# 数组相关

## 3.数组中重复的数字

**长度为n的数组中所有数字都在0-n-1内，查找重复数字**

```
// 面试题3（一）：找出数组中重复的数字
// 题目：在一个长度为n的数组里的所有数字都在0到n-1的范围内。数组中某些数字是重复的，但不知道有几个数字重复了，
// 也不知道每个数字重复了几次。请找出数组中任意一个重复的数字。例如，如果输入长度为7的数组{2, 3, 1, 0, 2, 5, 3}，
// 那么对应的输出是重复的数字2或者3。

#include <iostream>
using namespace std;

// 参数:
//        numbers:     一个整数数组
//        length:      数组的长度
//        duplication: (输出) 数组中的一个重复的数字

// 排序数组0，1,2,3，4.。。。。如果没有重复，则下标对应的就是值
//时间复杂度n,比3_1效率高
int *duplicate(int numbers[], int length,int duplication[])
{

    if(numbers == nullptr || length <= 0)
         return NULL;

    for(int i = 0; i < length; ++i)
    {
        if(numbers[i] < 0 || numbers[i] > length - 1)
            return NULL;
    }

    int k=0;

    for(int i = 0; i < length; ++i)
    {
        if(numbers[i] != i)
        {
            if(numbers[i] == numbers[numbers[i]])
            {
                duplication[k] = numbers[i];
                k++;
            }

            // 交换numbers[i]和numbers[numbers[i]]
            int temp = numbers[i];
            numbers[i] = numbers[temp];
            numbers[temp] = temp;
        }

    }
```

```
    return duplication;

}

int main()
{
    int data[]={2,3,1,0,2,5,3};
    int duplication[]={0};
    int *p;
    p=duplicate(data, sizeof(data)/sizeof(int),duplication);
    cout<<"sizeof"<<sizeof(p)/sizeof(int)<<endl;

    for(int h=0;h<sizeof(p)/sizeof(int);h++)
        cout<<"result"<<p[h]<<endl;


    return 0;
}
```

## 4.二维数组中的查找

**从左到右，上到下递增，查找是否包含该数字**

```
// 面试题4：二维数组中的查找
// 题目：在一个二维数组中，每一行都按照从左到右递增的顺序排序，每一列都按
// 照从上到下递增的顺序排序。请完成一个函数，输入这样的一个二维数组和一个
// 整数，判断数组中是否含有该整数。

#include <iostream>
using namespace std;

//用指针代替数组的输入
bool Find(int* matrix, int rows, int columns, int number)
{
    bool found = false;//常用的方法

    if(matrix != nullptr && rows > 0 && columns > 0)
    {
        int row = 0;
        int column = columns - 1;
        while(row < rows && column >=0)
        {
            if(matrix[row * columns + column] == number)//二维数组中的一个数。技巧
            {
                found = true;
                break;
            }
            else if(matrix[row * columns + column] > number)
                -- column;
            else
                ++ row;
        }
```

```
    }

    return found;
}


//  1   2   8   9
//  2   4   9   12
//  4   7   10  13
//  6   8   11  15
// 要查找的数在数组中
int main(int argc, char* argv[])
{
  int matrix[][4] = {{1, 2, 8, 9}, {2, 4, 9, 12}, {4, 7, 10, 13}, {6, 8, 11,
15}};
  bool result=Find((int*)matrix, 4, 4, 7);
  cout<<"result"<<result<<endl;

    return 0;
}
```

## 5.替换空格

**字符串中的每个空格替换成特定字符**

```
// 面试题5：替换空格
// 题目：请实现一个函数，把字符串中的每个空格替换成"%20"。例如输入“We are happy.”，
// 则输出“We%20are%20happy.”。

#include <iostream>
using namespace std;
#include <cstring>

/*length 为字符数组str的总容量，大于或等于字符串str的实际长度*/
void ReplaceBlank(char str[], int length,int& newLength)//此处必须加&，否则调用传出来
的还是原来的值
{
    if(str == nullptr && length <= 0)
        return;

    /*originalLength 为字符串str的实际长度*/
    int originalLength = 0;
    int numberOfBlank = 0;
    int i = 0;
    while(str[i] != '\0')
    {
        ++ originalLength;

        if(str[i] == ' ')
            ++ numberOfBlank;

        ++ i;
    }
```

```
    /*newLength 为把空格替换成'%20'之后的长度*/
    newLength = originalLength + numberOfBlank * 2;
    if(newLength > length)
        return;

    int indexOfOriginal = originalLength;
    int indexOfNew = newLength;
    while(indexOfOriginal >= 0 && indexOfNew > indexOfOriginal)
    {
        if(str[indexOfOriginal] == ' ')
        {
            str[indexOfNew --] = '0';
            str[indexOfNew --] = '2';
            str[indexOfNew --] = '%';
        }
        else
        {
            str[indexOfNew --] = str[indexOfOriginal];
        }

        -- indexOfOriginal;
    }

}


int main(int argc, char* argv[])
{
     const int length = 100;

    char str[length] = "hello world";
    int newLength=0;
     ReplaceBlank( str, length,newLength);
     for(int k=0;k<newLength;k++)
        cout<<str[k]<<endl;


    return 0;
}
```

## 11.旋转数组的最小数字

**输入一个递增排序的数组的一个旋转，输出该旋转数组的最小值**

```
/*
// 面试题11：旋转数组的最小数字
// 题目：把一个数组最开始的若干个元素搬到数组的末尾，我们称之为数组的旋转。
// 输入一个递增排序的数组的一个旋转，输出旋转数组的最小元素。例如数组
// {3, 4, 5, 1, 2}为{1, 2, 3, 4, 5}的一个旋转，该数组的最小值为1。
*/

#include <iostream>
```

```
using namespace std;

//定义两个指针，分别指向第一个p1和最后一个p2，
//判断两者中间值p>p1，说明在第一队列，使跑p1=p，再比较p1和p2的中间值p,p<p2,在第二队列，使p2=p，两者相差1,则返回p2
//{1,0,1,1,1}这种情况单独拿出来，for循环查找最小值
int MinInOrder(int* data,int index1,int index2);
int Min(int* data,int len)
{
    // if(data==NULL || len<0)
    //     throw new std::exception("Invaild parameters");

    int index1=0;
    int index2=len-1;

    int mid=index1;
    while(data[index1]>=data[index2])
    {
        //终止条件
        if(index2-index1==1)
        {
            mid=index2;
            break;
        }

        mid=index1+(index2-index1)/2;

        //如果三者相等，顺序查找
        if(data[index1]==data[index2] && data[mid]==data[index1])
            return MinInOrder(data,index1,index2);

        if(data[mid]>=data[index1])
            index1=mid;

        if(data[mid]<=data[index2])
            index2=mid;


    }

    return data[mid];
}

int MinInOrder(int* data,int index1,int index2)
{
    int result=data[index1];
    for(int i=index1+1;i<=index2;++i)
    {
        if(result>data[i])
            result=data[i];
    }

    return result;
}

int main()
{
    // 典型输入，单调升序的数组的一个旋转
```

```
    int data[] = { 3, 4, 5, 1, 2 };

    int result=Min(data,sizeof(data)/sizeof(int));

    cout<<result<<endl;

    return 0;
}
```

## 21.调整数组使奇数位于偶数前面

**奇数位于数组的前半部分，偶数位于后半部分**

```
/*
// 面试题21：调整数组顺序使奇数位于偶数前面
// 题目：输入一个整数数组，实现一个函数来调整该数组中数字的顺序，使得所有
// 奇数位于数组的前半部分，所有偶数位于数组的后半部分。
*/

#include <iostream>
using namespace std;

void Reorder(int *data,unsigned int leng,bool (*func)(int));
bool isEven(int n);

void ReorderOddEven(int *data,unsigned int leng)
{
    Reorder(data,leng,isEven);
}
//定义两个指针，最前和最后，最前指向偶数，最后指向奇数时候，交换
void Reorder(int *data,unsigned int leng,bool (*func)(int))
{
    if(data==NULL || leng==0)
        return;

    int *begin=data;
    int *end=data+leng-1;

    while(begin<end)
    {
        while(begin<end && !func(*begin))//从最开始查找满足func函数的位置
            begin++;

        while(begin<end && func(*end))//从最后往前找
            end--;

        if(begin<end)
        {
            int tmp=*begin;
            *begin=*end;
            *end=tmp;
        }
    }
}

bool isEven(int n)
{
```

```
    return (n&1)==0;
}



int main()
{
    int data[] = {1, 2, 3, 4, 5, 6, 7};
   ReorderOddEven(data,sizeof(data)/sizeof(int));
    for(int i=0;i<7;i++)
        cout<<data[i]<<endl;

    return 0;

}
```

## 39.数组中出现次数超过一半的数字

**数组中有一个数字出现的次数超过数组长度的一半，找出来**

```
// 面试题39：数组中出现次数超过一半的数字
// 题目：数组中有一个数字出现的次数超过数组长度的一半，请找出这个数字。例
// 如输入一个长度为9的数组{1, 2, 3, 2, 2, 2, 5, 4, 2}。由于数字2在数组中
// 出现了5次，超过数组长度的一半，因此输出2。

#include <iostream>
using namespace std;

//出现次数超过一半，则他出现的次数大于其他次数的和
//定义保存两个值，一个是次数，首次为1，一个数组中的一个数字
//遍历到下一个数字的时候，和前面相等就加1,不等就减去1
//当次数为0的时候，重新保存下一个数，继续这个步骤
//则最后一个不为0的数必然是超过一半的


bool g_bInputInvalid=false;

//判断输入的有效性
bool CheckInvalidArray(int* datas,int leng)
{
    g_bInputInvalid=false;
    if(datas==NULL && leng<=0)
        g_bInputInvalid=true;

    return g_bInputInvalid;
}

//检查次数是否超过一半
bool CheckMoreThanHalf(int* datas,int leng,int data )
{
    int times=0;
    for(int i=0;i<leng;++i)
    {
        if(datas[i]==data)
            times++;
    }
```

```
    bool isMoreThanHalf=true;
    if(times*2<=leng)
    {
        g_bInputInvalid=true;
        isMoreThanHalf=false;
    }

    return isMoreThanHalf;

}

int Solution(int* datas,int leng)
{
    if(CheckInvalidArray(datas,leng))
        return 0;

    int result=datas[0];
    int times=1;
    for(int i=1;i<leng;++i)
    {
        if(times==0)
        {
           result=datas[i];
           times=1;
        }

        else if(datas[i]==result)
            times++;

        else
            times--;
    }

    if(!CheckMoreThanHalf(datas,leng,result))
        result=0;

    return result;
}

int main()
{
   int datas[] = {1, 2, 3, 2, 2, 2, 5, 4, 2};
    int result=Solution(datas,sizeof(datas)/sizeof(int));

    cout<<result<<endl;


    return 0;
}
```

## 40.最小的K个数

**输入n个整数，找出其中最小的k个数**

```
// 面试题40：最小的k个数
// 题目：输入n个整数，找出其中最小的k个数。例如输入4、5、1、6、2、7、3、8
// 这8个数字，则最小的4个数字是1、2、3、4。

#include <iostream>
using namespace std;
#include <set>
#include <vector>
#include <functional>

//定义最大堆，最上面就是最大元素
//创建一个大小为k的数据容器来存储最小的k个数字，每次从输入的n个整数中读入一个数
//若容器中已有的数字少于k个，则直接把这次读入的整数放入容器；
//若已经有了k个数字，也就是容器满了，把大堆的最大元素和当前数字比较，替换

typedef multiset<int,greater<int>> intSet;
typedef multiset<int,greater<int>>::iterator setIterator;

void GetLeastNumbers(const vector<int>& data, intSet& leastNumbers, int k)
{
    leastNumbers.clear();

    if(k<1 || data.size()<k)
        return;

    vector<int>::const_iterator iter=data.begin();
    for(;iter!=data.end();++iter)
    {
        if(leastNumbers.size()<k)
            leastNumbers.insert(*iter);

        else
        {
            setIterator iterGreatest=leastNumbers.begin();//把大顶堆的最大数拿出来

            if(*iter<*(leastNumbers.begin()))
            {
                leastNumbers.erase(iterGreatest);
                leastNumbers.insert(*iter);
            }
        }

    }
}


int main(int argc, char* argv[])
{
   int data[] = {4, 5, 1, 6, 2, 7, 3, 8};
    intSet leastNumbers;

    vector<int> vectorData;
    for(int i = 0; i <  sizeof(data) / sizeof(int); ++ i)
        vectorData.push_back(data[i]);

    GetLeastNumbers(vectorData,leastNumbers,4);
   for(setIterator iter = leastNumbers.begin(); iter != leastNumbers.end();
++iter)
```

```
            printf("%d\t", *iter);
     printf("\n\n");


    return 0;
}
```

## 41.数据流中的中位数

```
// 面试题41：数据流中的中位数
// 题目：如何得到一个数据流中的中位数？如果从数据流中读出奇数个数值，那么
// 中位数就是所有数值排序之后位于中间的数值。如果从数据流中读出偶数个数值，
// 那么中位数就是所有数值排序之后中间两个数的平均值。

#include <iostream>
using namespace std;
#include <cstring>

#include <algorithm>
#include <vector>
#include <functional>

#include <stdexcept>
//设计一个大顶堆和一个小顶堆，max在前，min在后，满足两者的数目相等，或者仅仅相差1
//此时若相差1，则取min中的最小值，若相等取max的最大值和min最小值和的一半

template<typename T>
class DynamicArrary
{
    public:
        void Insert(T num)
        {
            //奇数
            if(((min.size() + max.size()) &1)==0 )
            {
                if(max.size()>0 && num<max[0])
                {
                    //新插入的数比大顶堆最大还大
                    //则把最大拿出来，放在小顶堆
                    //这样就保证大顶堆的数始终小于小顶堆，即排序前进
                    max.push_back(num);
                    push_heap(max.begin(),max.end(),less<T>());
                    //以上两个命令连用，把num放在合适位置，放在max的最大值位置

                    num=max[0];

                    //以下两个也连用，使最大值出来
                    pop_heap(max.begin(),max.end(),less<T>());
                    max.pop_back();
                }
                min.push_back(num);
                push_heap(min.begin(), min.end(), greater<T>());
        }
```

```
            else
            {
                //同理新插入的num比最小堆的最小值小，把最小值拿出来，放在最大堆里
                if(min.size()>0 && min[0]<num)
                {

                    min.push_back(num);
                   push_heap(min.begin(),min.end(),greater<T>());

                num=min[0];

                pop_heap(min.begin(),min.end(),greater<T>());
                min.pop_back();
                }

                max.push_back(num);
                push_heap(max.begin(),max.end(),less<T>());
            }
        }

        T GetMedian()
        {
            int size=min.size()+max.size();
            if(size==0)
                //throw exception("no numbers are available");
            {
                   std::logic_error ex("no numbers are available");
                   throw std::exception(ex);
            }


            T median=0;
            if((size & 1)==1)//奇数
                median=min[0];
            else
              median=(min[0]+max[0])/2;

            return median;
        }

    private:
        vector<T> min;
        vector<T> max;
};



int main(int argc, char* argv[])
{
    DynamicArrary<double> numbers;

    numbers.Insert(5);
    numbers.Insert(2);
    numbers.Insert(3);
    numbers.Insert(4);
```

```
    numbers.Insert(1);


        double result=numbers.GetMedian();

        cout<<result<<endl;



    return 0;
}
```

## 42.连续子数组的最大和

**输入一个整型数组，有正有负，数组中的一个或多个组成一个子数组，求最大和**

```
// 面试题42：连续子数组的最大和
// 题目：输入一个整型数组，数组里有正数也有负数。数组中一个或连续的多个整
// 数组成一个子数组。求所有子数组的和的最大值。要求时间复杂度为O(n)。
//{1,-2,3,10,-4,7,2,-5}  {3,10,-4,7,2} 18

 #include <iostream>
using namespace std;

//初始化和为0,第一步加上第一个数字1,此时和为1，第二步加上数字-2,和变为-1
//第三步加上数字3,则为2。比3本身小，所以不如从3开始的数组和大，因此之前的抛弃

//因此3+10-4=9，此时把之前得到的最大和13记录到最大数组和中，以后相大于13就更新
//遇到小于上一次更新的数就退出

//为了区分字子数组和的最大值是0和无效输入这两种情况，定义一个全局变量来标记是否有效输入
bool g_InvalidInput=false;

int FindGreatestSumOfSubArray(int *data,int leng)
{
    if(data==NULL || leng<0)
    {
        g_InvalidInput=true;
        return 0;
    }

    g_InvalidInput=false;

    int nCurSum=0;
    int nGreatestSum=0x80000000;
    for(int i=0;i<leng;i++)
    {
        if(nCurSum<=0)
            nCurSum=data[i];

        else
           nCurSum+=data[i];

        if(nCurSum>nGreatestSum)
            nGreatestSum=nCurSum;
```

```
    }

    return nGreatestSum;
}
int main()
{
    // 典型输入，单调升序的数组的一个旋转
      int data[] = {1, -2, 3, 10, -4, 7, 2, -5};

    int result=FindGreatestSumOfSubArray(data,sizeof(data)/sizeof(int));

    cout<<result<<endl;

    return 0;
}
```

## 43.1~n整数中1出现的次数

**输入一个整数，求1～n这n个整数的十进制表示中1出现的次数**

```
/*
// 面试题43：从1到n整数中1出现的次数
// 题目：输入一个整数n，求从1到n这n个整数的十进制表示中1出现的次数。例如
// 输入12，从1到12这些整数中包含1 的数字有1，10，11和12，1一共出现了5次。
*/

#include <iostream>
using namespace std;

#include <cstring>

//每次通过对10取余看看个位数是不是1
//如果这个数大于10,则除以10之后再判断个位数是不是1
int NumberOf1(unsigned int n);
int NOf1(unsigned int n)
{
    int count=0;
    for(unsigned int i=1;i<=n;++i)
        count+=NumberOf1(i);

    return count;
}

int NumberOf1(unsigned int n)
{
    int count=0;
    while(n)
    {
        if(n%10==1)
            count++;

        n=n/10;
    }
    return count;
}
```

```
//方法2
//例如输入的数字特别大 n=21345
//分成两段 1～1345  1346～21345
//先来分析1346～21345  1出现在10000～19999中的最高为万位的次数是1×10×10×10×10
//1出现在10000～12345中的次数是除去最高数字之后剩下的数字再加上1（2345+1=2346）

//分析1出现在除最高位之外的其他4位数的情况。
//1346～21345 分成1346～11345 和 11346～21345
//每一段上下的4位数选择其中一个为1,其余3个可有0～9的选择，即4×10×10×10×2

//至于在1～1345中1出现的次数，可用递归楼
//因为把21345最高位去掉得到的就是1345
int PowerBase10(unsigned int n);
int NumberOfOf1(const char* strN);

int NumOf1(int n)
{
    if(n<=0)
        return 0;

    char strN[50];
    //把数字转成字符串
    sprintf(strN,"%d",n);

    return NumberOfOf1(strN);
}

int NumberOfOf1(const char* strN)
{
    if(!strN || *strN<'0' || *strN>'9' || *strN=='\0')
        return 0;

    int first=*strN-'0';//取最高位
    cout<<"first "<<first<<endl;


    unsigned int leng=static_cast<unsigned int>(strlen(strN));

    if(leng==1 && first==0)
        return 0;

    if(leng==1 && first>0)
        return 1;

    //count1是数字10000～19999的第一位为1的数字出现的次数
    int count1=0;
    if(first>1)
        count1=PowerBase10(leng-1);
    else if(first==1)
        count1=atoi(strN+1)+1;//atoi为string转int

    //count2是1346～21345除第一位之外的数位中的数目
    int count2=first*(leng-1)*PowerBase10(leng-2);

    //count3是1～1345中的数目
    int count3=NumberOfOf1(strN+1)+1;

    return count1+count2+count3;
```

```
}

int PowerBase10(unsigned int n)
{
    int result=1;
    for(unsigned int i=0;i<n;++i)
        result*=10;

    return result;
}

int main()
{
    int result1= NOf1(21345);
    cout<<result1<<endl;

    int result2=NumOf1(21345);
    cout<<result2<<endl;


    return 0;

}
```

## 44.数字序列中某一位的数字

**数字以012345678910111213.....求第5位是5,第13位是1**

```
// 面试题44：数字序列中某一位的数字
// 题目：数字以0123456789101112131415…的格式序列化到一个字符序列中。在这
// 个序列中，第5位（从0开始计数）是5，第13位是1，第19位是4，等等。请写一
// 个函数求任意位对应的数字。

#include <iostream>
using namespace std;

#include <algorithm>

//判断1001位是什么
//序列的前10位0-9只有10位数字，显然不符合，在从后面紧跟着的序列中找 1001-10=991位

//接下来10～99 共90×2=180位数字，180<991,因此也跳过，从后面继续找 991-180=881

//接下来100～999 共900×3=2700 2700>881，因此很明显在这里面，开始找，因为881=270×3+1
//这意味着第811位是从100开始的第270个数字 即370的中间一位 ，即是7
int countOfIntegers(int digits);
int digitAtIndex(int index,int digits);
int beginNumber(int digits);

int digitAtIndex(int index)//1001
{
    if(index<0)
        return -1;
```

```
    int digits=1;
    while(true)
    {
        int numbers=countOfIntegers(digits);

        if(index<numbers*digits)
            return digitAtIndex(index,digits);

        index-=digits*numbers;//811-1*90

        digits++;

    }

    return -1;
}

//得到m位的数字共多少个。输入2是（10～99）90个，输入3是900个
int countOfIntegers(int digits)
{
    if(digits==1)
        return 10;

    int count=(int)std::pow(10,digits-1);

    return 9*count;
}

//当我们知道要找的那一位数字位于某m位数之中后，就可以用下面函数找出那一位数字了
int digitAtIndex(int index,int digits)
{
    int number=beginNumber(digits)+index/digits;//100+811/3=370
    int indexFromRight=digits-index%digits;//3-811%3=2
    for(int i=1;i<indexFromRight;++i)
        number/=10;//37

    return number%10;//7
}

//得到m位的第一个数字，第一个两位数是10,第一个三位数是100
int beginNumber(int digits)
{
    if(digits==1)
        return 0;

    return (int)std::pow(10,digits-1);
}

int main()
{

    int result=digitAtIndex(1001);

    cout<<result<<endl;


    return 0;
}
```

## 45.把数组排成最小的数

**输入数组{3,32,321}，排成的最小数为321323**

```
// 面试题45：把数组排成最小的数
// 题目：输入一个正整数数组，把数组里所有数字拼接起来排成一个数，打印能拼
// 接出的所有数字中最小的一个。例如输入数组{3, 32, 321}，则打印出这3个数
// 字能排成的最小数字321323。

#include <iostream>
using namespace std;
#include <cstring>
#include <algorithm>

//比较两个数，不是大小，而是m和n哪一个应该放在前面
//mn<nm则 m放前面，反之

//考虑可能出现大数问题，因此将数字转成字符串处理

// int型整数用十进制表示最多只有10位
const int g_MaxNumberLeng=10;

char* g_StrCombine1=new char[g_MaxNumberLeng*2+1];
char* g_StrCombine2=new char[g_MaxNumberLeng*2+1];
int compare(const void* strNumber1, const void* strNumber2);

void PrintMinNumber(int* numbers, int leng)
{
    if(numbers==NULL || leng<=0)
        return;

    char** strNumbers=(char**)(new int[leng]);
    //数字转字符串
    for(int i=0;i<leng;++i)
    {
        strNumbers[i]=new char[g_MaxNumberLeng+1];
        sprintf(strNumbers[i],"%d",numbers[i]);
    }

    //调用库函数进行排序
    qsort(strNumbers,leng,sizeof(char*),compare);

    for(int i=0;i<leng;++i)
        printf("%s",strNumbers[i]);
    printf("\n");

    for(int i=0;i<leng;++i)
        delete[] strNumbers[i];
    delete[] strNumbers;
}

// 如果[strNumber1][strNumber2] > [strNumber2][strNumber1], 返回值大于0
// 如果[strNumber1][strNumber2] = [strNumber2][strNumber1], 返回值等于0
// 如果[strNumber1][strNumber2] < [strNumber2][strNumber1], 返回值小于0
```

```
int compare(const void* strNumber1, const void* strNumber2)
{
    // [strNumber1][strNumber2]
    strcpy(g_StrCombine1, *(const char**)strNumber1);
    strcat(g_StrCombine1, *(const char**)strNumber2);

    // [strNumber2][strNumber1]
    strcpy(g_StrCombine2, *(const char**)strNumber2);
    strcat(g_StrCombine2, *(const char**)strNumber1);

    return strcmp(g_StrCombine1, g_StrCombine2);
}

int main(int argc, char* argv[])
{
  int numbers[] = {3, 32, 321};
  PrintMinNumber(numbers,sizeof(numbers)/sizeof(int));


    return 0;
}
```

## 49.丑数

**只包含质因子2,3,5的数称丑数，求从小到大的顺序的第1500个丑数**

```
// 面试题49：丑数
// 题目：我们把只包含因子2、3和5的数称作丑数（Ugly Number）。求按从小到
// 大的顺序的第1500个丑数。例如6、8都是丑数，但14不是，因为它包含因子7。
// 习惯上我们把1当做第一个丑数。

#include <iostream>
using namespace std;

//若一个数能被2整除，就连续除以2,同理连续除以3,5
//如果最后得到的是1,那么这个数就是丑数

bool IsUgly(int number)
{
    while(number % 2 == 0)
        number /= 2;
    while(number % 3 == 0)
        number /= 3;
    while(number % 5 == 0)
        number /= 5;

    return (number == 1) ? true : false;
}

//只需要按照顺序判断每个整数是不是丑数
int GetUglyNumber(int index)//1500
{
    if(index <= 0)
        return 0;
```

```
    int number = 0;
    int uglyFound = 0;
    while(uglyFound < index)
    {
        ++number;

        if(IsUgly(number))
            ++uglyFound;
    }

    return number;
}


int main()
{


    int result=GetUglyNumber(15);
    cout<<result<<endl;

    return 0;
}
```

## 51.数组中的逆序对

**前一个数字大于后面的数字，组成一个逆序对**

```
// 面试题51：数组中的逆序对
// 题目：在数组中的两个数字如果前面一个数字大于后面的数字，则这两个数字组
// 成一个逆序对。输入一个数组，求出这个数组中的逆序对的总数。

 #include <iostream>
using namespace std;

//数组{7,5,6,4}中一共存在5个逆序对，分别是（7,6），（7,5），（7,4），（6,4），（5,4）

//先分成两个数组，{7,5} {6,4}，判断可以构成
//然后进行排序{5,7 }  {4,6}
//在上面上个数组定义两个指针分别指向最后一个元素，p1-7  p2-6 同时定义一个辅助数组依次存放最大值
//p1>p2,则存在数值2中的逆序对，，，，跑p1<p2则把存在

 int InversePairsCore(int* data, int* copy,int start,int end);

int InversePairs(int* data,int leng)
{
    if(data==nullptr || leng<0)
        return 0;

    int* copy=new int[leng];
    for(int i=0;i<leng;++i)
        copy[i]=data[i];

    int count=InversePairsCore(data,copy,0,leng-1);

}
```

```
 int InversePairsCore(int* data, int* copy,int start,int end)
 {
     if(start==end)
     {
         copy[start]=data[start];
         return 0;
     }

    int leng=(end-start)/2;

    int left=InversePairsCore(copy,data,start,start+leng);//前半部分额
    int right=InversePairsCore(copy,data,start+leng+1,end);//后半部分

    //i初始化为前半段最后一个数字的下标
    int i=start+leng;
    //j初始化后半段最后一个数字的下标
    int j=end;

    int indexCopy=end;
    int count=0;

    while(i>=start && j>=start+leng+1)
    {
        if(data[i]>data[j])
        {
            //肯定存在逆对
            //把最大数放在辅助数组
            copy[indexCopy--]=data[i--];
            //就是i和后面的几个都组成逆对
            count+=j-start-leng;

        }

        else
        {
            copy[indexCopy--]=data[j--];//不存在逆对，把j放在辅助数组里，j--前移再继续
比较
        }
    }

    for(;i>=start;--i)
        copy[indexCopy--]=data[i];

    for(;j>=start+leng+1;--j)
        copy[indexCopy--]=data[j];

    return left+right+count;

 }

int main()
{
     int data[] = { 1, 2, 3, 4, 7, 6, 5 };

    int result=InversePairs(data,sizeof(data)/sizeof(int));

    cout<<result<<endl;
```

```
    return 0;
}
```

## 53_1.数字在排序数组中出现的次数

```
// 面试题53（一）：数字在排序数组中出现的次数
// 题目：统计一个数字在排序数组中出现的次数。例如输入排序数组{1, 2, 3, 3,
// 3, 3, 4, 5}和数字3，由于3在这个数组中出现了4次，因此输出4。
#include <iostream>
using namespace std;

#include <cstring>

//排序数组，想到用二分查找，找到第一个3和最后一个3不就完事了
//第一个
//中间数字比k大，则在左边，否则在右边，继续查找
//最后一个k

int GetFirstK(int* data,int leng,int k,int start,int end)
{
    if(start>end)
        return -1;

    int mid=(start+end)/2;
    int midData=data[mid];

    if(midData==k)
    {
        //比较左一个或者右一个是否为k，不是的话就是第一个左或右的
        if((mid>0 && data[mid-1]!=k) || mid==0)
            return mid;

        else
          end=mid-1;

    }
    else if(midData>k)
        end=mid-1;
    else
        start=mid+1;

    return GetFirstK(data,leng,k,start,end);
}

int GetLastK(int* data,int leng,int k,int start,int end)
{
    if(start>end)
        return -1;

    int mid=(start+end)/2;
    int midData=data[mid];

    if(midData==k)
    {
        if((mid<leng-1 && data[mid+1]!=k) || mid==leng-1)
            return mid;
```

```
        else
            start=mid+1;

    }
    else if(mid<k)
        start=mid+1;
    else
        end=mid-1;

    return GetLastK(data,leng,k,start,end);

}

int GetNumberOfK(int* data,int leng,int k)
{
    int count=0;
    if(data!=nullptr && leng>0)
    {
        int first=GetFirstK(data,leng,k,0,leng-1);
        int last=GetLastK(data,leng,k,0,leng-1);

        if(first>-1 && last>-1)
            count=last-first+1;
    }

    return count;
}

int main()
{
      int data[] = {3, 3, 3, 3, 4, 5};
    int result=GetNumberOfK(data,sizeof(data)/sizeof(int),3);
    cout<<result<<endl;


    return 0;

}
```

## 53_2.0~n-1中缺失的数字

**递增排序数组中的所有数字都是唯一的，在0～n-1有且只有一个数字不在该数组中，找出来**

```
// 面试题53（二）：0到n-1中缺失的数字
// 题目：一个长度为n-1的递增排序数组中的所有数字都是唯一的，并且每个数字
// 都在范围0到n-1之内。在范围0到n-1的n个数字中有且只有一个数字不在该数组
// 中，请找出这个数字。

#include <iostream>
using namespace std;

#include <algorithm>

//由于是排序好的，因此0在下标为0的位置，1在下标为1的位置。。。。
//由于m不再数组中，m+1处在下标为m的位置，m+2处在下标为m+1的位置。。。。。。
//因此问题转换为在排序数组中找到第一个值和下标不相等的元素
```

```
int GetMissingN(const int* data, int leng)
{
    if(data==nullptr || leng<0)
        return -1;

    int left=0;
    int right=leng-1;

    while(left<=right)
    {
        int mid=(right+left)>>1;
        if(data[mid]!=mid)
        {
            if(mid==0 || data[mid-1]==mid-1)
                return mid;

            right=mid-1;
        }
        else
            left=mid+1;
    }

    if(left==leng)
        return leng;

    return -1;
}

int main()
{
   int data[] = { 1, 2, 3, 4, 5 };

    int result=GetMissingN(data,sizeof(data)/sizeof(int));

    cout<<result<<endl;


    return 0;
}
```

## 57_1.和为S的两个数

**一个递增排序的数组和一个数字S,若有多对的数字和为S，输出任意一对就好**

```
// 面试题57（一）：和为s的两个数字
// 题目：输入一个递增排序的数组和一个数字s，在数组中查找两个数，使得它们
// 的和正好是s。如果有多对数字的和等于s，输出任意一对即可。

#include <iostream>
using namespace std;

#include <algorithm>


//定义两个指向，最前面后最后面，相加的和大于 s，则移动后面的数往前一个格子，依次
void  FindTwoOfSum(const int* data, int leng,int s,int* output)
```

```
{
    if(data==nullptr || leng<0)
        return;

    int left=0;
    int right=leng-1;

    while(left<=right)
    {
        int CurrentSum=data[left]+data[right];
        if(CurrentSum>s)
            right--;
        else if(CurrentSum<s)
            left++;
        else
        {
            output[0]=data[left];
            output[1]=data[right];
            break;//这个不能少啊啊啊啊啊，找到了就退出来啊啊啊啊啊
        }
    }
}

int main()
{
   int data[] = { 1, 2, 4, 7, 11,15 };
   int output[]={};
   FindTwoOfSum(data,sizeof(data)/sizeof(int),15,output);
   for(int i=0;i<2;i++)
        cout<<output[i]<<endl;



    return 0;
}
```

## 57_2.和为S的连续正数序列

**输入一个正数S，打印出所有和为S的连续正数序列，至少包含两个数**

```
// 面试题57（二）：为s的连续正数序列
// 题目：输入一个正数s，打印出所有和为s的连续正数序列（至少含有两个数）。
// 例如输入15，由于1+2+3+4+5=4+5+6=7+8=15，所以结果打印出3个连续序列1～5、
// 4～6和7～8。
#include <iostream>
using namespace std;

#include <cstring>


//定义small=1,big=2,计算和，当和小于s时候，增大big，当和大于s时候，增大small
//结束条件是 small= (1+s)/2
 void Print(int small,int big);
void FindContinuousSequence(int s)
{
    int small=1;
```

```
    int big=2;

    while(small<((s+1)/2))
    {
        int CurrentSum=0;
        for(int i=small;i<=big;i++)
            CurrentSum+=i;

        if(CurrentSum==s)
        {
            // cout<<small;
            // cout<<big;


            //此时已经满足一對了
            //满足了就调用打印函数呗
            Print(small,big);

            cout<<endl;

            //继续寻找下一对哦
            big++;
        }

        else if(CurrentSum>s)
            small++;
        else
            big++;
    }
}

 void Print(int small,int big)
 {
     for(int i=small;i<=big;i++)
        cout<<i<<endl;
 }

int main()
{
      int s=15;
    FindContinuousSequence(s);


    return 0;

}
```

## 61.扑克牌中的顺子

**随机抽出n张牌，大小王可以看成任意数字，是不是顺子**

```
// 面试题61：扑克牌的顺子
// 题目：从扑克牌中随机抽5张牌，判断是不是一个顺子，即这5张牌是不是连续的。
// 2～10为数字本身，A为1，J为11，Q为12，K为13，而大、小王可以看成任意数字。

 #include <iostream>
using namespace std;
```

```
//首先将数组排序，
//其次统计数组中0的个数
//最后统计排序之后的数组中相邻数字之间的空缺数
//如果空缺的总数小于或者等于0的个数，那么这个数组就是连续的
#include <cstdio>
#include <cstdlib>

int compare(const void *arg1, const void *arg2);
bool IsContinuous(int* data,int leng)
{
    if(data==nullptr || leng<1)
        return false;

    qsort(data,leng,sizeof(int),compare);

    int NumOf0=0;
    int NumOfK=0;

    //统计数组中0的个数
    for(int i=0;i<leng && data[i]==0;++i)
        ++NumOf0;

    //统计数组中的空数目
    int small=NumOf0;
    int big=small+1;
    while(big<leng)
    {
        //如果两个数字相等，有对子，不可能是顺子
        if(data[small]==data[big])
            return false;
        //空缺数
        NumOfK=data[big]-data[small]-1;
        small=big;
        ++big;
    }

    return (NumOfK>NumOf0)?false:true;
}

int compare(const void *arg1,const void* arg2)
{
    return *(int*)arg1-*(int*)arg2;
}
int main()
{
      int data[] = { 1, 3, 2, 5, 4 };

   bool result=IsContinuous(data,sizeof(data)/sizeof(int));

    cout<<result<<endl;

    return 0;
}
```

## 62.圆圈中最后剩下的数字

**约瑟夫环问题**

```cpp
// 面试题62：圆圈中最后剩下的数字
// 题目：0, 1, …, n-1这n个数字排成一个圆圈，从数字0开始每次从这个圆圈里
// 删除第m个数字。求出这个圆圈里剩下的最后一个数字。

#include <cstdio>
#include <list>

#include <iostream>


using namespace std;

//约瑟夫问题
//定义一个list,创建一个共有n个节点的环形链表，然后每次在这个环形链表中删除第m个节点
//std::list模拟一个环形链表，因此每当迭代器扫描到链表末尾的时候，记得把迭代器移动到链表头部
//这相当于按照顺序在一个圆圈里遍历了

int LastRemaining(unsigned int n, unsigned int m)
{
    if(n < 1 || m < 1)
        return -1;

    unsigned int i = 0;

    //环形链表赋值
    list<int> numbers;
    for(i = 0; i < n; ++ i)
        numbers.push_back(i);

    //迭代器
    list<int>::iterator current = numbers.begin();
    while(numbers.size() > 1)
    {
        //循环找到要删除的额节点
        for(int i = 1; i < m; ++ i)
        {
            current ++;
            //迭代器扫描到链表结尾时候，将其指向第一个数
            if(current == numbers.end())
                current = numbers.begin();
        }

        list<int>::iterator next = ++ current;
        if(next == numbers.end())
            next = numbers.begin();

        -- current;
        numbers.erase(current);
        //第二次循环的开始
        current = next;
    }

    return *(current);
}
```

```
int main()
{
    int result=LastRemaining(5,3);
    cout<<result<<endl;

    return 0;
}
```

## 63.股票的最大利润

**一组数组，找到两个数，是他们的差最大**

```
// 面试题63：股票的最大利润
// 题目：假设把某股票的价格按照时间先后顺序存储在数组中，请问买卖交易该股
// 票可能获得的利润是多少？例如一只股票在某些时间节点的价格为{9, 11, 8, 5,
// 7, 12, 16, 14}。如果我们能在价格为5的时候买入并在价格为16时卖出，则能
// 收获最大的利润11。

#include <iostream>
using namespace std;

//找到数组中相差最大值的两个数
//在卖价固定时候，买价最小，差值最大

int  MaxDiff(const int* data, unsigned leng)
{
    if(data==nullptr && leng<2)
        return 0;

    int min=data[0];
    int maxDiff=data[1]-min;

    for(int i=2;i<leng;++i)
    {
        if(data[i-1]<min)
            min=data[i-1];

        int currentDiff=data[i]-min;
        if(currentDiff>maxDiff)
            maxDiff=currentDiff;
    }
    return maxDiff;
}

int main()
{

    int data[] = { 4, 1, 3, 2, 5 };
    int result=MaxDiff(data, sizeof(data) / sizeof(int));
    cout<<result<<endl;

    return 0;
}
```

# 字符串

## 19.正则表达式匹配

**‘.’表示任意一个字符，’*‘前面字符可出现任意次，字符串的所有字符匹配整个模式**

```
/*
// 面试题19：正则表达式匹配
// 题目：请实现一个函数用来匹配包含'.'和'*'的正则表达式。模式中的字符'.'
// 表示任意一个字符，而'*'表示它前面的字符可以出现任意次（含0次）。在本题
// 中，匹配是指字符串的所有字符匹配整个模式。例如，字符串"aaa"与模式"a.a"
// 和"ab*ac*a"匹配，但与"aa.a"及"ab*a"均不匹配。
*/

#include <iostream>
using namespace std;

bool matchCore(const char* str,const char* pattern);
bool match(const char* str,const char* pattern)
{
    if(str==NULL || pattern==NULL)
        return false;

    return matchCore(str,pattern);
}

bool matchCore(const char* str,const char* pattern)
{
    if(*str=='\0' && *pattern=='\0')
        return true;

    if(*str!='\0' && *pattern=='\0')
        return false;

    //str:abc pattern: abs
    //str: abc pattern: .cd
    if(*str==*pattern || (*pattern=='.' && *str!='\0'))
        return matchCore(str+1,pattern+1);


    //str: abc pattern: a*c
    if(*(pattern+1)=='*')
    {
         if(*pattern==*str || (*pattern=='.' && *str!='\0'))
         {
             return matchCore(str+1,pattern+2)//abc  .*d
               || matchCore(str+1,pattern) //abc, a*d
               || matchCore(str,pattern+2);//abc .*d 忽略 .*


         }

         else
         {
             matchCore(str,pattern+2);//abc .*d 忽略 .*
         }

    }
```

```
    return false;
 }
int main()
{

    const char* str="abc";
    const  char* pattern="abc";
    bool result=match(str,pattern);

    cout<<result<<endl;

    return 0;

}
```

## 20.表示数值的字符串

### 判断字符串是否表示数值

```
/*
// 面试题20：表示数值的字符串
// 题目：请实现一个函数用来判断字符串是否表示数值（包括整数和小数）。例如，
// 字符串“+100”、“5e2”、“-123”、“3.1416”及“-1E-16”都表示数值，但“12e”、
// “1a3.14”、“1.2.3”、“+-5”及“12e+5.4”都不是
*/

#include <iostream>
using namespace std;

bool scanUnsignedInter(const char** str);
bool scanInteger(const char** str);
//A[.[B][e|E C]]或者 .B[e|E C]
bool isNumeric(const char* str)
{
    if(str==NULL)
        return false;

    //B是一个无符号整数
    bool numeric=scanInteger(&str);

     // 如果出现'.'，接下来是数字的小数部分
    if(*str == '.')
    {
        ++str;

        // 下面一行代码用||的原因：
        // 1. 小数可以没有整数部分，例如.123等于0.123；
        // 2. 小数点后面可以没有数字，例如233.等于233.0；
        // 3. 当然小数点前面和后面可以有数字，例如233.666
        numeric = scanUnsignedInter(&str) || numeric;
    }

    // 如果出现'e'或者'E'，接下来跟着的是数字的指数部分
    if(*str == 'e' || *str == 'E')
    {
        ++str;
```

```
        // 下面一行代码用&&的原因：
        // 1. 当e或E前面没有数字时，整个字符串不能表示数字，例如.e1、e1；
        // 2. 当e或E后面没有整数时，整个字符串不能表示数字，例如12e、12e+5.4
        numeric = numeric && scanInteger(&str);
    }

    return numeric && *str == '\0';
}

//扫描可能以表示正负的‘+’或者‘-’为起点的0-9的数位，带正负号的整数
//用来匹配A或C
bool scanInteger(const char** str)
{
    if(**str=='+' || **str=='-')
        ++(*str);

    return scanUnsignedInter(str);
}

//扫描字符串中0-9的位数，匹配B
bool scanUnsignedInter(const char** str)
{
   const char* before=*str;
   while(**str!='\0' && **str>='0' && **str<='9')
        ++(*str);

    //当str中存在若干0-9的数字，返回true
    return *str>before;
}
int main()
{
  const char* str="221e45";
   bool result=isNumeric(str);

    cout<<result<<endl;

    return 0;

}
```

## 38.字符串的排列

**输入一个字符串，打印该字符串中字符的所有排列**

```
/*
// 面试题38：字符串的排列
// 题目：输入一个字符串，打印出该字符串中字符的所有排列。例如输入字符串abc，
// 则打印出由字符a、b、c所能排列出来的所有字符串abc、acb、bac、bca、cab和cba。
*/

#include <iostream>
using namespace std;

//将第一个数，和后面的数分成两组，固定第一个，依次和后面的进行交换
void Permutation(char* pStr,char* pBegin);

void Permutation(char* pStr)
```

```
{
    if(pStr==NULL)
        return;

    Permutation(pStr,pStr);
}

 void Permutation(char* pStr,char* pBegin)
 {
     //指向当前我们执行排列操作的字符串的第一个字符
     if(*pBegin=='\0')
        printf("%s\n",pStr);

    else
    {
        for(char* pCh=pBegin;*pCh!='\0';++pCh)
        {
            char tmp=*pCh;
            *pCh=*pBegin;
            *pBegin=tmp;

            Permutation(pStr,pBegin+1);


            tmp=*pCh;
            *pCh=*pBegin;
            *pBegin=tmp;

        }
    }



 }
int main()
{
   char string4[] = "abc";
   Permutation(string4);

    return 0;

}
```

46.把数字翻译成字符串

0翻译成'a',1翻译成'b'....一个数字可能有多少种不同的翻译

```
#include <vector>
#include <string>
#include <iostream>
#include <algorithm>

using namespace std;

//递归
int getTransCount(const string& num,int k)
{
```

```
    int len=num.size();
    //超过长度，只有一种可能
    if(k>=len-1)
        return 1;

    //[k]和[k+1]可以组合成一个字符，有两种方案
    if(k+1<len)
    {
        int sum=(num[k]-'0')*10+num[k+1]-'0';
        if(sum>9 && sum<26)
            return getTransCount(num,k+1)+getTransCount(num,k+2);

    }

    return getTransCount(num,k+1);

}

//动态规划
int cnt[70]={1,1};//记录用的数组，初始化f(0)=1,f(1)=1
int getTranslation(int num)
{
    if(num<0)
        return 0;

    //整数转成字符串
    string numS=to_string(num);
    int numSize=numS.size();

    for(int i=1;i!=numSize;i++)
    {
        //如果前一个数字是1,和当前数字总能合并翻译
        //f(i)=f(i-1)+f(i-2)

        if(numS[i-1]=='1')
            cnt[i+1]=cnt[i]+cnt[i-1];
        //如果前一个数字是2,那么只有当前数字是0--5才能合并翻译
        else if(numS[i-1]=='2')
        {
            if(numS[i]>='0'&&numS[i]<='5')
                cnt[i+1]=cnt[i]+cnt[i-1];

        }

        //前一个数字是其他数字，都不能合并翻译
        else
            cnt[i+1]=cnt[i];
    }

    return cnt[numSize];
}



int main()
{
    int test=12258;
    int count1=getTranslation(test);
```

```
    string numS=to_string(test);
    int k=2;
    int count2=getTransCount(numS,k);
    cout<<count1<<" "<<count2<<endl;
}
```

## 48.最长不含重复字符的子字符串

### 并计算该字符串的长度

```
/*
// 面试题48：最长不含重复字符的子字符串
// 题目：请从字符串中找出一个最长的不包含重复字符的子字符串，计算该最长子
// 字符串的长度。假设字符串中只包含从'a'到'z'的字符。
*/

#include <iostream>
using namespace std;

#include <cstring>

 //暴力法，就是遍历它的子字符串，看看子字符串是否包含重复的
 bool hasDuplication(const std::string& str, int position[]);

 int longestSub(const std::string& str)
 {
     int longest=0;
     int* position=new int[26];
     for(int start=0;start<str.length();++start)
     {
         for(int end=start;end<str.length();++end)
         {
             int count=end-start+1;
             //将字符串str从start开始截取count个字符组成子字符串
             const std::string& substring=str.substr(start,count);
             if(!hasDuplication(substring,position))
             {
                   if(count>longest)
                        longest=count;

             }
             else

                 break;


         }
     }

     delete[] position;
     return longest;

 }

 bool hasDuplication(const std::string& str, int position[])
 {
```

```
     //每个字符上次出现在字符串中的下标
     for(int i=0;i<26;++i)
        position[i]=-1;

    for(int i=0;i<str.length();++i)
    {
        int indexInPosition=str[i]-'a';
       if(position[indexInPosition]>=0)
                    return true;

        position[indexInPosition]=indexInPosition;

    }

    return false;



 }


int main()
{
  const std::string input = "acfrarabc";
   int result=longestSub(input);
  cout<<result<<endl;

    return 0;

}
```

## 50.第一个只出现一次的字符

**在字符串中找出第一个只出现一次的字符**

```
/*
// 面试题50（一）：字符串中第一个只出现一次的字符
// 题目：在字符串中找出第一个只出现一次的字符。如输入"abaccdeff"，则输出
// 'b'。

*/

#include <iostream>
using namespace std;

#include <cstring>

 //创建一个哈系表，key是每个单词，value是出现的次数
 char FirstNotRepeatingChar(const char* pString)
 {
     if(pString==NULL)
        return '\0';

     const int tableSize=256;
     unsigned int hashTable[tableSize];

     for(unsigned int i=0;i<tableSize;++i)
```

```
        hashTable[i]=0;//value都初始化0次

    const char* pHashKey=pString;
    while(*(pHashKey)!='\0')
        hashTable[*(pHashKey++)]++;

    pHashKey=pString;
    while(*pHashKey!='\0')
    {
        if(hashTable[*pHashKey]==1)
            return *pHashKey;

        pHashKey++;
    }

    return '\0';
 }


int main()
{
  const  char* pString= "google";

   char result=FirstNotRepeatingChar(pString);
  cout<<result<<endl;

    return 0;

}
```

## 58_1.翻转单词顺序

**输入一个英文句子，翻转句子中单词的顺序**

```
/*
// 面试题58（一）：翻转单词顺序
// 题目：输入一个英文句子，翻转句子中单词的顺序，但单词内字符的顺序不变。
// 为简单起见，标点符号和普通字母一样处理。例如输入字符串"I am a student. "，
// 则输出"student. a am I"。
*/

#include <iostream>
using namespace std;

#include <cstring>

//先整体翻转，再单个单词进行翻转
void Reverse(char *pBegin,char *pEnd)
{
    if(pBegin==NULL || pEnd==NULL)
        return;

    while(pBegin<pEnd)
    {
        char tmp=*pBegin;
        *pBegin=*pEnd;
        *pEnd=tmp;
```

```
        pBegin++;
        pEnd--;
    }
}

//翻转句子
char* ReverseSentence(char *pData)
{
    if(pData==NULL)
        return NULL;

    char *pBegin=pData;

    //找到字符转最后一个字符
    char *pEnd=pData;
    while(*pEnd !='\0')
        pEnd++;
    pEnd--;

    //翻转整个句子
    Reverse(pBegin,pEnd);

    //翻转句子中的单词
    pBegin=pEnd=pData;
    while(*pBegin !='\0')
    {
        if(*pBegin==' ')
        {
            pBegin++;
            pEnd++;
        }

        else if(*pEnd==' ' || *pEnd=='\0')
        {
            Reverse(pBegin,--pEnd);
            pBegin=++pEnd;//放在第二个单词上了
        }
        else
        {
            pEnd++;
        }

    }

    return pData;
}
int main()
{
  char input[] = "I am a student.";;
  char* result=ReverseSentence(input);

  cout<<result<<endl;

    return 0;

}
```

58_2.左旋转字符串

把字符串前面的若干个字符转移到字符串的尾巴部分

```
/*
// 面试题58（一）：翻转单词顺序
// 题目：输入一个英文句子，翻转句子中单词的顺序，但单词内字符的顺序不变。
// 为简单起见，标点符号和普通字母一样处理。例如输入字符串"I am a student. ",
// 则输出"student. a am I"。
*/

#include <iostream>
using namespace std;

#include <cstring>

//先整体翻转，再单个单词进行翻转
void Reverse(char *pBegin,char *pEnd)
{
    if(pBegin==NULL || pEnd==NULL)
        return;

    while(pBegin<pEnd)
    {
        char tmp=*pBegin;
        *pBegin=*pEnd;
        *pEnd=tmp;

        pBegin++;
        pEnd--;
    }
}

//根据58_1的参考，先翻转字符串前面字符，再翻转后面的字符，最后翻转整体字符

char* LeftRotateString(char* pStr,int n)
{
    if(pStr!=NULL)
    {
        int leng=static_cast<int>(strlen(pStr));
        if(leng>0 && n>0 && n<leng)
        {
            char* pFirstStart=pStr;
            char* pFirstEnd=pStr+n-1;
            char* pSecondStart=pStr+n;
            char* pSecondEnd=pStr+leng-1;

            //翻转字符串前面n个字符
            Reverse(pFirstStart,pFirstEnd);

            //翻转字符串的后面字符
            Reverse(pSecondStart,pSecondEnd);

            //翻转整个字符
            Reverse(pFirstStart,pSecondEnd);
        }
    }
```

```
    return pStr;
}
int main()
{

    char input[] = "abcdefg";
    char* result=LeftRotateString(input,2);

    cout<<result<<endl;

    return 0;

}
```

## 单链表

### 头文件

```
#ifndef LIST_NODE
#define LIST_NODE

#include <iostream>
using namespace std;

struct ListNode
{
    int value;
    ListNode* next;
    ListNode* random;//复杂链表增加一项
};


   ListNode* CreateListNode(int value)
   {
    ListNode* pNode = new ListNode();
    pNode->value= value;
    pNode->next= nullptr;

    return pNode;
   }

  void ConnectListNodes(ListNode* pCurrent, ListNode* pNext)
  {
    if(pCurrent == nullptr)
         return;

     pCurrent->next= pNext;
   }

     void PrintListNode(ListNode* pNode)
    {
     if(pNode == nullptr)
         return;

     else
        cout<<pNode->value<<endl;
```

```
 }

void DestroyNode(ListNode* pHead)
{

 ListNode* pNode = pHead;
 while(pNode != nullptr)
 {
     pHead = pHead->next;
     delete pNode;
     pNode = pHead;
 }
}
 unsigned int GetListLength(ListNode* pHead)
{
 unsigned int nLength = 0;
 ListNode* pNode = pHead;
 while(pNode != nullptr)
 {
     ++nLength;
     pNode = pNode->next;
 }

 return nLength;
}

//复杂链表的创建
ListNode* CreatComplexNode(int val)
{
    ListNode* pNode=new ListNode();
    pNode->value=val;
    pNode->next=NULL;
    pNode->random=NULL;

    return pNode;
}

void BuildNodes(ListNode* pNode,ListNode* pNext,ListNode* pRandom)
{
    if(pNode!=NULL)
    {
        pNode->next=pNext;
        pNode->random=pRandom;
    }
}

void PrintComplexList(ListNode* phead)
{
    ListNode* pNode=phead;
    while(pNode!=NULL)
    {
        cout<<pNode->value<<endl;
        if(pNode->random!=NULL)
         cout<<pNode->random->value<<endl;;

        pNode=pNode->next;
    }
}
```

```
#endif
```

## 6.从尾到头打印链表

**输入一个链表的头节点，从尾到头反过来打印出每个节点**

```
#include <iostream>
#include <stack>//栈
#include "ListNode.h"

using namespace std;

//顺序打印
void PrintList(ListNode* root)
{
    if(root==nullptr)
       return;

    ListNode* phead=root;
    while(phead!=NULL)
    {
        PrintListNode(phead);
        phead=phead->next;
    }
}

//逆序打印
//利用栈的先进后出原理，链表入栈，出栈就完成逆序打印
void PrintList_Return(ListNode* root)
{
    if(root==NULL)
        return;

   std::stack<int> stk;
   ListNode* phead=root;
   while(phead!=NULL)
   {
       stk.push(phead->value);
       phead=phead->next;
   }

   while(!stk.empty())
   {
       cout<<stk.top()<<endl;
       stk.pop();
   }

}

//递归实现
//每访问到一个节点的时候，先递归输出它后面的节点，再输出该节点
void PrintList_DiGui(ListNode* root)
{
    if(root==NULL)
```

```
        return;
    cout<<"222"<<endl;
   ListNode* phead=root;
   if(phead!=NULL)
   {
       if(phead->next!=NULL)
       {
         PrintList_DiGui(phead->next);
         cout<<"111"<<endl;
       }
       cout<<"333"<<endl;
       cout<<phead->value<<endl;
   }
/*
222
222
222
222
222
333
5
111
333
4
111
333
3
111
333
2
111
333
1
*/

}

int main()
{

    ListNode* pNode1 =  CreateListNode(1);
    ListNode* pNode2 =  CreateListNode(2);
    ListNode* pNode3 =  CreateListNode(3);
    ListNode* pNode4 = CreateListNode(4);
    ListNode* pNode5 = CreateListNode(5);

     ConnectListNodes(pNode1, pNode2);
     ConnectListNodes(pNode2, pNode3);
     ConnectListNodes(pNode3, pNode4);
     ConnectListNodes(pNode4, pNode5);

     PrintList(pNode1);
     PrintList_Return(pNode1);
     PrintList_DiGui(pNode1);

    return 0;

}
```

## 18_1.在O(1)时间内删除链表节点

**给定单向链表的头指针和一个节点指针，删除该节点指针**

```
// 面试题18（一）：在O(1)时间删除链表结点
// 题目：给定单向链表的头指针和一个结点指针，定义一个函数在O(1)时间删除该
// 结点。

#include <cstdio>
#include "ListNode.h"

//思路：将被删除节点的值的下一个值拷贝给被删除节点，而此时被删除节点的指向其
//下一个节点的下一个节点
void DeleteNode(ListNode** pListHead, ListNode* pToBeDeleted)
{
    if(!pListHead || !pToBeDeleted)
        return;

    // 要删除的结点不是尾结点
    if(pToBeDeleted->next!= nullptr)
    {
        ListNode* pNext = pToBeDeleted->next;
        pToBeDeleted->value = pNext->value;
        pToBeDeleted->next = pNext->next;

        delete pNext;
        pNext = nullptr;
    }
    // 链表只有一个结点，删除头结点（也是尾结点）
    else if(*pListHead == pToBeDeleted)
    {
        delete pToBeDeleted;
        pToBeDeleted = nullptr;
        *pListHead = nullptr;
    }
    // 链表中有多个结点，删除尾结点
    else
    {
        ListNode* pNode = *pListHead;//双重指针
        while(pNode->next!= pToBeDeleted)
        {
            pNode = pNode->next;
        }

        pNode->next= nullptr;
        delete pToBeDeleted;
        pToBeDeleted = nullptr;
    }
}


void PrintfOri(ListNode* pListHead, ListNode* pNode)
{
    printf("The original list is: \n");
```

```
    ListNode* phead=pListHead;
    while(phead!=NULL)
    {
        PrintListNode(phead);
        phead=phead->next;
    }

    printf("The node to be deleted is: \n");
    PrintListNode(pNode);

}

void PrintfDel(ListNode* pListHead, ListNode* pNode)
{
     DeleteNode(&pListHead, pNode);

    printf("The result list is: \n");
    ListNode* phead=pListHead;
    while(phead!=NULL)
    {
        PrintListNode(phead);
        phead=phead->next;
    }

}

//补充知识
void change1(int *a,int &b)
{
     a=&b;//10
}


void change2(int *a,int &b)
{
    *a=b;//20
}

void change3(int **a,int &b)
{
    *a=&b;//20
}

void change4(int& a)
{
    a=20;//20
}

int main(int argc, char* argv[])
{
    ListNode* pNode1 = CreateListNode(1);
    ListNode* pNode2 = CreateListNode(2);
    ListNode* pNode3 = CreateListNode(3);
    ListNode* pNode4 = CreateListNode(4);
    ListNode* pNode5 = CreateListNode(5);

    ConnectListNodes(pNode1, pNode2);
    ConnectListNodes(pNode2, pNode3);
```

```
    ConnectListNodes(pNode3, pNode4);
    ConnectListNodes(pNode4, pNode5);

    PrintfOri(pNode1, pNode3);

    PrintfDel(pNode1, pNode3);

    //补充结果
    int a1=10,b1=20;
    int *p_a1=&a1;
    change1(p_a1,b1);
    printf("%d\n",*p_a1);//10

    /*
    其实系统有一个隐含的操作，就是_a=p_a,让_a也指向p_a所指向的地址，
    而我们对_a的指向进行改变的时候并不会影响p_a的指向。
    但是如果对_a指向的地址所存放的数据进行操作的话，
    p_a指向的地址所存放的数据也会改变，因为他们指向的同一地址
    */

    //这样输出就变成了20，因为是对_a和p_a共同指向的地址所存放的数据进行操作。
    int a2=10,b2=20;
    int *p_a2=&a2;
    change2(p_a2,b2);
    printf("%d\n",*p_a2);//20

    /*
    但有时候我们需要的是把p_a的指向改变，让p_a=&b,
    而不仅仅是对p_a所指向的地址所存放的数据进行改变。
    这时候我们就需要用到双重指针，顾名思义，即指向指针的指针，
    就是指向一个指针的地址,这样就可以把一个指针的地址传递进去，
    从而直接对那个指针进行操作，比如
    */

   //这里面隐含操作就是 int** _a=&p_a;这样*_a就是p_a了，所以输出为20
    int a3=10,b3=20;
    int *p_a3=&a3;
    change3(&p_a3,b3);
    printf("%d\n",*p_a3);//20

    int a4=10,b4=20;
    int *p_a4=&a4;
    change4(a4);
    printf("%d\n",a4);//20


    return 0;
}
```

## 18_2.删除链表中重复的节点

**1-2-3-3-4-4-5删除后变成1-2-5**

```

```

## 22.链表中倒数第K个节点

**链表的尾节点是倒数第1个节点**

```
// 面试题22：链表中倒数第k个结点
// 题目：输入一个链表，输出该链表中倒数第k个结点。为了符合大多数人的习惯，
// 本题从1开始计数，即链表的尾结点是倒数第1个结点。例如一个链表有6个结点，
// 从头结点开始它们的值依次是1、2、3、4、5、6。这个链表的倒数第3个结点是
// 值为4的结点。

#include <cstdio>
#include "ListNode.h"

ListNode* FindKthToTail(ListNode* pListHead, unsigned int k)
{
     //
    if(pListHead == nullptr || k == 0)
        return nullptr;

    ListNode *pAhead = pListHead;
    ListNode *pBehind = nullptr;

    //快的先往前走k-1之后，慢的开始从0走
    //当快的走到头，慢的就找到了k
    for(unsigned int i = 0; i < k - 1; ++ i)
    {
        if(pAhead->next != nullptr)
            pAhead = pAhead->next;
        else
        {
            return nullptr;
        }
    }

    pBehind = pListHead;
    while(pAhead->next != nullptr)
    {
        pAhead = pAhead->next;
        pBehind = pBehind->next;
    }

    return pBehind;
}



int main(int argc, char* argv[])
{


    ListNode* pNode1 = CreateListNode(1);
    ListNode* pNode2 = CreateListNode(2);
    ListNode* pNode3 = CreateListNode(3);
    ListNode* pNode4 = CreateListNode(4);
    ListNode* pNode5 = CreateListNode(5);

    ConnectListNodes(pNode1, pNode2);
    ConnectListNodes(pNode2, pNode3);
```

```
    ConnectListNodes(pNode3, pNode4);
    ConnectListNodes(pNode4, pNode5);

    printf("expected result: 4.\n");
    ListNode* pNode = FindKthToTail(pNode1, 2);
    PrintListNode(pNode);

    DestroyList(pNode1);

    return 0;
}
```

## 23.链表中环的入口节点

**一个链表中包含环，如何找出环的入口结点？**

```
// 面试题23：链表中环的入口结点
// 题目：一个链表中包含环，如何找出环的入口结点？例如，在图3.8的链表中，
// 环的入口结点是结点3。

#include <cstdio>
#include "ListNode.h"

ListNode* MeetingNode(ListNode* pHead)
{
    if(pHead == nullptr)
        return nullptr;

    ListNode* pSlow = pHead->next;//慢的每次走一步
    if(pSlow == nullptr)
        return nullptr;

    ListNode* pFast = pSlow->next;//快的每次走两步
    while(pFast != nullptr && pSlow != nullptr)
    {
        if(pFast == pSlow)
            return pFast;

        pSlow = pSlow->next;

        pFast = pFast->next;
        if(pFast != nullptr)
            pFast = pFast->next;
    }

    return nullptr;
}

ListNode* EntryNodeOfLoop(ListNode* pHead)
{
    ListNode* meetingNode = MeetingNode(pHead);
    if(meetingNode == nullptr)
        return nullptr;

    // 得到环中结点的数目
```

```
    int nodesInLoop = 1;
    ListNode* pNode1 = meetingNode;
    while(pNode1->next != meetingNode)
    {
        pNode1 = pNode1->next;
        ++nodesInLoop;
    }

    // 先移动pNode1，次数为环中结点的数目
    pNode1 = pHead;
    for(int i = 0; i < nodesInLoop; ++i)
        pNode1 = pNode1->next;

    // 再移动pNode1和pNode2
    ListNode* pNode2 = pHead;
    while(pNode1 != pNode2)
    {
        pNode1 = pNode1->next;
        pNode2 = pNode2->next;
    }

    return pNode1;
}


int main(int argc, char* argv[])
{

    ListNode* pNode1 = CreateListNode(1);
    ListNode* pNode2 = CreateListNode(2);
    ListNode* pNode3 = CreateListNode(3);
    ListNode* pNode4 = CreateListNode(4);
    ListNode* pNode5 = CreateListNode(5);

    ConnectListNodes(pNode1, pNode2);
    ConnectListNodes(pNode2, pNode3);
    ConnectListNodes(pNode3, pNode4);
    ConnectListNodes(pNode4, pNode5);
    ConnectListNodes(pNode5, pNode3);

     ListNode*  EntryNode=EntryNodeOfLoop(pNode1);
     PrintListNode(EntryNode);

   // DestroyNode(pNode1);
   // DestroyNode(pNode2);
   // DestroyNode(pNode3);
   // DestroyNode(pNode4);
   // DestroyNode(pNode5);

    return 0;
}
```

## 24.反转链表

**定义一个函数，输入一个链表的头结点，反转该链表并输出反转后链表的头节点**

```cpp
// 面试题24：反转链表
// 题目：定义一个函数，输入一个链表的头结点，反转该链表并输出反转后链表的
// 头结点。

#include <cstdio>
#include "ListNode.h"

//定义三个节点，一个当前节点，一个前一个，一个后一个

ListNode* ReverseList(ListNode* pHead)
{
    ListNode* pReversedHead = nullptr;//返回的翻转链表节点的头节点
    ListNode* pNode = pHead;//当前节点
    ListNode* pPrev = nullptr;//前一个节点
    while(pNode != nullptr)
    {

        ListNode* pNext = pNode->next;//后一个节点
        cout<<"00  "<<pNext->value<<endl;

        if(pNext == nullptr)
        {
            pReversedHead = pNode;
           // cout<<"0 "<<pReversedHead->value<<endl;
        }


       // cout<<"1 "<<pPrev->value<<endl;
       //当前节点的下一个为空，为了留给他的前一个节点空间，下面把当前节点的值给他的前一个
       //而自己指向下一节点
       //完成翻转
        pNode->next = pPrev;
       // cout<<"2 "<<pNode->value<<endl;

       // cout<<"33 "<<pPrev->value<<endl;//此时第一次还是空

        pPrev = pNode;
        cout<<"3 "<<pNode->value<<endl;
        cout<<"4 "<<pPrev->value<<endl;
        pNode = pNext;
        cout<<"5 "<<pNode->value<<endl;
    }

    return pReversedHead;
}




int main(int argc, char* argv[])
{

    ListNode* pNode1 = CreateListNode(1);
    ListNode* pNode2 = CreateListNode(2);
    ListNode* pNode3 = CreateListNode(3);
    ListNode* pNode4 = CreateListNode(4);
```

```
    ListNode* pNode5 = CreateListNode(5);

    ConnectListNodes(pNode1, pNode2);
    ConnectListNodes(pNode2, pNode3);
    ConnectListNodes(pNode3, pNode4);
    ConnectListNodes(pNode4, pNode5);


    ListNode* pReversedHead = ReverseList(pNode1);

    printf("The reversed list is: \n");

    while(pReversedHead!=NULL)
    {
        PrintListNode(pReversedHead);
        pReversedHead=pReversedHead->next;
    }


    return 0;
}
```

## 25.合并两个排序的链表

**输入两个递增排序的链表，合并这两个链表并使新链表中的结点仍然是按照递增排序的**

```
// 面试题25：合并两个排序的链表
// 题目：输入两个递增排序的链表，合并这两个链表并使新链表中的结点仍然是按
// 照递增排序的。例如输入图3.11中的链表1和链表2，则合并之后的升序链表如链
// 表3所示。

#include <cstdio>
#include "ListNode.h"

ListNode* Merge(ListNode* pHead1, ListNode* pHead2)
{
    if(pHead1 == nullptr)
        return pHead2;
    else if(pHead2 == nullptr)
        return pHead1;

    ListNode* pMergedHead = nullptr;

    if(pHead1->value< pHead2->value)
    {
        pMergedHead = pHead1;
        pMergedHead->next = Merge(pHead1->next, pHead2);//下一个节点跟上
    }
    else
    {
        pMergedHead = pHead2;
        pMergedHead->next = Merge(pHead1, pHead2->next);
    }
```

```
    return pMergedHead;//利用这样的返回
}

ListNode* List1()
{
    ListNode* pNode1 = CreateListNode(1);
    ListNode* pNode2 = CreateListNode(3);
    ListNode* pNode3 = CreateListNode(5);
    ListNode* pNode4 = CreateListNode(7);

    ConnectListNodes(pNode1, pNode2);
    ConnectListNodes(pNode2, pNode3);
    ConnectListNodes(pNode3, pNode4);

    return pNode1;
}

ListNode* List2()
{

    ListNode* pd1 = CreateListNode(2);
    ListNode* pd2 = CreateListNode(4);
    ListNode* pd3 = CreateListNode(6);
    ListNode* pd4 = CreateListNode(8);

    ConnectListNodes(pd1, pd2);
    ConnectListNodes(pd2, pd3);
    ConnectListNodes(pd3, pd4);

    return pd1;
}

int main(int argc, char* argv[])
{

    ListNode* pNode1=List1();
    ListNode* pd1=List2();

    ListNode* pHead = Merge(pNode1,pd1);

    printf("The HeBing list is: \n");

    while(pHead!=NULL)
    {
        PrintListNode(pHead);
        pHead=pHead->next;
    }


    return 0;
}
```

## 52.两个链表的第一个公共节点

**输入两个链表，找出它们的第一个公共结点**

```
```

```cpp
// 面试题52：两个链表的第一个公共结点
// 题目：输入两个链表，找出它们的第一个公共结点。

#include <cstdio>
#include "ListNode.h"

ListNode* FindFirstCommonNode(ListNode *pHead1, ListNode *pHead2)
{
    // 得到两个链表的长度
    unsigned int nLength1 = GetListLength(pHead1);
    unsigned int nLength2 = GetListLength(pHead2);
    int nLengthDif = nLength1 - nLength2;

    ListNode* pListHeadLong = pHead1;
    ListNode* pListHeadShort = pHead2;
    if(nLength2 > nLength1)
    {
        pListHeadLong = pHead2;
        pListHeadShort = pHead1;
        nLengthDif = nLength2 - nLength1;
    }

   cout<<"leng1 "<<nLength1<<endl;
   cout<<"leng2 "<<nLength2<<endl;

    // 先在长链表上走几步，再同时在两个链表上遍历
    for(int i = 0; i < nLengthDif; ++i)
        pListHeadLong = pListHeadLong->next;

    while((pListHeadLong != nullptr) &&
        (pListHeadLong->value != pListHeadShort->value))
    {
        pListHeadLong = pListHeadLong->next;
        pListHeadShort = pListHeadShort->next;
    }

    // 得到第一个公共结点
    ListNode* pFisrtCommonNode = pListHeadLong;

    return pFisrtCommonNode;
}




ListNode* List1()
{
    ListNode* pNode1 = CreateListNode(1);
    ListNode* pNode2 = CreateListNode(3);
    ListNode* pNode3 = CreateListNode(5);
    ListNode* pNode4 = CreateListNode(7);

    ConnectListNodes(pNode1, pNode2);
    ConnectListNodes(pNode2, pNode3);
    ConnectListNodes(pNode3, pNode4);

    return pNode1;
```

```
}

ListNode* List2()
{

    ListNode* pd1 = CreateListNode(2);
    ListNode* pd2 = CreateListNode(2);
    ListNode* pd3 = CreateListNode(7);


    ConnectListNodes(pd1, pd2);
    ConnectListNodes(pd2, pd3);

    return pd1;
}

int main(int argc, char* argv[])
{
    ListNode* pNode1=List1();
    ListNode* pd1=List2();

    ListNode* pResult = FindFirstCommonNode(pNode1, pd1);
    PrintListNode(pResult);


    return 0;
}
```

# 二叉树

## 头文件

```
#ifndef BINARY_TREE
#define BINARY_TREE

#include <iostream>
using namespace std;

struct BinaryTreeNode
{
   int value;
   BinaryTreeNode* left;
   BinaryTreeNode* right;

   //指向父节点的
   BinaryTreeNode* parent;
};

BinaryTreeNode* Create(int val)
{
    BinaryTreeNode* pNode=new BinaryTreeNode();
    pNode->value=val;
    pNode->left=NULL;
    pNode->right=NULL;

    //////////
```

```
    pNode->parent=NULL;

    return pNode;
}

void Connect(BinaryTreeNode* pParent,BinaryTreeNode* pLeft,
BinaryTreeNode* pRight)
{
    if(pParent!=NULL)
    {
        pParent->left=pLeft;
        pParent->right=pRight;

        /////////
        if(pLeft!=NULL)
            pLeft->parent=pParent;
        if(pRight!=NULL)
            pRight->parent=pParent;
    }
}

void PrintTreeNode(const BinaryTreeNode* pNode)
{
    if(pNode!=NULL)
    {
       cout<<"node: "<<pNode->value<<endl;

       if(pNode->left!=NULL)
            cout<<"left: "<<pNode->left->value<<endl;
        else
            cout<<"left is NULL"<<endl;

        if(pNode->right!=NULL)
             cout<<"right: "<<pNode->right->value<<endl;
        else
             cout<<"right is NULL"<<endl;
    }

    else
      cout<<"node is NULL "<<endl;

    cout<<endl;
}

void PrintTree(const BinaryTreeNode* pRoot)
{
    PrintTreeNode(pRoot);

    if(pRoot!=NULL)
    {
        if(pRoot->left!=NULL)
            PrintTree(pRoot->left);
        if(pRoot->right!=NULL)
             PrintTree(pRoot->right);
    }
}

void DestroyTree(BinaryTreeNode* pRoot)
```

```
{
    if(pRoot!=NULL)
    {
        BinaryTreeNode* pLeft=pRoot->left;
        BinaryTreeNode* pRight=pRoot->right;

        delete pRoot;
        pRoot=NULL;

        DestroyTree(pLeft);
        DestroyTree(pRight);
    }
}

#endif
```

## 遍历方式

```
#include <iostream>
using namespace std;

#include "BinaryTree.h"

#include <stack>

//前序遍历 ,递归实现，根左右
void QianXu(BinaryTreeNode* root)
{
    if(root==NULL)
        return;

   // BinaryTreeNode* pNode=root;
    if(root!=NULL)
     cout<<root->value<<endl;
    if(root->left!=NULL)
        QianXu(root->left);
    if(root->right!=NULL)
        QianXu(root->right);

}

//使用栈的前序遍历
/*
栈实现前序遍历较简单，由于每次先输出根节点，再输出左节点随后是右节点。
算法是:
1、若栈非空输出根节点，并出栈
2、将右节点压栈（如果存在）
3、将左节点压栈（如果存在）
4、重复第1步直到栈空
注意：之所以先压右节点是考虑了栈的特性，这样在迭代过程中可以先拿到左节点处理。（栈的先入后出）
*/
void StackQianXu(BinaryTreeNode* root)
{
    if(root==NULL)
        return;
```

```
    std::stack<BinaryTreeNode*> QianStack;
    BinaryTreeNode* p=root;

    QianStack.push(p);
    while(!QianStack.empty())
        {
            p=QianStack.top();
            cout<<p->value<<endl;
            QianStack.pop();
            if(p->right!=NULL)
                QianStack.push(p->right);
            if(p->left!=NULL)
                QianStack.push(p->left);
        }

}

//中序遍历 ,递归实现，左根右
void ZhongXu(BinaryTreeNode* root)
{
    if(root==NULL)
        return;

   // BinaryTreeNode* pNode=root;

        if(root->left!=NULL)
           ZhongXu(root->left);
        cout<<root->value<<endl;
         //if(root!=NULL)
          // ZhongXu(root);
         if(root->right!=NULL)
          ZhongXu(root->right);

}

/*
栈的中序遍历需要套两层循环，由于需要先输出左节点，
因此必须向下查找直到左节点为空才能输出。处理逻辑如下：

1、如果栈顶元素非空且左节点存在，将其入栈，重复该过程。若不存在则进入第2步
2、若栈非空，输出栈顶元素并出栈。判断刚出栈的元素的右节点是否存在，
不存在重复第2步，存在则将右节点入栈，跳至第1步
*/

//思路没问题，写的代码有问题，没出正确结果？？？？？
void StackZhongXu(BinaryTreeNode* root)
{
    if(root==NULL)
        return;

    std::stack<BinaryTreeNode*> ZhongStack;
    BinaryTreeNode* p=root;
    ZhongStack.push(p);

    while(!ZhongStack.empty())
        {

            while(ZhongStack.top()->left!=NULL)
```

```
            {
              ZhongStack.push(ZhongStack.top()->left);
            }

            while(!ZhongStack.empty() )
            {
                BinaryTreeNode* p=ZhongStack.top();
                cout<<p->value<<endl;
                ZhongStack.pop();

               if(p->right!=NULL)
                {
                   ZhongStack.push(p->right);
                    break;
                }

            }

/*
   while(p!=NULL)
       {

           while(p!=NULL)
           {
             //  if(p->right)
               //{
                   // ZhongStack.push(p->right);
              // }
               ZhongStack.push(p);
               cout<<"top "<<ZhongStack.top()->value<<endl;
               p=p->left;
           }

           p=ZhongStack.top();
           //cout<<"00"<<endl;
           cout<<"p "<<p->value<<endl;
           while(!ZhongStack.empty() )
           {
               // cout<<"00"<<endl;
               cout<<p->value<<endl;
               if(p->right==NULL)
               {
                   ZhongStack.pop();
                   p=ZhongStack.top();
               }
              else if(p->right!=NULL)
              {
                 ZhongStack.push(p->right);
                 cout<<"00 "<<p->right->value<<endl;
                 // p=p->right;
                  break;
              }
           }

          if(p->right!=NULL)
            {
                 ZhongStack.push(p->right);
                 cout<<"00 "<<p->right->value<<endl;
```

```
                    p=p->right;
                    break;
             }
             */
       }
}

//后序遍历 ,递归实现，左根右
void HouXu(BinaryTreeNode* root)
{
    if(root==NULL)
        return;

     if(root->left!=NULL)
        HouXu(root->left);
         //if(root!=NULL)
          // ZhongXu(root);
    if(root->right!=NULL)
        HouXu(root->right);
    cout<<root->value<<endl;

}

/*
后序遍历在中序的双层循环的基础上需要加入一个记录，专门记录上一次出栈的节点。步骤如下：

1、如果栈顶元素非空且左节点存在，将其入栈，重复该过程。若不存在则进入第2步（该过程和中序遍历一致）
2、判断上一次出栈节点是否当前节点的右节点，或者当前节点是否存在右节点，
满足任一条件，将当前节点输出，并出栈。否则将右节点压栈。跳至第1步
*/

//具体没有用图画出来 ？？？？？？？
void StackHouXu(BinaryTreeNode* root)
{
    if(root==NULL)
        return;

    std::stack<BinaryTreeNode*> HouStack;
    HouStack.push(root);

     BinaryTreeNode* lastpop=NULL;

    while(!HouStack.empty())
        {

            while(HouStack.top()->left!=NULL)
            {
               HouStack.push(HouStack.top()->left);
            }

            while(!HouStack.empty() )
            {
                if(lastpop==HouStack.top()->right ||
                 HouStack.top()->right==NULL )
                 {
                     cout<< HouStack.top()->value<<endl;
                     lastpop= HouStack.top();
```

```
                    HouStack.pop();
                }

                else if( HouStack.top()->right!=NULL)
                {
                    HouStack.push(HouStack.top()->right);
                    break;
                }

            }

        }
}

int main()
{

    BinaryTreeNode* node1= Create(10);
    BinaryTreeNode* node2= Create(6);
    BinaryTreeNode* node3= Create(14);
    BinaryTreeNode* node4= Create(4);
    BinaryTreeNode* node5= Create(8);
    BinaryTreeNode* node6= Create(12);
    BinaryTreeNode* node7= Create(16);

    Connect(node1,node2,node3);
    Connect(node2,node4,node5);
    Connect(node3,node6,node7);

   // PrintTree(node1);
    QianXu(node1);
    cout<<endl;
    StackQianXu(node1);
    cout<<endl;

    ZhongXu(node1);
    cout<<endl;
    StackZhongXu(node1);
    cout<<endl;

    HouXu(node1);
    cout<<endl;
    StackHouXu(node1);


    return 0;

}
```

## 7.重建二叉树

**输入某二叉树的前序遍历和中序遍历的结果，请重建出该二叉树**

```
/*
// 面试题7: 重建二叉树
```

```
// 题目：输入某二叉树的前序遍历和中序遍历的结果，请重建出该二叉树。假设输
// 入的前序遍历和中序遍历的结果中都不含重复的数字。例如输入前序遍历序列{1,
// 2, 4, 7, 3, 5, 6, 8}和中序遍历序列{4, 7, 2, 1, 5, 3, 8, 6}，则重建出
// 图2.6所示的二叉树并输出它的头结点。
*/


#include <iostream>
using namespace std;

#include "BinaryTree.h"

#include <exception>


BinaryTreeNode* ConstructCore(int* startPre,int* endPre,int* startMid,int*
endMid);

BinaryTreeNode* Construct(int* pre,int* mid,int leng)
{
    if(pre==NULL || mid==NULL || leng<=0)
        return NULL;

    return ConstructCore(pre,pre+leng-1,mid,mid+leng-1);
}

BinaryTreeNode* ConstructCore(int* startPre,int* endPre,int* startMid,int*
endMid)
{
    //前序遍历第一个数字是根节点
    int rootValue=startPre[0];
    BinaryTreeNode* root=new BinaryTreeNode();
    root->value=rootValue;
    root->left=root->right=NULL;

    if(startPre==endPre)
    {
        if(startMid==endMid && *startPre==*startMid)
            return root;

        else
         //throw std::exception("Invalid input.");
         return NULL;

    }

    //中序遍历中找到根节点的值
    int* rootMid=startMid;
    while(rootMid<=endMid && *rootMid !=rootValue)
        ++rootMid;

    if(rootMid == endMid && *rootMid!=rootValue)
        //throw std::exception("Invalid input.");
        return NULL;

   int leftLeng=rootMid-startMid;
   int* leftPreEnd=startPre+leftLeng;//左树的最后一个节点
```

```
    //构建左子树
    if(leftLeng>0)
    {
        root->left=ConstructCore(startPre+1,leftPreEnd,
        startMid,rootMid-1);
    }

    //构建右子树
    if(leftLeng<endPre-startPre)
    {
        root->right=ConstructCore(leftPreEnd+1,endPre,
        rootMid+1,endMid);
    }

    return root;
}



int main()
{
    const int leng= 8;
    int pre[leng] = {1, 2, 4, 7, 3, 5, 6, 8};
    int mid[leng] = {4, 7, 2, 1, 5, 3, 8, 6};

    BinaryTreeNode* root=Construct(pre,mid,leng);
    PrintTree(root);



}
```

## 8.二叉树的下一个结点

**给定一棵二叉树和其中的一个结点，如何找出中序遍历顺序的下一个结点**

```
// 面试题8：二叉树的下一个结点
// 题目：给定一棵二叉树和其中的一个结点，如何找出中序遍历顺序的下一个结点？
// 树中的结点除了有两个分别指向左右子结点的指针以外，还有一个指向父结点的指针。


#include <iostream>
using namespace std;

#include "BinaryTree.h"


BinaryTreeNode* GetNext(BinaryTreeNode* pNode)
{
    if(pNode == nullptr)
        return nullptr;

    BinaryTreeNode* pNext = nullptr;//定义返回值，在函数里面赋值
```

```
    if(pNode->right!= nullptr)
    {
        BinaryTreeNode* pRight = pNode->right;
        while(pRight->left != nullptr)
            pRight = pRight->left;

        pNext = pRight;
    }

    else if(pNode->parent!= nullptr)
    {
        BinaryTreeNode* pCurrent = pNode;
        BinaryTreeNode* pParent = pNode->parent;

   //该节点没有右子树，并且他还是他父节点的右子节点
   //沿着指向父节点的指针一直向上遍历，直到找到一个是他父节点的左子节点的节点
   //如果这样的节点存在，那么这个节点的父节点就是要找的节点
        while(pParent != nullptr && pCurrent == pParent->right)
        {
            pCurrent = pParent;
            pParent = pParent->parent;
        }
     //没有右子树，节点是他父节点的左子节点，
    //下一个节点就是他的父节点
        pNext = pParent;
    }

    return pNext;
}



int main()

{
    //       8
//        6      10
//       5 7    9  11
    BinaryTreeNode* pNode8 = Create(8);
    BinaryTreeNode* pNode6 = Create(6);
    BinaryTreeNode* pNode10 = Create(10);
    BinaryTreeNode* pNode5 = Create(5);
    BinaryTreeNode* pNode7 = Create(7);
    BinaryTreeNode* pNode9 = Create(9);
    BinaryTreeNode* pNode11 = Create(11);

    Connect(pNode8, pNode6, pNode10);
    Connect(pNode6, pNode5, pNode7);
    Connect(pNode10, pNode9, pNode11);


 BinaryTreeNode* next=GetNext(pNode10);
 cout<<next->value<<endl;

}
```

## 26.树的子结构

**输入两棵二叉树A和B，判断B是不是A的子结构**

```
/*
// 面试题26：树的子结构
// 题目：输入两棵二叉树A和B，判断B是不是A的子结构。
*/

#include <iostream>
using namespace std;

#include "BinaryTree.h"

bool Equal(double num1,double num2);
bool Tree1HaveTree2(BinaryTreeNode* p1,BinaryTreeNode* l8);

//在树1中查找与树2中根节点一样的节点
bool EqualRoot2(BinaryTreeNode* p1,BinaryTreeNode* l8)
{
    bool result=false;

    // if(p1==NULL || l8==NULL)
    //     return false;

    if(p1!=NULL && l8!=NULL)
    {
        //如果找到根相等，继续判断左右
         if(Equal(p1->value,l8->value))
            result=Tree1HaveTree2(p1,l8);

        //根不等
         if(!result)
            result=EqualRoot2(p1->left,l8);
        if(!result)
            result=EqualRoot2(p1->right,l8);
    }

    return result;
}

//根相等找到后，接着找左右是否相等
bool Tree1HaveTree2(BinaryTreeNode* p1,BinaryTreeNode* l8)
{
    bool equal=false;

    if(p1==NULL)
        return false;
    if(l8==NULL)
        return true;

    //不等时候
    if(!Equal(p1->value,l8->value))
        return false;

    //递归
    if(Equal(p1->value,l8->value))
    {
```

```
        Tree1HaveTree2(p1->left,l8->left);
        equal=true;
    }

    if(equal)
    {
        Tree1HaveTree2(p1->right,l8->right);
    }

    return equal;

    // return Tree1HaveTree2(p1->left,l8->left)
    // && Tree1HaveTree2(p1->right,l8->right);
}


//比较两者是否相等
bool Equal(double num1,double num2)
{
    if((num1-num2)>-0.000001 && (num1-num2)<0.0000001)
        return true;
    else
        return false;

}


int main()
{
    BinaryTreeNode* p1=Create(8);
    BinaryTreeNode* p2=Create(8);
    BinaryTreeNode* p3=Create(7);
    BinaryTreeNode* p4=Create(9);
    BinaryTreeNode* p5=Create(2);
    BinaryTreeNode* p6=Create(4);
    BinaryTreeNode* p7=Create(7);

    Connect(p1, p2, p3);
    Connect(p2, p4, p5);
    Connect(p5, p6, p7);
    PrintTree( p1);

   BinaryTreeNode* l8=Create(8);
   BinaryTreeNode* l9=Create(9);
   BinaryTreeNode* l2=Create(2);
    Connect(l8, l9, l2);
    PrintTree(l8);

   bool finalresult=true;
   finalresult=EqualRoot2(p1,l8);
   cout<<"finalresult  "<<finalresult<<endl;

   DestroyTree(p1);
    DestroyTree(l8);
}
```

## 27.二叉树的镜像

**请完成一个函数，输入一个二叉树，该函数输出它的镜像**

```
/*
// 面试题27：二叉树的镜像
// 题目：请完成一个函数，输入一个二叉树，该函数输出它的镜像。
*/

#include <iostream>
using namespace std;

#include "BinaryTree.h"

//前序遍历这个树的每个节点，若此节点有子节点，就交换它的两个子节点
//当交换完所有非叶子节点的左，右子节点之后，得到树的镜像

void MirrorRecursively(BinaryTreeNode *root)
{
    if(root==NULL)
        return;
    if(root->left==NULL && root->right==NULL)
        return;

    //交换根节点的6,10
    BinaryTreeNode *temp=root->left;
    root->left=root->right;
    root->right=temp;

   //交换10的左右子节点
    if(root->left)
    {
        cout<<"000"<<endl;
        MirrorRecursively(root->left);

    }

    //交换6的左右子节点
    if(root->right)
    {
        cout<<"111"<<endl;
       MirrorRecursively(root->right);

    }



}


int main()
{
    BinaryTreeNode* p1=Create(8);
    BinaryTreeNode* p2=Create(6);
    BinaryTreeNode* p3=Create(10);
    BinaryTreeNode* p4=Create(5);
    BinaryTreeNode* p5=Create(7);
    BinaryTreeNode* p6=Create(9);
    BinaryTreeNode* p7=Create(11);
```

```
    Connect(p1, p2, p3);
    Connect(p2, p4, p5);
    Connect(p3,p6,p7);
    PrintTree(p1);

   MirrorRecursively(p1);
     PrintTree(p1);
   return 0;


}
```

## 28.对称的二叉树

**请实现一个函数，用来判断一棵二叉树是不是对称的。如果一棵二叉树和它的镜像一样，那么它是对称的**

```
/*
// 面试题28：对称的二叉树
// 题目：请实现一个函数，用来判断一棵二叉树是不是对称的。如果一棵二叉树和
// 它的镜像一样，那么它是对称的。
*/

#include <iostream>
using namespace std;

#include "BinaryTree.h"

//前序遍历和对称前序遍历来判断

bool isSymmetrical(BinaryTreeNode* root1, BinaryTreeNode* root2);

bool isSymmetrical(BinaryTreeNode* root)
{
    return isSymmetrical(root,root);
}

bool isSymmetrical(BinaryTreeNode* root1, BinaryTreeNode* root2)
{
    if(root1==NULL && root2==NULL)
        return true;
    if(root1==NULL || root2==NULL)
        return false;

    if(root1->value!=root2->value)
        return false;

    return isSymmetrical(root1->left,root2->right) && isSymmetrical(root1-
>right,root2->left);
}

int main()
{
    BinaryTreeNode* p1=Create(8);
    BinaryTreeNode* p2=Create(6);
    BinaryTreeNode* p3=Create(6);
```

```
    BinaryTreeNode* p4=Create(5);
    BinaryTreeNode* p5=Create(7);
    BinaryTreeNode* p6=Create(7);
    BinaryTreeNode* p7=Create(5);

    Connect(p1, p2, p3);
    Connect(p2, p4, p5);
    Connect(p3,p6,p7);
    PrintTree(p1);

  bool result=isSymmetrical(p1);
  cout<<"result "<<result<<endl;

   return 0;


}
```

## 32_1.不分行从上往下打印二叉树

**从上往下打印出二叉树的每个结点，同一层的结点按照从左到右的顺序打印**

```
/*
// 面试题32（一）：不分行从上往下打印二叉树
// 题目：从上往下打印出二叉树的每个结点，同一层的结点按照从左到右的顺序打印
*/

#include <iostream>
using namespace std;

#include <deque>

#include "BinaryTree.h"

void PrintFromTopToBottom(BinaryTreeNode* root)
{
    if(root==NULL)
        return;

    //建立一个队列，存储打印节点的左右节点，
    //当打印此节点后，陆续打印左右节点
    std::deque<BinaryTreeNode*> deque;

    deque.push_back(root);

    while(deque.size())
    {
        BinaryTreeNode* pNode=deque.front();//返回队列中最早元素，并未删除
        deque.pop_front();//删除最早元素

        cout<<pNode->value<<endl;

        if(root->left)
            deque.push_back(pNode->left);

        if(root->right)
```

```
            deque.push_back(pNode->right);
    }
}

int main()
{
    BinaryTreeNode* p1=Create(8);
    BinaryTreeNode* p2=Create(6);
    BinaryTreeNode* p3=Create(10);
    BinaryTreeNode* p4=Create(5);
    BinaryTreeNode* p5=Create(7);
    BinaryTreeNode* p6=Create(9);
    BinaryTreeNode* p7=Create(11);

    Connect(p1, p2, p3);
    Connect(p2, p4,p5);
    Connect(p3,p6,p7);
    PrintTree(p1);

  PrintFromTopToBottom(p1);


}
```

## 32_2.分行从上往下打印二叉树

**从上往下打印出二叉树的每个结点，同一层的结点按照从左到右的顺序打印,每一层占一行**

```
/*
// 面试题32（二）：分行从上到下打印二叉树
// 题目：从上到下按层打印二叉树，同一层的结点按从左到右的顺序打印，每一层
// 打印到一行。
*/

#include <iostream>
using namespace std;

#include <deque>

#include "BinaryTree.h"

void PrintFromTopToBottom(BinaryTreeNode* root)
{
    if(root==NULL)
        return;

    //建立一个队列，存储打印节点的左右节点，
    //当打印此节点后，陆续打印左右节点
    std::deque<BinaryTreeNode*> deque;
    deque.push_back(root);

    //定义当前层中还没有打印的节点
    //下一层节点的数目
    int nextlevel=0;
    int tobeprinted=1;
```

```
    while(deque.size())
    {
        BinaryTreeNode* pNode=deque.front();//返回队列中最早元素，并未删除
         cout<<pNode->value<<endl;


        if(root->left)
        {
          deque.push_back(pNode->left);
          nextlevel++;
        }


        if(root->right)
        {
            deque.push_back(pNode->right);
            nextlevel++;
        }

         deque.pop_front();//删除最早元素
         --tobeprinted;

         if(tobeprinted==0)
         {
             cout<<endl;
             tobeprinted=nextlevel;
             nextlevel=0;
         }

    }
}

int main()
{
    BinaryTreeNode* p1=Create(8);
    BinaryTreeNode* p2=Create(6);
    BinaryTreeNode* p3=Create(10);
    BinaryTreeNode* p4=Create(5);
    BinaryTreeNode* p5=Create(7);
    BinaryTreeNode* p6=Create(9);
    BinaryTreeNode* p7=Create(11);

    Connect(p1, p2, p3);
    Connect(p2, p4,p5);
    Connect(p3,p6,p7);
    PrintTree(p1);

  PrintFromTopToBottom(p1);


}
```

## 32_3.之字形打印二叉树

**请实现一个函数按照之字形顺序打印二叉树**

```cpp
/*
// 面试题32（三）：之字形打印二叉树
// 题目：请实现一个函数按照之字形顺序打印二叉树，即第一行按照从左到右的顺
// 序打印，第二层按照从右到左的顺序打印，第三行再按照从左到右的顺序打印，
// 其他行以此类推。
*/

#include <iostream>
using namespace std;

#include <stack>

#include "BinaryTree.h"

void Print(BinaryTreeNode* pRoot)
{
    if(pRoot == NULL)
        return;

    //建立两个栈，辅助存储打印当前行和下一行的数据
    std::stack<BinaryTreeNode*> levels[2];

    int current = 0;
    int next = 1;

    levels[current].push(pRoot);
    while(!levels[0].empty() || !levels[1].empty())
    {
        BinaryTreeNode* pNode = levels[current].top();
        levels[current].pop();

        printf("%d ", pNode->value);

        //1,3,5.......层存储左，右边
        if(current == 0)
        {
            if(pNode->left != nullptr)
                levels[next].push(pNode->left);
            if(pNode->right != nullptr)
                levels[next].push(pNode->right);
        }

        //2,4,6......层存储右，左
        else
        {
            if(pNode->right != nullptr)
                levels[next].push(pNode->right);
            if(pNode->left != nullptr)
                levels[next].push(pNode->left);
        }

        if(levels[current].empty())
        {
            printf("\n");

            //交换两个栈继续打印下一层
            current = 1 - current;
            next = 1 - next;
```

```
        }
    }
}


int main()
{
    BinaryTreeNode* p1=Create(8);
    BinaryTreeNode* p2=Create(6);
    BinaryTreeNode* p3=Create(10);
    BinaryTreeNode* p4=Create(5);
    BinaryTreeNode* p5=Create(7);
    BinaryTreeNode* p6=Create(9);
    BinaryTreeNode* p7=Create(11);

    Connect(p1, p2, p3);
    Connect(p2, p4,p5);
    Connect(p3,p6,p7);
    PrintTree(p1);

  Print(p1);


}
```

## 33.二叉搜索树的后序遍历序列

**输入一个整数数组，判断该数组是不是某二叉搜索树的后序遍历的结果**

```
/*
// 面试题33：二叉搜索树的后序遍历序列
// 题目：输入一个整数数组，判断该数组是不是某二叉搜索树的后序遍历的结果。
// 如果是则返回true，否则返回false。假设输入的数组的任意两个数字都互不相同。
*/

#include <iostream>
using namespace std;

#include "BinaryTree.h"
#include <vector>


//反向考虑，判断数组是不是后序排列的二叉树
bool HaveHouxuOfTree(int data[],int leng)
{
    if(data==NULL || leng<=0)
        return false;

    //后序遍历中根节点的值是最后一个
    int root=data[leng-1];

    //左子树节点的值小于根节点
    int i=0;
    for(;i<leng-1;++i)
    {
        if(data[i]>root)
```

```
            break;
    }

    //右子树的值大于根节点
    int j=i;
    for(;j<leng-1;++j)
    {
        if(data[j]<root)
            return false;
    }

    //判断左子树是不是二叉树
    bool left=true;
    if(i>0)
        left=HaveHouxuOfTree(data,i);

    //判断右子树是不是二叉树
    bool right=true;
    if(j<leng-1)
      right=HaveHouxuOfTree(data+i,leng-i-1);

    return (left && right);

}


int main()
{
    BinaryTreeNode* p1=Create(8);
    BinaryTreeNode* p2=Create(6);
    BinaryTreeNode* p3=Create(10);
    BinaryTreeNode* p4=Create(5);
    BinaryTreeNode* p5=Create(7);
    BinaryTreeNode* p6=Create(9);
    BinaryTreeNode* p7=Create(11);

    Connect(p1, p2, p3);
    Connect(p2, p4, p5);
    Connect(p3, p6, p7);
    PrintTree(p1);

   int data[20]={5,7,6,9,11,10,8};
   int leng=sizeof(data)/sizeof(int);

   bool result=false;
   result=HaveHouxuOfTree( data, leng);
   cout<<result<<endl;


}
```

34.二叉树中和为某一值的路径

输入一棵二叉树和一个整数，打印出二叉树中结点值的和为输入整数的所有路径

```
/*
```

```
// 面试题34：二叉树中和为某一值的路径
// 题目：输入一棵二叉树和一个整数，打印出二叉树中结点值的和为输入整数的所
// 有路径。从树的根结点开始往下一直到叶结点所经过的结点形成一条路径。
*/

#include <iostream>
using namespace std;

#include "BinaryTree.h"
#include <vector>

void FindPath(BinaryTreeNode *root,int sum, std::vector<int> &path,int
currentsum);

void FindPath(BinaryTreeNode *root,int sum)
{
    if(root==NULL)
        return;

    std::vector<int> path;//利用vector实现栈，保证一次打印可出现多值
    int currentsum=0;
    FindPath(root,sum,path,currentsum);
}

void FindPath(BinaryTreeNode *root,int sum, std::vector<int> &path,int
currentsum)
{
   currentsum+=root->value;
   path.push_back(root->value);

   //如果到达叶子节点，并且路径上的和满足sum。则打印
   bool isLeaf=root->left==NULL && root->right==NULL;
   if(currentsum == sum && isLeaf)
   {
       std::vector<int>::iterator iter=path.begin();
       for(;iter!=path.end();++iter)
            cout<<*iter<<endl;
   }


   //非叶子节点
   if(root->left !=NULL)
    FindPath(root->left,sum,path,currentsum);

   if(root->right !=NULL)
    FindPath(root->right,sum,path,currentsum);

   //在返回父节点之前，在路径上删除该节点
   path.pop_back();

}


int main()
{
    BinaryTreeNode* p1=Create(10);
    BinaryTreeNode* p2=Create(5);
    BinaryTreeNode* p3=Create(12);
```

```
    BinaryTreeNode* p4=Create(4);
    BinaryTreeNode* p5=Create(7);

    Connect(p1, p2, p3);
    Connect(p2, p4, p5);
    PrintTree(p1);

   FindPath(p1,22);

}
```

## 54.二叉搜索树的第k个结点

**给定一棵二叉搜索树，请找出其中的第k大的结点**

```
/*
// 面试题54：二叉搜索树的第k个结点
// 题目：给定一棵二叉搜索树，请找出其中的第k大的结点。
*/


#include <iostream>
using namespace std;

#include "BinaryTree.h"

BinaryTreeNode* ZhongXu(BinaryTreeNode* root, unsigned int& k);

BinaryTreeNode* Knode(BinaryTreeNode* root,unsigned int k)
{
    if(root==NULL || k==0)
        return NULL;


    return ZhongXu(root,k);
}

//中序遍历 ,递归实现，左根右
BinaryTreeNode* ZhongXu(BinaryTreeNode* root, unsigned int& k)
{
    if(root==NULL)
        return NULL;

    BinaryTreeNode* target=NULL;

        if(root->left!=NULL)
           target=ZhongXu(root->left,k);

    if(target==NULL)
    {
        if(k==1)
            target=root;

        k--;
    }
```

```
    if(target==NULL && root->right!=NULL)
          target=ZhongXu(root->right,k);

    return target;


}

int main()
{
    BinaryTreeNode* p1=Create(1);
    BinaryTreeNode* p2=Create(2);
    BinaryTreeNode* p3=Create(3);
    BinaryTreeNode* p4=Create(4);
    BinaryTreeNode* p5=Create(5);
    BinaryTreeNode* p6=Create(6);
    BinaryTreeNode* p7=Create(7);

    Connect(p1, p2, p3);
    Connect(p3, NULL, p6);
    Connect(p5,p7,NULL);
    PrintTree(p1);


    unsigned int i=2;
    BinaryTreeNode*  target1= Knode(p1,i);

    cout<<"target "<<target1->value<<endl;


}
```

## 55_1.二叉树的深度

**输入一棵二叉树的根结点，求该树的深度**

```
/*
// 面试题55（一）：二叉树的深度
// 题目：输入一棵二叉树的根结点，求该树的深度。从根结点到叶结点依次经过的
// 结点（含根、叶结点）形成树的一条路径，最长路径的长度为树的深度。
*/

#include <iostream>
using namespace std;

#include "BinaryTree.h"
#include <vector>

//左右子树的最大值加1（1是根的那个）
//方法1
int TreeDepth(BinaryTreeNode *root)
{
    if(root==NULL)
        return 0;

    int nleft=TreeDepth(root->left);
```

```
    int nright=TreeDepth(root->right);

    return (nleft>nright) ? (nleft+1) :(nright+1);
}

//方法2
int FindPath(BinaryTreeNode *root, std::vector<int> &path,int &size,int
&maxsize)
{

   path.push_back(root->value);

   //如果到达叶子节点，并且路径上的和满足sum。则打印
   bool isLeaf=root->left==NULL && root->right==NULL;
   if(isLeaf)
   {
     cout<<"path.size= "<<path.size()<<endl;
     size=path.size();
     if(maxsize<size)
     {
         maxsize=size;
     }
   }

   //非叶子节点
   if(root->left !=NULL)
    FindPath(root->left,path,size,maxsize);

   if(root->right !=NULL)
    FindPath(root->right,path,size,maxsize);

   //在返回父节点之前，在路径上删除该节点
   path.pop_back();

   return maxsize;

}

int main()
{
    BinaryTreeNode* p1=Create(1);
    BinaryTreeNode* p2=Create(2);
    BinaryTreeNode* p3=Create(3);
    BinaryTreeNode* p4=Create(4);
    BinaryTreeNode* p5=Create(5);
    BinaryTreeNode* p6=Create(6);
    BinaryTreeNode* p7=Create(7);

    Connect(p1, p2, p3);
    Connect(p2, p4, p5);
    Connect(p3, NULL, p6);
     Connect(p5,p7,NULL);
    PrintTree(p1);

    int result;
    result=TreeDepth(p1);
    cout<<result<<endl;
```

```
    std::vector<int> path;
    int size=0;
    int maxsize=0;
    size=FindPath(p1, path,size,maxsize);
    cout<<size<<endl;



}
```

## 55_2.平衡二叉树

**输入一棵二叉树的根结点，判断该树是不是平衡二叉树**

```
/*
// 面试题55（二）：平衡二叉树
// 题目：输入一棵二叉树的根结点，判断该树是不是平衡二叉树。如果某二叉树中
// 任意结点的左右子树的深度相差不超过1，那么它就是一棵平衡二叉树。
*/

#include <iostream>
using namespace std;

#include "BinaryTree.h"
#include <vector>

//左右子树的最大值加1（1是根的那个）
//方法1
int TreeDepth(BinaryTreeNode *root)
{
    if(root==NULL)
        return 0;

    int nleft=TreeDepth(root->left);
    int nright=TreeDepth(root->right);

    return (nleft>nright) ? (nleft+1) :(nright+1);
}

//遍历树的每个节点时候，得到它的左右节点深度
//如果每个节点的左，右子树的深度相差都不超过1,即可
bool IsBalanced(BinaryTreeNode *root)
{
    if(root==NULL)
        return true;

    int left=TreeDepth(root->left);
    int right=TreeDepth(root->right);
    int diff=left-right;

    if(diff>1 || diff<-1)
        return false;

    return IsBalanced(root->left) && IsBalanced(root->right);
}
```

```
//方法2
bool IsBalanced(const BinaryTreeNode* pRoot, int* pDepth);

bool IsBalanced2(const BinaryTreeNode* pRoot)
{
    int depth = 0;
    return IsBalanced(pRoot, &depth);
}

bool IsBalanced(const BinaryTreeNode* pRoot, int* pDepth)
{
    if(pRoot == nullptr)
    {
        *pDepth = 0;
        return true;
    }

    int left, right;
    if(IsBalanced(pRoot->left, &left)
        && IsBalanced(pRoot->right, &right))
    {
        int diff = left - right;
        if(diff <= 1 && diff >= -1)
        {
            *pDepth = 1 + (left > right ? left : right);
            return true;
        }
    }

    return false;
}


int main()
{
    BinaryTreeNode* p1=Create(1);
    BinaryTreeNode* p2=Create(2);
    BinaryTreeNode* p3=Create(3);
    BinaryTreeNode* p4=Create(4);
    BinaryTreeNode* p5=Create(5);
    BinaryTreeNode* p6=Create(6);
    BinaryTreeNode* p7=Create(7);

    Connect(p1, p2, p3);
    Connect(p2, p4, p5);
    Connect(p3, NULL, p6);
     Connect(p5,p7,NULL);
    PrintTree(p1);

    bool result=false;
  result=IsBalanced(p1);
  cout<<result<<endl;

   bool result2=false;
  result2=IsBalanced(p1);
  cout<<result2<<endl;
```

```
}
```

## 68.树中两个结点的最低公共祖先

**输入两个树结点，求它们的最低公共祖先**

```
/*
// 面试题68：树中两个结点的最低公共祖先
// 题目：输入两个树结点，求它们的最低公共祖先。
*/

#include <iostream>
using namespace std;

#include "Tree.h"
#include <list>

//得到从根节点root开始到达节点node的路径，保存在path中。
bool GetNodePath(TreeNode* root,TreeNode* node, list<TreeNode*>& path)
{
    if(root==node)
        return true;

    path.push_back(root);

    bool found=false;

    vector<TreeNode*>::iterator i=root->children.begin();
    while(!found && i<root->children.end())
    {
        found=GetNodePath(*i, node,path);
        ++i;
    }

    if(!found)
        path.pop_back();

    return found;
}

//得到两条路径的最后一个公共节点
TreeNode* GetLastCommonNode(const list<TreeNode*>& path1, const list<TreeNode*>&
path2)
{
    list<TreeNode*>::const_iterator iterator1=path1.begin();
    list<TreeNode*>::const_iterator iterator2=path2.begin();

    TreeNode* pLast=NULL;

    while(iterator1 !=path1.end() && iterator2 !=path2.end())
    {
        if(*iterator1==*iterator2)
            pLast=*iterator1;

        iterator1++;
        iterator2++;
    }
```

```
    return pLast;
}


TreeNode* GetLastCommonParent(TreeNode* root, TreeNode* node1, TreeNode* node2)
{
    if(root==NULL || node1==NULL || node2==NULL)
        return NULL;

    list<TreeNode*> path1;
    GetNodePath(root,node1,path1);

    list<TreeNode*> path2;
    GetNodePath(root,node2,path2);

    return GetLastCommonNode(path1,path2);
}


// 形状普通的树
//              1
//            /   \
//           2     3
//       /       \
//      4         5
//     / \      / |  \
//    6   7    8  9  10

int main()
{
    TreeNode* pNode1 = CreateTreeNode(1);
    TreeNode* pNode2 = CreateTreeNode(2);
    TreeNode* pNode3 = CreateTreeNode(3);
    TreeNode* pNode4 = CreateTreeNode(4);
    TreeNode* pNode5 = CreateTreeNode(5);
    TreeNode* pNode6 = CreateTreeNode(6);
    TreeNode* pNode7 = CreateTreeNode(7);
    TreeNode* pNode8 = CreateTreeNode(8);
    TreeNode* pNode9 = CreateTreeNode(9);
    TreeNode* pNode10 = CreateTreeNode(10);

    ConnectTreeNodes(pNode1, pNode2);
    ConnectTreeNodes(pNode1, pNode3);

    ConnectTreeNodes(pNode2, pNode4);
    ConnectTreeNodes(pNode2, pNode5);

    ConnectTreeNodes(pNode4, pNode6);
    ConnectTreeNodes(pNode4, pNode7);

    ConnectTreeNodes(pNode5, pNode8);
    ConnectTreeNodes(pNode5, pNode9);
    ConnectTreeNodes(pNode5, pNode10);

    PrintTree(pNode1);

TreeNode* pResult = GetLastCommonParent(pNode1, pNode6, pNode8);
```

```
cout<<"pResult "<<pResult->value<<endl;

return 0;
}
```

## 栈和队列

### 9.两个队列实现一个栈

```
/*
两个栈实现一个队列
*/

#include <iostream>
using namespace std;

#include <stack>
#include <queue>
#include <exception>

//声明一个队列类
template <typename T>
 class Queue
{
    public:
        Queue(void);
        ~Queue(void);

        void appendTail(const T& node);
        T deleteHead();

    private:
        stack<T> stack1;
        stack<T> stack2;
};


template <typename T> Queue<T>::Queue(void)
{
}

template <typename T> Queue<T>::~Queue(void)
{
}

template<typename T>
 void Queue<T>::appendTail(const T& value)
{
    stack1.push(value);
}

template<typename T>
T Queue<T>::deleteHead()
{
    if(stack2.size()<=0)
    {
        while(stack1.size()>0)
```

```
        {
            T& data=stack1.top();
            stack1.pop();
            stack2.push(data);
        }
    }

    if(stack2.size()==0)
        return NULL;

    T head=stack2.top();
    stack2.pop();

    return head;
}

int main()
{
    Queue<char> queue;
    queue.appendTail('a');
    queue.appendTail('b');
    queue.appendTail('c');

    char head = queue.deleteHead();
    cout<<head<<endl;

    head = queue.deleteHead();

    queue.appendTail('d');
    head = queue.deleteHead();


    queue.appendTail('e');
    head = queue.deleteHead();
    head = queue.deleteHead();


    return 0;

}
```

## 30.一个栈中最小元素

```
/*
找到一个栈中的最小元素
*/

#include <iostream>
using namespace std;

#include <stack>
#include <assert.h>


//声明一个栈
template <typename T>
 class StackWithMin
```

```
{
    public:
        StackWithMin(){}
        virtual ~StackWithMin(){}

        T& top();
        const T& top() const;

        void push(const T& value);
        void pop();

        const T& min() const;

        bool empty() const;
        size_t size() const;

    private:
        stack<T> m_data;//数据栈，存放栈的所有元素
        stack<T> m_min;//辅助栈，存放栈的最小元素
};


template <typename T>
T& StackWithMin<T>::top()
{
    return m_data.top();
}

template <typename T>
const T& StackWithMin<T>::top() const
{
    return m_data.top();
}

template <typename T>
bool StackWithMin<T>::empty() const
{
    return m_data.empty();
}

template <typename T>
size_t StackWithMin<T>::size() const
{
    return m_data.size();
}

template <typename T>
void StackWithMin<T>::push(const T& value)
{
    m_data.push(value);

    if(m_min.size()==0 || value<m_min.top())
        m_min.push(value);

    else
    {
        m_min.push(m_min.top());//为了保存长度一致，把自身最小放在上面
    }
```

```
}

template <typename T>
void StackWithMin<T>::pop()
{
    assert(m_data.size()>0 && m_min.size()>0);

    m_data.pop();
    m_min.pop();
}

template <typename T>
 const T& StackWithMin<T>::min() const
{
    assert(m_data.size()>0 && m_min.size()>0);

    return m_min.top();
}

int main()
{

    StackWithMin<int> stack;

    stack.push(3);
    stack.push(4);
    stack.push(3);
    stack.push(2);

    int min=stack.min();
    cout<<min<<endl;


    return 0;

}
```

## 31.判断一个数组序列是不是栈的弹出序列

```
/*
判断一个数组序列是不是栈的弹出序列
*/

#include <iostream>
using namespace std;

#include <stack>
#include <assert.h>


bool IsPopOrder(const int* pPush, const int* pPop,int len)
{
    bool is=false;

    if(pPush!=NULL && pPop!=NULL && len>0)
    {
```

```
        const int* pNextPush=pPush;
        const int* pNextPop=pPop;

        std::stack<int> stackData;

        while(pNextPop-pPop <len)
        {
            //如果下一个弹出的数字不在栈顶，则把压栈序列中还没有入栈的数字压入辅助栈
            //直到把下一个需要弹出的数字压入栈顶为止
            while(stackData.empty() || stackData.top()!=*pNextPop)
            {
                if(pNextPush-pPush ==len)
                    break;

                stackData.push(*pNextPush);

                pNextPush++;
            }
        //如果所有数字都压入栈后仍然没有找到下一个弹出的数字，那么该序列不可能是一个弹出序列
            if(stackData.top()!=*pNextPop)
                break;

            stackData.pop();
            pNextPop++;

        }

        if(stackData.empty() && pNextPop-pPop==len)
            is=true;
    }
        return is;
}

int main()
{
     const int nLength = 5;
    int push[nLength] = {1, 2, 3, 4, 5};
    int pop[nLength] = {4, 5, 3, 2, 1};

    bool result=IsPopOrder(push,pop,nLength);
    cout<<result<<endl;
}
```

## 59.滑动窗口中的最大值

```
/*
// 面试题59（一）：滑动窗口的最大值
// 题目：给定一个数组和滑动窗口的大小，请找出所有滑动窗口里的最大值。例如，
// 如果输入数组{2, 3, 4, 2, 6, 2, 5, 1}及滑动窗口的大小3，那么一共存在6个
// 滑动窗口，它们的最大值分别为{4, 4, 6, 6, 6, 5}，
*/

#include <iostream>
using namespace std;

#include <stack>
#include <assert.h>
```

```cpp
#include <vector>
#include <deque>


vector<int> maxInWindows(const vector<int>& num,unsigned int size)
{
    vector<int> maxInWindows;

    if(num.size()>=size && size>=1)
    {
        deque<int> index;//建立一个双端队列，为了使他能从前后两端删除和插入数字，存放队列的下标志

       //前三个数字构成一个滑动窗口
        for(unsigned int i=0;i<size;i++)
        {
            //数组第一个数字是2,把它存入队列，第二个数字是3,由于它比前一个数字2大，因此2不可能成为
            //滑动窗口最大值，先把2从队列删除，再把3存入
            while(!index.empty() && num[i]>=num[index.back()])//存放可能最大值的索引
                index.pop_back();

            index.push_back(i);
        }
        //以上队列中已经有三个数字，而他的最大值4位于队列的头部

        for(unsigned int i=size;i<num.size();++i)
        {
            maxInWindows.push_back(num[index.front()]);//把可能最大值的下表对应的数字存入最大窗口中

            while(!index.empty() && num[i]>=num[index.back()])//把下一次滑动窗口中不可能的最大值的下标删除
                index.pop_back();

            //当一个数字的下标与当前处理的数字的下标之差大于或者等于滑动窗口大小时，这个数字已经从窗口中滑出
            //可以从队列中删除了
            if(!index.empty() && index.front()<=(int)(i-size))
                index.pop_front();//对头删除

            index.push_back(i);
        }

        maxInWindows.push_back(num[index.front()]);
    }

    return maxInWindows;
}

int main()
{
     int num[] = { 2, 3, 4, 2, 6, 2, 5, 1 };
    vector<int> vecNumbers(num, num + sizeof(num) / sizeof(int));

    //int expected[] = { 4, 4, 6, 6, 6, 5 };
```

```
   // vector<int> vecExpected(expected, expected + sizeof(expected) /
sizeof(int));

    unsigned int size = 3;

    vector<int> vecExpected=maxInWindows(vecNumbers,size);
    for(auto it=vecExpected.begin();it!=vecExpected.end();++it)
        cout<<*it<<endl;
}
```

# 位运算

## 15.二进制中1的个数

**请实现一个函数，输入一个整数，输出该数二进制表示中1的个数**

```
/*
// 面试题15：二进制中1的个数
// 题目：请实现一个函数，输入一个整数，输出该数二进制表示中1的个数。例如
// 把9表示成二进制是1001，有2位是1。因此如果输入9，该函数输出2。
*/

#include <iostream>
using namespace std;

//位运算
//n和1做与运算，判断n的最低位是不是1,接着把1左移动一位得到2，再和n做与运算，就能判断n的次低位
是不是1.。。。

int NumberOf1(int n)
{
    int count=0;
    unsigned int flag=1;

    while(flag)
    {
        if(n&flag)
            count++;

        flag=flag<<1;
    }

    return count;
}

//把一个整数减去1,再和原整数做与运算，会把该整数最右边的1变为0
//那么一个整数的二进制表示中有多少个1,就可以进行多少次这样的操作
int NumberOf1_1(int n)
{
    int count=0;

    while(n)
    {
        ++count;
        n=(n-1)&n;
    }
```

```
    return count;
}

int main()
{
    // 典型输入，单调升序的数组的一个旋转
    int n=10;

    int result= NumberOf1(n);

    cout<<result<<endl;

    return 0;
}
```

## 56_1.数组中只出现一次的两个数字

**一个整型数组里除了两个数字之外，其他的数字都出现了两次**

```
/*
// 面试题56（一）：数组中只出现一次的两个数字
// 题目：一个整型数组里除了两个数字之外，其他的数字都出现了两次。请写程序
// 找出这两个只出现一次的数字。要求时间复杂度是O(n)，空间复杂度是O(1)。
*/

#include <iostream>
using namespace std;

//位运算
//n和1做与运算，判断n的最低位是不是1,接着把1左移动一位得到2，再和n做与运算，就能判断n的次低位是不是1.。。。

//把一个整数减去1,再和原整数做与运算，会把该整数最右边的1变为0
//那么一个整数的二进制表示中有多少个1,就可以进行多少次这样的操作

 unsigned int FindFirstBitIs1(int num);
 bool IsBit1(int num,unsigned int indexBit);
//{2,4,3,6,3,2,5,5}
//依次异或数组中的每个数字，最终得到的就是两个只出现一次的数字的异或结果
//相同数字异或为0
void FindNumsApperaOnce(int data[],int leng,int* num1,int* num2)
{
    if(data==NULL || leng<2)
        return;

    int resultExclusiveOR=0;
    for(int i=0;i<leng;++i)
        resultExclusiveOR^=data[i];

    cout<<resultExclusiveOR<<endl;//最后得到的是4和6异或的额结果2

    //找到整数resultExclusiveOR的二进制表示中的最右边是1的位
    unsigned int indexOf1=FindFirstBitIs1(resultExclusiveOR);

    cout<<indexOf1<<endl;

    *num1=*num2=0;
```

```
    for(int j=0;j<leng;j++)
    {
        //判断data[j]从右边起的第indexOf1是不是1,进行分组
        //然后依次异或，得到的就是单个数
        if(IsBit1(data[j],indexOf1))
        {
             *num1^=data[j];
             cout<<"data"<<data[j]<<endl;
        }

        else
            *num2^=data[j];
    }

}

 unsigned int FindFirstBitIs1(int num)
 {
     int indexBit=0;
     //n和1做与运算，判断n的最低位是不是1,若是1,则结果为1，否则为0
     //接着把1左移动一位得到2，再和n做与运算，就能判断n的次低位是不是1.。。。
    // while(((num&1)==0) &&(indexBit<8*sizeof(int)))
      while(((num&1)==0) && num>=0)
     {
         cout<<"num"<<num<<endl;

         num=num>>1;
         ++indexBit;

     }

     return indexBit;
 }

 bool IsBit1(int num,unsigned int indexBit)
 {
     num=num>>indexBit;//2右移动1位是1
     return (num&1);//1&1返回1，倒数第2位是1
 }

int main()
{
    int data[] = { 2, 4, 3, 6, 3, 2, 5, 5 };

    int result1,result2;
   FindNumsApperaOnce(data,sizeof(data)/sizeof(int),&result1,&result2);

   cout<<result1<<" "<<result2<<endl;



    return 0;
}
```

## 65.不用加减乘除做加法

**写一个函数，求两个整数之和，要求在函数体内不得使用＋、－、×、÷**

```cpp
/*
// 面试题65：不用加减乘除做加法
// 题目：写一个函数，求两个整数之和，要求在函数体内不得使用＋、－、×、÷
// 四则运算符号。
*/

#include <iostream>
using namespace std;

//位运算
//n和1做与运算，判断n的最低位是不是1,接着把1左移动一位得到2，再和n做与运算，就能判断n的次低位是不是1.。。。
//把一个整数减去1,再和原整数做与运算，会把该整数最右边的1变为0
//那么一个整数的二进制表示中有多少个1,就可以进行多少次这样的操作



//第一步各位相加但不计进位，101+10001=10100
//0+0,1+1结果都是0,0+1,1+0结果都是1,用异或

//第二步：记下进位10,和第一步结果相加 10100+10=10110
//1+1时候产生进位，两个数先做位与运算，然后再向左移动一位，只有两个数都是1的时候，位与得到结果才是1,其余的都是0

//第三步：将前两步结果相加,依然是重复前面两步，直到不产生进位为止

int Add(int num1,int num2)
{
    int sum,carry;

    do
    {
        //第一步
        sum=num1^num2;

        //第二步
        carry=(num1 & num2)<<1;

        num1=sum;
        num2=carry;
    }while(num2!=0);

    return num1;
}



int main()
{

    int result= Add(111, 899);

    cout<<result<<endl;

    return 0;
}
```

# 动态规划

## 10.斐波那契数列

**写一个函数，输入n，求斐波那契（Fibonacci）数列的第n项**

```
/*
// 面试题10：斐波那契数列
// 题目：写一个函数，输入n，求斐波那契（Fibonacci）数列的第n项。
*/

#include <iostream>
using namespace std;

//递归
long long Sol_1(unsigned int n)
{
    if(n<=0)
        return 0;
    if(n==1)
        return 1;

    return Sol_1(n-1)+Sol_1(n-2);
}

//循环
//从下往上算，首先根据f(0)+f(1)=f(2)，再f(1)+f(2)=f(3)....
long long Sol_2(unsigned int n)
{
    int result[2]={0,1};
    if(n<2)
        return result[n];

    long long f_one=1;
    long long f_two=0;
    long long f=0;
    for(unsigned int i=2;i<=n;++i)
    {
        f=f_one+f_two;

        f_two=f_one;
        f_one=f;
    }

    return f;
}

int main()
{
    int result_1=Sol_1(10);
    int result_2=Sol_2(10);

    cout<<result_1<<" "<<result_2<<endl;

    return 0;

}
```

## 12.矩阵中的路径

**用来判断在一个矩阵中是否存在一条包含某字符串所有字符的路径**

```
/*
// 面试题12：矩阵中的路径
// 题目：请设计一个函数，用来判断在一个矩阵中是否存在一条包含某字符串所有
// 字符的路径。路径可以从矩阵中任意一格开始，每一步可以在矩阵中向左、右、
// 上、下移动一格。如果一条路径经过了矩阵的某一格，那么该路径不能再次进入
// 该格子。例如在下面的3×4的矩阵中包含一条字符串“bfce”的路径（路径中的字
// 母用下划线标出）。但矩阵中不包含字符串“abfb”的路径，因为字符串的第一个
// 字符b占据了矩阵中的第一行第二个格子之后，路径不能再次进入这个格子。
// A B T G
// C F C S
// J D E H
*/

#include <iostream>
using namespace std;
#include <cstdio>
#include <cstring>
#include <stack>


bool hasPathCore(const char* matrix,int rows,int cols,int row,int col,const
char* str,int& pathLeng,bool* visited);

//回溯法，从一点出发，查找它周围的四个点是否满足下一个点
bool hasPath(const char* matrix,int rows,int cols,const char* str)
{
    if(matrix==NULL || rows<1 || cols<1 || str==NULL)
        return false;

    //定一个是否遍历到的矩阵，记录这个数已经遍历过了
    bool *visited=new bool[rows*cols];
    memset(visited,0,rows*cols);

    //要找的字符串的路径长度
    int pathLeng=0;
    for(int row=0;row<rows;++row)
    {
        for(int col=0;col<cols;++col)
        {
            if(hasPathCore(matrix,rows,cols,row,col,str,pathLeng,visited))
                return true;
        }
    }

    delete[] visited;

    return false;
}

bool hasPathCore(const char* matrix,int rows,int cols,int row,int col,const
char* str,int& pathLeng,bool* visited)
{
    //找到结尾了
```

```
    if(str[pathLeng]=='\0')
        return true;

    //开始查找matrix中是否有str中的字符
    bool hasPath=true;
    if(row>=0 && row<rows && col>=0 && col<=cols &&
matrix[row*cols+col]==str[pathLeng] && !visited[row*cols+col])
    {
        ++pathLeng;//找到一个相等
        //路过了
        visited[row*cols+col]=true;

        //从它的四周开始查找下一个
        hasPath=hasPathCore(matrix,rows,cols,row,col-1,str,pathLeng,visited)
            || hasPathCore(matrix,rows,cols,row-1,col,str,pathLeng,visited)
            || hasPathCore(matrix,rows,cols,row,col+1,str,pathLeng,visited)
            || hasPathCore(matrix,rows,cols,row+1,col,str,pathLeng,visited);

        //如果都没找到，回到上一个节点
        if(!hasPath)
        {
            --pathLeng;
            visited[row*cols+col]=false;
        }
    }

    return hasPath;
}

int main()
{
//ABTG
//CFCS
//JDEH

//BFCE

     const char* matrix = "ABTGCFCSJDEH";
     const char* str = "BFCE";

     bool result=hasPath(matrix,3,4,str);
     cout<<result<<endl;


    return 0;
}
```

## 13.机器人的运动范围

**地上有一个m行n列的方格。一个机器人从坐标(0, 0)的格子开始移动，它每一次可以向左、右、上、下移动一格，但不能进入行坐标和列坐标的数位之和 大于k的格子**

```
/*
// 面试题13：机器人的运动范围
// 题目：地上有一个m行n列的方格。一个机器人从坐标(0, 0)的格子开始移动，它
// 每一次可以向左、右、上、下移动一格，但不能进入行坐标和列坐标的数位之和
// 大于k的格子。例如，当k为18时，机器人能够进入方格(35, 37)，因为3+5+3+7=18。
```

```
// 但它不能进入方格(35, 38)，因为3+5+3+8=19。请问该机器人能够到达多少个格子？
*/

#include <iostream>
using namespace std;

//同12，回溯法
//机器人从（0,0）开始，判断满足不？然后四周查找
int movingCountCore(int threshold,int rows,int cols,int row,int col,bool*
visited);
bool check(int threshold,int rows,int cols,int row,int col,bool* visited);
int getDigitSum(int data);

int movingCount(int threshold,int rows,int cols)
{
    if(threshold<0 || rows<=0 || cols<=0)
        return 0;

    //定义一个矩阵，判断走过没哦
    bool *visited=new bool[rows*cols];
    for(int i=0;i<rows*cols;++i)
        visited[i]=false;

    //开始计算多少个格子
    int count=movingCountCore(threshold,rows,cols,0,0,visited);

    delete[] visited;

    return count;
}

int movingCountCore(int threshold,int rows,int cols,int row,int col,bool*
visited)
{
    //计算机器人可以走多少格子
    int count=0;

    if(check(threshold,rows,cols,row,col,visited))
    {
       visited[row*cols+col]=true;

       count=1+movingCountCore(threshold,rows,cols,row,col-1,visited)+
                movingCountCore(threshold,rows,cols,row-1,col,visited)+
                movingCountCore(threshold,rows,cols,row,col+1,visited)+
                movingCountCore(threshold,rows,cols,row+1,col,visited);
    }

    return count;
}

bool check(int threshold,int rows,int cols,int row,int col,bool* visited)
{
    //检查是否满足阈值条件
    if(row>=0 && row<=rows && col>=0 && col<cols
    && getDigitSum(row)+getDigitSum(col)<=threshold
    && !visited[row*cols+col])
        return true;
```

```
    return false;
}

//计算位数的和
int getDigitSum(int data)
{
    int sum=0;
    while(data>0)
    {
        sum+=data%10;
        data/=10;
    }

    return sum;
}

int main()
{
    int result=movingCount(5,10,10);

    cout<<result<<endl;

    return 0;

}
```

## 14.剪绳子

**给你一根长度为n绳子，请把绳子剪成m段（m、n都是整数，n>1并且m≥1）。每段的绳子的长度记为k[0]、k[1]、……、k[m]。k[0]*k[1]*…*k[m]可能的最大乘积是多少？**

```
/*
// 面试题14：剪绳子
// 题目：给你一根长度为n绳子，请把绳子剪成m段（m、n都是整数，n>1并且m≥1）。
// 每段的绳子的长度记为k[0]、k[1]、……、k[m]。k[0]*k[1]*…*k[m]可能的最大乘
// 积是多少？例如当绳子的长度是8时，我们把它剪成长度分别为2、3、3的三段，此
// 时得到最大的乘积18。
*/

#include <iostream>
using namespace std;

//动态规划
//递归会造成重复计算，因此从小到大避免重复
//f(n)=f(i)*f(n-i)

int maxProduct(int leng)
{
    if(leng<2)
        return 0;
    if(leng==2)
        return 1;
    if(leng==3)
        return 2;

    //定义一个数组，存放每次的乘
    int* products=new int[leng+1];
```

```
    products[0]=0;
    products[1]=1;
    products[2]=2;
    products[3]=3;

    int max=0;

    //从下到上的递归
    for(int i=4;i<=leng;++i)
    {
        max=0;

        //f(n)=f(i)*f(n-i)的最大值
        for(int j=1;j<i/2;++j)
        {
            int product=products[j]*products[i-j];
            if(max<product)
                max=product;

            products[i]=max;
        }
    }

    max=products[leng];
    delete[] products;

    return max;
}


int main()
{

    int result=maxProduct(12);

    cout<<result<<endl;

    return 0;
}
```

## 29.顺时针打印矩阵

**输入一个矩阵，按照从外向里以顺时针的顺序依次打印出每一个数字。**

```
// 面试题29：顺时针打印矩阵
// 题目：输入一个矩阵，按照从外向里以顺时针的顺序依次打印出每一个数字。



#include <algorithm>
#include <iostream>
using namespace std;
#include <cstdio>

void PrintMatrixInCircle(int** numbers, int columns, int rows, int start);
void printNumber(int number);
```

```
void PrintMatrixClockwisely(int** numbers, int columns, int rows)
{
    if(numbers == nullptr || columns <= 0 || rows <= 0)
        return;

    int start = 0;

   //选择左上角（start,start)的一圈分析
   //对于5×5的矩阵，最后一圈只有一个数字，对应的坐标（2,2） 5>2*2
   //对于6×6的矩阵，最后一圈有4个数字，左上角坐标依然是（2,2） 6>2*2
   //于是下面是循环结束的条件
    while(columns > start * 2 && rows > start * 2)
    {
        PrintMatrixInCircle(numbers, columns, rows, start);

        ++start;
    }
}

void PrintMatrixInCircle(int** numbers, int columns, int rows, int start)
{
    int endX = columns - 1 - start;
    int endY = rows - 1 - start;

    // 从左到右打印一行
    for(int i = start; i <= endX; ++i)
    {
        int number = numbers[start][i];
        printNumber(number);
    }

    // 从上到下打印一列
    //保证终止列大于开始列
    if(start < endY)
    {
        for(int i = start + 1; i <= endY; ++i)
        {
            int number = numbers[i][endX];
            printNumber(number);
        }
    }

    // 从右到左打印一行
    //终止行大于开始行，终止列大于开始列
    if(start < endX && start < endY)
    {
        for(int i = endX - 1; i >= start; --i)
        {
            int number = numbers[endY][i];
            printNumber(number);
        }
    }

    // 从下到上打印一行
    if(start < endX && start < endY - 1)
    {
        for(int i = endY - 1; i >= start + 1; --i)
        {
```

```
            int number = numbers[i][start];
            printNumber(number);
        }
    }
}

void printNumber(int number)
{
    printf("%d\t", number);
}


int main()
{
    int rows=4,columns=4;

      int** numbers = new int*[rows];
    for(int i = 0; i < rows; ++i)
    {
        numbers[i] = new int[columns];

        for(int j = 0; j < columns; ++j)
        {
            numbers[i][j] = i * columns + j + 1;
        }
    }


    PrintMatrixClockwisely( numbers, 4,4);


    return 0;
}
```

## 47.礼物的最大价值

**在一个m×n的棋盘的每一格都放有一个礼物，每个礼物都有一定的价值（价值大于0）**

```
// 面试题47：礼物的最大价值
// 题目：在一个m×n的棋盘的每一格都放有一个礼物，每个礼物都有一定的价值
// （价值大于0）。你可以从棋盘的左上角开始拿格子里的礼物，并每次向右或
// 者向下移动一格直到到达棋盘的右下角。给定一个棋盘及其上面的礼物，请计
// 算你最多能拿到多少价值的礼物？

/*
    //四行四列
    int values[][4] = {
        { 1, 10, 3, 8 },
        { 12, 2, 9, 6 },
        { 5, 7, 4, 11 },
        { 3, 7, 16, 5 }
    };

    1-12-5-7-7-16
*/

#include <algorithm>
```

```cpp
#include <iostream>
using namespace std;

//定义一个函数f(i,j)表示达到坐标（i，j)的格子时能拿到的礼物总和最大值
//两种途径到达f(i,j),第一中f(i-1,j)和f(i,j-1)
//f(i,j)=max(f(i-1,j),f(i,j-1))+gift(i,j)

//由于递归有大量的重复计算。为了缓存中间的计算结果，需要一个辅助的二维数组
//数组中坐标为（i，j)的元素表示到达坐标为（i，j)的格子时能拿到的礼物价值总和的最大值

/*int getMax_value(const int* values, int rows, int cols)
{
    if(values==NULL || rows<=0 || cols<=0)
        return 0;

    int** max_Values=new int*[rows];//定义一个二维数组
    for(int i=0;i<rows;++i)
        max_Values[i]=new int[cols];

    for(int i=0;i<rows;++i)
    {
        for(int j=0;j<cols;++j)
        {
            int left=0;
            int up=0;

            if(i>0)
                up=max_Values[i-1][j];

            if(j>0)
                left=max_Values[i][j-1];

            max_Values=std::max(left,up)+values[i*cols+j];
        }
    }

    int max_Value=max_Values[rows-1][cols-1];

    for(int i=0;i<rows;++i)
        delete[] max_Values[i];
    delete[] max_Values;

    return max_Value;
}*/

int getMaxValue(const int* values, int rows, int cols)
{
    if(values == nullptr || rows <= 0 || cols <= 0)
        return 0;

    int** maxValues = new int*[rows];
    for(int i = 0; i < rows; ++i)
        maxValues[i] = new int[cols];

    for(int i = 0; i < rows; ++i)
    {
        for(int j = 0; j < cols; ++j)
        {
```

```
            int left = 0;
            int up = 0;

            if(i > 0)
                up = maxValues[i - 1][j];

            if(j > 0)
                left = maxValues[i][j - 1];

            maxValues[i][j] = std::max(left, up) + values[i * cols + j];
        }
    }

    int maxValue = maxValues[rows - 1][cols - 1];

    for(int i = 0; i < rows; ++i)
        delete[] maxValues[i];
    delete[] maxValues;

    return maxValue;
}

int main()
{
     int values[][4] = {
        { 1, 10, 3, 8 },
        { 12, 2, 9, 6 },
        { 5, 7, 4, 11 },
        { 3, 7, 16, 5 }
    };

    int result=getMaxValue( (const int*) values, 4, 4);
    cout<<result<<endl;

    return 0;
}
```

## 查找算法

### 1.一个关键值数组，任意输入一个数字k，使用折半查找算法找到K在数组中的下标

```
/*
一个关键值数组，任意输入一个数字k，使用折半查找算法找到K在数组中的下标
*/

#include <iostream>
using namespace std;

int bin_search(int key[],int len,int k)
{
    int low=0,high=len-1,mid;

    while(low<=high)
    {
        mid=(low+high)/2;
```

```
        if(key[mid]==k)
            return mid;

        if(k>key[mid])
            low=mid+1;

        else
        {
            high=mid-1;
        }
    }

    return -1;

}

//递归
int bin_search_recur(int key[],int low,int high,int k)
{
    int mid;
    if(low>high)
        return -1;

    else
    {
        mid=(low+high)/2;

        if(key[mid]==k)
            return mid;
        if(k>key[mid])
            return bin_search_recur(key,mid+1,high,k);

        else
           return bin_search_recur(key,low,mid-1,k);
    }
}

int main()
{
    int data[10]={1,3,5,7,10,12,15,19,21,50};

    int result=bin_search(data,10,15);
    cout<<result<<endl;
    int result2=bin_search_recur(data,0,9,15);
    cout<<result2<<endl;

    return 0;
}
```

## 2.已知一个从小到大排列的有序整型数组，从中找出某个数出现的次数

```
/*
已知一个从小到大排列的有序整型数组，从中找出某个数出现的次数
*/

#include <iostream>
```

```
using namespace std;

//折半查找这个数第一次出现的位置loc_a和最后一次出现的位置loc_b,loc_b-loc_a+1;

//loc_a是key左边的，而loc_b是key右边的，依然折半查找，直到子序列中查找不到key,
//则最后一次查找到key时记录下来的位置就是loc_a或者loc_b的位置

int bin_search(int data[],int key,int len,int loc_flag)
{
    int low=0,high=len-1;
    int mid=0;

    int last=-1;//记录最终的loc_a或者loc_b

    while(low<=high)
    {
        mid=(low+high)/2;

        if(data[mid]<key)
            low=mid+1;

        else if(data[mid]>key)
            high=mid-1;

        else
        {
            //找到key
            last=mid;

            if(loc_flag==0)
                high=mid-1;//查找loc_a,调整high的值继续在左边查找
            else
                low=mid+1;//查找loc_b，在右边查找
        }
    }

    return last;//他是最终的loc_a或者loc_b为位置
}

int getDataCount(int data[],int len,int key)
{
    int loc_a,loc_b;

    loc_a=bin_search(data,key,len,0);
    loc_b=bin_search(data,key,len,1);

    if(loc_a==-1 || loc_b==-1)
        return 0;
    else
        return loc_b-loc_a+1;
}

int  main()
 {
    int array[] = {0,1,3,5,5,5,5,5,5,5,5,5,5,5,5,6,9,12};
    int result=getDataCount(array,19,5);
    cout<<"result "<<result<<endl;
```

```
    return 0;
}
```

# 排序算法

## 1.冒泡排序

```
/*
编写一个程序，将一个整型数组中的数据从大到小排列，要求使用冒泡
*/

#include <iostream>
using namespace std;

void bubbleSort(int data[],int len)
{

   int tmp;
   int i,j;
   int flag=1;
   for(i=0;i<len-1 && flag==1;i++)//排序的汤数
   {
       flag=0;
       for(j=i+1;j<len;j++)
       {
           if( data[i]<data[j])
           {
               tmp=data[j];
               data[j]=data[i];
               data[i]=tmp;
               flag=1;//发生数据交换，标志为1
           }
       }
   }


}

int main()
{
    int data[9]={0,3,1,2,4,6,6,10,5};
    bubbleSort( data,9);
    for(int i=0;i<9;i++)
        cout<<data[i]<<endl;

    return 0;
}
```

## 2.直接插入排序

**{5,6,3,9,2,7,1}，将序列的第一个元素看作有序序列，之后的元素插入第一序列组成新的第一序列，依次类推**

```
/*
```

```
编写一个程序，将一个整型数组中的数据从大到小排列，要求使用直接插入排序
*/

#include <iostream>
using namespace std;

void insertSort(int data[],int len)
{
    int i,j,tmp;

    for(i=1;i<len;i++)
    {
        tmp=data[i];

        j=i-1;
        while(j>=0 && tmp>data[j])//从大到小排序
        {
            data[j+1]=data[j];//循环右移，直到找到data[i]应该放置的位置
            j--;
        }

        data[j+1]=tmp;//将元素tmp插入指定位置
    }



}

int main()
{
    int data[9]={0,3,1,2,4,6,6,10,5};
    insertSort( data,9);
    for(int i=0;i<9;i++)
        cout<<data[i]<<endl;

    return 0;
}
```

## 3.选择排序

**{5,6,3,9,2,7,1},将次看作第一序列，找到最小元素和第一个元素交换;{1,6,3,9,2,7,5}**

**其次把第二个元素到最后一个元素看作第二序列，找到最小的与第二序列的第一个元素交换**

```
/*
编写一个程序，将一个整型数组中的数据从大到小排列，要求使用选择排序
*/

#include <iostream>
using namespace std;

//每次选择序列中的最小元素，让他它未排序的第一个元素交换
void selectSort(int data[],int len)
{

   int tmp;
   int i,j;
```

```
    int min;
    for(i=0;i<len-1;i++)
    {
        min=i;

        for(j=i+1;j<len;j++)
        {
            if(data[min]>=data[j])
            {
                min=j;//记录最小元素的位置
            }
        }

        if(min!=i)
        {
            tmp=data[min];
            data[min]=data[i];
            data[i]=tmp;
        }

    }


}

int main()
{
    int data[9]={0,3,1,2,4,6,6,10,5};
    selectSort( data,9);
    for(int i=0;i<9;i++)
        cout<<data[i]<<endl;

    return 0;
}
```

## 4.快速排序

**快速排序是先挑选一个基准，把比它小的放在左边，比它大的放在右边**

**之后在他的左边继续挑选一个基准，执行上述过程**

**在它右边也是一样**

```
/*
编写一个程序，将一个整型数组中的数据从大到小排列，要求使用快速排序
*/

#include <iostream>
using namespace std;

//快速排序是先挑选一个基准，把比它小的放在左边，比它大的放在右边
//之后在他的左边继续挑选一个基准，执行上述过程
//在它右边也是一样
void swap(int *a,int *b)
{                               /*交换序列中元素的位置*/
  int tmp;
  tmp = *a;
  *a = *b;
```

```
  *b = tmp;
}

//s,t表示每一序列的首尾元素
void quickSortArray(int array[], int s,int t)
{
    int low,high;
    if(s<t)
    {
        low = s;
        high = t+1;
        while(1)
        {
            do low++;
            while(array[low]>=array[s] && low!=t);        /*array[s]为基准元素，重
复执行low++操作*/

            do high--;
            while(array[high]<=array[s] && high!=s);      /*array[s]为基准元素，重
复执行high--操作*/

            if(low<high)
                swap(&array[low],&array[high]);           /*交换array[low]和
array[high]的位置*/
            else
                break;
        }

        swap(&array[s],&array[high]);                    /*将基准元素与array[high]进
行交换*/
        quickSortArray (array,s,high-1);                 /*将基准元素前面的子序列快速
排序*/
        quickSortArray (array,high+1,t);                 /*将基准元素后面的子序列快速
排序*/
    }
}

void quickSort(int array[], int arraySize) {
    quickSortArray(array, 0, arraySize-1);
}

main()
{
    int i,array[10];
    printf("Input ten integer\n");
    for(i=0;i<10;i++)
    {
        scanf("%d",&array[i]);                  /*循环输入10个整数*/
    }

    quickSort(array,10);                       /*执行快速选择排序*/
    printf("\nThe result of quick sort is\n");
    for(i=0;i<10;i++) {
        printf("%d ",array[i]);                     /*输出排序后的结果*/
    }
    getchar();
    getchar();
}
```

## 5.希尔排序

**希尔排序分段进行排序，使得元素交换是跳跃的，减少了元素交换次数**

```
/*
编写一个程序，将一个整型数组中的数据从大到小排列，要求使用希尔排序
*/

#include <iostream>
using namespace std;

//希尔排序分段进行排序，使得元素交换是跳跃的，减少了元素交换次数
void shellSort(int data[],int len)
{
   int i,j,flag,tmp,gap=len;

   while(gap>1)//间隔为1比较结束
   {
       gap=gap/2;//按照经验值，每次间隔缩小一半
       do
       {
          flag=0;//子序列可以使用冒泡
          for(i=0;i<len-gap;i++)
          {
              j=i+gap;
              if(data[i]>data[j])
              {
                  tmp=data[i];
                  data[i]=data[j];
                  data[j]=tmp;
                  flag=1;
              }
          }
       }while(flag!=0);//改进了冒泡排序
   }


}

int main()
{
    int data[9]={0,3,1,2,4,6,6,10,5};
    shellSort( data,9);
    for(int i=0;i<9;i++)
        cout<<data[i]<<endl;

    return 0;
}
```

####