实验题第 1 题

第 1 小题

(1) 设计思路

本题要求对于一个不带头节点的单链表，设计递归算法逆置所有节点。

- 根据递归程序设计的方法，有：
  - 转化方法：把逆置所有节点转化为逆置递归逆置头节点和后继链表的问题；
  - 递归模型：$\begin{cases} f(\text{head})=f(\text{head}\to\text{next}) & \text{head}\to\text{next}\neq\text{NULL} \\ f(\text{head})=\text{不做处理} & \text{head}\to\text{next}=\text{NULL} \end{cases}$

  根据以上模型，可以写出代码如下。
- 需要注意的是，不能直接在递归函数中对头指针 head 进行操作。因为 head 必定是在最后一层递归时置给原尾节点的，那么回溯到第一层递归时，head 必定已经改变，导致原头节点地址的丢失，无法正常完成头节点和后继链表的逆置。我对此进行了思考，最终还是选择引入一个 newhead 来存放原尾节点（新头节点）的地址，不在递归中修改 head，等完成所有节点的逆置之后，在递归外把 newhead 的值传给 head，从而完成整个链表的逆置。

(2) 源代码

void Reverse(Node<DataType> *&h, Node<DataType> *&f, Node<DataType> *&newhead); **//逆置辅助函数**

```
template <class DataType>
void LinkList<DataType>::Reverse(Node<DataType> *&h, Node<DataType> *&f,
Node<DataType> * &newhead){
  if(h->next != NULL){                  //如果头节点后面还有节点
    Reverse(h->next,h,newhead);         //递归逆置头节点和之后的链表
    h->next = f;                        //让头节点指向其前驱节点 f
  }
  if(h->next == NULL && h != head){     //如果头节点后没有节点且不是首轮递归（h 为原尾节点）
    h->next = f;                        //使头节点指向其前驱节点 f
    newhead = h;                        //把 h 的地址传入新的头指针中
  }
}
```

void Reverse(); **//逆置主调函数**

```
template <class DataType>
void LinkList<DataType>::Reverse(){
  Node<DataType> * null = NULL; //初始化空指针
  Node<DataType> * newhead;     //初始化新头指针
  Reverse(head, null, newhead); //进行逆置，并获取新头节点地址存入新头指针
  head = newhead;               //把新头节点的地址赋给 head，完成逆置
}
```

(3)说明

上述代码可以应用测试程序进行测试并得到正确结果。

- 针对前文提到的 head 指针问题，如果形象地比喻一下，把链表比作一条吊着的鱼，那么头指针 head 就是鱼钩。逆置的含义是把鱼整条倒过来挂，是针对鱼这个整体的操作，而不是带着鱼钩一起反过来。因此逆置的过程必须把鱼钩隔离开，先进行鱼的倒置，最后再重新把鱼挂上鱼钩。newhead 就是拎着鱼尾巴，把鱼反过来的那只手。

第 2 小题

(1) 设计思路

本题要求假设二叉树采用二叉链表存储节点，设计递归算法判断两棵二叉树是否同构（即形态相同）。

- 根据递归程序设计的方法，有：
  - 转化方法：把判断二叉树同构转化为判断左右子树同构的问题；
  - 递归模型：$\begin{cases} f(\text{root})=f(\text{root}\to\text{lChild})\wedge f(\text{root}\to\text{rChild}) & \text{root1,root2}\neq\text{NULL} \\ f(\text{root})=\text{true} & \text{root1,root2}=\text{NULL} \\ f(\text{root})=\text{false} & \text{root1}\vee\text{root2}\neq\text{NULL} \end{cases}$

  根据以上模型，可以写出代码如下。

(2) 源代码

bool Isomorphism(BTNode<DataType> *r1,BTNode<DataType> *r2, BTNode<DataType> *f1, BTNode<DataType> *f2){}; **//辅助函数**

```
template <class DataType>
bool BinaryTree<DataType>::Isomorphism(BTNode<DataType> *r1,BTNode<DataType> *r2,
BTNode<DataType> *f1, BTNode<DataType> *f2){
  if (r1 == NULL && r2 == NULL) return true;     //如果上一层父节点的左/右孩子同时为空节点，
返回 true
  if ((r1 == NULL && r2 != NULL) || (r1 != NULL && r2 == NULL)) return false;
  //如果一个是空节点，另一个不是空节点，则不同构，返回 false
  if(
    Isomorphism(r1->lChild, r2->lChild, r1, r2) &&
    Isomorphism(r1->rChild, r2->rChild, r1, r2)
  ) //如果左右子树都同构，返回 true
    return true;
  else
    return false;
}
```

bool Isomorphism(BinaryTree<char> tree){}; **//主调函数**

```
template <class DataType>
bool BinaryTree<DataType>::Isomorphism(BinaryTree<char> tree){
  return Isomorphism(root,tree.root,nullptr,nullptr);
}
```

(3) 说明

上述代码可以应用测试程序进行测试并得到正确结果。

值得注意的是，检测是否同构是按照相同的结构，同时顺着左/右分支深入下去的，所以只要判断下一次检测的节点是否同时为空，即可判断是否同构。

第 3 小题

(1) 设计思路

本题要求有 n（n>3）个硬币（编号为 1 n），其中一枚是假币，由于假币的重量较轻，可以采用天平称重的方式找到这枚假币。请利用分治法设计算法找到这枚假币，输出假币的编号。

- 根据分治法的设计步骤，得：
  - 分解：将从一大堆硬币中找假币分解为从三小堆中找假币；
  - 求解：如果只有一个硬币（左边界=有边界），直接返回编号；
  - 合并：直接返回，无需合并。

  根据以上步骤，可以写出程序如下。
- 将硬币分成三堆，只需要对前两堆进行称重，即可找到假币在的那一堆里，这样遍历的次数永远比直接从头到尾遍历一遍少。称重的情况如下：
  - sum1=sum2：假币在第 3 堆里；
  - sum1<sum2：假币在第 1 堆里；
  - sum1>sum2：假币在第 2 堆里。

  如此进行范围的缩小，直到一堆只有一个硬币，规模达到最小，可以直接求解。

(2) 源代码

int Solve(int w[], int a, int b); **//解题函数**

```
int Solve(int w[], int a, int b){
    if (a == b) // 如果只有一个硬币，直接返回它的编号
        return a;

    int mid1 = 0;
    int mid2 = 0;
    if ((b-a)>3){//硬币数大于 4，使用公式对硬币进行分组，求出两个边界
        mid1 = a + (b - a) / 3;  // 第一组的右端
        mid2 = b - (b - a) / 3;  // 第二组的左端
    }
    else{//银币数小于等于 4，直接给 mid1 和 mid2 赋值
        mid1 = a;
        mid2 = a + 1;
    }

    //计算 sum1，sum2
    int sum1 = 0, sum2 = 0;
    for (int i = a; i <= mid1; i++)
        sum1 += w[i]; // 计算第一组的总重量
    for (int i = mid1 + 1; i <= mid2; i++)
        sum2 += w[i]; // 计算第二组的总重量

    //conquer
    if (sum1 == sum2) {// 如果第一组和第二组相等，假币在第三组
        return Solve(w, mid2 + 1, b); // 递归查找第三组
    }
    else if (sum1 < sum2) {// 如果第一组较轻，假币在第一组
        return Solve(w, a, mid1); // 递归查找第一组
    }
    else { // 如果第二组较轻，假币在第二组
        return Solve(w, mid1 + 1, mid2); // 递归查找第二组
```

```
  }
}
```

(3)说明

上述代码可以应用测试程序进行测试并得到正确结果。

需要注意的是，每次进行分堆，第 1 堆和第 2 堆硬币个数必须相同，否则失去比较的意义，无法按照理论得出结论。因此，在银币数小于等于 4 时，已经不能按照公式进行分组，否则会出现 1、2 两组失衡的情况，此时直接给 1、2 两枚硬币分为两组即可。

第 4 小题

(1) 设计思路

本题要求**在一个整数序列中，每个元素出现的次数称为重数，重数最大的元素称为众数。现已知一个长度为 n 的递增有序的整数序列，请利用分治法设计算法寻找该序列的众数**。

- 根据分治法的设计步骤，得：
  - 分解：将从一个序列中寻找众数分解为从两个小序列中分别寻找众数；
  - 求解：如果序列区间只有一个元素，直接返回为众数，记重数为 1；
  - 合并：
    - 如果两个区间众数相同，则返回，并合并重数为两边之和；
    - 如果两个区间众数不同，则需要比较左众数、右众数，以及被间隔分开的数跨界后的重数，返回三者中最大者。

根据以上步骤，可以写出程序如下。

(2) 源代码

int GetMode(int arr[], int a, int b, int &m); **//求众数**

```
int GetMode(int arr[], int a, int b, int &m) {
    if (a == b) {    //序列区间只有一个元素
        m = 1;       //重数为 1
        return arr[a];//直接返回
    }

    int mid = (a + b) / 2;
    int leftM, rightM; // 两边的众数重数
    //分开求解
    int leftMode = GetMode(arr, a, mid, leftM);
    int rightMode = GetMode(arr, mid + 1, b, rightM);

    // 合并结果
    if (leftMode == rightMode) {//两边众数相同
        m = leftM + rightM;
        return leftMode;
    } else {
        // 如果两边众数不同，计算跨界的重数
        int leftCount = leftM;
        int rightCount = rightM;
        int midCount = 0;

        // 向左统计右众数
        for (int i = mid; i >= a; i--) {
            if(arr[i] == rightMode)
                rightCount++;
        }
        // 向右统计左众数
        for (int i = mid + 1; i <= b; i++) {
            if(arr[i] == leftMode)
                leftCount++;
        }
        // 统计间隔两边的数的重数
        if (arr[mid]==arr[mid+1])
        {
```

```
            for (int i = mid; i >= a; i--) {
            if(arr[i] == arr[mid])
                midCount++;
            }

            for (int i = mid + 1; i <= b; i++) {
            if(arr[i] == arr[mid])
                midCount++;
            }
        }

        // 比较跨界后的重数，确定最终众数
       if (leftCount < midCount && midCount > rightCount){
            m = midCount;
            return arr[mid];
        } else if (leftCount > rightCount) {
            m = leftCount;
            return leftMode;
        } else {
            m = rightCount;
            return rightMode;
        }
    }
}
```

(3)说明

上述代码可以应用测试程序进行测试并得到正确结果。

需要注意的是，要考虑众数是否被间隔一分为二，从而导致左右众数都不是真正的众数。

此外，本题使用分治法解决似乎加大了问题的时间复杂度。