2026 考研数据结构代码题
强化 75 题必背版

来源：b 站：我头发还多还能学

微信：Rain—Splash 或 petrichoryin

注：有些强化题目比较复杂或者过于简单，因此没有编写核心思想描述，大家观看视频理解即可

408 版资料包含基础 63 题+强化 75 题+最终预测 15 题（25 年 10 月份出），每道题均有讲解视频，建议配套使用，有需要的同学可添加微信获取。资料题目的答案均已优化，相比王道更加准确且简洁。***<u>本 pdf 加阴影的题目是重点题，务必可以理解并默写出来！</u>***

强化篇目录

顺序表：

1、将（a1,a2,…am,b1,b2,…bn）转换成(b1,b2,…bn,a1,a2,…am)【逆置】

2、将 n 个整数存放在数组 R 中，设计尽可能高效的算法，将 R 序列循环左移 p 个位置(0<p<n)，即将 R 中的数据由(X0,X1,...Xn-1)变换为(Xp,Xp+1,...Xn-1,X0,X1...Xp-1) ***【2010 统考：逆置】***

3、尽可能高效地从有序表中删除值重复元素，仅保留第一个【有序去重】

4、顺序表中删除值重复的元素，仅保留第一个【无序去重】

5、尽可能高效地从有序表中找出出现次数最多的元素及对应次数。若有多个出现次数最多的优先选取值最小（最大呢?）的元素【经典双指针】

6、一个整数序列 A={a0,a1,...an-1}，其中 0≤ai<n（0≤i<n）。若整数序列中有过半相同元素，则称其为主元素。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 没有主元素。设计算法找出数组 A 中的主元素，若存在主元素则输出，否则输出-1 ***【2013 统考：空换时】***

7、给定一个含 n 个整数的数组，设计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。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* ***【2018 统考：空换时】***

8、两个递增有序表合并成一个递增有序表【有序表合并：双指针】

9、两个递增有序表合并成一个递减有序表【有序表合并：双指针】

10、三个序列 A, B, C 长度均为 n，均为无重复元素的递增序列，设计一个尽可能高效的算法，输出三个序列中共同存在的所有元素【交集：三指针】

11、三元组(a,b,c)的距离 D=|a-b|+|b-c|+|c-a|。给定 3 个非空整数集合 S1,S2 和 S3，按升序分别存储在 3 个数组中。设计一个尽可能高效的算法，计算所有可能的三元组(a,b,c)中的最小距离 ***【2020 统考：三指针】***

12、一个长度为 L 的升序序列 S，处在第 L/2(向上取整)位置的数称为 S 的中位数。例 S1=(11,13,15,17,19)，S1 的中位数为 15，两个序列的中位数是他们所有元素升序序列的中位数。例 S2=(2,4,6,8,20)，则 S1 和 S2 的中位数是 11。现有两个等长的升序序列 A 和 B，设计一个尽可能高效的算法，找出两个序列 A 和 B 的中位数 ***【2011 统考：双指针合并】***

链表：

1、尽可能高效地删除递增链表重复的结点，仅保留第一个【有序去重】

2、删除单链表中重复的结点，仅保留第一个【无序去重】

3、用单链表保存 m 个整数，且|data|≤n，设计时间尽可能高效的算法，对于 data 绝对值相等的点，仅保留第一次出现的点 ***【2015 统考：空换时】***

4、将一个带头结点的单链表 A 分解成两个带头结点的单链表 A 和 B，使 A 中含奇数位置元素，B 中含偶数位置元素，且相对位置不变【链表拆分】

5、将单链表{a1,b1,a2,b2……an,bn}拆分成{ a1,a2……an }和{ bn,bn-1,……b1 }【链表拆分+头插】

6、将一个单链表 A 分解成两个单链表 A 和 B，使 A 中保留值>=0 的结点，B 中保留值<0 的结点。（拓展：A 中包含数字、英文字母和其他字符，将其拆分到 A, B, C 中，使每个链表只包含一类）【链表拆分】

7、两个递增有序的单链表，将 B 合并到 A 后，仍非递减有序【链表合并】

8、两个递增有序的单链表，将 B 合并到 A 中，保证非递增有序【链表合并+头插】

9、A, B 两个单链表递增有序，从 A, B 中找出公共元素产生单链表 C，要求不破坏 A 和 B 结点（即不使用 A、B 原有结点空间）【链表合并+交集】

10、A, B 两个单链表递增有序，从 A, B 中找出公共元素产生单链表 C。要利用 A 和 B 的原有结点空间【链表合并+交集】

11、A, B 两个单链表递增有序，从 A, B 中找出所有元素后再去重并存放于 A 链表中，且要求利用 A 和 B 的原有空间【链表合并+并集】

12、A, B 两个单链表递增有序，从 A, B 中找出 A 有但 B 没有的元素并存放于 A 链表中，且要求利用 A 的原有空间【链表合并+差集】

13、假定采用带头节点的单链表保存单词，当两个单词有相同的后缀时，可共享相同的后缀存储空间。设 str1 和 str2 分别指向两个单词所在链表的头结点，请设计一个时间上尽可能高效的算法，找出这两个链表的共同后缀的起始位置【2012 统考：链表比较】

14、查找单链表中倒数第 k 个结点，若查找成功，则输出该结点的 data，并返回 1，否则返回 0 ***【2009 统考：双指针】***

15、给定一个单链表 $L(a_1, a_2, a_3, \dots, a_n)$,将其重新排列为$(a_1, a_n, a_2, a_{n-1}, \dots)$ ***【2019 统考题：双指针+头插+合并】***

16、两个递增有序的循环单链表，设计算法将 B 合并到 A 中，仍保证一个递增的循环单链表【对比第 7 题】

17、有两个循环单链表，链表头指针分别为 h1,h2，试编写函数将 h2 链表接到 h1 之后，要求链接后仍保持循环链表形式【链表拼接】

18、单链表有环，是指单链表的最后一个结点的指针指向了链表中的某个结点，编写算法判断单链表是否有环【双指针】

树

1、求树的度（树中结点度的最大值为树的度）

2、判断两个二叉树是否相似（拓展：相等呢）

3、(a-(b+c))*(d/e)存储在二叉树，求四则运算后的结果

4、将给定的二叉树转化为等价的中缀表达式 ***（2017 统考：中序遍历）***

5、将二叉树的叶子结点从左向右连接成一个单链表（拓展：双链表呢）

6、输出根结点到每个叶子结点的路径（如果是根结点到值为 x 结点的路径呢？如果是根结点到所有结点的路径呢？均为祖先问题！）

7、使用递归方法输出根结点到叶子结点最长的一条路径

8、增加一个指向双亲结点的 parent 指针，输出所有结点到根结点的路径

9、找到 p 和 q 最近公共祖先结点 r（拓展：多种方法，多种存储结构）

10、试给出自下而上从右到左的层次遍历

11、层次遍历将二叉树所有结点的左右子树交换

12、层次遍历求二叉树中叶子结点个数、度为 1 的结点个数。

13、用层次遍历求解二叉树的高度

14、计算二叉树的带权路径长度（叶子结点）

15、求解二叉树的宽度

16、判断二叉树是否为完全二叉树

17、满二叉树先序序列存在于数组中，设计算法将其变成后序序列

18、先序与中序遍历分别存在两个一维数组 A, B 中，试建立二叉链表

19、二叉树以二叉链表表示，设计算法存储到一维数组中

拓展：二叉树以顺序方式存在于数组 A 的中，设计算法以二叉链表表示

20、二叉树采用顺序存储，统计该树中所有叶子结点的个数

图

1、已知无向连通图 G 由顶点集 V 和边集 E 组成，|E|>0,当 G 中度为奇数的顶点个数为不大于 2 的偶数时，G 存在包含所有边且长度为|E|的路径（称为 EL 路径）。设图 G 采用邻接矩阵存储，设计算法判断图中是否存在 EL 路径，若存在返回 1，否则返回 0。***【2021 年统考题】***

2、有向图 G 采用邻接矩阵存储，将图中出度大于入度的顶点称为 K 顶点。要求输出 G 中所有的 K 顶点，并返回 K 顶点的个数。***【2023 年统考题）***

3、有向图采用邻接表存储，计算每个顶点的入度和出度

4、设计算法判断无向图是否是一棵树

5、输出有向图 Vi 顶点到 Vj 顶点所有简单路径。（拓展：输出 Vi 到 Vj 之间长度为 k 的简单路径）

6、利用 BFS 求不带权的无/有向图中 v 顶点到其他顶点的的最短路径，存储到 d 数组中。（拓展：输出距离顶点 V 的最短路径长度为 k 的所有顶点）

7、Prim 算法求无向图的最小生成树

8、Kruskal 算法求无向图的最小生成树

9、Dijkstra 算法求有/无向图的最短路径

10、Floyed 求有/无向图的最短路径长度（拓展：最短路径与村庄问题！）

11、Floyed 求有/无向图的最短路径长度

12、拓扑排序判断有/无向图是否有环（邻接矩阵）

13、判断领接矩阵存储的有向图的拓扑排序是否唯一***【2024 年统考题】***

查找

1、查找二叉排序树中第 k 小的结点

2、输出二叉搜索树中所有值大于 key 的值

3、判断顺序存储的二叉树是否为二叉搜索树***【2022 年统考题】***

4、判断一个二叉排序树是否为平衡二叉树（拓展：顺序存储呢？）

排序

1、二路归并排序

2、判断数组是否是小根堆

3、堆排序

4、已知由 n(n>=2)个正整数构成的集合 A={ak|0<=ak<n}，将其划分为两个不相交的子集 A1 和 A2，元素个数分别是 n1 和 n2。A1 和 A2 中元素之和分别为 S1 和 S2。设计一个尽可能高效的划分算法，满足 |n1-n2|最小且|S1-S2|最大。***【2016 年统考题，快排 的应用】***

顺序表默认结构体：

```
#define maxsize 50
typedef struct{
    int data[maxsize];
    int length;
} Sqlist;
```

1、将序列（a1,a2,⋯am,b1,b2,⋯bn）转换成(b1,b2,⋯bn,a1,a2,⋯am)

```
void Reverse(int R[], int start, int end){
    int i = start, j = end; //也可以写到 for 循环内
    for (; i < j; ++i, --j){
        int temp = R[i];
        R[i] = R[j];
        R[j] = temp;
    }
}//对撞指针
void change(int R[], int m, int n){
    Reverse(R, 0, m + n - 1);
    Reverse(R, 0, n - 1);
    Reverse(R, n, m + n - 1);
} //时空复杂度分别为 O(N)和 O(1)
```

核心思想：

1、对撞指针交换：使用两个指针 i 和 j，从数组的两端向中间靠拢，交换元素，完成数组的反转

2、整体反转：先将整个数组（包括两部分）反转。数组的顺序就变成了（bn, ..., b1, am, ..., a1）

3、局部反转：再分别对前半部分和后半部分进行反转。前半部分变回原来的顺序，后半部分也恢复到原来的顺序，即将(bn, ..., b1, am, ..., a1)转换成（b1, b2, ..., bn, a1, a2, ..., am)

2、将 n 个整数存放在数组 R 中，设计尽可能高效的算法，将 R 序列循环左移 p 个位置(0<p<n)，即将 R 中的数据由(X0,X1,...Xn-1)变换为(Xp,Xp+1,...Xn-1,X0,X1...Xp-1)（2010 年统考题）

```
void Reverse(int R[], int start, int end){
    int i = start, j = end; //也可以写到 for 循环内
    for (; i < j; ++i, --j){
        int temp = R[i];
        R[i] = R[j];
        R[j] = temp;
    }
}
//要清楚函数每个参数的意思
void change(int R[], int n, int p){
    Reverse(R, 0, n - 1);
    Reverse(R, 0, n - p - 1);
    Reverse(R, n - p, n - 1);
} //时空复杂度分别为 O(N)和 O(1)
```

核心思想：同第一题，这里不过多介绍。

3、尽可能高效地从顺序表中删除所有值重复的元素，仅保留第一个

```
void del(Sqlist *L){
    for (int i = 0; i < L->length; i++){
        int k = i + 1;
        for (int j = i + 1; j < L->length; j++)
            if (L->data[j] != L->data[i])
                L->data[k++] = L->data[j];
        L->length = k;
    }
}//时空复杂度分别为 O(N²)和 O(1)
```

核心思想：

1、逐个检查元素：外层循环遍历顺序表中的每个元素，假设当前元素是第一个出现的元素

2、去除重复元素：内层循环从当前元素的下一个元素开始，检查与当前元素相同的元素。如果找到了不同的元素，就将其移动到当前检查位置 k。这样，重复的元素被覆盖掉

3、更新表长：在内层循环结束后，更新顺序表的长度 为 k，即新的元素个数。此时表中只有第一个出现的元素被保留，重复的元素都被删除

4、尽可能高效地从有序表中删除所有值重复的元素，仅保留第一个

```
void del(Sqlist *L){
    int i = 0;
    int k = i + 1;
    for (int j = i + 1; j < L->length; j++)
        if (L->data[j] != L->data[i]){
            L->data[k++] = L->data[j];
            i++;
        }
    L->length = k;
} //时空复杂度分别为 O(N)和 O(1)
```

核心思想：

1、维护唯一元素位置：通过一个指针 i 指向当前已处理的唯一元素的最后位置。初始时，i 从 0 开始，表示第一个元素

2、遍历并判断元素是否重复：内层循环通过指针 j 遍历剩余的元素，若当前元素 L->data[j] 与已处理的元素 L->data[i] 不相同，说明是一个新的唯一元素。此时，将该元素放到位置 k，并将 k 和 i 都加一，继续重复循环

3、更新顺序表长度：遍历完成后，更新顺序表的长度为 k，重复的元素都被删除

5、尽可能高效的从有序表中找出出现次数最多的元素及对应次数。（若有多个出现次数最多的优先选取值最小的元素）

```
void find(Sqlist L) {
    int max_count = 1;
    int max_val, count = 1;
    int i = 0; //  初始化双指针
    for ( ; i < L.length - 1; i++)
        if (L.data[i] == L.data[i + 1])
            count++;
        else{
            if (count > max_count ) {
                max_count = count;
                max_val = L.data[i];
            }
            count = 1;
        }
    if (count > max_count ) {
        max_count = count;
        max_val = L.data[i];
    }
    printf("对应元素为: %d", max_val);
    printf("最大次数为: %d", max_count);
} //时空复杂度分别为 O(N)和 O(1)
```

1、循环遍历：使用 i 遍历，检查当前元素与下一个元素是否相同。若相同，说明重复出现，count 计数增加；否则表示当前重复元素的序列结束

2、更新最大重复次数：每次序列结束时，比较当前重复次数和最大重复次数，若当前重复次数大于最大重复次数，则更新最大值和最大次数

3、处理末尾元素：遍历结束后，最后一个元素的重复次数可能是最大的，因此需要在循环外再次检查一次，确保最大值和最大次数的更新

6、已知整数序列 A={a0,a1,...an-1}。其中 0≤ai<n（0≤i< n）。若一个整数序列中有过半相同元素，则称其为主元素。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 没有主元素。设计尽可能高效的算法找出数组 A 中的主元素，若存在主元素则输出，否则输出-1（2013 年统考题）

```
int func(int A[], int n){
    int *B = (int *)malloc(sizeof(int) * n);
    for (int i = 0; i < n; i++)
        B[i] = 0;
    for (int i = 0; i < n; i++)
        B[A[i]]++;
    for (int i = 0; i < n; ++i)
        if (B[i] > n/2)
            return i;
    return -1;
}//时空复杂度分别为 O(N)和 O(N)
```

核心思想：

1、创建计数数组：创建一个大小为 n 的数组 B[] 来记录数组 A[] 中每个元素出现的次数

2、统计元素出现次数：遍历数组 A[]，每遇到一个元素 A[i]，就将 B[A[i]]加 1，记录该元素的出现次数

3、查找主元素：遍历计数数组 B[]，如果某个元素的计数大于 n/2，返回该元素作为主元素；如果没有，返回 -1

7、给定一个含 n 个整数的数组，设计一个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：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。（2018 年统考题）

```
int func(int A[], int n){
    int *B = (int *)malloc(sizeof(int)*(n+1));
    for (int i = 0; i < n+1; ++i)
        B[i] = 0;
    for (int i = 0; i < n; ++i)
        if (A[i] > 0 && A[i] <= n)
            B[A[i]]++;
    for (int i = 1; i < n+1; ++i)
        if (B[i] == 0)
            return i;
    return n+1;
}//时空复杂度分别为 O(N)和 O(N)
```

核心思想：

1、创建计数数组：创建一个大小为 n+1 的辅助数组 B[]，用于记录数组 A[]中每个正整数 1 到 n 的出现次数

2、统计出现次数：遍历数组 A[]，将其中每个大于 0 且小于等于 n 元素对应位置的 B[]值加 1。

3、查找最小未出现的正整数：再次遍历 B[]，找到第一个值为 0 的索引，返回对应的正整数；如果没有，返回 n+1

8、两个递增有序表合并成一个递增有序表

```
void merge(Sqlist A, Sqlist B, Sqlist *C){
    int i = 0, j = 0, k = 0;
    while (i < A.length && j < B.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 = A.length + B.length;
} //时空复杂度分别为 O(M+N)和 O(1)
```

核心思想：

1、双指针合并：使用两个指针 i 和 j 分别遍历两个递增有序表 A 和 B，比较每对元素，较小的元素放入结果数组 C 中，同时移动对应的指针

2、处理剩余元素：如果其中一个表已经遍历完，但另一个表仍有剩余元素，直接将剩余部分全部复制到 C 中

3、更新结果长度：合并完成后，更新 C 的长度为 A.length + B.length，即合并后的总长度

9、两个递增有序表合并成一个递减有序表

```
void merge(Sqlist A, Sqlist B, Sqlist *C){
    int i = A.length-1, j = B.length-1, k = 0;
    while (i >= 0 && j >= 0){
        if (A.data[i] > B.data[j])
            C->data[k++] = A.data[i--];
        else
            C->data[k++] = B.data[j--];
    }
    while (i >= 0)
        C->data[k++] = A.data[i--];
    while (j >= 0)
        C->data[k++] = B.data[j--];
    C->length = A.length + B.length;
} //时空复杂度分别为 O(M+N)和 O(1)
```

核心思想：

1、双指针从后向前合并：使用两个指针 i 和 j 从两个递增有序表 A 和 B 的末尾开始遍历，比较每对元素，较大的元素放入结果数组 C 中，并向前移动对应的指针

2、处理剩余元素：当一个表遍历完后，直接将剩余部分的元素（递增有序的）按照递减顺序复制到 C 中

3、更新结果长度：合并完成后，更新 C 的长度为 A.length + B.length，即合并后的总长度

10、三个序列 A, B, C 长度均为 n，且均为无重复元素的递增序列，设计一个尽可能高效的算法，输出三个序列中共同存在的所有元素（三指针）

```
int fmax(int a, int b, int c){
   if (a > b && a > c)
      return a;
   else if (b > a && b > c)
      return b;
   else
      return c;
}
void samekey(int A[], int B[], int C[], int n){
   int i = 0, j = 0, k = 0;
   while (i < n && j < n && k < n)
      if(A[i] == B[j] && A[i] == C[k]){
         printf("%d", A[i]);
         i++;
         j++;
         k++;
      }
      else{
         int max_num = fmax(A[i], B[j], C[k]);
         if (A[i] < max_num)
            i++;
         if (B[j] < max_num)
            j++;
         if (C[k] < max_num)
            k++;
      } //if else 是个整体，while 循环不用加括号
} //时空复杂度分别为 O(N)和 O(1)
```

第 10 题核心思想：

1、三指针遍历：通过三个指针 i, j, k 分别遍历三个递增有序数组 A[], B[], 和 C[]，并依次比较对应位置的元素

2、找共同元素：当 A[i] == B[j] == C[k] 时，说明当前元素在三个序列中都存在，输出该元素，并同时移动三个指针。

3、跳过较小元素：若三个指针指向的元素不相等，找出最大的元素，通过比较当前三个元素，移动指向最小元素的指针，跳过较小的元素，保持比较的高效性

第 11 题最优解核心思想：

1、使用三指针遍历：利用三个指针 i, j, k 分别遍历三个升序有序集合 S1[], S2[]和 S3[]，并依次计算当前三元组（a, b, c）的距离

2、计算距离：对于每一个三元组（S1[i], S2[j], S3[k]），计算其距离 D = |a-b| + |b-c| + |c-a|，并更新最小距离

3、移动指针：根据当前的三个元素，移动指向最小元素的指针，这样能更快速地找到可能更小的距离，减少无效的计算

11、三元组(a,b,c)的距离 D=|a-b|+|b-c|+|c-a|。给定 3 个非空整数集合 S1,S2 和 S3，按升序分别存储在 3 个数组中。设计一个尽可能高效的算法，计算所有可能的三元组(a,b,c)中的最小距离。如 S1={-1,0,9}, S2={-25,-10,10,11},S3={2,9,17,30,41}，则最小距离为 2，对应的三元组为(9,10,9)。（2020 年统考题）

```
int abs(int a){
   if (a < 0)
      return -a;
   else
      return a;
} //abs 函数，c 语言需要调用<stdlib.h>
int f1(int S1[], int m, int S2[], int n, int S3[], int p){
   int i = 0, j = 0, k = 0, min_dis = 1e5;
   while (i < m && j < n && k < p) {
      int dis = abs(S1[i] - S2[j]) + abs(S1[i] - S3[k])
               +abs(S2[j] - S3[k]);
      if (dis < min_dis)
         min_dis = dis;
      if (S1[i] <= S2[j] && S1[i] <= S3[k])
         i++;
      else if (S2[j] <= S1[i] && S2[j] <= S3[k])
         j++;
      else
         k++;
   }
   return min_dis;
} //时空复杂度分别为 O(N)和 O(1)
```

暴力解：

```
int f(int S1[], int m, int S2[], int n, int S3[], int p){
    int min_dis = 1e5;
    for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++)
            for (int k = 0; k < p; k++){
                int dis = abs(S1[i]-S2[j])+
                           abs(S2[j]-S3[k]) +
                           abs(S3[k]-S1[i]);
                if (dis < min_dis)
                    min_dis = dis;
            }
    return min_val;
} //时空复杂度分别为 O(N^3)和 O(1)
```

核心思想：

1、三重循环遍历所有可能三元组：使用三重循环遍历集合 S1[], S2[], 和 S3[] 中的每一对元素，计算所有可能的三元组 (a, b, c)

2、计算三元组距离：对于每个三元组，计算其距离 D = |a-b| + |b-c| + |c-a|，并更新最小距离

3、返回最小距离：在所有三元组中找到最小的距离并返回

12、一个长度为 L 的升序序列 S，处在第 L/2(向上取整)个位置的数称为 S 的中位数。例如 S1=(11,13,15,17,19)，S1 的中位数为 15，两个序列的中位数是含他们所有元素升序序列的中位数。例如，若 S2=(2,4,6,8,20)，则 S1 和 S2 的中位数是 11。现在有两个等长的升序序列 A 和 B，试设计一个在时间和空间都尽可能高效的算法，找出两个序列 A 和 B 的中位数（2011 年统考题）

```
int Find(Sqlist A, Sqlist B){
    Sqlist C;
    int i = 0, j = 0, k = 0;
    while (i < A.length && j < B.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++];
    return C.data[k/2 - 1];
} //时空复杂度分别为 O(N)和 O(N)
```

核心思想：

1、合并两个升序序列：通过使用两个指针 i 和 j，将两个升序序列合并成一个新的升序序列 C

2、直接返回其中位数，直接返回 C[k/2 - 1]即可

13、将数组中所有负数移动到数组最前端，所有非负数移动到数组的最后端

```
void func(int A[], int n){
        int left = 0, right = n - 1;
        while(left < right){
                while(A[left] < 0 && left < right)
                        left++;
                while(A[right] >= 0 && left < right)
                        right--;
                if (left < right){
                        int temp = A[left];
                        A[left] = A[right];
                        A[right] = temp;
                }
        }
}
```

核心思想：

1、双指针方法：使用两个指针 left 和 right，left 指向数组的开头，right 指向数组的末尾

2、左指针移动负数：left 指针向右移动，跳过所有负数元素，直到遇到非负数或者与 right 指针重合

3、右指针移动非负数：right 指针向左移动，跳过所有非负数元素，直到遇到负数或者与 left 指针重合

4、交换元素：当 left 和 right 指针指向的元素需要交换时，交换它们的值，继续向中间推进指针，直到 left 不再小于 right

1、尽可能高效地删除递增链表中重复的结点，仅保留第一个。

```
void del(LinkList L){
    LinkList p = L->next;
    while (p->next != NULL){
        LinkList q = p->next;
        if (p->data == q->data){
            p->next = q->next;
            free(q);
        }
        else
            p = p->next;
    }
} //时空复杂度分别为 O(N)和 O(1)
```

核心思想：

1、遍历链表：使用指针 p 遍历链表，从第二个节点开始，逐个比较相邻节点的数据

2、删除重复节点：当发现相邻的两个节点数据相同时，删除后一个节点，并将前一个节点的 next 指针指向下一个节点。这样，删除了重复节点，保留了第一个出现的节点

3、继续遍历：如果当前节点与下一个节点的数据不同，则移动指针 p 到下一个节点，继续比较

2、删除单链表中重复的结点，仅保留第一个

```
void del(LinkList L){
    LinkList p = L->next;
    while (p != NULL){
        LinkList r = p;
        while (r->next != NULL)
            if (r->next->data == p->data ){
                LinkList q = r->next;
                r->next = q->next;
                free(q);
            }
            else
                r = r->next;
        p = p->next;
    }
} //时空复杂度分别为 O(N^2)和 O(1)
```

1、外层遍历：使用指针 p 遍历链表中的每个节点，从第二个节点开始，逐个检查是否存在与当前节点 p 相同数据的后续节点。

2、内层检查重复节点：用指针 r 从 p 的下一个节点开始，检查其后续节点是否与当前节点 p 数据相同。若相同，删除重复节点，将 r 的 next 指向下一个节点，并释放其内存

3、继续遍历：如果当前节点 p 和后续节点的数据不同，则继续检查下一个节点；外层指针 p 每次向后移动，直至遍历完整个链表

3、用单链表保存 m 个整数，且|data|≤n，设计时间上尽可能高效的算法，对于 data 绝对值相等的点，仅保留第一次出现的点（2015 统考题）

```
void del(LinkList L, int n){
    int *A = (int *)malloc( sizeof(int)* (n + 1) );
    for (int i = 0; i < n + 1; ++i)
        A[i] = 0;
    LinkList p = L;
    int m;
    while (p->next != NULL){
        if (p->next->data >= 0)
            m = p->next->data;
        else
            m = -p->next->data;
        if (A[m] == 0){
            A[m] = 1;    p = p->next;
        }
        else{
            LinkList r = p->next;
            p->next = r->next;
            free(r);
        }
    }
} //时空复杂度分别为 O(N)和 O(N)
```

1、初始化数组 A：创建数组 A，所有元素初始化为 0。A[i]用来标记值 i 是否已经出现过

2、遍历链表：使用指针 p 遍历链表，对于每个节点，取其值的绝对值 m，并检查 A[m]是否为 0（表示 m 未出现过）。如果为 0，则标记该值已出现，并移动 p 指针到下一个节点

3、删除重复节点：如果节点值 m 已经在数组 A 中标记过，说明该节点是重复节点，需要删除。通过调整指针将当前节点删除并释放内存

4、将一个带头结点的单链表 A 分解成两个带头结点的单链表 A 和 B，使 A 中保留值大于等于 0 的结点，B 中为小于 0 的结点。

```
LinkList split(LinkList A){
    LinkList B = (LinkList)malloc(sizeof(LNode));
    LinkList p = A, rb = B;
    while (p->next != NULL){
        if (p->next->data >= 0)
            p = p->next;
        else{
            LinkList r = p->next;
            p->next = r->next;
            rb->next = r;
            rb = r;
        }
    }
    rb->next = NULL;
    return B;
} //时空复杂度分别为 O(N)和 O(1)
```

核心思想：

1、初始化链表：创建一个新的链表 B，初始化指针 p 遍历链表 A，指针 rb 用于构建链表 B

2、遍历链表 A： 遍历链表 A，如果节点值小于 0，将该节点从链表 A 删除并添加到链表 B，否则保持在链表 A 中

3、返回链表 B：遍历结束后，确保链表 B 末尾指针为 NULL，返回链表 B

5、将一个带头结点的单链表 A 分解成两个带头结点的单链表 A 和 B，使 A 中含奇数位置元素，B 中含偶数位置元素，且相对位置不变

```
LinkList split(LinkList A){
    LinkList B = (LinkList)malloc(sizeof(LNode));
    LinkList rb = B, p = A;
    int pos = 1;
    while (p->next != NULL){
        if (pos % 2 == 1)
            p = p->next;
        else{
            LinkList r = p->next;
            p->next = r->next;
            rb->next = r;
            rb = r;
        }
        pos++;
    }
    rb->next = NULL;
    return B;
} //时空复杂度分别为 O(N)和 O(1)
```

1. 遍历链表并判断位置：从头指针 A 开始遍历，用 pos 记录当前位置编号，每次循环判断当前是否为奇数位置，是的话仅将指针 p 后移一位；否则进入偶数位置处理逻辑

2. 偶数位置摘节点到 B 链表：当 pos 为偶数时，将该节点从原链表中断开，并接到链表 B 的尾部，实现摘节点并挂到 B 上。

3. 循环结束处理收尾：最后将 B 链表尾节点空，避免出现脏数据。函数返回新链表 B 的头指针

6、将一个单链表{a1,b1,a2,b2……an,bn}拆分成{ a1,a2……an }和{ bn,bn-1,……b1 }

```
LinkList split(LinkList A){
    LinkList B = (LinkList)malloc(sizeof(LNode));
    B->next = NULL;
    LinkList p = A;
    int pos = 1;
  while (p->next != NULL){
      if (pos % 2 == 1)
          p = p->next;
      else{
          LinkList r = p->next;
          p->next = r->next;
          r->next = B->next;
          B->next = r;
      }
      pos++;
    }
    return B;
}//时空复杂度分别为 O(N)和 O(1)
```

1. 遍历链表按位置分类：从头指针 A 开始，用 pos 记录当前位置，奇数位置跳过，偶数位置节点需要摘出处理

2. 偶数节点头插进 B 链表：将偶数位置节点从 A 中断开后，使用头插法插入到 B 链表头部，实现逆序构建 {bn, ..., b1}

3. 构建两个链表并返回：最终 A 保留奇数位置节点 {a1, ..., an}，B 为逆序的偶数位置节点，函数返回 B 的头指针

7、两个递增有序的单链表，设计算法将 B 合并到 A 中，仍保证一个非递减有序的链表（双指针基础，很重要！）

```
void merge(LinkList A, LinkList B){
    LinkList ra = A, p= A->next, q = B->next;
    while (p != NULL && q != NULL){
        if (p->data <= q->data){
            ra->next = p;
            ra = p;
            p = p->next;
        }
        else{
            ra->next = q;
            ra = q;
            q = q->next;
        }
    }
    if (p != NULL)
        ra->next = p;
    if (q != NULL)
        ra->next = q;
} //时空复杂度分别为 O(M+N))和 O(1)
```

1、双指针比较合并：用指针 p 和 q 同时遍历 A 和 B，比较当前节点值，将较小节点接到结果链表后，保持非递减顺序

2、尾指针维护链表：用 ra 记录合并链表的最后一个节点，实现原地合并，无需新建节点

3、拼接剩余部分：当一个链表遍历完后，直接将另一个链表剩余部分连接到结果链表末尾，确保完整合并

8、两个递增有序的单链表，设计算法将 B 合并到 A 中，保证一个非递增有序的链表

```
void merge(LinkList A, LinkList B){
    LinkList p = A->next, q = B->next, r;
    A->next = NULL;
    B->next = NULL;
    while (p != NULL && q != NULL){
        if (p->data <= q->data){
            r = p;   //我的 r 在第一行定义了哦
            p = p->next;
        }
        else{
            r = q;
            q = q->next;
        }
        r->next = A->next;
        A->next = r;
    }
    while (p != NULL){
        r = p;   p = p->next;
        r->next = A->next;
        A->next = r;
    }
    while (q != NULL){
        r = q;   q = q->next;
        r->next = A->next;
        A->next = r;
    }
} //时空复杂度分别为 O(M+N)和 O(1)
```

核心思想：

1、头插法逆序构建链表：在合并过程中始终将较小的节点用头插法插入到 A 的表头，逐步构建一个非递增链表

2、遍历比较并插入：用指针 p 和 q 分别遍历 A 和 B，依次比较当前节点，将较小节点插入到 A 的头部，实现从大到小排序

3、处理剩余节点：当某一链表先遍历完后，继续将另一链表剩下的节点依次头插到 A 中，完成所有节点的合并

9、A, B 两个单链表递增有序，从 A, B 中找出公共元素产生单链表 C，要求：不破坏 A 和 B 结点（交集）

```
LinkList common(LinkList A, LinkList B){
    LinkList C = (LinkList)malloc(sizeof(LNode));
    LinkList p = A->next, q = B->next, rc = C;
    while (p != NULL && q != NULL){
        if (p->data > q->data)
            q = q->next;
        else if (p->data < q->data)
            p = p->next;
        else{ //下面这行位置不够所以没对齐 hh
        LinkList r = (LinkList)malloc(sizeof(LNode));
            r->data = p->data;
            rc->next = r;
            rc = r;
            p = p->next;
            q = q->next;
        }
    }
    rc->next = NULL;
    return C;
} //时空复杂度分别为 O(M+N)和 O(1)
```

核心思想：

1、双指针同步遍历：用 p 和 q 同时遍历 A 和 B，通过比较 data 值跳过不相等的节点，只处理相等的公共元素

2、只复制不破坏原链表：当找到相同元素时，动态创建新节点加入结果链表 C，不修改 A、B 中任何节点，满足"不破坏原链表"的要求

3、保持有序、构造新链表：利用 rc 指针始终指向 C 的尾部，依次接入新节点，保证 C 也是递增有序链表

10、A, B 两个单链表递增有序，从 A, B 中找出公共元素存放到 A 链表中。要利用 A 链表的原有结点空间（交集）

```
void common(LinkList A, LinkList B){
    LinkList p = A, q = B->next;
    while (p->next != NULL && q != NULL){
        if (p->next->data > q->data)
            q = q->next;
        else if (p->next->data < q->data){
            LinkList r = p->next;
            p->next = r->next;
            free(r);
        }
        else{
            p = p->next;
            q = q->next;
        }
    }
    if (p->next != NULL)
        p->next = NULL;
} //时空复杂度分别为 O(M+N)和 O(1)
```

核心思想：

1、双指针对比遍历：用 p->next 和 q 比较 A 和 B 的元素值，逐步同步推进，寻找公共元素

2、就地保留交集元素：当 A 中元素不属于交集时，直接删除 p->next 结点，复用原链表空间，避免创建新节点

3、截断多余尾部：当 B 遍历结束而 A 还有剩余节点时，将其直接截断，确保 A 中只保留公共元素，构成交集链表

11、A, B 两个单链表递增有序，从 A, B 中找出 A 有但 B 没有的元素并存放于 A 链表中，且要求利用 A 的原有空间（差集 A-B）

```
void Diff(LinkList A, LinkList B){
    LinkList p = A, q = B->next;
    while (p->next != NULL && q != NULL)
        if (p->next->data > q->data){
            q = q->next;
        }
        else if (p->next->data < q->data){
            p = p->next;
        }
        else{
            LinkList r = p->next;
            p->next = r->next;
            free(r);
            q = q->next;
        }
}//时空复杂度分别为 O(M+N)和 O(1)
```

核心思想：

1、双指针同步遍历：用 p->next 和 q 比较 A、B 的元素值，依序查找 A 中独有的元素

2、删除公共节点：当遇到 A 和 B 中相同的元素时，从 A 中删除该节点，实现差集效果（只保留 A 有但 B 没有的）

3、原地操作节省空间：复用 A 链表原有节点空间，边遍历边修改，不创建新节点，空间复杂度为 O(1)

12、A, B 两个单链表递增有序，从 A, B 中找出所有元素后再去重并存放于 A 链表中，且要求利用 A 和 B 的原有空间（并集）

```
void Union(LinkList A, LinkList B){
   LinkList ra = A, p = A->next, q = B->next;
   B->next = NULL;
   while (p != NULL && q != NULL){
      if (p->data < q->data){
         ra->next = p;
         ra = p;    p = p->next;
      }
      else if (p->data > q->data){
         ra->next = q;
         ra = q;    q = q->next;
      }
      else{
         ra->next = p;
         ra = p;
         p = p->next;
         LinkList r = q;
         q = q->next;
         r->next = NULL;
         free(r);
      }
   }
   if(p != NULL)
      ra->next = p;
   if(q != NULL)
      ra->next = q;
} //时空复杂度分别为 O(M+N)和 O(1)
```

第 12 题核心思想：

1、双指针归并去重：用 p 和 q 同步遍历 A 和 B，比较当前节点值，较小者接入结果链表，相等则保留一个并释放另一个，去除重复

2、原地合并复用空间：通过尾指针 ra 构建新链表，优先复用 A、B 原有节点，不新建结点，满足空间复用要求

3、处理剩余节点：当其中一个链表遍历完后，将另一链表剩余部分直接链接到结果链表末尾，完成并集合并

第 13 题核心思想：

1、先求链表长度：分别遍历两个链表，统计长度 len1 和 len2，用于后续对齐操作

2、对齐两个链表：将较长链表先走差值步，使两个指针从"末尾等距"的位置同时出发，为比较后缀做准备

3、同步查找公共尾：同时遍历两个链表，若某一节点地址相同（即 p == q），说明找到公共后缀的起始位置。若无，则返回 NULL

13、假定采用带头节点的单链表保存单词，当两个单词有相同的后缀时，可共享相同的后缀存储空间。设 str1 和 str2 分别指向两个单词所在链表的头结点，请设计一个时间上尽可能高效的算法，找出这两个链表的共同后缀的起始位置。（2012 年统考题）

```
LinkList find(LinkList str1, LinkList str2){
   LinkList p = str1->next, q = str2->next;
   int len1 = 0, len2 = 0;
   while (p != NULL){
      p = p->next;
      len1++;
   }
   while (q != NULL){
      q = q->next;
      len2++;
   }
   for (p = str1->next; len1 > len2; len2++)
      p = p->next;
   for (q = str2->next; len2 > len1; len1++)
      q = q->next;
   while(p != NULL && q != NULL){
      if (p == q)
         return p;
      p = p->next;
      q = q->next;
   }
   return NULL;
} //时空复杂度分别为 O(max(M,N))和 O(1)
```

14、查找单链表中倒数第 k 个结点，若查找成功，则输出该结点的 data，并返回 1，否则返回 0（2009 年统考题）

```
//法一最优解（双指针思想）
int find(LinkList L, int k){
   LinkList p = L, q = L;
   int i = 0;
   while (q != NULL){
      q = q->next;
      i++;
      if (i > k)
         p = p->next;
   }
   if (p == L || k <= 0)
      return 0;
   else{
      printf("%d", p->data);
      return 1;
   }
} //时空复杂度分别为 O(N)和 O(1)
```

核心思想：

1、双指针拉开间距：使用两个指针 p 和 q，先让 q 向前走 k 步，建立 p 和 q 间距。

2、同步前进找位置：当 q 继续向前走时，p 也同步移动，直到 q 为 NULL，此时 p 指向倒数第 k 个结点。

3、合理性判断与输出：若 k 非法或链表长度不足 k，返回 0，否则输出 p->data 并返回 1。

```
//法二暴力解
int len(LinkList L){
   LinkList p = L->next;
   int n = 0;
   while (p != NULL){
      p = p->next;
      n++;
   }
   return n;
}//先统计单链表长度

int find(LinkList L, int k){
   if (k > len(L) || k <= 0)
      return 0;
   int m = len(L) - k + 1;
   int i = 1;
   LinkList p = L->next;
   while (i < m) {
      p = p->next;
      i++;
   }
   printf("%d", p->data);
   return 1;
} //时空复杂度分别为 O(N)和 O(1)
```

第 14 题暴力解核心思想：

1、先计算链表长度：用 len 函数遍历链表，统计节点总数

2、计算目标位置：根据倒数第 k 个节点转换成正数第 m = len - k + 1 个节点

3、遍历到目标节点输出：从头遍历链表到第 m 个节点，打印数据并返回成功，否则返回失败

第 15 题核心思想：

1、快慢指针找中点：利用快慢指针将链表分成前后两段，slow 最终指向中点，实现链表对半划分

2、逆置后半部分：将后半段链表逆序插入到 slow 后，实现后半段原地逆置，方便后续交叉合并

3、交替合并两段：将前半段和逆置后的后半段节点依次穿插链接，形成 a1, an, a2, an-1... 的排列顺序

15、给定一个单链表 L(a_1, a_2, a_3,..., a_n),将其重新排列为(a_1, a_n, a_2, a_{n-1},...)（2019 统考题）

```
void func(LinkList L){
   LinkList slow = L, fast = L;
   while (fast != NULL && fast->next != NULL){
      slow = slow->next;
      fast = fast->next->next;
   }
   LinkList p = slow->next;
   slow->next = NULL;
   while (p != NULL){
      LinkList r = p->next;
      p->next = slow->next;
      slow->next = p;
      p = r;
   } //逆置
   LinkList mid = slow->next;
   slow->next = NULL;
   LinkList beg = L->next;
   while (mid != NULL){
      LinkList r= mid->next;
      mid->next = beg->next;
      beg->next = mid;
      beg = mid->next;
      mid = r;
   }
} //时空复杂度分别为 O(N)和 O(1)
```

16、两个递增有序的循环单链表，设计算法将 B 合并到 A 中，仍保证一个非递减的循环单链表

```
void merge(LinkList A, LinkList B){
   LinkList ra = A, p = A->next, q = B->next;
   B->next = NULL;
   while (p != A && q != B){
      if (p->data <= q->data){
         ra->next = p;
         ra = p;
         p = p->next;
      }
      else{
         ra->next = q;
         ra = q;
         q = q->next;
      }
   }
   if (p != A)
      ra->next = p;
   if (q != B){
      ra->next = q;
      while (q->next != B)
         q = q->next;
      q->next = A;
   }
} //时空复杂度分别为 O(M+N)和 O(1)
```

第 16 题核心思想：

1、打破循环便于合并：先将循环链表 B 的尾指针断开，临时变为普通单链表，便于按顺序合并

2、归并连接两表：使用三个指针 p、q、ra，按递增顺序将 A、B 中结点逐一连接到 A 中，类似归并排序

3、恢复循环结构：若 B 仍有剩余结点，接到 A 后，并遍历 B 找到尾结点，将其 next 指向 A，恢复循环链表结构

17、有两个循环单链表，链表头指针分别为 h1,h2，试编写函数将 h2 链表接到 h1 之后，要求链接后仍保持循环链表形式（保留 h1 头结点，剔除 h2）

```
void link(LinkList h1, LinkList h2){
   LinkList p, q;
   p = h1, q = h2;
   while (p->next != h1)
      p = p->next;
   while (q->next != h2)
      q = q->next;
   p->next = h2->next;
   q->next = h1;
} //时空复杂度分别为 O(max(M,N))和 O(1)
```

1、找到两个链表的尾节点：遍历 h1 和 h2，分别找到以 h1、h2 为头循环链表的尾节点 p 和 q

2、连接链表内容部分：将 h1 的尾节点 p 指向 h2 的第一个有效结点（即 h2->next），完成链表内容拼接

3、恢复循环并剔除 h2：将 h2 的尾节点 q 的 next 指向 h1，构成新的循环链表，原 h2 的头结点被剔除

18、单链表有环，是指单链表的最后一个结点的指针指向了链表中的某个结点，编写算法判断单链表是否有环

```
bool findloop(LinkList L){
    LinkList fast = L, slow = L;
    while (fast != NULL && fast->next != NULL){
        slow = slow->next;
        fast = fast->next->next;
        if (slow == fast)
            return true;//有环
    }
    return false;//无环
} //时空复杂度分别为 O(N)和 O(1)
```

核心思想：

1、设置快慢指针：使用两个指针 slow 和 fast，初始都指向链表头，slow 每次走一步，fast 每次走两步

2、判断是否相遇：如果链表中存在环，fast 会在环中追上 slow，出现 slow == fast，说明链表有环

3、遍历终止无环：若 fast 或 fast->next 为 NULL，说明遍历到链表尾部，没有环

树的结构体：（二叉链表形式）

```
typedef struct BTNode{
    char data;
    struct BTNode *lchild, *rchild;
} BTNode, *BiTree;
```

1、求树的度（树中结点度取最大值为树的度）

```
void func(BiTree p, int *max_degree){
    if (p != NULL){
        int degree;
        if (p->lchild && p->rchild)
            degree = 2;
        else if (p->lchild || p->rchild)
            degree = 1;
        else
            degree = 0;
        if (degree > *max_degree)
            *max_degree = degree;
        func(p->lchild, max_degree);
        func(p->rchild, max_degree);
    }
} //操作型
//时空复杂度分别为 O(N)和 O(N)
```

树的递归这部分的核心思想不展开描述，直接看视频，操作型和计算型两种都要掌握！

```
int func(BiTree p){
    if (p == NULL || (!p->lchild && !p->rchild))
        return 0;
    if (p->lchild && p->rchild)
        return 2;
    if (p->lchild || p->rchild){
        int A = func(p->lchild);
        int B = func(p->rchild);
        int max_degree = A > B ? A : B;
        return max_degree > 1 ? max_degree : 1;
    }
} //计算型
```

2、判断两个二叉树是否相似

```
int func(BiTree T1, BiTree T2){
    if (T1 == NULL && T2 == NULL)
        return 1;
    if (T1 == NULL || T2 == NULL)
        return 0;
    if (T1 ->data != T2->data)
        return 0; //本语句在判断两树相等时再加
    int left = func(T1->lchild, T2->lchild);
    int right = func(T1->rchild, T2->rchild);
    return left && right;
} //时空复杂度分别为 O(N)和 O(N)
```

3、将给定的二叉树转化为等价的中缀表达式(具体细节图在视频中会提到)（2017 年统考题）

```
void func(BiTree p, int *deep){
    if (p != NULL){
        ++(*deep);
        if ((p->lchild && p->rchild ) && *deep>1)
            printf("(");
        func(p->lchild, deep);
        printf("%c", p->data);
        func(p->rchild, deep);
        if ((p->lchild && p->rchild ) && *deep>1)
            printf(")");
        --(*deep);
    }
} //时空复杂度分别为 O(N)和 O(N)
```

4、(a-(b+c))*(d/e)存储在二叉树(图在视频里出现)，求四则运算后的结果

```
int func(BiTree p){
    if (p == NULL)
        return 0;
    if (!p->lchild && !p->rchild)
        return p->data - '0';
    if (p->lchild && p->rchild){
        int A = func(p->lchild);
        int B = func(p->rchild);
        if (p->data == '+')
            return A + B;
        else if (p->data == '-')
            return A - B;
        else if (p->data == '*')
            return A * B;
        else if (p->data == '/')
            return A / B;
    }
} //时空复杂度分别为 O(N)和 O(N)
```

5、将一个二叉树的叶子结点从左向右连接成一个单链表(head 指向第一个，tail 指向最后一个)

```
void link(BiTree p, BiTree *head, BiTree *tail){
    if (p != NULL){
        if (!p->lchild && !p->rchild)
            if (*head == NULL){
                *head = p; *tail = p;
            }else{
                (*tail)->rchild = p;
                *tail = p;
            }
        link(p->lchild, head, tail);
        link(p->rchild, head, tail);
    }
}//时空复杂度分别为 O(N)和 O(N)

void link(BiTree p, BiTree *head, BiTree *tail){
    if (p != NULL){
        link(p->lchild, head, tail);
        link(p->rchild, head, tail);
        if (!p->lchild && !p->rchild)
            if (*head == NULL){
                *head = p; *tail = p;
            }else{
                (*tail)->rchild = p;
                p->lchild = *tail;
                *tail = p;
            }
    }
}
```

6、输出根结点到每个叶子结点的路径

```
void allpath(BiTree p, char st[], int *top){
    if (p != NULL){
        st[++(*top)] = p->data;
        if (!p->lchild && !p->rchild){
            for (int i = 0; i <= *top; i++)
                printf("%c", st[i]);
        }
        allpath(p->lchild, st, top);
        allpath(p->rchild, st, top);
        --(*top);
    }
}//根结点到每个叶子点的顺序
//时空复杂度分别为 O(N)和 O(N)

void allpath(BiTree p, char st[], int* top){
    if (p != NULL){
        st[++(*top)] = p->data;
        if (!p->lchild && !p->rchild){
            for (int i = *top; i >= 0 ; i--)
                printf("%c", st[i]);
        }
        allpath(p->lchild, st, top);
        allpath(p->rchild, st, top);
        (*top)--;
    }
}//每个叶子到根结点的顺序
```

如果是根结点到值为 x 结点的路径呢？根结点到所有结点的路径呢？均为祖先问题，直接秒杀！

7、使用递归方法输出根结点到叶子结点最长的一条路径。

```
void allpath(BiTree p, char st[], int *top, int *max, char res[]) {
    if (p != NULL){
        st[++(*top)] = p->data;
        if (!p->lchild && !p->rchild) {
            if (*top > *max) {
                *max = *top;
                for (int i = 0; i <= *top; i++)
                    res[i] = st[i];
            }
        }
        allpath(p->lchild, st, top, max, res);
        allpath(p->rchild, st, top, max, res);
        (*top)--;
    }
} //时空复杂度分别为 O(N)和 O(N)
```

8、增加一个指向双亲结点的 parent 指针，输出所有结点到根结点的路径

```
typedef struct BTNode{
    char data;
    struct BTNode *lchild, *rchild, *parent;
} BTNode, *BiTree;
void fun(BiTree p, BiTree q){
    if (p != NULL){
        p->parent = q ;
        q = p ;
        fun(p->lchild, q) ;
        fun(p->rchild, q) ;
    }
}
void allpath(BiTree p){
    if (p != NULL){
        BiTree q = p;
        while (q != NULL){
            printf("%c ",q->data);
            q = q->parent;
        }
        allpath(p->lchild);
        allpath(p->rchild);
    }
} //时空复杂度分别为 O(N)和 O(N)
```

9、找到 p 和 q 最近公共祖先结点 r

```
BiTree find_LCA(BiTree root, BiTree p, BiTree q) {
    if (root==NULL || root == p || root == q)
        return root;
    BiTree left = find_LCA(root->lchild, p, q);
    BiTree right = find_LCA(root->rchild, p, q);
    if (left == NULL && right == NULL)
        return NULL;
    if (left != NULL && right != NULL)
        return root;
    return left != NULL ? left : right;
}//递归思想
//时空复杂度分别为 O(N)和 O(N)
```

如果二叉树顺序存储呢？（408 必会）

```
BiTree find_LCA(BiTree aa[], int i, int j){
    while (i != j){
        if (j > i)
            j = (j - 1) / 2;
        else
            i = (i - 1) / 2;
    }
    return aa[i];
} //时空复杂度分别为 O(logN)和 O(1)
```

10、试给出自下而上从右到左的层次遍历

```
void level(BiTree bt){
    BiTree que[maxsize], st[maxsize];
    int front = 0, rear = 0, top = -1;
    if (bt != NULL){
        que[++rear] = bt;
        while (front != rear){
            bt = que[++front];
            st[++top] = bt;
            if (bt->lchild != NULL)
                que[++rear] = bt->lchild;
            if (bt->rchild != NULL)
                que[++rear] = bt->rchild;
        }
    }
    while (top != -1)
        printf("%c", st[top--]->data);
} //时空复杂度分别为 O(N)和 O(N)
```

核心思想：

1、正序层序遍历入队入栈：使用队列按正常层序遍历顺序（从上到下、从左到右）访问结点，并依次压入栈中

2、利用栈实现反序输出：出栈的顺序即为从下到上、从右到左，达到所需遍历顺序

3、借助两个辅助结构：分别用队列进行层次遍历，用栈反转访问顺序，实现自下而上的从右到左层序遍历

11、层次遍历将二叉树所有结点的左右子树交换

```
void level(BiTree bt) {
    BiTree que[maxsize];
    int front = 0, rear = 0;
    if (bt != NULL){
        que[++rear] = bt;
        while (front != rear){
            bt = que[++front];
            BiTree temp = bt->lchild;
            bt->lchild = bt->rchild;
            bt->rchild = temp;
            if (bt->lchild != NULL)
                que[++rear] = bt->lchild;
            if (bt->rchild != NULL)
                que[++rear] = bt->rchild;
        }
    }
} //时空复杂度分别为 O(N)和 O(N)
```

核心思想：

1、层序遍历整棵树： 使用队列进行标准的层序遍历，依次访问树中每个结点

2、每个结点交换左右子树： 在访问到每个结点时，交换它的左右孩子，实现整棵树的镜像翻转

12、层次遍历求二叉树中叶子结点个数

```
void level(BiTree bt, int *n) {
    BiTree que[maxsize];
    int front = 0, rear = 0;
    if (bt != NULL){
        que[++rear] = bt;
        while (front != rear) {
            bt = que[++front];
            if (!bt ->lchild && !bt->rchild)
                ++(*n);
            if (bt ->lchild != NULL)
                que[++rear] = bt ->lchild;
            if (bt ->rchild != NULL)
                que[++rear] = bt ->rchild;
        }
    }
} //时空复杂度分别为 O(N)和 O(N)
```

核心思想：

1、层序遍历整棵树：利用队列从根节点开始按层访问所有结点。

2、判断并统计叶子结点：如果当前结点没有左、右孩子，则将叶子结点数量 *n 加一

3、通过参数指针返回结果：使用整型指针 *n 将叶子结点的总数返回给调用者

13、层次遍历求解二叉树的高度

```
int level(BiTree bt){
    BiTree que[maxsize];
    int front = 0, rear = 0, last = 1, level = 0;
    if (bt != NULL){
        que[++rear] = bt;
        while (front != rear){
            bt = que[++front];
            if (bt->lchild != NULL)
                que[++rear] = bt->lchild;
            if (bt->rchild != NULL)
                que[++rear] = bt->rchild;
            if (front == last){
                level++;
                last = rear;
            }
        }
        return level;
    }
} //时空复杂度分别为 O(N)和 O(N)
```

核心思想：

1、层序遍历整个树：使用队列从根节点逐层遍历所有结点。

2、用 last 记录当前层的最后一个结点索引：当 front == last 时说明该层访问结束，层数加一，更新 last

3、最终返回遍历的层数：即二叉树的高度。整个过程不依赖递归，适合非递归实现。

14、计算二叉树的带权路径长度（叶子结点）

```
int func(BiTree bt){
    BiTree que[maxsize];
    int front=0, rear=0, wpl=0, last = 1, level = 0;
    if (bt != NULL){
        que[++rear] = bt;
        while (front != rear){
            bt = que[++front];
            if (!bt->lchild && !bt->rchild)
                wpl += level * (bt->data);
            if (bt->lchild != NULL)
                que[++rear] = bt->lchild;
            if (bt->rchild != NULL)
                que[++rear] = bt->rchild;
            if (front == last){
                level++;
                last = rear;
            }
        }
    }
    return wpl;
}//法一，层次遍历思想
```

核心思想：

1、按层序遍历整棵树：使用队列进行广度优先搜索，同时用 level 记录当前层数

2、只计算叶子结点权值：对于没有左右孩子的结点（即叶子），将其权值乘以所在层数并累加到 wpl 中

3、最终返回总权重路径长度：所有叶子结点的权值 × 深度 的累加值即为树的 WPL

```
void func(BiTree bt, int *deep, int *wpl){
    if (bt != NULL){
        ++(*deep);
        if (!bt->lchild && !bt->rchild)
            *wpl += (bt->data) * (*deep - 1);
        func(bt->lchild, deep, wpl);
        func(bt->rchild, deep, wpl);
        --(*deep);
    }
} //法二，递归遍历思想
```

需要讲义对应讲解视频，可加下方微信获取

15、求解二叉树的宽度

```
int level(BiTree bt){
    BiTree que[maxsize];
    int front=0, rear=0, res=0, last=1, width=0;
    if (bt != NULL){
        que[++rear] = bt;
        while (front != rear){
            bt = que[++front];
            res++;
            if (bt->lchild != NULL)
                que[++rear] = bt->lchild;
            if (bt->rchild != NULL)
                que[++rear] = bt->rchild;
            if (front == last){
                if (res > width)
                    width = res;
                res=0;
                last = rear;
            }
        }
    }
    return width;
} //法一，层次遍历思想
```

核心思想：

1、层序遍历整个树

2、res 计数每层结点数：每遇到一个结点就加一，等当前层结束后（front == last））更新最大宽度 width

3、更新最大值：比较当前层结点数与最大值并更新，然后重置 res 为 0，同时更新 last

```
void func(BiTree bt, int A[], int *L){
    if (bt != NULL){
        ++(*L);
        A[(*L)]++;
        func(bt->lchild, A, L);
        func(bt->rchild, A, L);
        --(*L);
    }
}
int max_width(BiTree bt){
    int A[maxsize] = {0};
    int L = 0;
    func(bt, A, &L);
    int width = 0;
    for (int i = 1; i < maxsize; i++)
        if (A[i] > width)
            width = A[i];
    return width;
} //法二，递归遍历思想
```

16、判断二叉树是否为完全二叉树

```
bool fun(BiTree bt){
    BiTree que[maxsize];
    int front = 0, rear = 0;
    if (bt != NULL){
        que[++rear] = bt;
        while (front != rear){
            bt = que[++front];
            if (bt != NULL){
                que[++rear] = bt->lchild;
                que[++rear] = bt->rchild;
            }
            else{
                while (front != rear)
                    if (que[++front] != NULL)
                        return false;
            }
        }
    }
    return true;
} //时空复杂度分别为 O(N)和 O(N)
```

核心思想：

1、层序遍历判断结构完整性：使用队列逐层遍历所有节点

2、遇到空节点后检查后续：一旦出现 NULL，则之后所有节点都必须为 NULL，否则不是完全二叉树

3、满足条件返回 true，否则 false 若遍历中违反完全二叉树结构，立即返回 false，否则最后返回 true

拓展：判断二叉树是否为满二叉树

```
void func(BiTree p, int *n, int *L, int *max_L){
    if (p != NULL){
        ++(*L);
        if ((*L) >= (*max_L))
            *max_L = *L;
        ++(*n);
        func(p->lchild, n, L, max_L);
        func(p->rchild, n, L, max_L);
        --(*L);
    }
}
bool Is_True(BiTree p){
    int n=0, L=0, max_L = 0;
    func(p, &n, &L, &max_L);
    if (pow(2, max_L) - 1 == n)
        return true;
    else
        return false;
}
```

核心思想：

1、递归遍历统计节点数和最大深度：利用递归更新当前深度和最大深度，并累加节点数。

2、判断节点数是否满足满二叉树节点公式：满二叉树节点数为 2^高度 - 1。

3、返回判断结果：满足节点数公式则返回 true，否则返回 false。

19、二叉树以二叉链表表示，设计算法存储到一维数组中

```
void create(BiTree p, char A[], int i){
    if (p != NULL){
        A[i] = p->data;
        create(p->lchild, A, 2 * i + 1);
        create(p->rchild, A, 2 * i + 2);
    }
}注意这是第 19 题，先看完 17 和 18 再看这个
//时空复杂度分别为 O(N)和 O(N)
```

17、满二叉树先序序列存在于数组中，设计算法将其变成后序序列

```
void preTopost(char pre[], int L1, int R1, char post[], int L2, int R2){
   if (L1 <= R1){
      post[R2] = pre[L1];
      preTopost(pre, L1 + 1, (L1 + R1 + 1)/2, post, L2, (L2 + R2 - 1)/2);
      preTopost(pre, (L1 + R1 + 1)/2 + 1, R1, post, (L2 + R2 - 1)/2 + 1, R2 - 1);
   }
}//先序转后序
void postTopre(char post[], int L1, int R1, char pre[], int L2, int R2){
   if (L1 <= R1){
      pre[L2] = post[R1];
      if (L1 == R1)
         return;   //防止 L1 和 R1 均为 0 时陷入无穷递归
      postTopre(post, L1, (L1 + R1 - 1)/2, pre, L2 + 1, (L2 + R2 + 1)/2);
      postTopre(post, (L1 + R1 - 1)/2 + 1, R1 - 1, pre, (L2 + R2 + 1)/2 + 1, R2);
   }
}//后序转先序!

//时空复杂度分别为 O(N)和 O(N)
```

18、先序与中序遍历分别存在两个一维数组 A, B 中，试建立二叉链表

```
BiTree func(char pre[], char mid[], int L1, int R1, int L2, int R2){
   BiTree root = (BiTree)malloc(sizeof(BTNode));
   root->data = pre[L1];
   int i = L2;
   for (; mid[i] != pre[L1]; i++); //注意 for 循环内无内容
   if (i > L2)
      root->lchild = func(pre, mid, L1 + 1, L1 + i - L2, L2, i - 1);
   else
      root->lchild = NULL;
   if (i < R2)
      root->rchild = func(pre, mid, L1 + i - L2 + 1, R1, i + 1, R2);
   else
      root->rchild = NULL;
   return root;
}//先序和中序构建二叉链表
BiTree func(char post[], char mid[], int L1, int R1, int L2, int R2){
   BiTree root = (BiTree)malloc(sizeof(BTNode));
   root->data = post[R1];
   int i = L2;
   for (; mid[i] != post[R1]; i++); //注意 for 循环内无内容
   if (i > L2)
      root->lchild = func(post, mid, L1, i - L2 + L1 - 1, L2, i - 1);
   else
      root->lchild = NULL;
   if (i < R2)
      root->rchild = func(post, mid, i - L2 + L1, R1 - 1, i + 1, R2);
   else
      root->rchild = NULL;
   return root;
} //后序和中序构建二叉链表   //时空复杂度分别为 O(N)和 O(N)
```

19 题拓展：二叉树以顺序方式存在于数组 A 的中，设计算法以二叉链表表示

```
BiTree create(char A[], int i, int n){
    if (i < n && A[i] != '#'){
        BiTree t = (BiTree)malloc(sizeof(BTNode));
        t->data = A[i];
        t->lchild = create(A, 2 * i + 1, n);
        t->rchild = create(A, 2 * i + 2, n);
        return t;
    }
    return NULL;
} //时空复杂度分别为 O(N)和 O(N)
```

20、已知二叉树采用顺序存储结构，编写算法统计该树中所有叶子结点个数。

```
int count(char A[], int i, int N) {
    if ( i >= N || A[i] == '#')
        return 0;
    if ((2 * i + 1 >= N || A[2 * i + 1] == '#') &&
        (2 * i + 2 >= N || A[2 * i + 2] == '#'))
        return 1;
    int n1 = count(A, 2 * i + 1, N);
    int n2 = count(A, 2 * i + 2, N);
    return n1 + n2;
}//时空复杂度分别为 O(N)和 O(N)
//408 近几年爱考顺序存储的树和图，务必掌握！！
```

图的结构体

邻接表存储：

```
typedef struct ANode{
    int adjvex; //边所指向结点的位置
    struct ANode *nextarc;
} ANode, *Node; //边结点结构体

typedef struct{
    int data;
    ANode *firstarc;
} Vnode; //顶点结构体

typedef struct{
    Vnode adjlist[maxsize];
    int numver, numedg;
} Graph;
```

邻接矩阵存储：

```
typedef struct{
    char verticle[maxsize];
    int Edge[maxsize][maxsize];
    int numver, numedg;
} mGraph;
```

1、已知无向连通图 G 由顶点集 V 和边集 E 组成，|E|>0,当 G 中度为奇数的顶点个数为不大于 2 的偶数时，G 存在包含所有边且长度为|E|的路径（称为 EL 路径）。设图 G 采用邻接矩阵存储，设计算法判断图中是否存在 EL 路径，若存在返回 1，否则返回 0。（2021 年统考题）

```
int IsExistEL(MGraph G){
    int count = 0;
    for (int i = 0; i < G.numver; i++){
        int degree = 0;
        for (int j = 0; j < G.numver; j++)
            degree += G.Edge[i][j];
        if (degree % 2 != 0)
            count++;
    }
    if (count == 0 || count == 2)
        return 1;
    else
        return 0;
}//时空复杂度分别为 O(V²)和 O(1)
//408 注意，结构体要和题目给出的保持一致
```

核心思想：

1、统计每个顶点的度：遍历邻接矩阵每一行，累加每个顶点的度数

2、统计奇度顶点个数：统计图中度为奇数的顶点数量

3、判断是否满足欧拉路径条件：若奇度顶点个数为 0 或 2，则存在 EL 路径，返回 1；否则返回 0

2、有向图 G 采用邻接矩阵存储，将图中出度大于入度的顶点称为 K 顶点。要求输出 G 中所有的 K 顶点，并返回 K 顶点的个数。（2023 年统考题）

```
int func(MGraph G){
    int count = 0;
    for(int i = 0; i<G.numver; i++){
        int indegree = 0, outdegree = 0;
        for (int j = 0; j < G.numver; j++){
            outdegree += G.Edge[i][j];
            indegree += G.Edge[j][i];
        }
        if (outdegree > indegree){
            printf("%c", G.verticle[i]);
            count++;
        }
    }
    return count;
} //时空复杂度分别为 O(V²)和 O(1)
```

核心思想：

1、分别统计每个顶点的出度和入度：通过遍历邻接矩阵的行和列，计算每个顶点的出度和入度

2、判断出度是否大于入度：若某个顶点的出度大于入度，则将其输出，并计数

3、返回出度大于入度的顶点个数：遍历结束后返回符合条件的顶点总数

3、有向图采用邻接表存储，计算每个顶点的入度和出度，将其存储到两个数组中

```
void func(Graph G, int inres[], int outres[]){
    for(int i = 0; i < G.numver; i++){
        inres[i] = 0;
        outres[i] = 0;
    }
    for (int i = 0; i < G.numver; i++){
        Node p = G.adjlist[i].firstarc;
        for(; p != NULL; p = p->nextarc){
            outres[i]++;
            inres[p->adjvex]++;
        }
    }
} //时空复杂度分别为 O(V+E)和 O(V)
```

核心思想：

1、初始化度数数组：将所有顶点的入度和出度数组清零，为后续统计做准备

2、遍历邻接表访问所有边：依次扫描每个顶点的边表，获取边的起点和终点信息

3、分别更新入度和出度：对起点的出度加 1，对终点的入度加 1，完成度数统计

4、设计算法判断无向图是否是一棵树

```
void DFS(Graph G, int v, int visit[], int *vn){
    visit[v] = 1;
    ++(*vn);
    Node p = G.adjlist[v].firstarc;
    for(; p != NULL; p = p->nextarc){
        if (visit[p->adjvex] == 0)
            func(G, p->adjvex, visit, vn);
    }
} //时空复杂度分别为 O(V+E)和 O(V)

bool IsTree(Graph G){
    int v = 0, vn = 0;
    int visit[maxsize] = {0};
    DFS(G, v, visit, &vn);
    if (vn == G.numver && (G.numver-G.numedg) == 1)
        return true;
    else
        return false;
}
```

核心思想：

1、深度优先遍历图的连通部分：从顶点 0 开始 DFS，统计访问过的顶点数 vn，确保连通性

2、判断顶点数和边数关系：树的性质是顶点数减去边数等于 1，即 |V| - |E| = 1

3、综合连通性和边数判断：若遍历访问所有顶点且满足边数关系，则图为树，否则不是

5、输出有向图 Vi 顶点到 Vj 顶点所有简单路径。

```
void DFS(Graph G, int i, int j, int visit[], int path[],
    int *top){
    visit[i] = 1;
    path[++(*top)] = i;
    if (i == j){
        for(int k = 0; k <= *top; k++)
            printf("%d ", path[k]);
    }
    Node p = G.adjlist[i].firstarc;
    for( ; p != NULL; p = p->nextarc)
        if (visit[p->adjvex] == 0)
            DFS(G, p->adjvex, j, visit, path, top);
    --(*top);
    visit[i] = 0;
} //时空复杂度分别为 O(V+E)和 O(V)
```

核心思想：

1、路径记录与回溯机制：利用 path 数组和 top 指针记录当前路径，递归过程中路径动态更新，回溯时恢复状态

2、遍历所有可能路径：从起点 i 开始，对未访问的邻接点递归调用 DFS，探索所有通往终点 j 的路径

3、到达目标时输出路径：当递归到达终点 j 时，打印当前完整路径，实现所有路径的枚举

6、求不带权的无/有向图中 v 顶点到其他顶点的的最短路径(长度)，存储到 dist 数组中。

```
void BFS(Graph G, int v, int dist[]){
    int visit[maxsize]={0};
    for (int i = 0; i < G.numver; i++)
        dist[i] = INT16_MAX;
    int que[maxsize];
    int front = 0, rear = 0;
    visit[v] = 1;
    dist[v] = 0;
    que[++rear] = v;
    while (front != rear){
        v = que[++front];
        Node p = G.adjlist[v].firstarc;
        for(; p != NULL; p = p->nextarc)
            if (visit[p->adjvex] == 0){
                visit[p->adjvex] = 1;
                que[++rear] = p->adjvex;
                dist[p->adjvex] = dist[v] + 1;
            }
    }
} //时空复杂度分别为 O(V+E)和 O(V)
```

1、初始化距离与访问状态：所有顶点距离初始化为极大值，起点距离设为 0，访问标记清零。

2、层序遍历：通过队列维护待访问顶点，按层依次访问，保证先访问距离近的顶点

3、更新距离并标记访问：访问邻接顶点时更新其距离（当前顶点距离+1），并标记已访问防止重复入队

7、Prim 算法求无向图的最小生成树

```
void prim(MGraph G, int v) {
    int visit[maxsize] = {0};
    int lowcost[G.numver], parent[G.numver];
    for (int i = 0; i < G.numver; i++){
        lowcost[i] = INT16_MAX;
        parent[i] = -1;
    }
    lowcost[v] = 0;
    for (int i = 0; i < G.numver - 1; i++) {
        int pos, min_val = INT16_MAX;
        for (int j = 0; j < G.numver; j++)
            if (lowcost[j]<min_val && visit[j]==0){
                pos = j;
                min_val = lowcost[j];
            }
        visit[pos] = 1;
        for (int j = 0; j < G.numver; j++)
            if (lowcost[j]>G.Edge[pos][j] && !visit[j]){
                lowcost[j] = G.Edge[pos][j];
                parent[j] = pos;
            }
    }
    for (int i = 0; i < G.numver; i++)
        printf("%d_%d:%d",i, parent[i], lowcost[i]);
} //时空复杂度分别为 O(V²)和 O(V)
```

8、Kruskal 算法求无向图的最小生成树

```
int find(int v, int parent[]) {
    while (parent[v] != v)
        v = parent[v];
    return v;
}
void Kruskal(EGraph G){
    int parent[G.numver];
    for (int i = 0; i <G.numver; i++)
        parent[i] = i;
    for (int i = 2; i <= G.numedg; i++) {
        G.A[0] = G.A[i];
        int j = i - 1;
        for( ; G.A[0].weight < G.A[j].weight; j--)
            G.A[j + 1] = G.A[j];
        G.A[j + 1] = G.A[0];
    }//直接插入排序
    for (int i = 1; i <= G.numedg; i++) {
        if (find(G.A[i].start, parent) != find(G.A[i].end, parent)){
            printf("%d-%d:%d",G.A[i].start, G.A[i].end, G.A[i].weight);
            parent[find(G.A[i].start, parent)] = parent[find(G.A[i].end, parent)];
        }
    }
} //时空复杂度分别为 O(E^2)和 O(V)
```

9、Dijkstra 算法求有/无向图的最短路径

```
void Dijkstra(MGraph G, int v) {
    int visit[maxsize] = {0};    int lowcost[G.numver], parent[G.numver];
    for (int i = 0; i < G.numver; i++){
        lowcost[i] = INT16_MAX;    parent[i] = -1; //考试尽量两句话写到两行
    }
    lowcost[v] = 0;
    for (int i = 0; i < G.numver - 1; i++) {
        int pos, min_val = INT16_MAX;
        for (int j = 0; j < G.numver; j++)
            if (lowcost[j]<min_val && visit[j]==0){
                pos = j;    min_val = lowcost[j];
            }
        visit[pos] = 1;
        for (int j = 0; j < G.numver; j++)
            if (lowcost[j] > G.edge[pos][j] + lowcost[pos] && visit[j] == 0){
                lowcost[j] = G.edge[pos][j] + lowcost[pos];    parent[j] = pos;
            }
    }
    for (int i = 0; i < G.numver; i++){
        int st[G.numver], top = -1, j = i;
        while(parent[j] != -1){
            st[++top] = j;    j = parent[j];
        }
        printf("%d " v);
        while (top != -1)    printf("%d ", st[top--]);
        printf("%d ", lowcost[i]);
    }
} //时空复杂度分别为 O(V^2)和 O(V)
```

10、Floyed 求有/无向图的最短路径长度

```
void floyed(MGraph G,int dist[][maxsize],
int path[][maxsize]) {
    for (int i = 0; i < G.numver; i++)
        for (int j = 0; j < G.numver; j++){
            dist[i][j] = G.Edge[i][j];
            path[i][j] = -1;
        }
    for (int m = 0; m < G.numver; m++)
        for (int s = 0; s < G.numver; s++)
            for (int e = 0; e < G.numver; e++)
                if (dist[s][m] + dist[m][e] < dist[s][e]){
                    dist[s][e] = dist[s][m] + dist[m][e];
                    path[s][e] = m;
                }
} //时空复杂度分别为 O(V³)和 O(V²)

void printPath(int u, int v, int path[][maxsize]){
    if (path[u][v] == -1){
        printf("%d ", u);
        return;
    }
    printPath(u, path[u][v], path);
    printPath(path[u][v], v, path);
}
```

11、拓扑排序判断有/无向图是否有环（邻接表）

```
bool tpsort (Graph G){
    int inres[maxsize]={0};
    int st[maxsize], top = -1, n = 0;
    for (int i = 0; i < G.numver; i++){
        Node p = G.adjlist[i].firstarc;
        for ( ; p != NULL; p = p->nextarc)
            inres[p->adjvex]++;
    }
    for (int i = 0; i < G.numver; i++)
        if (inres[i] == 0)//无向图为 1
            st[++top] = i;
    while (top != -1){
        int v = st[top--];
        n++;
        Node p = G.adjlist[v].firstarc;
        for (; p != NULL; p = p->nextarc)
            if (--inres[p->adjvex] == 0) //无向图为 1
                st[++top] = p->adjvex;
    }
    if (n == G.numver)
        return false; //无环
    else
        return true; //有环
} //时空复杂度分别为 O(V+E)和 O(V)
```

1、统计每个顶点的入度：遍历邻接表，计算所有顶点的入度值。

2、将入度为 0 的顶点入栈准备遍历：将所有入度为 0 的顶点入栈，作为拓扑排序的起点。

3、模拟拓扑排序过程并检测环：依次弹栈访问顶点，减少其邻接点入度，若过程中顶点访问数不等于总顶点数，说明存在环。

12、拓扑判断有/无向图是否有环（邻接矩阵）

```
bool tpsort(MGraph G) {
    int inres[maxsize] = {0};
    int st[G.numver], top = -1, n = 0;
    for (int i = 0; i < G.numver; i++)
        for (int j = 0; j < G.numver; j++)
            if (G.Edge[j][i] != 0)
                inres[i]++;
    for (int i = 0; i < G.numver; ++i)
        if (inres[i] == 0)
            st[++top] = i;
    while (top != -1) {
        int v= st[top--];
        n++;
        for (int i = 0; i < G.numver; i++)
            if (G.Edge[v][i] != 0 && (--inres[i] == 0))
                st[++top] = i;
    }
    if (n == G.numver)
        return false; //无环
    else
        return true; //有环
} //时空复杂度分别为 O(V²)和 O(V)
```

13、判断领接矩阵存储的有向图的拓扑排序是否唯一（2024 年统考题）

```
bool tpsort(MGraph G) {
    int inres[G.numver] = {0};
    int st[G.numver], top = -1, n = 0;
    for (int i = 0; i < G.numver; i++)
        for (int j = 0; j < G.numver; j++)
            if (G.Edge[j][i] != 0)
                inres[i]++;
    for (int i = 0; i < G.numver; ++i)
        if (inres[i] == 0)
            st[++top] = i;
    if (top > 0)
        return false;
    while (top != -1) {
        int v= st[top--];
        n++;
        for (int i = 0; i < G.numver; i++)
            if (G.Edge[v][i] != 0 && (--inres[i] == 0))
                    st[++top] = i;
        if (top > 0)
            return false;
    }
    if (n == G.numver)
        return true; //无环且唯一
    else
        return false; //有环
} //时空复杂度分别为 O(V²)和 O(V)
```

第 13 题核心思想：

1、统计每个顶点的入度：遍历邻接矩阵统计每个顶点的入度值

2、初始化入度为 0 的顶点栈，并判断唯一性：将所有入度为 0 的顶点入栈，若某时刻栈中顶点数超过 1，说明有多种拓扑序列，返回无唯一

3、执行拓扑排序并判断图的性质：依次出栈访问顶点，更新邻接点入度，最后判断是否遍历所有顶点，确认无环且唯一拓扑序列

1、查找二叉排序树中第 k 小的结点

```
void find(BiTree bt, int k, int *count, BiTree *p){
    if (bt != NULL){
        find(bt->lchild, k, count, p);
        (*count)++;
        if ((*count) == k)
            *p = bt;
        find(bt->rchild, k, count, p);
    }
} //时空复杂度分别为 O(N)和 O(N)
```

2、输出二叉搜索树中所有值大于 key 的值

```
void func(BiTree bt, char key){
    if (bt != NULL){
        func(bt->lchild, key);
        if (bt->data > key)
            printf("%d ", bt->data);
        func(bt->rchild, key);
    }
} //时空复杂度分别为 O(N)和 O(N)
```

3、判断顺序存储的二叉树是否为二叉搜索树

```
bool func(SqBiTree T, int i, Elemtype low,
Elemtype high){
    if(i >= T.ElemNum || T.Node[i] == -1)
        return true;
    if(T.Node[i] <= low || T.Node[i] >= high)
        return false;
    bool left = func(T, 2 * i + 1, low, T.Node[i]);
    bool right = func(T, 2 * i + 2, T.Node[i], high);
    return left && right ;
} //时空复杂度分别为 O(N)和 O(N)
```

核心思想：

1、利用递归遍历顺序存储的二叉树：通过索引关系（左子节点为 2*i+1，右子节点为 2*i+2）递归访问所有节点

2、维护上下界判断节点值合法性：每个节点的值必须严格在 low 和 high 之间，保证二叉搜索树的性质

3、递归检查左右子树的合法性：分别递归检查左子树的值范围是 [low，当前节点值)，右子树的值范围是（当前节点值, high]，确保全树满足二叉搜索树条件

4、判断一个二叉排序树是否为平衡二叉树

```
int height(BiTree T) {
    if (T == NULL)
        return 0;
    int A = height(T->lchild);
    int B = height(T->rchild);
    return (A > B ? A : B) + 1;
} //计算高度
bool istrue(BiTree T) {
    if (T == NULL)
        return true;
    int left = height(T->lchild);
    int right = height(T->rchild);
    if (abs(left - right ) > 1)
        return false;
    return istrue(T->lchild) && istrue(T->rchild);
} //时空复杂度分别为 O(N)和 O(N)
```

核心思想：

1、递归计算每个子树的高度：通过 height() 函数递归地求出当前节点左右子树的高度，以便用于平衡性判断

2、判断每个节点的左右子树是否平衡：在 istrue() 函数中，通过判断 abs(左高 - 右高) <= 1，确定当前节点是否满足平衡二叉树的条件

3、逐层递归检查整棵树是否平衡：继续递归检查左右子树是否也都平衡，最终确定整棵树是否为平衡二叉树（AVL 树的弱化定义）

1、二路归并排序

```
void merge(int A[],int low, int mid, int high){
    int *B = (int*)malloc((high + 1)*sizeof(int));
    for (int i = low; i <= high; i++)
        B[i] = A[i];
    int i = low, j = mid + 1, k = low;
    while (i <= mid && j <= high)
        if (B[i] <= B[j])
            A[k++] = B[i++];
        else
            A[k++] = B[j++];
    while (i <= mid)
        A[k++] = B[i++];
    while (j <= high)
        A[k++] = B[j++];
}
void mergeSort(int A[], int low, int high){
    if (low < high){
        int mid = (low + high) / 2;
        mergeSort(A, low, mid);
        mergeSort(A, mid + 1, high);
        merge(A, low, mid, high);
    }
} //时空复杂度分别为 O(NlogN)和 O(N)
```

二路归并排序核心思想：

1、递归拆分数组：通过递归不断将数组从中间分成两半，直到子数组长度为 1，实现分治思想的“分”阶段

2、合并有序子数组：利用辅助数组，将拆分后的两个有序子数组按顺序合并成一个有序数组，完成“治”的过程

3、借助辅助数组实现稳定合并：使用临时数组复制当前区间元素，保证合并时不会破坏原数组数据，实现稳定排序

堆排序核心思想：

1、构建初始大顶堆：从最后一个非叶子节点开始，依次向上调用 sift，将整个数组调整为一个大顶堆，使得堆顶是当前未排序部分的最大值。

2、依次将堆顶元素交换到末尾：每轮将堆顶元素与当前未排序部分的最后一个元素交换，相当于把当前最大值固定到最终位置。

3、交换后重新调整剩余堆结构：交换后对堆顶调用 sift，继续保持剩余未排序部分为大顶堆，直到整个数组有序为止。

3、堆排序

```
void sift(int A[], int low, int high){
    int i = low, j = 2 * i + 1;
    int temp = A[i];
    while (j <= high){
        if (j < high && A[j] < A[j + 1])
            ++j;
        if (temp < A[j]){
            A[i] = A[j];
            i = j;
            j = 2 * i + 1;
        }
        else
            break;
    }
    A[i] = temp;
}//堆调整的时空复杂度分别为 O(logN)和 O(1)
void heapSort(int A[], int n){
    for (int i = n / 2 - 1; i >= 0; --i)
        sift(A, i, n - 1);
    for (int i = n - 1; i > 0; --i){
        int temp = A[0];
        A[0] = A[i];
        A[i] = temp;
        sift(A, 0, i - 1);
    }
} //堆排序时空复杂度分别为 O(NlogN)和 O(1)
```

2、判断数组是否是小根堆

```
bool isMinHeap(int A[], int n) {
    for (int i = 0; i <= (n - 2) / 2; i++) {
        if (A[i] > A[2 * i + 1])
            return false;
        if (2 * i + 2 < n && A[i] > A[2 * i + 2])
            return false;
    }
    return true;
} //时空复杂度分别为 O(N)和 O(1)
```

核心思想：

1、遍历所有非叶子节点：通过循环遍历索引范围在 [0, (n-2)/2] 内的节点，确保所有父节点都检查一遍。

2、比较父节点与左右子节点：判断每个父节点是否小于或等于其左子节点 2i+1 和右子节点 2i+2（存在时），确保最小堆性质。

3、一旦发现违反堆性立即返回：只要发现一个父节点大于任一子节点，立即返回 false，避免不必要的检查，提升效率。

4、已知由 n(n>=2)个正整数构成的集合 A={ak|0<=ak<n}，将其划分为两个不相交的子集 A1 和 A2，元素个数分别是 n1 和 n2。A1 和 A2 中元素之和分别为 S1 和 S2。设计一个尽可能高效的划分算法，满足 |n1-n2|最小且|S1-S2|最大。（2016 年统考题）

```
void func(int A[],int A1[], int A2[], int n){
        Quicksort(A, 0, n-1);//考试时快排代码要写
        for (int i = 0; i < n/2; ++i)
            A1[i] = A[i];
        for (int i = n/2; i < n; ++i)
            A2[i-n/2] = A[i];
} //时空复杂度分别为 O(NlogN)和 O(logN)
```

最优解有一定难度，听不懂就暴力吧

核心思想：

1、先对数组整体排序：调用快速排序将原数组按升序排序，为后续均分做准备

2、将排序后数组一分为二：将前一半元素赋值给 A1，后一半赋值给 A2，确保两个子数组有序且元素平均分布

3、借助额外数组实现划分：通过额外数组 A1 和 A2 存储结果，逻辑清晰但需额外空间支持