(node->left) que.push(node->left);
                if(node->right) que.push(node->right);
            }
            result.push_back(vec);
        }
        return result;
    }
};

/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 * 递归二叉树的深度
 */
class Solution {
public:
    int max(int a, int b){
        return a>b ? a : b;
    }
    int maxDepth(TreeNode* root) {
        if(!root) return 0;
        int left_depth = maxDepth(root->left);
        int right_depth = maxDepth(root->right);
        return max(left_depth,right_depth)+1;
    }
};
```

```
/**
 *  给定一个二叉树，检查它是否是镜像对称的。
 *  例如，二叉树 [1,2,2,3,4,4,3] 是对称的。
 */
class Solution {
public:
    bool check(TreeNode *p, TreeNode *q){
        if(!p && !q) return true;
        if(!p || !q) return false;
        return check(p->left,q->right) && check(p->right,q->left) && p->val == q->val;
    }
    bool isSymmetric(TreeNode* root) {
        return check(root,root);
    }
};
```

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 * 给定一个二叉树和一个目标和，判断该树中是否存在根节点到叶子节点的路径，这条路径上所有节点值相加等于目标和。
 * 说明：叶子节点是指没有子节点的节点。
 * 示例:
 * 给定如下二叉树，以及目标和 sum = 22
 */
class Solution {
public:
    bool hasPathSum(TreeNode* root, int sum) {
        queue<TreeNode*> node_que;
        queue<int> val_que;
        if(root){
            node_que.push(root);
            val_que.push(root->val);
        }
        while(!node_que.empty() && !val_que.empty()){
            TreeNode *node = node_que.front(); node_que.pop();
            int val = val_que.front(); val_que.pop();
            if(!node->left && !node->right) {
                if( val==sum )
                    return true;
                continue;
            }
            if(node->left) {
                node_que.push(node->left);
                val_que.push(node->left->val + val);
            }
            if(node->right) {
                node_que.push(node->right);
                val_que.push(node->right->val + val);
            }
        }
        return false;
    }
};
```

```
/**
* 《从中序与后序遍历序列构造二叉树》
* 根据一棵树的中序遍历与后序遍历构造二叉树。
* 注意：
* 你可以假设树中没有重复的元素。
* 例如，给出：中序遍历 inorder = [9,3,15,20,7]，后序遍历 postorder = [9,15,7,20,3]
* 返回如下的二叉树：
     3
    / \
   9  20
     /  \
    15   7
*/
class Solution {
public:
    // 找到Inorder中根节点的位置
    int index;
    int getInorderIndex(vector<int> &inorder, int val){
        for(int i=0;i<inorder.size();i++) {
            if(val==inorder[i]) return i;
        }
        return 0;
    }

    TreeNode* create(vector<int>& inorder, vector<int>& postorder, int left, int right){
        if( left>right ) return nullptr;
        int val=postorder[index];
        TreeNode* root=new TreeNode(val);
        index--;
        int inIndex = getInorderIndex(inorder, val);
        root->right=create(inorder,postorder, inIndex+1, right);    // 后序(左右根)顺序表 index-- 先指向右子树

        root->left=create(inorder, postorder, left, inIndex-1);
        return root;
    }

    TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
        if (!postorder.size()) return nullptr;
        // get root index
        index=postorder.size()-1;
        TreeNode* root=create(inorder, postorder, 0, inorder.size()-1);
        return root;
    }
};
```

```
/**
根据一棵树的前序遍历与中序遍历构造二叉树。
注意:
你可以假设树中没有重复的元素。
例如，给出 前序遍历 preorder = [3,9,20,15,7], 中序遍历 inorder = [9,3,15,20,7]
返回如下的二叉树:
    3
   / \
  9  20
    /  \
   15   7
*/

class Solution{
public:
    int index;
    int find(vector<int> inorder, int val){
        for(int i=0; i<inorder.size(); i++){
            if( inorder[i]==val ){
                return i;
            }
            return -1;
        }
    }
    TreeNode *create(vector<int> preorder, vector<int> inorder, int left, int right){
        if(left > right) return nullptr;
        int val = preorder[index];
        TreeNode *root = new TreeNode(val);
        index++;
        int rootIndexOfInorder = find(inorder, val);
        root->left = create(preoder, inorder, 0, rootIndexOfInorder-1); //前序(根左右)遍历序列, index++先指向左子树
        root->right = create(preorder, inorder, rootIndexOfInorder+1, right);
        return root;
    }
    TreeNode *buildTree(vector<int>& preorder, vector<int>& inorder){
        index = 0;
        return create(preoder, inorder, 0, inorder.size()-1);
    }
};
```

```
//顺序存储结构
/**
 *  i的左孩子 ---2i
 *  i的右孩子 ---2i+1
 *  i的父结点 ---i/2(向下取整)
 *  i是否是叶子结点/分支结点 --- i 是否大于 ( n/2(向下取整) )
 *  i是否有左孩子 ---2i <= n
 *  i是否有右孩子 ---2i+1 <= n
*/
char comm_Ancestor(Tree T,int i,int j){
    if(T.nodes[i].data!='#' && T.nodes[j].data != '#'){
        while(i!=j){
            if(i>j)
                i = i/2;
            else
                j = j/2;
        }
        return T.nodes[i].data;
    }
}
```

```
//判断是否是完全二叉树
/**
    层序遍历 所有节点入队，包括空节点
    空树 return true
    if node == nullptr
        check 后面节点是否为空
            if 后面节点为空
                return false
            return true
*/
class Solution{
public:
    bool check(TreeNode *root){
        queue<TreeNode*> q;
        if(T == nullptr){
            return true;
        }
        q.push(root);
        while(!q.empty()){
            TreeNode *node = q.front();
            q.pop();
            if(node){
                q.push(node->left);
                q.push(node->right);
            }else{
                while(!q.empty()){
                    node = q.front();
                    q.pop();
                    if(p){
                        return false;
                    }
                }
            }
        }
        return true;
    }
};
```

```
// 双分支节点个数
/**
    model:
    f(b)=0                          // 若b=null
    f(b)=f(b->left)+f(b->right)+1   // 若b是双分支结点
    f(b)=f(b->left)+f(b->right)     // 若b是叶结点或单分支节点
*/
int DsonNodes(BinTree T){
    if(T==NULL)
        return 0;
    else if(T->left!=NULL &&T->right!=NULL) return DsonNodes(T->left)+DsonNodes(T->right)+1;
    else return DsonNodes(T->left)+DsonNodes(T->right);
}

/* 交换左右子树 */
/* 后序遍历思想：先交换b的左子树的左右子树，再交换b的右子树的左右子树，最后交换b的左右子树 */
class Solution {
public:
    TreeNode* invertTree(TreeNode* root) {
        if(!root) return nullptr;
        TreeNode *left = invertTree(root->left);
        TreeNode *right = invertTree(root->right);

        root->left = right;
        root->right = left;
        return root;
    }
};
```

```
// 最近公共祖
/**
 * 具体思路:
(1) 如果当前结点 root 等于 NULL，则直接返回 NULL
(2) 如果 root 等于 p 或者 q ，那这棵树一定返回 p 或者 q
(3) 然后递归左右子树，因为是递归，使用函数后可认为左右子树已经算出结果，用 left 和 right 表示
(4) 此时若left为空，那最终结果只要看 right；若 right 为空，那最终结果只要看 left
(5) 如果 left 和 right 都非空，因为只给了 p 和 q 两个结点，都非空，说明一边一个，因此 root 是他们的最近公共祖先
(6) 如果 left 和 right 都为空，则返回空（其实已经包含在前面的情况中了）

 时间复杂度是 O(n)：每个结点最多遍历一次或用主定理，空间复杂度是 O(n)：需要系统栈空间
*/
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if(root==nullptr) return nullptr;
        if(root==p || root==q) return root;

        TreeNode *left = lowestCommonAncestor(root->left, p, q);
        TreeNode *right = lowestCommonAncestor(root->right, p, q);

        if(left==nullptr) return right;
        if(right==nullptr) return left;

        if(right && left) return root;

        return nullptr;
    }
};

/* 递归所有祖先 */
int PrintAncestors(BinTree *root, ElemType x)
{
    if (!root)  return 0;
    if (root->data == x)    return 1;
    //如果子树中可以找到匹配值 那么此节点肯定是祖先结点
    if (PrintAncestors(root->left, x) || PrintAncestors(root->right, x))
    {
        printf("%4c", root->data);
        return 1;
    }
    return 0;
}//打印祖先      结果: 5 1
```

```
/* 满二叉树 前序序列转换为后序序列 */
void preTopost(char pre[],int l1,int h1,char post[],int l2,int h2){
    int half;
    if(l1<=h1){
        post[h2]=pre[l1];
        half=(h1-l1)/2;
        preTopost(pre,l1+1,l1+half,post,l2,l2+half-1);
        preTopost(pre,l1+half+1,h1,post,l2+half,h2-1);
    }
}

/* 将叶子结点从左到右连成一个单链表，表头指向head */
BinTree *head=NULL;
BinTree *pre=NULL;

BinTree *leafNodeToList(BinTree *root){
    if(root){
        leafNodeToList(root->left);
        if(root->left==NULL && root->right==NULL){
            if(pre==NULL){
                head=root;
                pre=root;
            }else{
                pre->right=root;
                pre=root;
            }
        }
        leafNodeToList(root->right);
        pre->right=NULL;
    }
    return head;
}

/* 判断两棵树是否相似 */
int similar_tree(BinTree *t1,BinTree *t2){
    if(t1==NULL && t2==NULL) return 1;
    else if(t1==NULL || t2==NULL) return 0;
    else{
        return similar_tree(t1->left,t2->left) && similar_tree(t1->right,t2->right);
    }
}
```

```
/* 中序线索二叉树里查找指定节点在后序的前驱结点 */
BinTree *InPostPre(BinTree *t,BinTree *p){
    BinTree *q;
    if(p->rtag==0) q=p->right;
    else if(p->ltag==0) q=p->left;
    else if(p->left==NULL)
        q=NULL;
    else{
        while(p->ltag==1&&p->left!=NULL){
            p=p->left;
        }
        if(p->ltag==0)
            q=p->left;
        else
            q=NULL;
    }
    return q;
}

/* 将给定的表示式树（二叉树）转换为等价的中缀表达式（添加括号反映操作符计算次序） */
void BTreeToE(BTree *root){
    BTreeToExp(root,1); //根的高度为1
}
void BTreeToExp(BTree *root, int deep){
    if(root==nullptr) return;    // 空节点返回
    else if(root->left==nullptr && root->right==null)    // 若为叶子结点
        printf(root->data);             // 输出操作数，不加括号
    else{
        if(deep > 1) printf("(");    // 若有子表达式则加 1层括号
        BTreeToExp(root->left, deep + 1);
        printf(root->data);          //  输出操作符
        BTreeToExp(root->right, deep + 1);
        if(deep > 1) printf(")");    // 若有子表达式则加 1层括号
    }
}
```

```
/* 判断是否是二叉搜索树 */
// 递归
class Solution {
public:
    bool helper(TreeNode* root, long long  low, long long high){
        if(!root) return true;
        if(root->val >= high || root->val <=low) return false;
        return helper(root->left, low, root->val) && helper(root->right, root->val, high);
    }
    bool isValidBST(TreeNode* root) {
        return helper(root, LONG_MIN, LONG_MAX);
    }
};

// 迭代 判断二叉搜索树
class Solution {
public:
    long long inorder = LONG_MIN;
    bool isValidBST(TreeNode* root) {
        if(!root) return true;
        stack<TreeNode*> st;
        vector<long long> res;
        while (root != nullptr || !st.empty()) {
            while (root != nullptr) {
                st.push(root);
                root = root->left;
            }
            root = st.top();
            st.pop();
            if(root->val <= inorder) return false;
            inorder = root->val;
            root = root->right;
        }
        return true;
    }
};
```

```
/* 判断是否是二叉平衡树 */
class Solution {
public:
    int max(int a, int b){
        return a > b ? a : b;
    }
    int maxDepth(TreeNode *root){
        if(!root) return 0;
        int leftdepth = maxDepth(root->left);
        int rightdepth = maxDepth(root->right);
        if(!root->left || !root->right) return leftdepth + rightdepth + 1;
        return max(leftdepth,rightdepth) + 1;
    }
    bool isBalanced(TreeNode* root) {
        if(!root)  return true;
        if(maxDepth(root->left)-maxDepth(root->right) > 1 || maxDepth(root->left) - maxDepth(root->right) < -1
            return false;
        return isBalanced(root->left) && isBalanced(root->right);
    }
};

/* 二叉树最大宽度 */
class Solution {
public:
    int widthOfBinaryTree(TreeNode* root) {
        if(!root) return 0;
        queue<pair<TreeNode*, unsigned long long>> que; // 结点，位置下标
        int ans = 1;
        que.push({root,1});
        while(!que.empty()){
            int sz = que.size();
            // 当前层的宽度是队尾(最右)编号-对头(最左)编号+1
            ans = max(int(que.back().second - que.front().second + 1),ans);
            while(sz--){
                TreeNode *node = que.front().first;
                unsigned long long pos = que.front().second;
                que.pop();
                if(node->left) que.push({node->left, pos * 2});
                if(node->right) que.push({node->right, pos * 2 + 1});
            }
        }
        return ans;
    }
};
```

图

图的数据结构(邻接矩阵)

```
typedef struct GNode *PtrToGNode;
struct GNode{
    int Nv;  /* 顶点数 */
    int Ne;  /* 边数   */
    VertexType vex[Maxsize]; /* 顶点表*/
    EdgeType G[MaxVertexNum][MaxVertexNum]; /* 邻接矩阵, 边表*/
};
typedef PtrToGNode MGraph; /* 以邻接矩阵存储的图类型 */
```

深度优先遍历（递归）

```
void DFS(MGraph graph, int v){
    visited[v] = true;
    visit(v);
    for(int i = 0; i < graph->Nv; ++i){
        if(graph->G[v][i] == 1 && !visited[vex[i]])
            DFS(graph, vex[i]);
    }
}
```

深度优先遍历（迭代）

```
void DFS(MGraph graph, int v){
    stack<int> st;
    visit(v);
    visited[v];
    st.push(v);
    while(!st.empty()){
        int data, i;
        data = st.top();
        for(i = 0; i < graph->Nv; ++i){
            if(graph->G[data][i] == 1 && visited[vex[i]] == 1){
                visit(vex[i]);
                visited[vex[i]] = true;
                st.push(v);
                break;
            }
        }
        if(i == graph->Nv) st.pop();
    }
}
```

图的广度优先遍历(迭代)

```
void BFS(MGraph graph, int v){
    queue<int> que;
    int vertex;
    visit(v);
    visited[v] = true;
    que.push(v);
    while(!que.empty()){
        vertex = que.front();
        que.pop();
        for(int i = 0; i < graph->Nv; ++i){
            if(graph->G[v][i] == 1 && !visited[vex[i]]){
                visit(vex[i]);
                visited[vex[i]] = true;
                que.push(vex[i]);
            }
        }
    }
}
```

图的数据结构描述(邻接表)

```
typedef struct ArcNode{      // 边表结点
    int adjvex;              // 该弧所指向的顶点的位置
    struct ArcNode *nextArc;    // 指向下一条弧的指针
}ArcNode;

typedef struct VNode{        // 顶点表结点
    int data;                // 顶点信息
    ArcNode* firstArc;          // 指向第一条依附该顶点的弧的指针
}VNode, AdjList[MaxVertexNum];

typedef struct{
    AdjList vertices;        // 邻接表
    int vexnum;              // 顶点数目
    int arcnum;              // 边数目
}ALGraph;
```

```
vector<bool> visited;
void DFS(ALGraph graph, int v){ //深度优先遍历(递归)
    visited[v] = true;
    ArcNode* p;
    visit(graph.vertices[v].data);
    p = G.verties[v].firstArc;
    while(p){
        if(!visited[p->adjvex])
            DFS(graph, p->adjvex);
        p = p->nextArc;
    }
}

void DFSTraverse(Graph graph,int v){ //图的非递归深度优先遍历
    int i,visited[MaxSize],top;
    ArcNode *stack[MaxSize],*p;
    for(i = 0; i < graph.vexnum; i++){    //将所有顶点都添加未访问标志0
        visited[i] = 0;
    }
    printf("%4c",graph.vertices[v].data); //访问顶点v并将访问标志置为1
    visited[v] = 1;

    top = -1; //初始化栈
    p = graph.vertices[v].firstArc;    //p指向顶点v的第一个邻接点
    while(top > -1 || p != NULL){
        while(p!=NULL){
            if(visited[p->adjvex] == 1){
                p = p->nextarc;
            }else{
                printf("%4c",graph.vertices[p->adjvex].data);
                visited[p->adjvex]=1;
                stack[++top] = p;
                p = graph.vertices[p->adjvex].firstArc;
            }
        }
        if(top > -1){
            p = stack[top--];
            p = p->nextArc;
        }
    }
}
```

```
// 图的广度优先遍历
int visited[maxsize];
void BFS(ALGraph *G, int v){
    ArcNode* p;
    int que[maxsize], front = 0, rear = 0;
    int j;
    Visit(v);
    visited[v] = 1;
    rear = (rear + 1) % maxsize;
    que[rear] = v;
    while (front != rear)
    {
        front = (front + 1) % maxsize;
        j = que[front];
        p = G->adjlist[j].firstArc;
        while(p){
            if(visited[p->adjvex] == 0){
                Visit(p->adjvex);
                visited[p->adjvex] = 1;
                rear = (rear + 1) % maxsize;
                que[rear] = p->adjvex;
            }
            p = p->nextarc;
        }
    }
}
```

```
//拓扑
bool TopologicalSort(Graph G){
    //若G存在拓扑排序，返回true，  否则返回false
    InitStack(S);
    for(int i=0; i<G.vexnum; ++i){
        if(indegree[i] == 0)
            S.push(i);
        int count = 0;
        while(!IsEmpty(S)){
            Pop(S,i);
            print[count++] = i;
            for(p = G.vertices[i].firstarc; p; p = p->next){
                //将所有i所指向的顶点的入度减1，  并且将入度减为0的顶点压入栈S
                    v = p->adjvex;
                    if(!(--indegree[v]))
                        S.push(v);       // 入度为0 则入栈
            }//for
        }//while
        if(count < G.vexnum)     // 拓扑排序失败 有回路
            return false;
        else
            return true;
    }
}
```

```
// dijstra
int g[N][N];  // 存储每条边
int dist[N];  // 存储1号点到每个点的最短距离
bool st[N];   // 存储每个点的最短路是否已经确定

// 求1号点到n号点的最短路，如果不存在则返回-1
int dijkstra()
{
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;

    for (int i = 0; i < n - 1; i ++ )
    {
        int t = -1;     // 在还未确定最短路的点中，寻找距离最小的点
        for (int j = 1; j <= n; j ++ )
            if (!st[j] && (t == -1 || dist[t] > dist[j]))
                t = j;

        // 用t更新其他点的距离
        for (int j = 1; j <= n; j ++ )
            dist[j] = min(dist[j], dist[t] + g[t][j]);

        st[t] = true;
    }

    if (dist[n] == 0x3f3f3f3f) return -1;
    return dist[n];
}

// floyd
```

初始化：

```
    for (int i = 1; i <= n; i ++ )
        for (int j = 1; j <= n; j ++ )
            if (i == j) d[i][j] = 0;
            else d[i][j] = INF;

// 算法结束后，d[a][b]表示a到b的最短距离
void floyd()
{
    for (int k = 1; k <= n; k ++ )
        for (int i = 1; i <= n; i ++ )
            for (int j = 1; j <= n; j ++ )
                d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
}
```

```
// prim
int n;      // n表示点数
int g[N][N];        // 邻接矩阵，存储所有边
int dist[N];        // 存储其他点到当前最小生成树的距离
bool st[N];     // 存储每个点是否已经在生成树中


// 如果图不连通，则返回INF(值是0x3f3f3f3f)，否则返回最小生成树的树边权重之和
int prim()
{
    memset(dist, 0x3f, sizeof dist);

    int res = 0;
    for (int i = 0; i < n; i ++ )
    {
        int t = -1;
        for (int j = 1; j <= n; j ++ )
            if (!st[j] && (t == -1 || dist[t] > dist[j]))
                t = j;

        if (i && dist[t] == INF) return INF;

        if (i) res += dist[t];
        st[t] = true;

        for (int j = 1; j <= n; j ++ ) dist[j] = min(dist[j], g[t][j]);
    }

    return res;
}
```

查找 4个

```
//折半查找

int Bsearch(int R[], int low, int high, int k){
    int mid;
    while(low <= high)  // 当子表长度大于等于1时进行循环
    {
        mid = (low + high) / 2;
        if(R[mid] == k)
            return mid;
        else if(R[mid] > k)
            high = mid - 1;
        else
            low = mid + 1;
    }
    return 0;
}
```

```
//二叉排序树 查找
BTNode* BSTSearch(BTNode* bt, int key)
{
    if(bt == null)
        return null;
    else
    {
        if(bt->val == key)
            return bt;
        else if(key < bt->val)
            return BSTSearch(bt->left, key);
        else
            return BSTSearch(bt->right, key);
    }
}

//二叉排序树 插入
int BSTInsert(BTNode* bt, int key)
{
    if(bt == null)
    {
        BTNode* bt = new BTNode(key);
        return 1;
    }
    else
    {
        if(key == bt->val)
            return 0;
        else if(key < bt->val)
            BSTInsert(bt->left, key);
        else
            BStInsert(bt->right, key);
    }
}
```

排序算法 8个

```
//直接插入排序
void InserSort(int R[], int n)
{
    int i, j;
    int temp;
    for(i = 1; i < n; ++i)
    {
        temp = R[i];
        j = i - 1;
        while(j >= 0 && temp < R[j])
        {
            R[j + 1] = R[j];
            --j;
        }
        R[j + 1] = temp; // 插入
    }
}

// 折半查找排序
void InsertSort(int A[]){
    int n = A.size();
    int low, high;
    for(int i = 2; i < n; ++i){
        A[0] = A[i];
        low = 1;
        high = i - 1;
        while(low < high){
            mid = (low + high) / 2;
            if(A[mid] > A[0])
                high = mid - 1;
            else
                low = mid + 1;
        }
        for(j = i - 1; j >= high + 1; --j)
            A[j + 1] = A[j];
        A[high + 1] = A[0];
    }
}
```

```
// ShellSort
/*
    记录前后位置的增量是dk，不是1
    A[0]暂存单元，不是哨兵，当 j<=0 时，插入位置已到
*/
void ShellSort(int A[]){
    int n = A.size();
    for(int dk = n / 2; dk >= 1; dk = dk / 2)
        for(int i = dk + 1; i <= n; ++i)
            if(A[i] < A[i - dk]){
                A[0] = A[i];
                for(int j = i - dk; j > 0 && A[0] < A[j]; j -= dk)
                    A[j + dk] = A[j];
                A[j + dk] = A[0];
            }
}

// 冒泡排序
void BubbleSort(int R[]){
    int n = R.size();
    for(int i = n - 1; i >= 1; --i){
        int flag = 0;
        for(j = 1; j <=i; ++j){
            if(R[j - 1] > R[j]){
                int temp = R[j];
                R[j] = R[j - 1];
                R[j - 1] = temp;
                flag = 1;
            }
        }
        if(flag == 0)
            return;
    }
}
```

```
// 快速排序
#include<iostream>
using namespace std;

const int N = 1000010;
int n;
int arr[N];

void qiuck_sort(int arr[], int l, int r){
    if(l >= r) return;
    int i = l - 1, j = r + 1, x = arr[l + r >> 1];
    while(i < j){
        do i++; while(arr[i] < x);
        do j--; while(arr[j] > x);

        if(i < j) swap(arr[i], arr[j]);
    }
    qiuck_sort(arr, l, j), qiuck_sort(arr, j + 1, r);
}

int main(){
    scanf("%d", &n);

    for(int i = 0; i < n; i++)  scanf("%d", &arr[i]);

    qiuck_sort(arr, 0 , n - 1);

    for(int i = 0; i < n; i++)  printf("%d ", arr[i]);
}

// 简单选择排序
void SelectSort(int R[], int n){
    int i, j, k;
    int temp;
    for(int = 0; i < n; ++i){
        k = i;
        //从无序列中选择最小的一个关键字
        for(j = i + 1; j < n; ++j){
            if(R[k] > R[j])
                k = j;
        }
        // 最小关键字与无序序列第一个位置交换
        temp = R[i];
        R[i] = R[k];
        R[k] = temp;
    }
}
```

```cpp
// 归并排序
#include<iostream>
using namespace std;

const int N = 1000010;
int n;
int tmp[N];
int q[N];

void merge(int q[], int l, int r){
    if(l >= r) return;
    int mid = l + r >> 1;
    merge(q, l, mid);
    merge(q, mid + 1, r);
    int k = 0, i = l, j = mid + 1;
    while(i <= mid && j <= r){
        if(q[i] < q[j]) tmp[k++] = q[i++];
        else tmp[k++] = q[j++];
    }
    while(i <= mid) tmp[k++] = q[i++];
    while(j <= r) tmp[k++] = q[j++];

    for(i = l, j = 0; i <= r; i++, j++) q[i] = tmp[j];
}
int main(){
    scanf("%d", &n);
    for(int i = 0; i < n; i++){
        scanf("%d", &q[i]);
    }
    merge(q, 0, n - 1);

    for(int i = 0; i < n; i++){
        printf("%d ", q[i]);
    }
}
```

```
// 堆排序
#include<iostream>
using namespace std;
    void swap(int tree[], int a, int b){
        int temp = tree[a];
        tree[a] = tree[b];
        tree[b] = temp;
    }
    void heapify(int tree[], int n, int i){
        int c1 = i * 2 + 1;
        int c2 = i * 2 + 2;
        int max = i;
        if(c1 < n && tree[c1] > tree[max])
            max = c1;
        if(c2 < n && tree[c2] > tree[max])
            max = c2;
        if(max != i){
            swap(tree, max, i);
            heapify(tree, n, max);
        }
    }
    void build_heap(int tree[], int n){
        int last_node = n - 1;
        int parent = (last_node - 1) / 2;
        for(int i = parent; i >= 0; i--){
            heapify(tree, n, i);
        }
    }
    void heapSort(int tree[], int n){
        build_heap(tree, n);
        for(int i = n - 1; i >= 0; i--){
            swap(tree, i, 0);
            heapify(tree, i, 0);
        }
    }
    int main(){
        int n,m;
        scanf("%d%d", &n, &m);
        int tree[n];
        for(int i = 0; i < n; i++){
            scanf("%d", &tree[i]);
        }
        heapSort(tree, n);
         int i;
        for(i = 0; i < m; i++){
            printf("%d ", tree[i]);
        }
    }
```

```cpp
// 01背包
#include<iostream>
using namespace std;

const int N = 1010;
int n, m;
int v[N], w[N];
int f[N][N];

int main(){
    cin >> n >> m;
    for(int i = 1; i <= n; i++) cin >> v[i] >> w[i];

    for(int i = 1; i <= n; i++)
        for(int j = 1; j <= m; j++){
            f[i][j] = f[i - 1][j];
            if(j >= v[i]) f[i][j] = max(f[i][j], f[i - 1][j - v[i]] + w[i]);
        }

    cout << f[n][m] << endl;
    return 0;
}
```