第一章 基础数学思维与技巧

[算法中的数学思维 - 芯笑 (lamper.top)]

最大公约数

求最大公约数 --- 欧几里得辗转相除法

```
public static int gcd(int a,int b){
    while(b>0){
        int temp = a%b;
        a=b;
        b=temp;
    }
    return a;
}
```

```
public static int gcd(int a,int b){
return  b==0?a:gcd(b,a%b);3
}
```

最小公倍数

求最小公倍数

```
public static int lcm(int a,int b){
return a * b / gcd(a,b);3
}
```

进制转换

```
String  s  =  Integer.toString(a,m);//10进制a数转m进制数,结果为字符串

int  a  =  Integer.parseInt(s,m);//把字符串s当做m进制数,将结果转为10进制数

BigInteger  biginteger  =  new  BigInteger(s,m);//把m进制的字符串s转换成10进制数后封装成大数对象
```

位运算

与 & (全1为1,有0为0)

判断奇偶数

奇数-二进制最后一位一定为1
偶数-二进制最后一位一定为0

```
public static boolean check(int m){
    return (m&1)==1;
}
```

判断m是否为2的x次方

若m为2的x次方:m的二进制只有最高位为1,其余全为0,(m-1)的二进制除最高位都为1.

```
public static boolean check(int m){
    return  m&(m-1)==0;
}
```

异或 ^ (相同为0,不同为1)

找到数组中只出现了一次的数

按位异或：相同为0，不同为1

```
x^x=0;
0^x=x;
a^b^c=a^c^b;

public static int num(int[] s){
    int ans = 0;
    for(int
        i=0;i<s.length;i++){ans
        = ans ^ s[i];
    }
    return ans;
```

移位 >> 和<<

```
8>>1 == 4
4>>1 == 2

2<<1 == 4
4<<1 == 8

n >> m == n / (2 ^ m)
n << m == n * (2 ^ m)
```

素数

判断素数

素数:只有1和它本身是因数 。

首先，0和1不是素数，然后 i 从 2 开始判断 i 是不是 n 的因数，如果是因数，则直接返回 n 不是素数，否则，判断 i+1是不是 n 的因数，直到 i=√n 的时候，如果 i 仍然不是 n 的因数，那么 n 就是素数。

注：如果一个数 a 能够整除 i ，那么 i 和 a/i 一定满足：假设 i<=a/i , 那么 i<=√n , && a/i>= √n 。

```
public static boolean isprime(int n){
    if(n==0 || n==1)
        return false;
    for(int i=2;i<=n/i;i++){
        if(n%i==0)
            return false;
    }
    return true;
}
```

求1~n中的所有素数 ---埃氏筛法

思路：如果一个数不是素数，那么这个数一定是 n 个素数的乘积（0和1除外），同理，素数的 k 倍数一定是合数（k>=2）。

```
public static void isprime(int n){
    boolean[] isprime = new boolean[n+1];//false表示素数，true表示合数
    for(int i=2;i*i<=n;i++)
        if(!isprime[i]) //i是质数
            for(int  j=2;j*i<=n;j++)//将i的倍数全部标记为合数
                isprime[i*j] = true;
    for(int i=2;i<=n;i++)
        if(!isprime[i])
            System.out.println(i);
}
```

求1~n中的所有素数 ---欧拉筛法

思路：每个合数，只被他最小的质因子筛一次。

注：与埃氏筛法不同，埃氏筛法是将素数的倍数，标记为合数；欧拉筛法是将目前已经找到的每一个素数的 i 倍标记为合数，无论 i 是否是素数，同时，如果 i 本身就是素数的倍数，那么就去执行下一个 i 。

```
public static void isprime(int n){
    boolean[] isprime = new boolean[n+1];
    int[] prime = new int[n];//存储素数
    int count = 0;//统计目前素数个数
    for(int i=2;i<=n;i++) {
        if(!isprime[i]) //i是质数
            prime[count++] = i;//把当前素数存储到数组中count位置
        for(int  j=0;j<count && i*prime[j]<=n;j++){//将i的倍数全部标记为合数
            isprime[i*prime[j]] = true;
        if(i%prime[j]==0)  break;//欧拉筛法精髓11
        }
    }
    for(int i=0;i<count;i++)
```

```
        System.out.println(prime[i]);
}
```

例题：最小质因子之和

题目链接：最小质因子之和(Easy Version) - 蓝桥云课 (lanqiao.cn)

题目描述

定义 $F(i)$ 表示整数 i 的最小质因子。现给定一个正整数 N，请你求出 $\sum_2^n F(i)$。

输入描述

第 1 行为一个整数 T，表示测试数据数量。

接下来的 T 行每行包含一个正整数 N。

$1 \leq T \leq 10^6$，$2 \leq N \leq 3 \times 10^6$。

输出描述

输出共 T 行，每行包含一个整数，表示答案。

思路：因为题目输入为T组数据，如果单独计算每组数据，则会有部分区间的数据被重复计算，所以先通过埃氏筛法，求出每一个数的最小质因子，将结果存放在 ans 数组中，然后将 ans 数组表示为前缀和数组，此时 ans 数组中的结果就为2~n的质因子之和，此时，题目若输入 15 ，则直接输出 ans[15] 即可。

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import  java.io.PrintWriter;

public class 最小质因子之和 {
    static boolean[] isprime  = new boolean[3000001];//是否是素数
    static long[] ans = new long[3000001];//存储最小质因子 i的最小质因子为ans[i]，例：ans[4] = 2
    static BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    static PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    public static void main(String[] args) throws IOException{
        get(3000000);//题目数据范围，N最大值为3*10^6，将2~3*10^6中每一个数的最小质因子全部求出
        for(int i=2;i<=3000000;i++) {
            ans[i]  =  ans[i]  +  ans[i-1];//求前缀和，此时ans[i]中存放的数就是2~i中每一个数的最小质因子的和
        }
        int n = Integer.parseInt(in.readLine());
        while(n-->0) {
            out.println(ans[Integer.parseInt(in.readLine())]);
        }
        out.flush();
    }

    //找出每个数的质因子
    static void get(int n)
{ 27    for(int i=2;i<=n;i++) {
            if(isprime[i])//i不是质数直接跳过,不考虑,i不能作为筛除条件
                continue;
            ans[i]  =  i;//i为素数，素数的最小质因子就是其本身
```

```
            for(int   j=2;j<=n/i;j++) {//j为倍数，将素数i的j倍数标记为合数，并将此数的最小质因子标记为i
                if(!isprime[j*i]) {//判断是否已经被标记过
                    isprime[j*i] = true;//将i*j标记为合数
                    ans[j*i] = i;//j*i的最小质因子是i
                }
            }
        }
    }
}
```

回文数

判断回文数

思路：将数字转换为字符串类型后，将此字符串倒转后，判断与原字符串是否相同

```
public static boolean check(int m){
    return Integer.toString(m).equals(new StringBuffer(Integer.toString(m)).reverse().toString());
}
```

判断数组中元素是否相同

思路：若数组中元素全部相同，则数组中的最大值应当==最小值。

```
public static boolean check(int[] n){
    Arrays.sort(n);
return  n[0]==n[n.length-1];4
}
```

思路：利用Set集合自动去重，将数组中所有元素全部添加到集合中后，如果集合中只有一个元素，则表示数组中所有元素全部相同。

```
public static boolean check(int[] n){
    Set<Integer> set = new HashSet<>();
    for(int i=0;i<n.length;i++) {
        set.add(n[i]);
    }
    return set.size()==1;
}
```

日期+星期模拟

[Calendar类 - 芯笑 (lamper.top)]

```
public static Main{
    static int[] date = {0,31,28,31,30,31,30,31,31,30,31,30,31};//存储每月天数
    static int y = 2001,m = 1;d = 1,week = 1;//初始年,月,日,星期(根据题意选择是否需要)
    //week==0,表示周日,week==1,表示周一 ...  week==6,表示周六
    public static void main(String[] args){
```

```java
        int ans = 0;//计数
        while(y!=9999 || m!=12|| d!=31){//设置日期判断范围
            //判断闰年(满足其一即可):
            //1.可以整除400
            //2.可以整除4但不能整除100
            if(y%400==0 || (y%4==0&& y%100!=0) date[2] = 29;
            else date[2] = 28;
            if(check()) ans++;//满足条件,计数器++;
            d++;
            week++;
            week%=7;
            if(d>date[m]){
                d = 1;
                m++;
            }
            if(m>12){
                m = 1;
                y++;
            }
        }
        if(check()) ans++;//之前结束日期并未判断,判断结束日期
        System.out.println(ans);
    }
    public static boolean check(){}//根据题目要求完成
}
```

约数

[求约数 - 芯笑 (lamper.top)]

唯一分解定理

[唯一分解定理 - 芯笑 (lamper.top)]

唯一分解定理

唯一分解定理又称为算数基本定理，基本内容是：

每个大于1的自然数，要么本身就是质数，要么可以写为2个或以上的质数的积，而且这些质因子按大小排列之后，写法仅有一种方式。

用另一种方法表示就是：

对于任何一个大于1的正整数,都存在一个标准的分解式：N=p1^a1 * p2^a2*…*pn^an;（其中一系列an为指数，pn为质数）

此定理表明：任何一个大于 1 的正整数都可以表示为素数的积。

有这样几个式子：

设F(n)代表n的正因子的数量，则F(n)=(a1+1)*(a2+1)*(a3+1)*……*(an+1);

设G(n)代表n的正因子的和，则G(n)=(1+p1^2+p1^3+...+p1^a1)*(1+p2^2+p2^3+...p2^a2)*....*

(1+pn^1+pn^2+...+pn^an)= $\prod_{i=1}^{n} \frac{pi^{ei+1}-1}{pi-1}$;

n的质因数个数 ---唯一分解定理

```java
public static int num(long r){
    int ans = 0;
    for(int i=2;i<n/i;i++){
        while(n%i==0){
            ans++;
            n/=i;
        }
    }
    if(n>1)
        ans++;
    return ans;
}
```

n的约数个数 ---唯一分解定理

```java
public static int num(int n){
    int cnt = 1;//乘法初始值为1
    int bak = n;//备份n
    for(int i=2;i*i<bak;i++){
        int sum = 0;
        while(bak%i==0){
            sum++;
            bak = bak / i;
        }
        cnt = cnt * (sum + 1);
    }
    if(bak>1) cnt*=2;
    return cnt;
}
```

求n!的约数个数 ---唯一分解定理

```java
public static long num(int n){
    int[] prime = new int[n+1];//prime[i]表示素数i这个因子出现的次数
    for(int i=2;i<=n;i++){
        int bak = i;
        for(int j=2;j*j<bak;j++){
            int sum = 0;
            while(bak%j==0){
                prime[j]++;
                bak = bak / j;
            }
        }
        if(bak>1) prime[bak]++;
    }
    long ans = 1;
    for(int i=2;i<=n;i++){
        if(prime[i]>1)
            ans = ans * (prime[i]+1);
    }
    return ans;
}
```

例题：数数

题目链接：数数 - 蓝桥云课 (lanqiao.cn)

问题描述

任何一个大于 1 的正整数都能被分解为若干个质数相乘, 比如 $28 = 2 \times 2 \times 7$ 被分解为了三个质数相乘。请问在区间 [2333333, 23333333] 中有多少个正整数 可以被分解为 12 个质数相乘?

思路：将这个区间中的每一个数都根据唯一分解定理进行拆分，统计有多少个数的拆分结果为12

```java
import java.util.Scanner;

public class 数数 {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int ans = 0;
        for(int i=2333333;i<=23333333;i++)
            if(num(i)==12)
                ans++;
        System.out.println(ans);
    }
    static int num(int n)
        {int ans = 0;
        for(int i=2;i<n/i;i++)
            {while(n%i==0) {
                n/=i;
                ans++;
            }
        }
        if(n>1)
            ans++;
        return ans;
    }
}

```

例题：求阶乘

题目链接：求阶乘 - 蓝桥云课 (lanqiao.cn)

问题描述

满足 $N!$ 的末尾恰好有 K 个 0 的最小的 N 是多少?

如果这样的 N 不存在输出 -1 。

输入格式

一个整数 K 。

输出格式

一个整数代表答案。

思路:

1.根据唯一分解定理可知：每一个数都可以写为 n 个素数的乘积；

2.如果一个数的结尾有 0 的存在，那么这个数分解后一定有 2 和 5 （素数中，只有2 * 5才能使结尾产生 0 ）；

3.从 1 ~ N ，将每一个数都分解后，2 的数量一定比 5 多（ 2 每隔两个数就会最少出现一个，5 每隔5个数，最少出现一个），那么，N! 末尾 0 的数量，就是将 1 ~ N 中每个数分解后，5 的数量；

4.如果用一个循环从 5 开始，每次 +5 ，判断这些数可以拆分出几个 5 ，然后去找结尾有 k 个 0 的最小的 N 是多少，这个方法结果正确，但是时间复杂度会比较高，所以借助二分，去找到结尾有 k 个 0 的最小的 N 是多少；

5.用二分去查找，就必须做到：已知 N ，求出 1 ~ N 中可以拆分出多少个 5 ，以 125 为例，因为每五个数才拆分出 5 ，所以，如果 1~125 都只拆一个 5 ，则可以拆分出 125 / 5 共 25 个 5 ，拆分后的结果为 1 ~ 25 ，然后继续拆分 5 ，1 ~ 25 可以拆分出 25 / 5 个 5 ，拆分后结果为 1 ~ 5 ，1 ~ 5 可以拆分出 5 / 5 个 5 ，最后剩余 1 ，1 无法继续拆分出 5 ，所以 125 可以拆分出 25 + 5 + 1 = 31 个 5 ；

6.二分：如果mid拆分出的 5 的数量 >= k，那么可以 right = mid ，反之left = mid + 1，二分结果后，还需要判断它是否确实能拆分出 k 个 5 ，因为存在一个 N! 能恰好末尾有 k 个 0 ；

```java
import java.util.Scanner;

public class 求阶乘 {
    public static long find(long x) {//求x能拆分出多少个5
        long res = 0;
        while(x != 0) {
            res = res + x / 5;
            x/=5;
        }
        return res;
    }
    public static void main(String[] args)
        {     Scanner      sc      =      new
        Scanner(System.in);   long    k    =
        sc.nextLong();
        long l = 0,r = 100000;//防止溢出
        while(l < r) {
            long mid = (l + r) / 2;
            if(k <= find(mid)) {
                r = mid;
            }else {
                l = mid + 1;
            }
        }
```

```
            System.out.println(-1);
        }else {
            System.out.println(r);
        }
    }
}
```

第二章 字符串基础

常用API

```
String m = "abcde";

char ch = m.charAt(String n);//获取字符串m的第(n+1)个字符
int length = m.length();//获取字符串m的长度
boolean flag = m.equals(String n);//判断字符串m和n是否相等,严格区分大小写
boolean flag = m.equalsIgnoreCase(String n);//判断字符串m和n是否相等,不区分大小写
int len = m.index(String s);//返回字符串s在m中第一次出现的位置
int compare = m.compareTo(String anotherString);//按字典序比较两个字符串,若compare>0,m大,若compare<0,m小
String s = m.concat(n);//将字符串n拼接到字符串m的结尾
boolean flag = m.contains(String n);//判断字符串m是否包含字符串n
boolean flag = m.endsWith(String s);//判断字符串m是否以字符串s结尾
String[] s = m.split(" ");//根据正则表达式拆分字符串m
String s = m.trim();//删除字符串m的前导空格和尾部空格
String s = m.subString(int i,int j);//截取字符串m中下标为i至下标为j-1的部分,即[i,j);
...
```

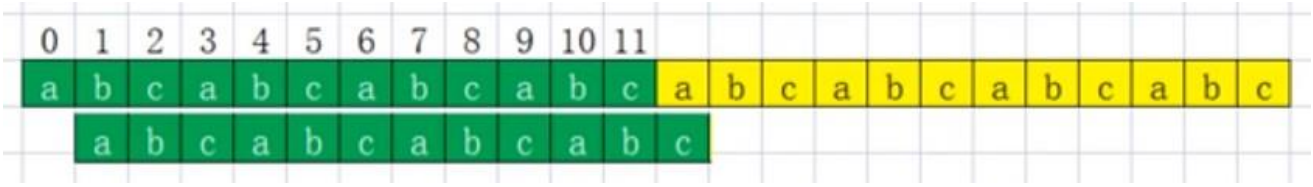
周期串

思路：从 1 开始枚举周期 T 的大小，然后判断每个周期内的对应字符是否相同，如果不同，则直接判断下一个 T 。

```
public static int cycle(String s){
    char[] ch = s.toCharArray();
    int T;
    for(T=1;T<=ch.length;T++){
        if(ch.length%T==0){//周期串的长度一定是周期T的倍数
            boolean flag = true;
            for(int start = T;start<ch.length;start++){
                if(ch[start]!=ch[start%T]){
                    flag = false;
                    break;
                }
            }
            if(flag){
                break;
            }
        }
```

```
    }
    return T;
}
```

思路：pos 表示第二行的字符串向右移动的格数，如果移动后，第二行的字符串与第一行字符串对应位置的字符全部相同，则 pos 就是这个字符串的周期。




```
public static int cycle(String s){
    String m = s+s;
    int pos;
    for(pos=1;pos<=s.length();pos++){
        if(s.length()%pos!=0)
            continue;
        String x = m.substring(pos,pos+s.length());
        if(x.equals(s))
            break;
    }
    return pos;
}
```

思路：如果一个字符串 sub 是字符串 s 的周期，那么将字符串 s 中所有的 sub 全部替换为空字符串之后，字符串的长度如果为 0，就表示字符串 sub 是字符串 s 的周期。

```
public static int cycle(String s){
    for(int i=1;i<=s.length();i++){
        if(s.length()%i==0){
            String sub = s.substring(0,i);
            if(s.replace(sub,"").length()==0)
                return i;
        }

    }
    return 0;
}
```

例题：重复字符串

题目链接：重复字符串 - 蓝桥云课 (lanqiao.cn)

题目描述

如果一个字符串 S 恰好可以由某个字符串重复 K 次得到，我们就称 S 是 K 次重复字符串。例如 abcabcabc 可以看作是 abc 重复 3 次得到，所以 abcabcabc 是 3 次重复字符串。

同理 aaaaaa 既是 2 次重复字符串、又是 3 次重复字符串和 6 次重复字符串。

现在给定一个字符串 S，请你计算最少要修改其中几个字符，可以使 S 变为一个 K 次字符串?

输入描述

输入第一行包含一个整数 K。

第二行包含一个只含小写字母的字符串 S。

其中，$1 \leq K \leq 10^5, 1 \leq |S| \leq 10^5$。其中 $|S|$ 表示 S 的长度。

输出描述

输出一个整数代表答案。如果 S 无法修改成 K 次重复字符串，输出 -1。

思路：已知重复次数为 K ，那么周期就是 S.length() / K ，然后只需要求出每一个周期的第 i 个字符，出现次数最多的字符是哪个，然后将其余字符全部改为它，那么就将 S 改为了重复 K 次的字符串，此时修改次数也是最少的。以abdcbbcaabca， 重复 3 次为例：

将此字符串拆分为三个部分后，每个周期写在一行，结果为：

abdc
bbca
abca

只需要求出每一个竖列出现次数最多的字符出现的次数，然后将其余字符全部改为它，那么这一列修改次数为（K - max），然后将每一列的结果加起来，即为答案。

```
import java.io.BufferedReader;
import java.io.IOException;
import  java.io.InputStreamReader;
import  java.io.OutputStreamWriter;
import java.io.PrintWriter;

public class 重复字符串 {

    static BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    static PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
    public static void main(String[] args) throws IOException {
        int n = Integer.parseInt(in.readLine());
        String a = in.readLine();
        if(a.length()%n!=0 || n>a.length()) {
            System.out.println(-1);
            return;
        }
        int t = a.length() / n;
        int index = 0;
        char[][] ch = new char[n][t];
        for(int i=0;i<n;i++)
            for(int j=0;j<t;j++)
                ch[i][j] = a.charAt(index++);
        int ans = 0;
        for(int i=0;i<t;i++) {
            int[] num = new int[26];
            int max = 0;
```

```
            for(int j=0;j<n;j++) {
                num[ch[j][i]-'a']++;
                if(max<num[ch[j][i]-'a']) {
                    max = num[ch[j][i]-'a'];
                }
            }
            ans = ans + (n-max);
        }
        System.out.println(ans);
    }
}
```

第三章 排序

[Java自定义排序 - 芯笑 (lamper.top)]

冒泡排序

思路：每一次循环将最大值 / 最小值放于向后移动。

```
public static int[] sort(int[] a){
    for(int  i=0;i<a.length-1;i++){
        for(int j=0;j<a.length-1-
i;j++){4    if(a[j]>a[j+1]){
                int temp = a[j+1];
                a[j+1] = a[j];
                a[j] = temp;
            }
        }
    }
    return a;
}
```

插入排序

思路：第 i 趟，把第 i 个元素放到前 i - 1 个有序的序列中 。

```
public static int[] InsertSort(int[] a){
    for(int i=1;i<a.length;i++){
        int temp = a[i];//处理第i个元素
        int j = i-1;
        for(;j>=0 && a[j]>temp;j--){
            a[j+1] = a[j];//大的元素往后移
        }
        a[j+1] = temp;
    }
    return a;
}
```

选择排序

思路：第 i 趟把从 i ~ 结尾最小的元素找到，放到 i 位置。

```
public static int[] SelectedSort(int[] a){
    for(int i=0;i<a.length;i++){
        int min = i;//存放i+1到最后最小的元素所在的下标
        for(int  j=i+1;j<a.length;j++){
            if(a[j]<a[min])
                min = j;
        }
        int temp = a[i];
        a[i] = a[min];
        a[min] = temp;
    }
    return a;
}
```

希尔排序

思路：将排序的区间分成若干个有跨度的子区间，对每一个子区间，进行插入排序，跨度不断 / 2 ，最终当跨度为 1 的时候，进行一个插入排序。

```
public static int[] shell(int[]
    a){ for(int gap =
    a.length/2;gap>0;gap/=2){
        //对每一分组进行直接插入排序
        for(int i=gap;i<a.length;i++){
            int j = i;
            while(j-gap>=0  &&  a[j-gap]>a[j]){//大的往后移动
                int temp = a[j];
                a[j] = a[j-gap];
                a[j-gap] = temp;
                j  =  j-gap;//下一次继续从分组的前一个位置开始
            }
        }
    }
    return a;
}
```

计数排序

思路：找出数组中的最大值和最小值，每个数都是在 min 和 max 之间，用一个长度为（max - min + 1）的数组 c 来存储每一个数出现的次数，然后将数组 c 转换为前缀和数组，则 c[i]，就表示不大于（i+min）的元素的个数，按照 c 数组还原排序结果。

```
public static void countSort(int[] a){
    int[] b = new int[a.length];
    int max = a[0];min = a[0];
    for(int i=0;i<a.length;i++){
        if(a[i]>max) max = a[i];
    if(a[i]<min) min = a[i];7  }
    int dis = max - min + ;
```

```
    int[] c = new int[dis];
    for(int i=0;i<a.length;i++)
        c[a[i]-min]++;
    for(int i=1;i<c.length;i++)
        c[i] = c[i] + c[i-1];
    for(int  i=a.length-1;i>=0;i--){
        b[c[a[i]-min]-1] = a[i];
        c[a[i]-min]--;
    }
    System.out.println(Arrays.toString(b));
}
```

第四章 数据结构基础

链表

为什么要用链表

数组作为一个顺序储存方式的数据结构，可是有大作为的，它的灵活使用为我们的程序设计带来了大量的便利；但是，数组最大的缺点就是我们的插入和删除时需要移动大量的元素，所以呢，大量的消耗时间，以及冗余度难以接收。

链表可以灵活地去解决这个问题，插入删除操作只需要修改指向的对象就可以了，不需要进行大量的数据移动操作。

单链表

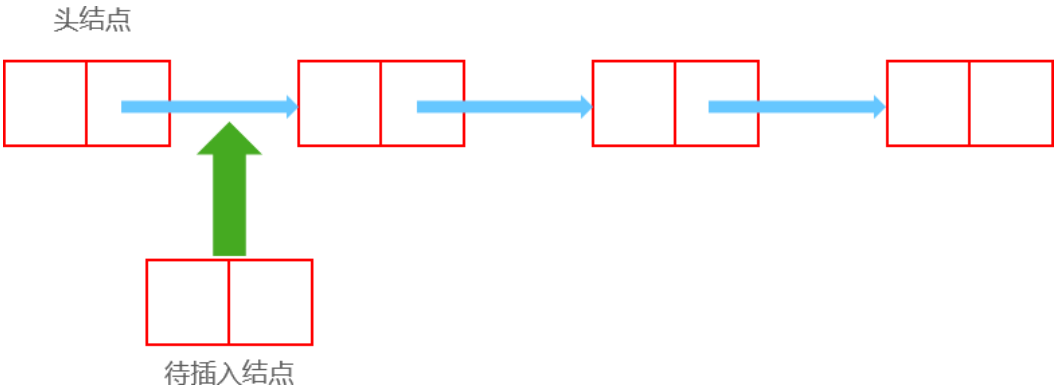
初始化

```
static class Node{//定义结点类
    int value;//本身的值
    Node next;//指向下一个结点
    public Node(int value, Node next) {
        this.value = value;
    this.next = next;7
    }
}
```

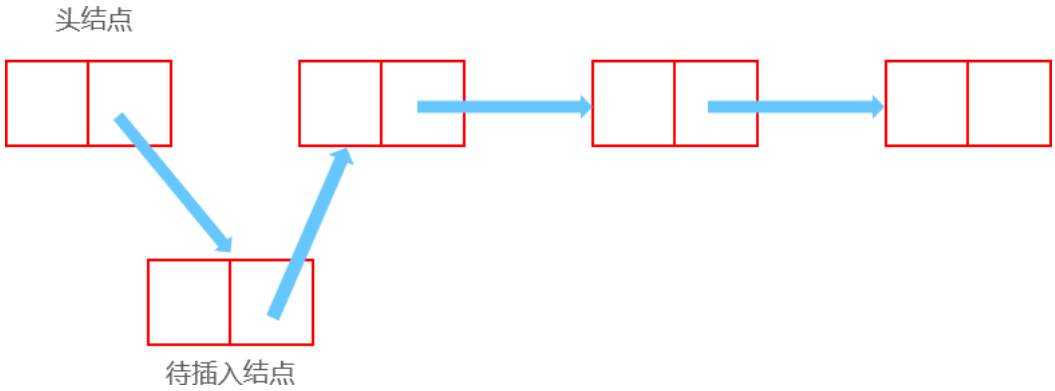
```
Node head = new Node(-1,null);//头结点
Node end = new Node(-1, null);//尾结点
Node per = head;
for(int i=1;i<=10;i++) {
    per.next = new Node(i, null);
per = per.next;7  }
per.next = end;
```

插入

插入前：



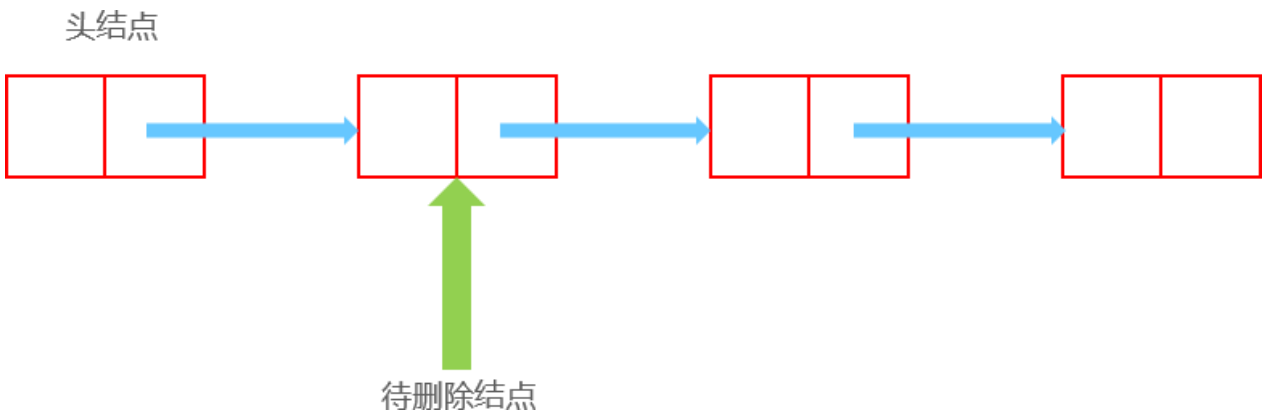
插入后：



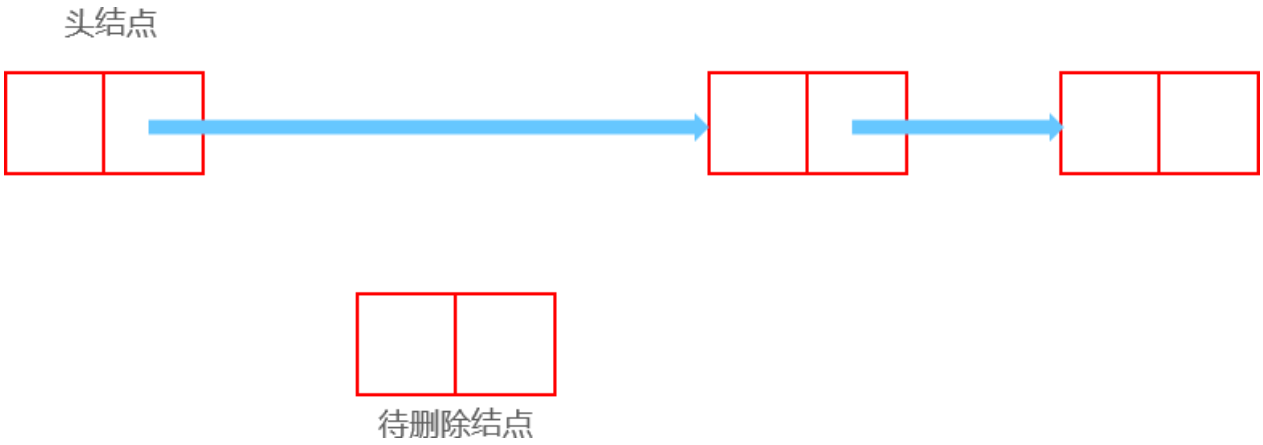

```
Node now;//待插入结点
now.next  =  head.next;//此节点的next为插入位置上一个结点的下一个结点
head.next  =  now;//此节点位置的上一个结点的下一个结点为now
```

删除

删除前：



删除后：




```
Node now;//待删除结点
head.next = now.next;
```

双链表

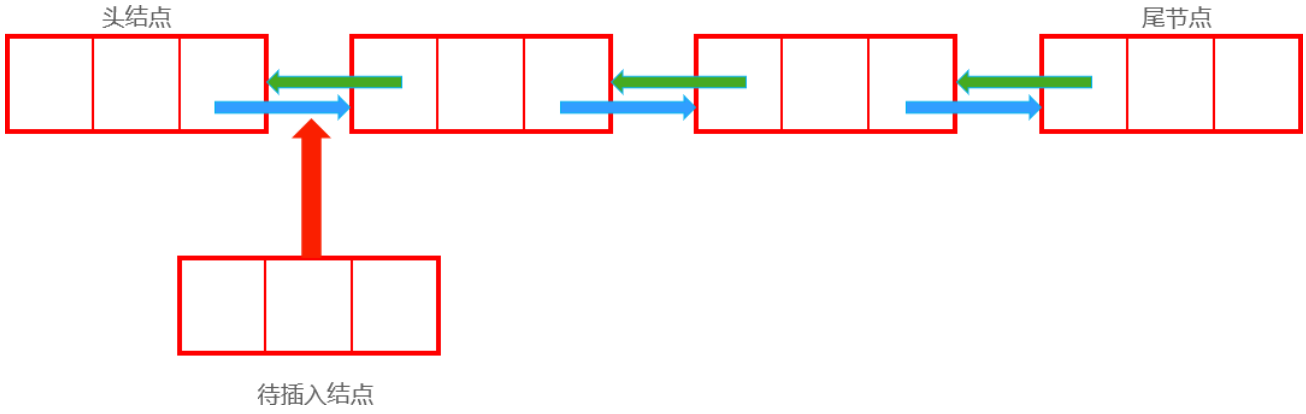
初始化

```
static class N{
    N last;
    int value;
    N next;
    public N(N last, int value, N next) {
        this.last = last;
        this.value = value;
    this.next = next;9  }
}
```

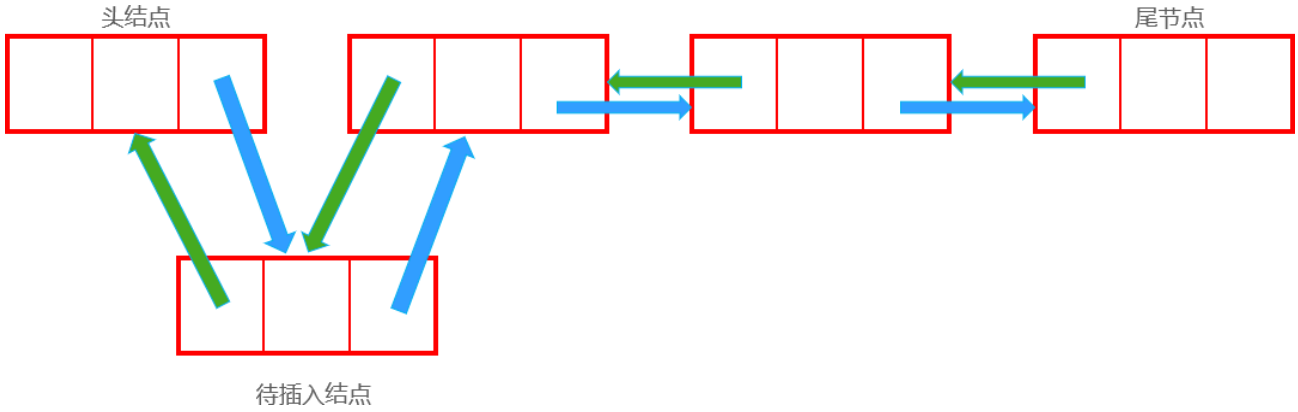
```
Node first = new Node(null,-1,null);//头结点
Node end = new Node(null,-1, null);//尾节点
Node per = first;
for(int i=1;i<=10;i++) {
    per.next = new N(per,i, null);
per = per.next;7  }
end.last = per;
per.next = end;
```

插入

插入前:



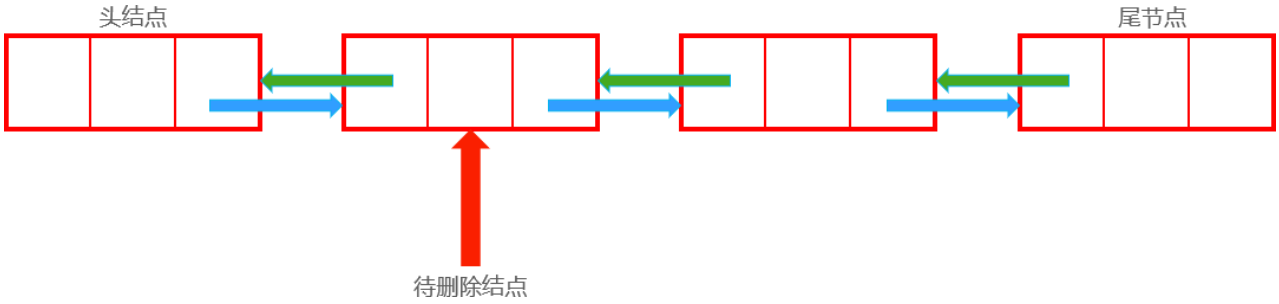
插入后：



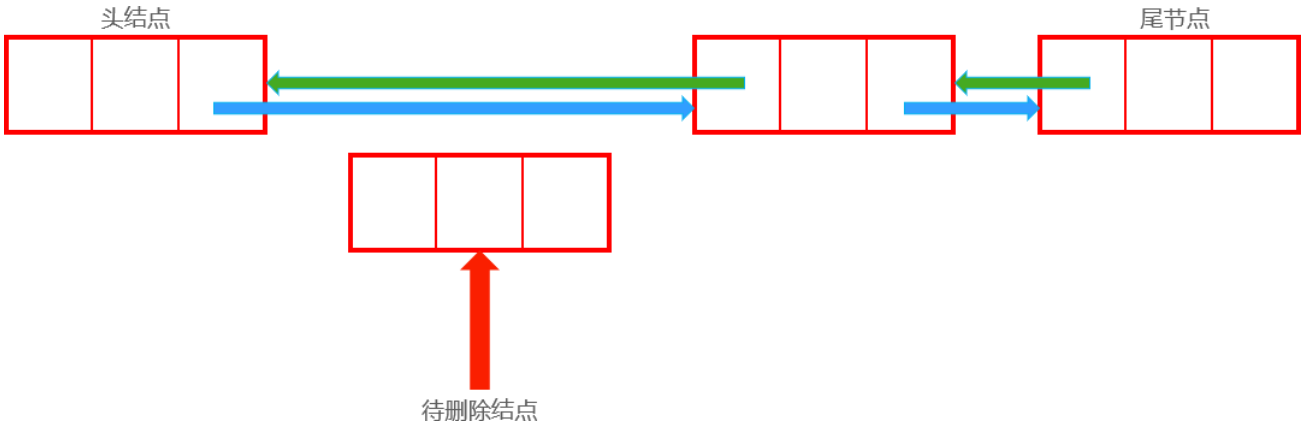

```
Node now;//待插入结点
now.next = first.next;
first.next.last = now;
first.next = now;
now.last = first;
```

删除

删除前：



删除后：




```
Node now;//待删除结点
now.last.next = now.next;
now.next.last = now.last;
```

例题：左移右移（双链表解法）

题目链接：左移右移 - 蓝桥云课 (lanqiao.cn)

问题描述

小蓝有一个长度为 N 的数组, 初始时从左到右依次是 $1, 2, 3, \ldots N$ 。

之后小蓝对这个数组进行了 M 次操作, 每次操作可能是以下 2 种之一:

1. 左移 x, 即把 x 移动到最左边。

2. 右移 x, 即把 x 移动到最右边。

请你回答经过 M 次操作之后, 数组从左到右每个数是多少?

思路：

1.创建双链表并完成初始化，初始元素为 1 ~ n；

2.无论 x 左移或右移，都要先将 x 从原位置删除，为了便于获取 x 对应的 Node 结点，用 Map 存储 x 和 value 为 x 的结点；

3.如果 x 为左移，就将 x 对应的 Node 结点插入到头结点后；

4.如果 x 为右移，就将 x 对应的 Node 结点插入到尾节点前；

```
import java.io.BufferedReader;
import java.io.IOException;
import  java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.util.HashMap;
import java.util.Map;

public class 左移右移_双链表 {
    static class
        Node{Node up;
        int value;
        Node down;
        public Node(Node up, int value, Node down) {
```

```
            this.up = up;
            this.value = value;
            this.down = down;
        }
    }
    public static void main(String[] args) throws IOException{
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
        String[] s = in.readLine().split(" ");
        int n = Integer.parseInt(s[0]);
        int m = Integer.parseInt(s[1]);
        Map<Integer, Node> map = new HashMap<>();
        Node first = new Node(null, -1, null);
        Node last = new Node(null, -1, null);
        Node no = first;
        for(int i=1;i<=n;i++) {
            no.down = new Node(no, i, null);
            no = no.down;
            map.put(i, no);
        }
        last.up = no;
        no.down = last;
        for(int i=0;i<m;i++) {
            s = in.readLine().split(" ");
            char ch = s[0].charAt(0);
            int x = Integer.parseInt(s[1]);
            Node node = map.get(x);
            node.up.down = node.down;
            node.down.up = node.up;
            if(ch=='L') {
                node.down = first.down;
                first.down.up = node;
                first.down = node;
                node.up = first;
            }else {
                node.up = last.up;
                last.up.down = node;
                node.down = last;
                last.up = node;
            }
        }
        no = first.down;
        while(no!=last) {
            System.out.print(no.value+" ");
            no = no.down;
        }
    }
}
```

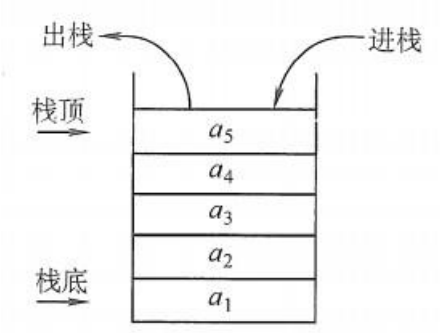
栈

栈

栈（Stack）：是只允许在一端进行插入或删除的线性表。首先栈是一种线性表，但限定这种线性表只能在某一端进行插入和删除操作。

栈顶（Top）：线性表允许进行插入删除的那一端。

栈底（Bottom）：固定的，不允许进行插入和删除的另一端。



常用方法

```
Stack<Integer> stack = new Stack();
boolean is = stack.isEmpty();//判断此栈是否为空
int n = stack.peek();//获取栈顶的元素，但不删除
int m = stacl.pop();//获取并删除栈顶的元素
stack.push(10);//将10压入栈中
stack.clear();//清空栈
```

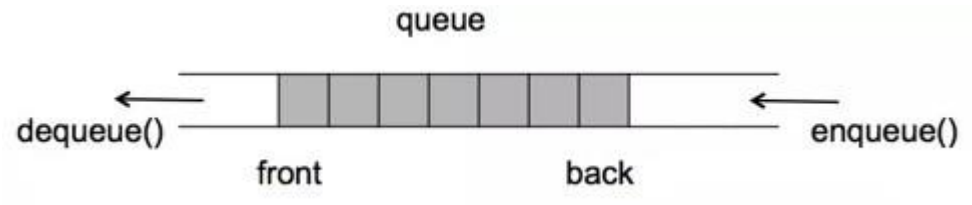
判断括号序列是否合法

```
public static boolean check(String s){
    Stack<Character> stack = new Stack();
    char[] ch = s.toCharArray();
    for(int i=0;i<ch.length;i++){
        if(ch[i]=='(')
            stack.push(ch[i]);
        else if(stack.isEmpty())
            return false;
        else
            stack.pop();
    }
    return stack.isEmpty();
}
```

队列

队列

队列（queue）是一种先进先出的、操作受限的线性表。



队列这种数据结构非常容易理解，就像我们平时去超市买东西，在收银台结账的时候需要排队，先去排队的就先结账出去，排在后面的就后结账，有其他人再要过来结账，必须排在队尾不能在队中间插队。

常用方法

```
Queue<Integer> queue = new LinkedList<>();
queue.peek();//获取队头元素，但不删除
queue.poll();//获取并删除队头元素
queue.clear();//清空队列
queue.push(11);//将11存放到队列中
```

例题：左移右移（栈 + 队列解法）

题目链接：左移右移 - 蓝桥云课 (lanqiao.cn)

问题描述

小蓝有一个长度为 N 的数组, 初始时从左到右依次是 $1, 2, 3, \ldots N$ 。

之后小蓝对这个数组进行了 M 次操作, 每次操作可能是以下 2 种之一:

1. 左移 x, 即把 x 移动到最左边。
2. 右移 x, 即把 x 移动到最右边。

请你回答经过 M 次操作之后, 数组从左到右每个数是多少?

思路：

1.如果一个数先移动到最左边，再移动到最右边，那么最后输出的时候这个数一定是在最右边，也就是一个数最终出现在哪里，以他最后一次出现为准；

2.为了避免一个数重复判断，而且要以他最后一次出现时的 L 和 R 操作为最终操作，所以可以先将全部输入分别存放到 char 类型数组和 int 类型数组中，然后逆序判断，并且用一个数组来表示这个 x 有没有出现过；

3.因为要对输入做逆序操作，所以，逆序时最后出现的 L 对应的 x 在输出的最前面，然后之后出现的 L 对应的 x 依次输出，即先入先出，可以用队列来存储进行 L 操作的 x ；

4.逆序时最后出现的 R 对应的 x 在输出的最后面，然后之后出现的 R 对应的 x 依次在前，即后入先出，可以用栈来存储进行 R 操作的 x ；

5.输出时，先输出队列中的元素，然后将 1 ~ n 中没有出现过的值按序输出，最后输出栈中的元素；

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
```

```java
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import  java.util.LinkedList;
import java.util.Queue;
import java.util.Stack;

public class 左移右移_栈_队列 {

    public static void main(String[] args) throws IOException{
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
        String[] s = in.readLine().split(" ");
        int n = Integer.parseInt(s[0]);
        int m = Integer.parseInt(s[1]);
        int[] a = new int[n+1];
        char[] c = new char[m];
        int[] x = new int[m];
        for(int i=0;i<m;i++) {
            s = in.readLine().split(" ");
            c[i] = s[0].charAt(0);
            x[i] = Integer.parseInt(s[1]);
        }
        Stack<Integer> r = new Stack<>();
        Queue<Integer> l = new LinkedList<>();
        for(int i=m-1;i>=0;i--) {
            if(a[x[i]]==0) {//判断x[i]是否出现过
                a[x[i]] = 1;//若x[i]没有出现过
                if(c[i]=='L')
                    l.add(x[i]);
                else
                    r.push(x[i]);
            }
        }
        while(l.size()!=0)  //输出队列中元素
            System.out.print(l.poll()+" ");
        for(int i=1;i<=n;i++)
            if(a[i]==0)  //a[i]为0，表示i没有出现过
                System.out.print(i+" ");
        while(r.size()!=0)  //输出栈中元素
            System.out.print(r.pop()+" ");
    }
}
```

第五章 分治算法

归并排序

思路：先把数组从中间分成前后两部分，然后分别对前后两部分进行排序，再将排好序的两部分数据合并在一起

```java
public static void mergeSort(int[] a,int left,int right){//待排序数组，要排序的范围[left,right]
    int mid = (left+right)>>1;
    if(left<right){
        mergeSort(a,left,mid);
```

```
        mergeSort(a,mid+1,right);
        merge(a,left,mid,right);
    }
}
public static void merge(int[] a,int left,int mid,int
    right){int[] temp = new int[right-left+1];//临时数组，用
    来归并
    int  i=left,j=mid+1,k=0;//左半段用i指向，右半段用j指向，temp数组用k指向
    while(i<=mid && j<=right){
        if(a[i]<a[j])
            temp[k++] = a[i++];
        else
            temp[k++] = a[j++];
    }
    while(i<=mid) temp[k++] = a[i++];
    while(j<=right) temp[k++] = a[j++];
    for(int x=0;x<temp.length;x++){
        a[left+x] = temp[x];
    }
```

快速排序

思路：

1.首先设定一个分界值，通过该分界值将数组分成左右两部分。

2.将大于或等于分界值的数据集中到数组右边，小于分界值的数据集中到数组的左边。此时，左边部分中各元素都小于分界值，而右边部分中各元素都大于或等于分界值。

3.然后，左边和右边的数据可以独立排序。对于左侧的数组数据，又可以取一个分界值，将该部分数据分成左右两部分，同样在左边放置较小值，右边放置较大值。右侧的数组数据也可以做类似处理。

4.重复上述过程，可以看出，这是一个递归定义。通过递归将左侧部分排好序后，再递归排好右侧部分的顺序。当左、右两个部分各数据排序完成后，整个数组的排序也就完成了。

```
public static void quickSort(int[] a,int left,int right){
    if(left>right)  return;//区间擦肩而过，无效，不需要进行递归
    int i=left,j=right,temp = a[left];//a[left]作为基准点
    while(i!=j){
        while(a[j]>=a[temp] && j>i)
            j--;//只要a[j]大于基准点继续往前移动j
        if(j>i)
            a[i++] = a[j];
        while(a[i]<=a[temp] && i<j)
            i++;
        if(i<j)
            a[j--] = a[i];
    }
    a[i] = temp;//基准点元素放到最终位置
    quickSort(a,left,i-1);
    quickSort(a,i+1,right);
}
```

快速幂

思路：

1.每一步都把指数分成两半，而相应的底数做平方运算。这样不仅能把非常大的指数给不断变小，所需要执行的循环次数也变小，而最后表示的结果却一直不会变。

例：3^{10} = 3*3*3*3*3*3*3*3*3*3 ，尽量想办法把指数变小来，这里的指数为10。

$3^{10}=(3*3)(3*3)(3*3)(3*3)(3*3)$

$3^{10}=(3*3)^5$

$3^{10}=9^5$

2.此时指数由10缩减一半变成了5，而底数变成了原来的平方，求3^{10}原本需要执行10次循环操作，求9^5却只需要执行5次循环操作，但是3^{10}却等于9^5,用一次（底数做平方操作）的操作减少了原本一半的循环量，特别是在幂特别大的时候效果非常好，例如$2^{10000}=4^{5000}$,底数只是做了一个小小的平方操作，而指数就从10000变成了5000，减少了5000次的循环操作。

3.现在问题是如何把指数5变成原来的一半，5是一个奇数，5的一半是2.5，但是指数不能为小数，因此不能简单粗暴地直接执行5/2，然而，这里还有另一种方法能表示9^5，$9^5=9^4*9^1$

4.此时抽出了一个底数的一次方，这里即为9^1，这个9^1先单独移出来,剩下的9^4又能够在执行“缩指数”操作了，把指数缩小一半，底数执行平方操作。$9^5 = 81^2*9^1$

5.把指数缩小一半，底数执行平方操作，9^5 = 6561^1*9^1

6.此时，发现指数又变成了一个奇数1，按照上面对指数为奇数的操作方法，应该抽出了一个底数的一次方，这里即为6561^1，这个 6561^1 ^先单独移出来，但是此时指数却变成了0，也就意味着我们无法再进行“缩指数”操作了。

$9^5=(6561^0)(9^1)(6561^1)=1(9^1)(6561^1)=(9^1)(6561^1)=9*6561=59049$

7.能够发现，最后的结果是 9*6561。所以能发现一个规律：最后求出的幂结果实际上就是在变化过程中所有当指数为奇数时底数的乘积。

8.继续优化：b%2==1可以用更快的“位运算”来代替，例如：b&1。因为如果b为偶数，则其二进制表示的最后一位一定是0；如果 b 是奇数，则其二进制表示的最后一位一定是 1 。将他们分别与 1 的二进制做“与”运算，得到的就是b二进制最后一位的数字了，是0则为偶数，是1则为奇数。例如9是奇数，则 9&1=1；而8是偶数，则8&1=0；因此奇偶数的判断就可以用“位运算”来替换了。

9.m = m / 2也可以用更快的移位操作来代替，例如：6的四位二进制为0110，而6/2=3，3的四位二进制为0011，可以发现，a 的一半，结果为 a 的二进制码向右移一位，即 m >>=1。

```
public static long num(long n, long m, long p) {
    long result = 1;
    while (m > 0) {
        if ((m & 1 ) == 1) {
            result = result * n % p;
        }
        m >>= 1;
        n = (n * n) % p;
    }
    return result;
}
```

第六章 搜索

BFS 和 DFS的思路

BFS（广度优先搜索）

思路：BFS类似于树的层次遍历过程,从根节点开始，沿着树的宽度遍历树的节点。如果所有节点均被访问，则算法中止。舍去空间换时间（每一个的状态都需要用一个对象存储起来，放到一个队列中去）。

实现：

1.创建一个空队列 queue（用来存放节点）和一个空列表 visit（用来存放已访问的节点）2.

依次将起始点及邻接点加入 queue 和 visit 中

3.poo出队列中最先进入的节点,从图中获取该节点的邻接点

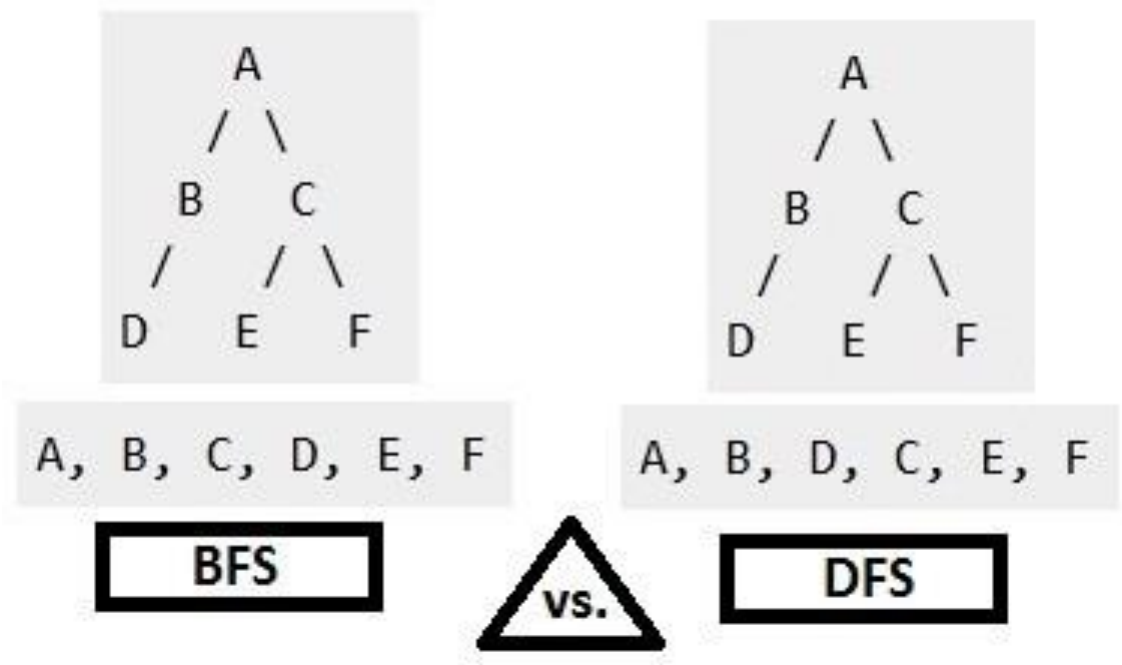
4.如果邻接点不在 visit 中（这个情况还没有判断），则将该邻接点加入 queue 和 visit 中

5.输出pop出的节点

6.重复3、4、5，直至队列为空

DFS（深度优先搜索）

思路：沿着树的深度遍历树的节点，选一条路一直走到底，回到之前有路的点，换一个方向继续走，知道遍历所有的子节点（所有情况全部判断完成），进而达到全局搜索的目的。



全排列

DFS解法

思路：将此过程看做一棵树，每一个结点下都会有 n 个结点表示下一个数，首先先将全部 n^n 个结果全部得出，然后剪枝，减去有重复数字出现的情况。

```
public static void dfs(int depth,String ans,int n){//当前深搜的层数，目前的结果，目标层数
    if(depth==n){//当前深搜层数=目标层数
        System.out.println(ans);
        return;
    }
    for(int i=1;i<=n;i++){
        if(!ans.contains(i+""))//只有当还没有用过i的时候，才会在现在的基础上继续往下拓展
             dfs(depth+1,ans+i;n);//进入下一层，ans记录为进入下一层的值，n不变
    }
}
```

BFS解法

思路：先将有重复数字的结果得出，每一个数后都可以跟 n 中可能，那么将这 n 中可能存入队列中，然后重复此过程，直到字符串的长度为 n 时，得到结果；剪枝，如果这个数字已经用过了，就直接只用下一个数字。

```
public static void bfs(int n){
    Queue<String> queue = new LinkedList<>();
    for(int i=1;i<=n;i++)
        queue.offer(i+"");
    while(!queue.isEmpty()){
        String now = queue.poll();
        for(int  i=1;i<=n;i++){//每个结点都向下产生n个结果
            if(now.contains(i+""))//i已经使用过了
                continue;
            String son = now + i;
            if(son.length()==n)
                System.out.println(son);
            else
                queue.offer(son);
        }
    }
}
```

整数划分

思路：对 n 进行划分后， n 可以被不超过 n 个数累加得到，进行累加的每一个数，也可以被不超过它本身个数累加得到。

```java
public static void dfs(int n,int nowget,int max,String ans){//要划分的数，现在已经得到的值，目前划分已经用到的最大值，具体拆分方法
    if(nowget==n){
        ans = ans.substring(0,ans.length()-1);
        System.out.println(n+"="+ans);
        return;
    }
    for(int  i=1;i<=n-nowget;i++){//从nowget累加到n
        if(i>=max)//只有当下一个数不小于我之前用过的最大值时，才能保证整个结果为非递减
            dfs(n,nowget+i,i,ans+i+"+");
    }
}
```

例题

例题：路径之谜

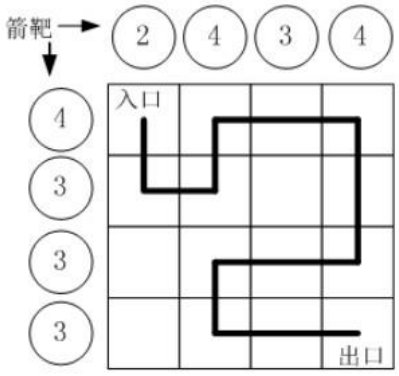
题目链接：路径之谜 - 蓝桥云课 (lanqiao.cn)

题目描述

小明冒充 X 星球的骑士，进入了一个奇怪的城堡。

城堡里边什么都没有，只有方形石头铺成的地面。

假设城堡地面是 $n \times n$ 个方格。如下图所示。



按习俗，骑士要从西北角走到东南角。可以横向或纵向移动，但不能斜着走，也不能跳跃。每走到一个新方格，就要向正北方和正西方各射一箭。（城堡的西墙和北墙内各有 n 个靶子）同一个方格只允许经过一次。但不必走完所有的方格。如果只给出靶子上箭的数目，你能推断出骑士的行走路线吗？有时是可以的，比如上图中的例子。

本题的要求就是已知箭靶数字，求骑士的行走路径（测试数据保证路径唯一）

思路：

1.从入口点开始，到达每一个点都将对应位置北墙和西墙的箭靶数减一，每一个点，都可以继续向四个方向继续前进（前提是这个点没有走过，在城堡范围内，且这个点对应的两个箭靶的数字不为 0 ）。

2.如果已经到了终点，就要判断现在每一个箭靶上的数字是否都已经变为 0 ，如果是，那么此时走的路径就是正确解，否则就需要回溯，考虑其他的行走路线。

3.回溯：因为要从已经走过的点退回来，所以在已经走过的点上射的箭要收回，箭靶数加一，并且标记此点为还没有走过。

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import  java.io.PrintWriter;

public class 路径之谜 {
    static  int[]  path;//记录最终路径，因为底面为n*n，所以走出需要2*n步
    static int n;
    static int[] cntx;//存储北墙箭靶数字
    static int[] cnty;//存储西墙箭靶数字
    static boolean[][] visited;//判断此点有没有走过
    static int dx[] = {1, 0, -1, 0};//到下一个点x坐标的变化量
    static int dy[] = {0, 1, 0, -1};//到下一个点y坐标的变化量
    static BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    static PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
    public static void main(String[] args) throws IOException {
        n = Integer.parseInt(in.readLine());
        cntx = new int[n];
        cnty = new int[n];
        path = new int[n * n];
        visited = new boolean[n][n];
        String[] s = in.readLine().split(" ");
        for (int i = 0; i < n; i++) {
            cntx[i] = Integer.parseInt(s[i]);
        }
        s = in.readLine().split(" ");
        for (int i = 0; i < n; i++) {
            cnty[i] = Integer.parseInt(s[i]);
        }
        dfs(0, 0, 0);//从0,0位置开始走,目前走了0步
  }
    private static void dfs(int x, int y, int step) {
        path[step] = y * n + x; //将该点编号记录到路径中
        visited[x][y]  =  true;//将该点标记为已经走过的状态
        cntx[x]--;//拔掉对应北墙的箭
        cnty[y]--;//拔掉对应西墙的箭
        if (x == n - 1 && y == n - 1 && check()){//判断是否到达终点
            for (int i = 0; i <= step; i++){//输出答案
                System.out.print(path[i]+" ");
            }
            return;
        }
        for (int i = 0; i < 4; i++){//上下左右四个方向搜索下一步
            int xx = x + dx[i], yy = y + dy[i];
             //下一步(xx,yy)未走过且在地图范围内
            if (0 <= xx && xx <= n-1 && yy >= 0 && yy <= n-
                1&& !visited[xx][yy] ){ if (cntx[xx] > 0 && cnty[yy] > 0){//该点对
                应箭靶上有箭，说明该点可以走
                    dfs(xx, yy, step + 1);//搜索下一步
                    //要从xx,yy点回来,在xx,yy点射的箭要复原，并重新标记xx,yy点没有走过
                    visited[xx][yy] = false;
                    cntx[xx]++; cnty[yy]++;
                }
            }
        }
    }
    private static boolean check() {//判断到达终点时,是否箭靶数都已经归零
```

```
        for (int i = 0; i < n; i++) {
            if (cntx[i] != 0 || cnty[i] != 0)
                return false;
        }
        return true;
    }
}
```

例题：迷宫

题目链接：迷宫 - 蓝桥云课 (lanqiao.cn)

题目描述

本题为填空题，只需要算出结果后，在代码中使用输出语句将所填结果输出即可。

下图给出了一个迷宫的平面图，其中标记为 1 的为障碍，标记为 0 的为可以通行的地方。

```
010000
000100
001001
110000
```

迷宫的入口为左上角，出口为右下角，在迷宫中，只能从一个位置走到这 个它的上、下、左、右四个方向之一。

对于上面的迷宫，从入口开始，可以按 DRRURRDDDR 的顺序通过迷宫， 一共 10 步。其中 D、U、L、R 分别表示向下、向上、向左、向右走。 对于下面这个更复杂的迷宫（30 行 50 列），请找出一种通过迷宫的方式，其使用的步数最少，在步数最少的前提下，请找出字典序最小的一个作为答案。

请注意在字典序中 $D < L < R < U$。

思路：从起点开始，将从此点能到达的点存储到队列中，每次获取并删除队列中的第一个元素，并将其能到达且还未到达过的点（若此点已经到达过，则表示当前处理的这条路径不是最短路径）存储到队列中，若已经到达终点，则此路径为最短路径。如果队列中已经没有元素，但仍未到达迷宫终点，则表示此迷宫无解

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.util.LinkedList;
import java.util.Queue;

public class 迷宫 {
    static  int  num;//存储迷宫最短路径所需要的步数
    static int xsize = 30;//迷宫大小30行50列
    static int ysize = 50;
    static char[][] arr = new char[xsize][ysize];//存储迷宫：0表示路，1表示墙
    static boolean[][] help = new boolean[xsize][ysize];//判断此点是否已经做过
    static  int[][]  dir  =  {{1,0},{0,-1},{0,1},{-1,0}};//四个方向横纵坐标的变化量
    static char[] sign = {'D','L','R','U'};//表示四个方向
    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
        for(int i=0;i<xsize;i++){
            arr[i] = in.readLine().toCharArray();
        }
        out.println(bfs());
        out.print(num);//额外输出最短路径需要多少步
        out.flush();
    }
```

```
    private static String bfs() {
        Queue<Node> list = new LinkedList<>();//队列
        int x = 0;
        int y = 0;
        int runnum = 0;
        list.add(new  Node(x,y,"",runnum));//将起点存储到队列中
        while(!list.isEmpty()){//判断队列是否为空，若为空，则此迷宫没有通路
            Node now = list.poll();//获取队列中的第一个元素并删除
            help[now.x][now.y]  =  true;//将此点标记为已经走过
            for(int  i=0;i<4;i++){//循环四次，对四个方向进行处理
                int xx = now.x + dir[i][0];//移动后的x坐标
                int yy = now.y + dir[i][1];//移动后的y坐标
                  //此点在迷宫范围内，未走过，不是墙
                if(check(xx,yy) && help[xx][yy]==false && arr[xx][yy]=='0'){
                    list.add(new Node(xx,yy,now.num + sign[i],now.runnum + 1));//将此点存入队列中
                    if(xx==xsize-1  &&  yy==ysize-1){//如果已经到了迷宫终点
                        num = now.runnum + 1;//所需步数+1（now.runnum是到达迷宫终点前一步所需要的步数）
                        return now.num + sign[i];//返回通过迷宫的方式
                    }
                }
            }
        }
        return  "";//空字符串，表示此迷宫无通路
    }
    private static boolean check(int xx, int yy) {//判断此点是否在迷宫范围内
        return xx>=0 && yy>=0 && xx<xsize && yy<ysize;
    }
    static class
        Node{int
        x;//x坐标int
        y;//y坐标
        int  runnum;//到达此点最短步数
        String num;//到达此点的方式
        public Node(int x, int y,String num ,int runnum)
            {super();
            this.x = x;
            this.y = y;
            this.num = num;
            this.runnum = runnum;
        }
}   }
```

第七章 贪心

基本概念

所谓贪心算法是指，在对问题求解时，总是做出在当前看来是最好的选择。也就是说，不从整体最优上加以考虑，它所做出的仅仅是在某种意义上的局部最优解。

贪心算法没有固定的算法框架，算法设计的关键是贪心策略的选择。必须注意的是，贪心算法不是对所有问题都能得到整体最优解，选择的贪心策略必须具备无后效性（即某个状态以后的过程不会影响以前的状态，只与当前状态有关。

例题

例题：合并果子

题目链接：合并果子 - 蓝桥云课 (lanqiao.cn)

题目描述

在一个果园里，多多已经将所有的果子打了下来，而且按果子的不同种类分成了不同的堆。多多决定把所有的果子合成一堆。

每一次合并，多多可以把两堆果子合并到一起，消耗的体力等于两堆果子的重量之和。可以看出，所有的果子经过 $n-1$ 次合并之后，就只剩下一堆了。多多在合并果子时总共消耗的体力等于每次合并所耗体力之和。

因为还要花大力气把这些果子搬回家，所以多多在合并果子时要尽可能地节省体力。假定每个果子重量都为 1，并且已知果子的种类数和每种果子的数目，你的任务是设计出合并的次序方案，使多多耗费的体力最少，并输出这个最小的体力耗费值。

例如有 3 种果子，数目依次为 1，2，9。可以先将 1、2 堆合并，新堆数目为 3，耗费体力为 3。接着，将新堆与原先的第三堆合并，又得到新的堆，数目为 12，耗费体力为 12。所以多多总共耗费体力 $=3+12=15$。可以证明 15 为最小的体力耗费值。

思路：

1.要保证最终耗费的体力最小，那么就可以每次合并都把目前数量最少的两堆果子合并，耗费的体力就是这两堆果子树木的和，然后合并后又可以作为新的一堆果子继续去判断，直到最终只剩一堆。

2.可以借助PriorityQueue优先队列，队列中第一个元素就是最小值，即可每次获取队列中前两个元素，然后将他们的和再次添加至队列中，直到最终队列中只剩一个元素。

3.以题目样例为例：对于数组{pi}={1，2，9}，Huffman树的构造过程如下：

3.1 找到{1, 2, 9}中最小的两个数，分别是 1 和 2 ，

3.2 从{pi}中删除它们并将和 3 加入，得到{3, 9}，体力消耗为 3 。

3.3 找到{3， 9}中最小的两个数，分别是 3 和 9 ，

3.4 从{pi}中删除它们并将和 12 加入，得到{12}，费用为 12 。

3.5现在，数组中只剩下一个数12，构造过程结束，总费用为3 + 12 = 15。

```java
import java.io.BufferedReader;
import java.io.IOException;
import  java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.util.PriorityQueue;
import java.util.Scanner;
import  java.util.Spliterator;

public class Main {

    public static void main(String[] args) throws IOException{
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
        int n = Integer.parseInt(in.readLine());
        long sum = 0;
        PriorityQueue<Long> queue = new PriorityQueue<>();
        String[] s = in.readLine().split(" ");
        for(int i=0;i<s.length;i++) {
```

```
            queue.add(Long.parseLong(s[i]));
        }
        long number = 0;
        while(queue.size()!=1) {
            long a = queue.poll();//获取最小的一堆
            long b = queue.poll();//获取最小的一堆
            number = number + ( a + b );//合并这两堆耗费的体力
            queue.add((a+b));//将合并后的结果放回优先队列中
        }
        System.out.println(number);
    }
}
```

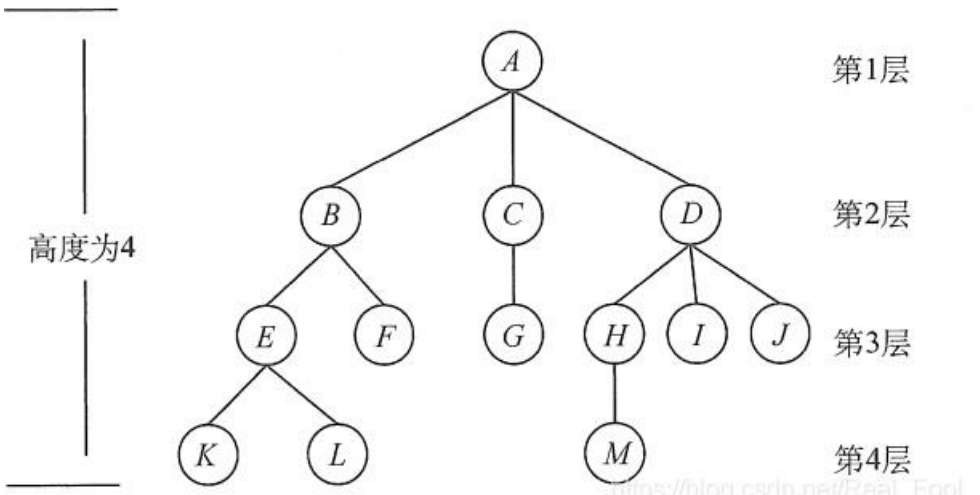
第八章 树

树的相关概念

什么是树

树(Tree)是 n (n ≥ 0)个结点的有限集。n=0时称为空树。在任意一颗非空树中：有且仅有一个特定的称为根的结点。当n>1时，其余结点可分为 m (m > 0) 个互不相交的有限集T1、T2、T3……、Tm，其中每个集合本身又是一棵树，并且称为根的子树。

树的基本概念



二叉树

什么是二叉树

二叉树是n(n>=0)个结点的有限集合，该集合或者为空集（空二叉树）、或者由一个根结点和两颗互不相交的、分 别称为根结点的左子树和右子树的二叉树组成。

二叉树的特点

1.二叉树中每个结点最多有两颗子树，度没有超过2的。

2.左子树和右子树是有顺序的，不能颠倒。

满二叉树

在二叉树中，所有的分支节点都有左子树和右子树，并且所有的叶子都在同一层。

完全二叉树

1.叶子结点只能出现在最下面两层。

2.最下层的叶子一定集中在左部连续位置。

3.倒数第二层，若有叶子结点，一定在右部连续位置。

4.如果结点度为1，则该结点只有左孩子。

5.同样结点的二叉树，完全二叉树的深度最小。

二叉树的创建和嵌套打印

```
//结点类
public class TreeNode{
    int data;//结点存放的数据
    TreeNode left;//左孩子
    TreeNode right;//右孩子
    public TreeNode(int data,TreeNode left,TreeNOde right){
        this.data = data;
        this.left = left;
    this.right = right;10
    }
}
```

```
import java.util.Scanner;
public class Tree{
    TreeNode  root;//整棵树的根节点
    Scanner sc = new Scanner(System.in);
    public Tree(){
    root = null;7
    }
    public TreeNode createBinaryTree(){//树的创建
        TreeNode  t;//当前树的根节点
        int x = sc.nextInt();
        if(x==0) t=null;
        else{
            t = new TreeNode();
            t.data = x;
            t.left = createBinaryTree();
        t.right = createBinaryTree();17
        }
        return t;
    }
    public void printTree(TreeNode t){//树的打印
        if(t!=null){
```

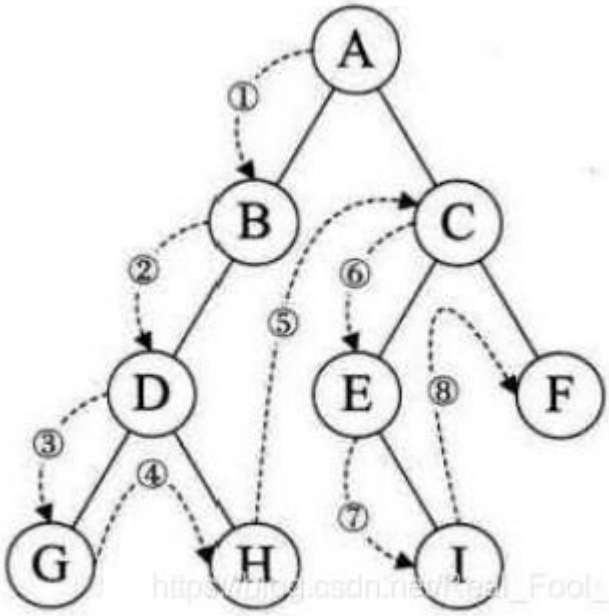
```
            System.out.print(t.data);
            if(t.left!=null || t.right!=null){
                System.out.print("(");
                printTree(t.left);
                if(t.right!=null)  System.out.print(",");
                printTree(t.left);
                System.out.print(")");
            }
        }
    }
}
```

前中后序层次遍历

前序遍历

[二叉树的递归与非递归遍历 - 芯笑 (lamper.top)]

思路：对于每个结点，优先处理结点本身，再处理它的左孩子，最后处理它的右孩子。

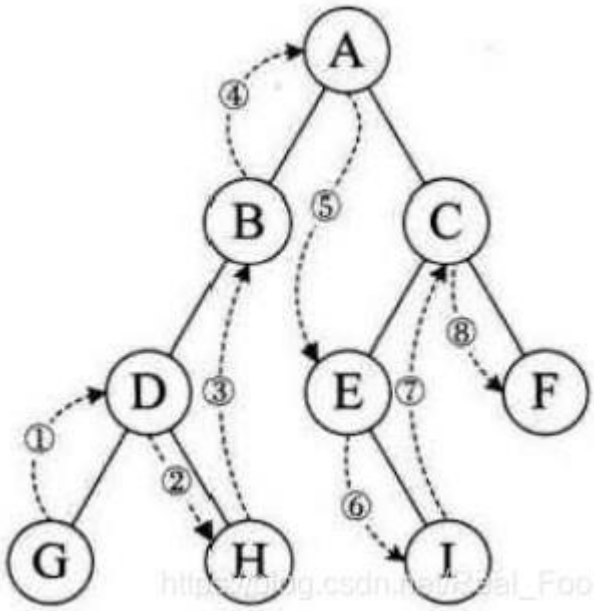


此图结果为：ABDGHCEIF

```
public void preOrder(TreeNode root){
    if(root!=null){
        System.out.print(root.data+" ");
        preOrder(root.left);
        preOrder(root.right);
    }
}
```

中序遍历

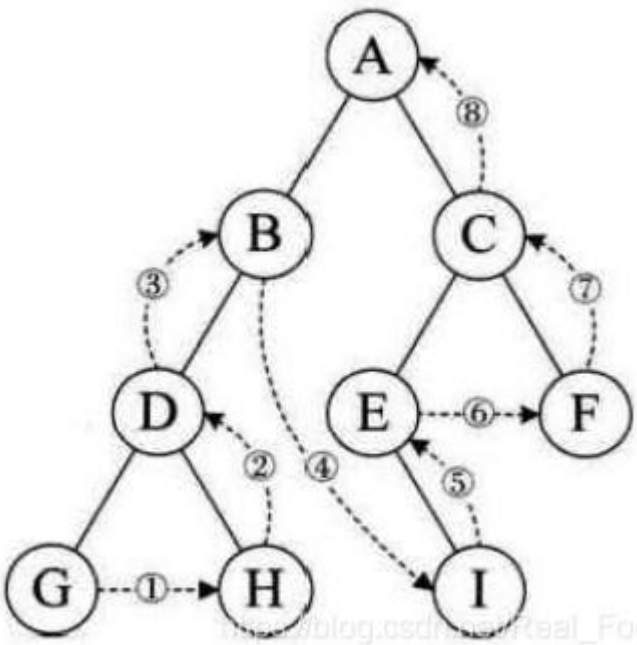
思路：对于每个结点，优先处理它的左孩子，再处理它本身，最后处理它的右孩子。



此图结果为：GDHBAEICF

```
public void midOrder(TreeNode root){
    if(root!=null){
        midOrder(root.left);
        System.out.print(root.data+" ");
        midOrder(root.right);
    }
}
```

后序遍历



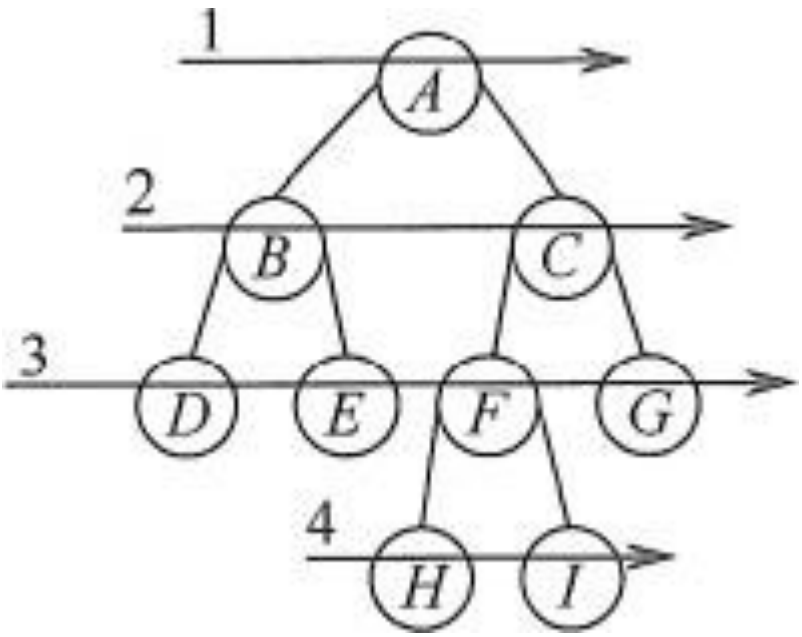
此图结果为：GHDBIEFCA

思路：对于每个结点，优先处理它的左节点，再处理它的右节点，最后处理它本身。

```
public void postOrder(TreeNode root){
    if(root!=null){
        postOrder(root.left);
        postOrder(root.right);
        System.out.print(root.data+" ");
    }
}
```

层次遍历

思路：广度优先搜索；借助一个队列；先将二叉树根结点入队，然后出队，访问出队结点，若它有左子树，则将左子树根结点入队;若它有右子树，则将右子树根结点入队。然后出队，访问出队结…如此反复，直至队列为空。



此图结果为：ABCDEFGHI

```
public void levelOrder(TreeNode t){
    Queue<TreeNode> queue = new LinkedList<>();
    if(t==null) return;
    queue.offer(t);
    while(!queue.isEmpty()){
        TreeNode head = queue.poll();
        System.out.print(head.data);
        if(head.left!=null)
            queue.offer(head.left);
        if(head.right!=null)
            queue.offer(head.right);
    }
}
```

求二叉树深度

```
public int treeDepth(TreeNode root){
    if(root==null) return 0;//此结点不存在
return  Math.max(treeDepth(root.left),treeDepth(root.right))+1; 4
}
```

求二叉树叶子结点个数

```
public int TreeLeaf(TreeNode root){
    if(root==null) return 0;
    if(root.left==null  &&  root.right==null)  return  1;//此结点没有孩子，表示此结点为叶子结点
else return treeLeaf(root.left) + treeLeaf(root.right);5
}
```

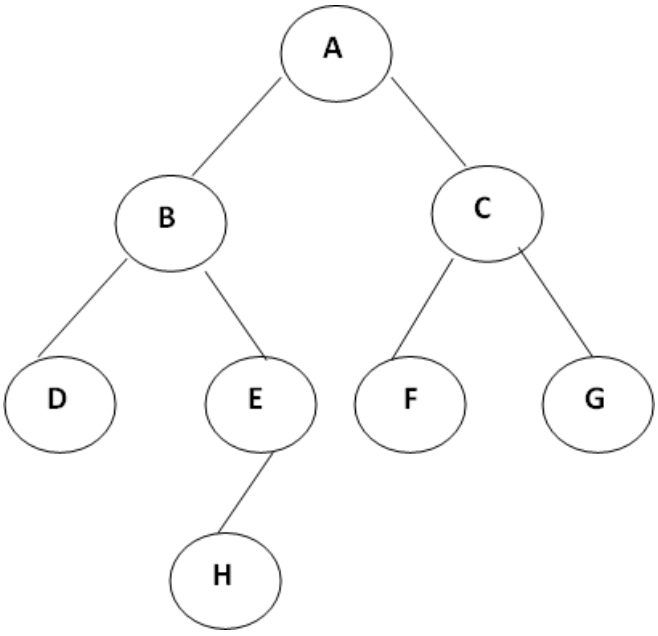
重建二叉树

思路：

1.前序遍历为：根，{左子树}，{右子树}；可得，前序遍历的第一个结点为根结点；

2.中序遍历为：{左子树}，根，{右子树}；可得，结点的左侧为它的左孩子树，右侧为它的右孩子树；

3.重复此过程，重建此二叉树（求后序遍历结果）；



以此树为例：

1.前序遍历为 ABDEHCFG，中序遍历为 DBHEAFCG；

2.前序遍历的第一个点（A）一定是根节点，那么中序遍历中，A 左边的部分（DBHE）为 A 的左子树，右边的部分（FCG）为 A 的右子树；

3.每一个子树也都是一个完整的树，那么对于树（BDEH），B 是这个子树的根节点，这个子树的左子树为（D），右子树为（EH）；对于树（CFG），C 是这个子树的根节点，这个子树的左子树为 F，右子树为 G；

4.重复此过程，直至每一个子树都检索完成为止，此时，树其实已经重建出来了，后序遍历也可得到（对于每一个子树，优先输出左子树，输出右子树，最后输出根节点）；

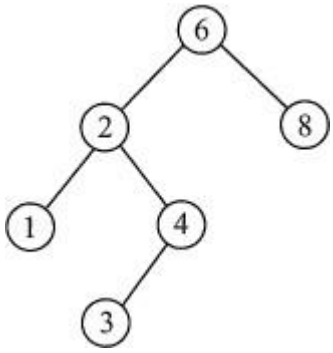
```java
public static String f(String pre,String mid){//前序遍历结果，中序遍历结果
    if(pre.length()==0) return "";
    else if(pre.length==1) return pre;
    else{
        int pos = mid.indexOf(pre.charAt(0));
        String left = f(pre.substring(1,pos+1),mid.substring(0,pos));
        String right = f(pre.substring(pos+1),mid.substring(pos+1));
        return left+right+pre.charAt(0);
    }
}
```

二叉排序树

二叉排序树(也称二叉查找树)或者是一棵空树，或者是具有下列特性的二叉树:

若左子树非空，则左子树上所有结点的值均小于根结点的值。
若右子树非空，则右子树上所有结点的值均大于根结点的值。
左、右子树也分别是一棵二叉排序树。
根据二叉排序树的定义，左子树结点值 < 根结点值 < 右子树结点值，所以对二叉排序树进行中序遍历，可以得到一个递增的有序序列。

此图二叉排序树的中序遍历序列为 123468 。

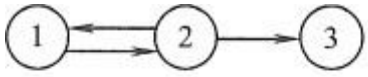


第九章 图

图的概念

有向图

若E是有向边(也称弧)的有限集合时，则图G为有向图。弧是顶点的有序对，记为<v, w>，其中v,w是顶点，v称为弧尾，w称为弧头，<v,w>称为从顶点v到顶点w的弧，也称v邻接到w，或w邻接自v。



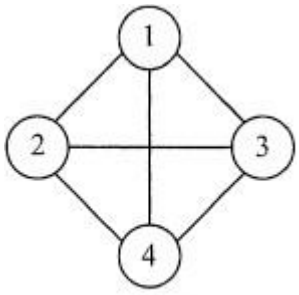
(a) 有向图G_1

图(a)所示的有向图G_1可表示为

$G_1 = (V_1, E_1)$　　$V_1 = \{1, 2, 3\}$　　$E_1 = \{<1,2>,<2,1>,<2,3>\}$

无向图

若E是无向边(简称边)的有限集合时，则图G为无向图。边是顶点的无序对，记为(v,　w)或(w,v),因为(v,w)=(w,v),其中v,w是顶点。可以说顶点w和顶点v互为邻接点。边(v，w)依附于顶点w和v，或者说边(v，w)和顶点v，w相关联。



(b) 无向图G_2

图(b)所示的无向图G2可表示为

$G_2=(V_2,E_2)$
$V_2 =\{1,2,3,4\}$
$E_2 =\{(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)\}$

简单图

满足条件：

1.不存在重复边

2.不存在顶点到自身的边

多重图

若图 G 中某两个结点之间的边数多于一条，又允许顶点通过同一条边和自己关联，则 G 为多重图。多重图的定义和简单图是相对的。

完全图

对于无向图，| E | 的取值范围是 0 到 n (n − 1) / 2 ，有 n (n − 1) / 2 条边的无向图称为完全图，在完全图中任意两个顶点之间都存在边。对于有向图，| E |的取值范围是 0 到n (n − 1) ，有n (n − 1) 条弧的有向图称为有向完全图，在有向完全图中任意两个顶点之间都存在方向相反的两条弧。

图的存储

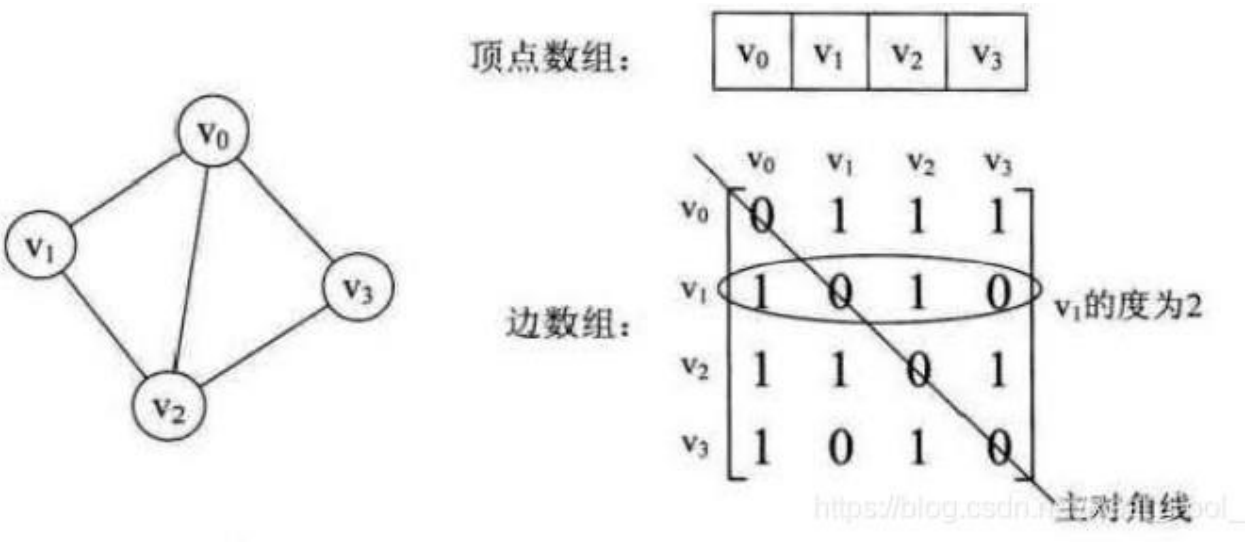
领接矩阵

图的邻接矩阵(Adjacency Matrix) 存储方式是用两个数组来表示图。一个一维数组存储图中顶点信息，一个二维数组(称为邻接矩阵)存储图中的边或弧的信息。

设图 G 有 n 个顶点，则邻接矩阵 A 是一个 n * n 的方阵，定义为：

$$A[i][j]=\begin{cases}1, & 若(v_i,v_j)或<v_i,v_j>是E(G)中的边\\ 0, & 若(v_i,v_j)或<v_i,v_j>不是E(G)中的边\end{cases}$$

无向图和它的领接矩阵



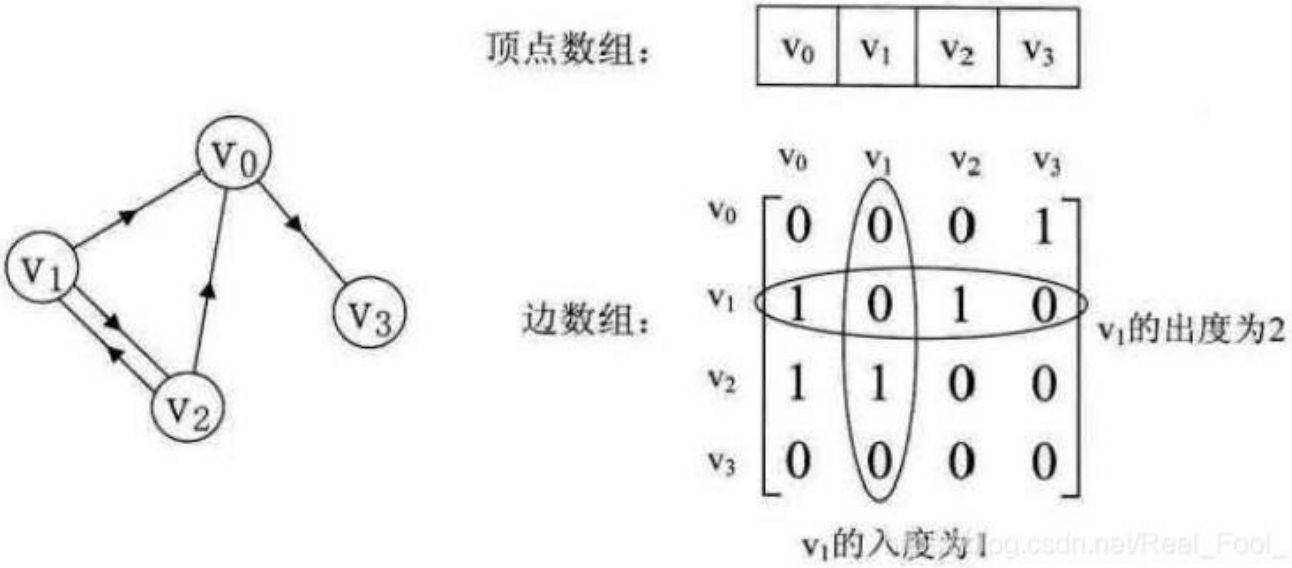
可以看出：

1.无向图的邻接矩阵一定是一个对称矩阵(即从矩阵的左上角到右下角的主对角线为轴，右上角的元与左下角相对应的元全都是相等的)。 因此，在实际存储邻接矩阵时只需存储上(或下)三角矩阵的元素。

2.对于无向图，邻接矩阵的第 i 行(或第 i 列)非零元素(或非 ∞ 元素)的个数正好是第 i 个顶点的度T D (v_i) 。比如顶点 v_1 的度就是1 + 0 + 1 + 0 = 2 。

3.求顶点 v_i 的所有邻接点就是将矩阵中第i行元素扫描一遍， A [i] [j] 为 1就是邻接点。

有向图和它的邻接矩阵：



可以看出:

1.主对角线上数值依然为0。但因为是有向图，所以此矩阵并不对称。

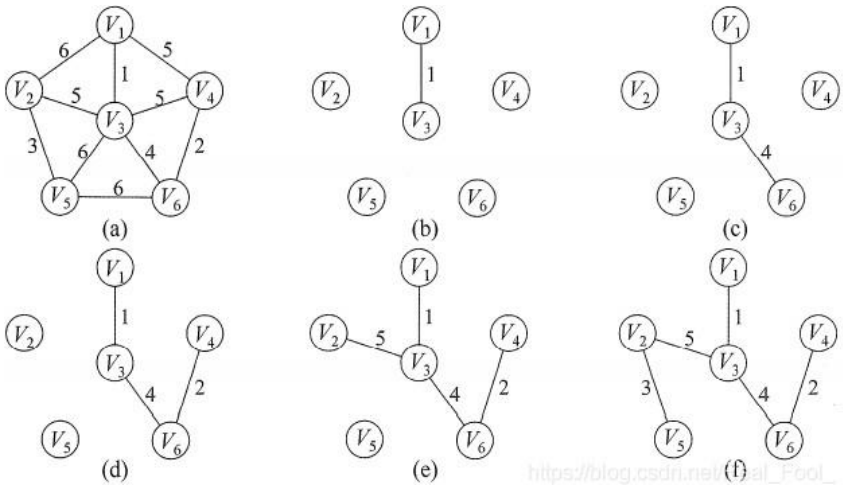
2.有向图讲究入度与出度，顶点 v_1 的入度为1,正好是第 v_1 列各数之和。顶点 v_1 的出度为2，即第 v_1 行的各数之和。

3.与无向图同样的办法，判断顶点 v_i 到 v_j 是否存在弧，只需要查找矩阵中A [i] [j] 是否为1即可。

最小生成树

Prim算法

思路：Prim算法基于贪心，我们每次总是选出一个离生成树距离最小的点去加入生成树，最后实现最小生成树。

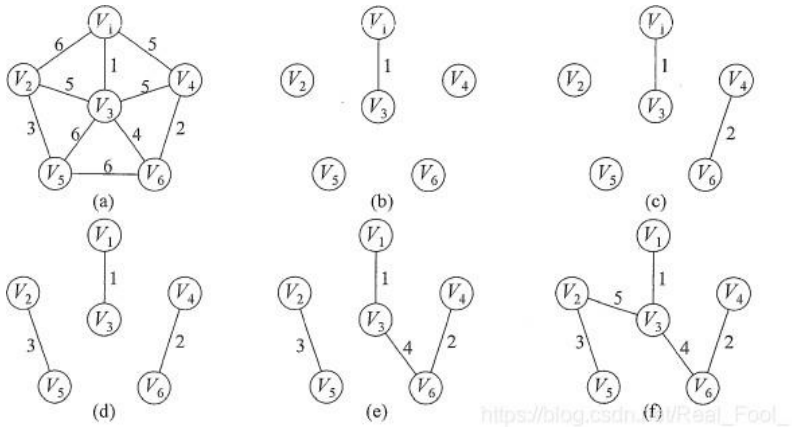



```
public static int Prim(){
    ArrayList<Integer> listU = new ArrayList<>();//已存入结点
    ArrayList<Integer> listV = new ArrayList<>();//未存入结点
    listU.add(0);
    for(int i=1;i<n;i++){
    listV.add(i);7
    }
    double min;
    int form = -1;//起点
```

```
    int to = -1;//终点
    double sum = 0;
    while(!listV.isEmpty()) {
        min = Double.MAX_VALUE;
        for(int i:listU) {
            for(int j:listV) {
                if(edge[i][j]!=0 && edge[i][j]<min) {
                    min = edge[i][j];
                    form = i;
                    to = j;
                }
            }
        }
        if(min==Double.MAX_VALUE) {
            System.out.println("-1");
            return;
        }
        listU.add(to);
        listV.remove(new Integer(to));
        sum = sum + edge[form][to];
    }
    return sum;
}
```

Kruskal算法

思路：将图的存储结构使用边集数组的形式表示，并将边集数组按权值从小到大排序，遍历边集数组，每次选取一条边并判断是否构成环路，不会构成环路则将其加入最小生成树，最终只会包含 n-1 条边（ n 为无向图的顶点数)。




```
import java.util.ArrayList;
import java.util.Collections;
//...数据读入包省略
public class Main{
    public void static main(String[] args) {
        //...数据读入省略
        Collections.sort(list);
        for(int i=0;i<list.size();i++) {
            Edge e = list.get(i);
            if(find(e.x)!=find(e.y)) {
                merge(e.x,e.y);
                num++;
                ans = ans + e.w;
                if(num==n-1) {
                    System.out.println(ans);//最小生成树边长之和
                    return;
                }
```

```
            }
        }
    }
    static int find(int x)
        {if(f[x] == x)
            return f[x];
        f[x] = find(f[x]);
        return f[x];
    }
    static void merge(int x,int y)
        {int xx = find(x);
        int yy = find(y);
        if(xx!=yy)
            f[yy] = xx;
    }
}
class Edge implements
    Comparable<Edge>{int x;//起点
    int y;//终点
    int w;//边长
    public Edge(int x, int y, int w)
        {super();
        this.x = x;
        this.y = y;
        this.w = w;
    }
    @Override
    public int compareTo(Edge o) {
        // TODO Auto-generated method stub
        return this.w>o.w?1:-1;
    }
}
```

例题：聪明的猴子

题目链接：聪明的猴子 - 蓝桥云课 (lanqiao.cn)

题目描述

在一个热带雨林中生存着一群猴子，它们以树上的果子为生。昨天下了一场大雨，现在雨过天晴，但整个雨林的地表还是被大水淹没着，部分植物的树冠露在水面上。猴子不会游泳，但跳跃能力比较强，它们仍然可以在露出水面的不同树冠上来回穿梭，以找到喜欢吃的果实。

现在，在这个地区露出水面的有 N 棵树，假设每棵树本身的直径都很小，可以忽略不计。我们在这块区域上建立直角坐标系，则每一棵树的位置由其所对应的坐标表示(任意两棵树的坐标都不相同)。

在这个地区住着的猴子有 M 个，下雨时，它们都躲到了茂密高大的树冠中，没有被大水冲走。由于各个猴子的年龄不同、身体素质不同，它们跳跃的能力不同。有的猴子跳跃的距离比较远(当然也可以跳到较近的树上)，而有些猴子跳跃的距离就比较近。这些猴子非常聪明，它们通过目测就可以准确地判断出自己能否跳到对面的树上。

现已知猴子的数量及每一个猴子的最大跳跃距离，还知道露出水面的每一棵树的坐标，你的任务是统计有多少个猴子可以在这个地区露出水面的所有树冠上觅食。

kruskal解法思路：

1.在构造最小生成树的过程中，找到所用的最长的一条边；

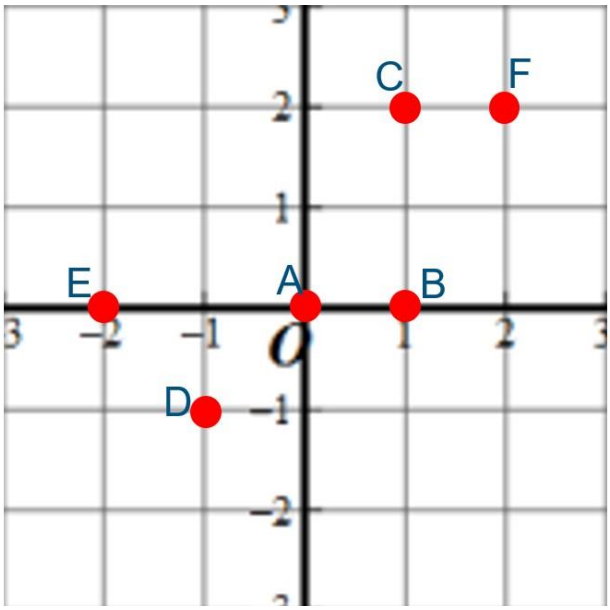
2.判断有几只猴子的跳跃距离>=最小生成树最长边；

样例分析：

```
4
1 2 3 4
6
0 0
1 0
1 2
-1 -1
-2 0
2 2
```

1.求出任意两棵树之间的距离，并且按照距离的大小进行排序，完成 f 数组初始化，得到；

A B 1.0
C F 1.0
A D 1.4142135623730951
D E 1.4142135623730951
A E 2.0
B C 2.0
A C 2.23606797749979
B D 2.23606797749979
B F 2.23606797749979
A F 2.8284271247461903
B E 3.0
C D 3.605551275463989
C E 3.605551275463989
D F 4.242640687119285
E F 4.47213595499958



f [1] = 1
f [2] = 2
f [3] = 3
f [4] = 4
f [5] = 5
f [6] = 6

2.接下来，从最短距离开始，如果此路的起点和终点在一个整体中，便不对此边进行操作，否则，选择此条 边，并更改 f 数组中的内容；

```java
import java.io.BufferedReader;
import java.io.IOException;
import  java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.util.ArrayList;
import  java.util.Collections;

public class Main {
    static int ans = 0;
    static int n;
    static int[] a;
    static int m;
    static int[] x;
    static int[] y;
    static double max;
    static int[] f;
    static int num = 0;
    static BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    static PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
    public static void main(String[] args) throws IOException{
        m  =  Integer.parseInt(in.readLine().trim());//猴子的数量
        a = new int[m];//每一只猴子的最大跳跃距离
        String[] s = in.readLine().trim().split(" ");
        for(int i=0;i<m;i++)
            a[i] = Integer.parseInt(s[i]);
```

```
        n  =  Integer.parseInt(in.readLine().trim());//树的数量
        x = new int[n];//每一棵树的横坐标
        y = new int[n];//每一棵树的纵坐标
        for(int i=0;i<n;i++) {
            s = in.readLine().trim().split(" ");
            x[i] = Integer.parseInt(s[0]);
            y[i] = Integer.parseInt(s[1]);
        }
        ArrayList<Edge> list = new ArrayList<>();//存储Edge，获取顺序就是按照边长从小到大获取
        for(int i=0;i<n-1;i++) {//起点
            for(int j=i+1;j<n;j++) {//终点
                double l = Math.sqrt((x[i]-x[j])*(x[i]-x[j])+(y[i]-y[j])*(y[i]-y[j]));
                Edge edge = new Edge(i, j, l);
                list.add(edge);
            }
        }
        Collections.sort(list);
        f = new int[n];
        for(int i=0;i<n;i++)
            f[i] = i;
        for(int i=0;i<list.size();i++)
            {Edge e = list.get(i);
            if(find(e.x)!=find(e.y)) {
                merge(e.x,e.y);
                max = Math.max(max, e.w);
                num++;//用了多少条边
                if(num==n-1)//最小生成树已经完成
                    break;
            }
        }
        for(int i=0;i<m;i++)
            if(a[i]>=max)
                ans++;
        System.out.println(ans);
    }
    static int find(int x) {//连接过的那一个点的最上一层
        if(f[x] != x)
            f[x] = find(f[x]);
        return f[x];
    }
    static void merge(int x,int y) {//把两个点之间建立一个关系，表示这两个是一个整体
        int xx = find(x);
        int yy = find(y);
        f[yy] = xx;
    }
}
class Edge implements Comparable<Edge>{//起点、终点、边长
    int x;
    int y;
    double w;
    public Edge(int x, int y, double w) {
        super();
        this.x = x;
        this.y = y;
    this.w = w;82
    }
    @Override
    public int compareTo(Edge o) {//排序时是根据边的长度，从小到大排序
    return  this.w>o.w?1:-1;86
    }
}
```

最短路

Disjkstra算法（源点至其余点的最短距离）

思路：

1.用一个 dis 数组存储结点，用一个数组 visit 存储中转点；

2.每次从源点开始选择一个距离源点最近并且还未使用过的结点 u ，判断通过 u 能否中转到 v ，使 dist[v]更小。即 dis[v] = Math.min(dis[v] , dis[u] + g[u][v]);

```
public static int[] Disjkstra(int vn,int[][] g){//结点数，两节点间距离
    int[] dis = new int[vn];
    int[] vis = new int[vn];
    for(int i=0;i<vn;i++)
        dis[i] = g[0][i];//初始化距离数组，0为源点
    vis[0] = 1;//标记已经使用过了
    int minindex = 0;//记录当前从v0到哪一个点距离最小
    for(int i=0;i<vn-1;i++){
        int min = Integer.MAX_VALUE;
        for(int j=0;j<vn;j++){
            if(vis[j]==0 && dis[j]<min){
                min = dis[j];//记录最小的距离
                minindex = j;//记录最小的结点编号
            }
        }
        vis[minindex] = 1;
        for(int j=0;j<vn;j++){
            if(g[minindex][j]<Integer.MAX_VALUE && minindex!=j)
                dis[j] = Math.min(dis[j],dis[minindex]+g[minindex][j]);
        }
    }
    return dis;
}
```

优化：

1.已知Dijkstra算法每次在集合B中寻找能够加入集合A的顶点的时候，都要将符合条件的边进行遍历，当边较多的时候，该循环会带来较大的时间复杂度。

2.这里可以维护一个优先队列，队列中存放的是点(额外定义的一种数据类型)，该点离顶点距离越小优先级越高。这样每次取队首元素，就可以得到当前状态下距离起点距离最短的点，并让其出队。若该点已在集合A中，则继续出队，否则将其加入集合A，并以该点为中介点进行松弛操作。(若此点i不在集合A中，则 dis[i] 表示起点到点i的距离或起点到点 i 经过松弛之后的距离，若点i不在集合A中且其位于队首位置，那么它到集合A中某点的距离一定是最小的，这样便可保证我们取队首元素就能取到算法中描述的“最短边”如此一来便可以大大减小算法的时间复杂度。

```
import java.util.*;

public class Main {

    public static void main(String[] args)
        {Scanner cin = new
        Scanner(System.in);while (true) {
            int n = cin.nextInt(), m = cin.nextInt();
            if(n==0&m==0)break;
            List<List<Edge>> vec = new ArrayList<>();
            for (int i = 0; i < n + 5; i++) {
```

```
                vec.add(new ArrayList<>());//点
            }
            int dis[] = new int[n + 5];
            boolean vis[] = new boolean[n + 5];
            for (int i = 0; i < m; i++) {
                int a = cin.nextInt(), b = cin.nextInt(), c = cin.nextInt();//起点，终点，路长
                vec.get(a).add(new Edge(b, c));
                vec.get(b).add(new Edge(a,c));
            }
            for (int i = 0; i <= n; i++) {
                dis[i] = Integer.MAX_VALUE;
                vis[i] = false;
            }
            dis[1] = 0;
            PriorityQueue<Node> queue = new PriorityQueue<>();
            queue.add(new Node(1, 0));
            while (!queue.isEmpty()) {
                Node now = queue.poll();
                if (vis[now.point]) {
                    continue;
                }
                vis[now.point] = true;
                for (int i = 0; i < vec.get(now.point).size(); i++) {
                    Edge nxt = vec.get(now.point).get(i);
                    if (nxt.val + dis[now.point] < dis[nxt.to]) {
                        dis[nxt.to] = nxt.val + dis[now.point];
                        queue.add(new Node(nxt.to, dis[nxt.to]));
                    }
                }
            }
            System.out.println(dis[n]);
        }
    }
}
class Edge{
    int to,val;
    public Edge(int to,int val){
        this.to=to;
        this.val=val;
    }
}
class Node implements Comparable<Node>{
    int point,val;
    public Node(int point,int val){
        this.val=val;
        this.point=point;
    }
    @Override
    public int compareTo(Node node){
        return  val-node.val;
    }
}
```

Floyd算法（任意两点间的最短距离）

思路：以每一个点为中转结点，优化任意两点之间的距离；

```
public static void Floyd(){
    for(int  k=0;k<vn;k++){//中转点
        for(int i=0;i<vn;i++){//起点
            for(int j=0;j<vn;j++){//终点
                if(i!=j && j!=k && i!=k && g[i][k]!=Integer.MAX_VALUE && g[k]
[j]!=Integer.MAX_VALUE)
                    g[i][j]  =  Math.min(g[i][j],g[i][k]+g[k][j]);
            }
        }
    }
}
```

第十章 并查集

并查集 - 芯笑 (lamper.top)

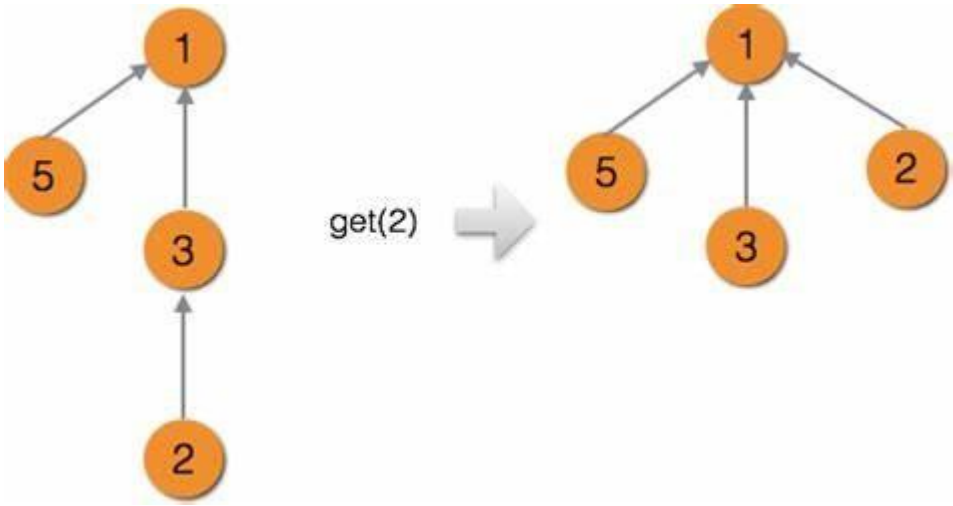
普通并查集（朋友的朋友是朋友）

思路：

1.用一个数组 f 表示 i 的上一层是 f[i]，初始情况下 f[i] = i 全部满足；

2.每当两个点连接时，都可以将其中一个点的 f 值修改为另一个点的 f 值，表示此两点的上一层相同——两个点为同一整体；

3.每次连接时，必须将 a 的最上层 f 值和 b 的最上层 f 值连接，这样 a 、b 以及 a、b 过程中连接过得所有点的 f 值都相同，他们都在同一个整体中；




```
public static int find(int x){
    if(f[x] ! = x)
        f[x]  =  find(f[x]);//将过程中的所有点的f值全部赋值为顶点的f值
return f[x];5
}
```

```java
public static void merge(int x,int y){
    int xx = find(x);
    int yy = find(y);
    if(xx!=yy)
        f[yy] = xx;
}
```

例题：蓝桥幼儿园

题目链接：蓝桥幼儿园 - 蓝桥云课 (lanqiao.cn)

题目描述

蓝桥幼儿园的学生是如此的天真无邪，以至于对他们来说，朋友的朋友就是自己的朋友。

小明是蓝桥幼儿园的老师，这天他决定为学生们举办一个交友活动，活动规则如下：

小明会用红绳连接两名学生，被连中的两个学生将成为朋友。

小明想让所有学生都互相成为朋友，但是蓝桥幼儿园的学生实在太多了，他无法用肉眼判断某两个学生是否为朋友。于是他起来了作为编程大师的你，请你帮忙写程序判断某两个学生是否为朋友（默认自己和自己也是朋友）。

输入描述

第 1 行包含两个正整数 N, M，其中 N 表示蓝桥幼儿园的学生数量，学生的编号分别为 $1 \sim N$。

之后的第 $2 \sim M+1$ 行每行输入三个整数，op, x, y：

- 如果 $op = 1$，表示小明用红绳连接了学生 x 和学生 y。
- 如果 $op = 2$，请你回答小明学生 x 和 学生 y 是否为朋友。

$1 \leq N, M \leq 2 \times 10^5$，$1 \leq x, y \leq N$。

思路：用 f 数组表示关系，如果 op 为 1，便进行 merge 操作，如果 op 为 2，判断两个学生的 f 值是否相同。

```java
import java.io.BufferedReader;
import java.io.IOException;
import  java.io.InputStreamReader;
import  java.io.OutputStreamWriter;
import java.io.PrintWriter;

public class Main {
    static BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    static PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
    static int[] p;//p[i]表示i的头结点是p[i]
    public static void main(String[] args) throws
        IOException{String[] s = in.readLine().split(" ");
        int n = Integer.parseInt(s[0]);
        int m = Integer.parseInt(s[1]);
        p = new int[n+1];
        for(int i=1;i<=n;i++) {
            p[i] = i;//第i个小朋友的根节点
        }
        for(int i=0;i<m;i++) {
            s = in.readLine().split(" ");
            int op = Integer.parseInt(s[0]);
            int x = Integer.parseInt(s[1]);
            int y = Integer.parseInt(s[2]);
            if(op==1)
                merge(x,y);
            else
                System.out.println(find(x)==find(y)?"YES":"NO");
```

```
        }
    }
    static int find(int x) {
        if(p[x]!=x)
            p[x]=find(p[x]);
        return p[x];
    }
    static void merge(int x,int y){
        int xx = find(x);
        int yy = find(y);
        if(xx != yy)
            f[yy] = xx;
    }
}

```

种类并查集（敌人的敌人是朋友）

思路：

1.在普通并查集的基础上进行完善，如果 a 和 b 是敌人，b 和 c 是敌人，那么 a 和 c 是朋友，这时可以表示为：a 和 !b 是朋友，b和 !a 是朋友，c 和 !b 是朋友，b 和 !c 是朋友；

2.数组 f 的大小需要为普通并查集的二倍，前半部分表示 i 的 f[i] 值，后半部分表示 !i 的 f[i] 值；

例题：蓝桥侦探

题目链接：蓝桥侦探 - 蓝桥云课 (lanqiao.cn)

题目描述

小明是蓝桥王国的侦探。

这天，他接收到一个任务，任务的名字叫分辨是非，具体如下：

蓝桥皇宫的国宝被人偷了，犯罪嫌疑人锁定在 N 个大臣之中，他们的编号分别为 $1 \sim N$。

在案发时这 N 个大臣要么在大厅1，要么在大厅2，但具体在哪个大厅他们也不记得了。

审讯完他们之后，小明把他们的提供的信息按顺序记了下来，一共 M 条，形式如下：

- x y，表示大臣 x 提供的信息，信息内容为：案发时他和大臣 y 不在一个大厅。

小明喜欢按顺序读信息，他会根据信息内容尽可能对案发时大臣的位置进行编排。

他推理得出第一个与先前信息产生矛盾的信息提出者就是偷窃者，但推理的过程已经耗费了他全部的脑力，他筋疲力尽的睡了过去。作为他的侦探助手，请你帮助他找出偷窃者！

思路：

1.种类并查集，如果 a 和 b 不在一起， b 和 c 不在一起，那么 a 和 c 一定在一起，否则，有人说谎；

2.题目输入 x，y 表示 x 和 y 不在一起，那么便可以表示为：x 和 !y 在一起，y 和 !x 在一起，两个内容都进行 merge 操作；

```
import java.io.BufferedReader;
import java.io.IOException;
import  java.io.InputStreamReader;

public class Main {
    static BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
```

```
    static int[] f;
    public static void main(String[] args) throws IOException{
        String[] s = in.readLine().split(" ");
        int n = Integer.parseInt(s[0]);
        int m = Integer.parseInt(s[1]);
        f = new int[2*n+2];
        for(int i=0;i<2*n+2;i++) {
            f[i] = i;
        }
        for(int i=0;i<m;i++) {
            s = in.readLine().split(" ");
            int x = Integer.parseInt(s[0]);
            int y = Integer.parseInt(s[1]);
            if(find(x)!=find(y)) {
                merge(x, y+n);
                merge(x+n, y);
            }else {
                System.out.println(x);
                return;
            }
        }
    }
    static int find(int x) {
        if(f[x]!=x)//夫点不是自己
            f[x]=find(f[x]);
        return f[x];
    }
    static void merge(int x,int y) {
        x = find(x);
        y = find(y);
        f[y] = x;
    }
}
```

带全并查集

思路：在普通并查集的基础上还需要记录长度。

例题：蓝桥部队

题目链接：蓝桥部队 - 蓝桥云课 (lanqiao.cn)

题目描述

小明是蓝桥部队的长官，他的班上有 N 名军人和 1 名军师。

这天，N 名军人在操场上站成一排，起初编号为 i 的军人站在第 i 列。

作为长官，小明可以对军人和军师下达 M 条命令，命令有两种类型，格式如下：

- `1 x y`，让军人 x 所在列的所有人作为一个整体移动到和军人 y 所在列的后面，使两列合并为一列。
- `2 x y`，询问军师军人 x 和军人 y 是否站在同一列。若是，则军师要回应小明 x, y 之间有多少人，否则军师要回应 -1。

你就是小明的军师，请你回答小明的每个询问。

```
import java.io.BufferedReader;
import java.io.IOException;
```

```java
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import  java.io.PrintWriter;

public class Main {
    static BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    static PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
    static int[] f;//存放此列的排头元素
    static int[] length;//存放此列的军人数量
    static int[] num;//此军人前面有多少人
    public static void main(String[] args) throws IOException{
        String[] s = in.readLine().split(" ");
        int n = Integer.parseInt(s[0]);
        int m = Integer.parseInt(s[1]);
        f = new int[n+1];
        length = new int[n+1];
        num = new int[n+1];
        for(int i=1;i<=n;i++) {
            f[i] = i;
            length[i] = 1;
        }
        for(int i=0;i<m;i++) {
            s = in.readLine().split(" ");
            int op = Integer.parseInt(s[0]);
            int x = Integer.parseInt(s[1]);
            int y = Integer.parseInt(s[2]);
            if(op==1) {//合并x和y所在列 x到y后面
                merge(x, y);
            }else {//求x和y间隔
                if(find(x)!=find(y)) {
                    System.out.println(-1);
                }else {
                    System.out.println(Math.abs(num[x]-num[y])-1);
                }
            }
        }
    }
    static void merge(int x,int y) {
        x = find(x);
        y = find(y);
        if(x==y)
            return;
        num[x] = num[x] + length[y];
        length[y] = length[y] + length[x];
        f[x] = y;
    }
    static int find(int x) {
        if(x!=f[x]) {
            int root = find(f[x]);
            num[x] = num[x] + num[f[x]];
            return f[x] = root;
        }
        return f[x];
    }
}
```

第十一章 动态规划

最大连续子段和

思路：dp[i] 表示以 a[i]结尾的最大连续子段和，那么dp[i] = Math.max(dp[i - 1] + a[i] , a[i])，如果 dp[i - 1] 是小于0的，那么最大值就是 a[i]，否则，最大值就是 dp[i - 1] + a[i]，最终结果就是 dp 数组中的最大值。

```
public static int len(int[] a){
    int[] dp = new int[a.length];
    dp[0] = Math.max(0,a[0]);
    int max = Integer.MIN_VALUE;
    for(int i=1;i<a.length;i++){
        dp[i] = Math.max(dp[i-1]+a[i],a[i]);
        if(max<dp[i])
            max = dp[i];
    }
    return max;
}
```

LCS 最长公共子序列

思路：

1.用一个数组 dp[i][j] 表示 S 字符串中前 i 个字符与 T 字符串中前 j 个字符的最长上升子序列，那么 dp[i+1][j+1] 就是S 字符串中前 i+1 个字符与 T 字符串中前 j+1 个字符的最长上升子序列；

2.如果此时 S 中的第 i+1 个字符与 T 中的第 j+1 个字符相同，那么 dp[i+1][j+1] = dp[i][j] + 1;

3.如果此时 S 中的第 i+1 个字符与 T 中的第 j+1 个字符不同，那么 dp[i+1][j+1] = Math.max (dp[i+1][j] , dp[i][j+1]);

```
public static int LCS(String s,String t){
    int[][] dp=new int[s.length()+1][t.length()+1];
    for(int i=1;i<=s.length();i++){
        for(int j=1;j<=t.length();j++){
            if(s.charAt(i-1)==t.charAt(j-1))
                dp[i][j]=dp[i-1][j-1]+1;
            else
                dp[i][j]=Math.max(dp[i][j-1],dp[i-1][j]);
        }
    }
    return dp[len1][len2];
}
```

LCS最长公共子序列及其个数

思路：

1.在最长公共子序列的基础上稍作修改即可；

2.如果 s[i] == t[j]，此时公共子序列的长度增加1，其子序列个数至少与 num[i-1][j-1] 的个数相同，所以个数不变；

3.如果 dp[i][j] == dp[i][j-1] (即两者的长度相同,但内容不同或顺序不同) 则 num[i][j] += num[i][j-1];

4.如果 dp[i][j] == dp[i-1][j], 则 num[i][j] += num[i-1][j];

5.如果 dp[i][j] == dp[i-1][j-1], 则 num[i][j] -= num[i-1][j-1]； （因为如果 3 和 4 同时触发则会出现加了两次 num[i-1][j-1] 的情况，所以要剪掉）；

```
public static void LCS(String a,String b){
    char[] s = s.toCharArray();
    char[] t = b.toCharArray();
    int[][] dp = new int[s.length+1][t.length+1];
    int[][] num = new int[s.length+1][t.length+1];
    for(int i=0;i<s.length;i++) num[i][0] = 1;
    for(int i=0;i<t.length;i++) num[0][i] = 1;
    for(int i=1;i<=s.length;i++) {
        for(int j=1;j<=t.length;j++) {
            if(s[i-1]==t[j-1]) {
                dp[i][j] = dp[i-1][j-1] + 1;
                num[i][j] = num[i-1][j-1];
            }else {
                dp[i][j] = Math.max(dp[i-1][j], dp[i][j-1]);
            }
            if(dp[i][j]==dp[i][j-1])
                num[i][j] = num[i][j] + num[i][j-1];
            if(dp[i][j]==dp[i-1][j])
                num[i][j] = num[i][j] + num[i-1][j];
            if(dp[i][j]==dp[i-1][j-1])
                num[i][j] = num[i][j] - num[i-1][j-1];
            num[i][j] %= N;
        }
    }
}
```

LIS最长上升子序列

动态规划方法 O(n^2):

思路：用 dp 数组表示以 i 结尾的最长上升子序列，那么 dp[i] 的结果就为：前 i 个比 a[i] 小的数中 dp 数组中的最大值 + 1， dp 数组中的最大值，便是此序列的最长上升子序列长度；

```
public static int LIS(int[] a){
    int[] dp = new int[a.length+1];
    dp[0] = 0;
    int num  = 0;
    for(int i=1;i<a.length;i++) {
        int max = 0;//前面的最大dp值
        for(int j=1;j<=i;j++) {//判断前面的数
        if(a[i]>a[j]) {//保证上升
                if(dp[j]>max)
                    max  =  dp[j];//在保证上升的前提之下，找到最大值
            }
```

```
        }
        dp[i] = max+1;//长度加上i本身
        num = Math.max(max, dp[i]);
    }
    return num;
}
```

贪心方法 O(nlog$_2$n)：

思路：用一个数组 f 表示上升子序列的长度为 i 时的末尾元素最小值，用二分发查找 a[i] 可以放入 f 数组中的位置，并将其放入，记录 f 数组用到的长度即为最长上升子序列的长度，数组 f 的内容便是最长上升子序列的一种形式（大小最小）；

```
public static int number(int[] a){
    int[] f = new int[a.length];//长度为i的上升子序列的末尾元素最小值为f[i]
    int length = 0;
    for(int i=0;i<n;i++) {
        int l = 0;
        int r = length;
        while(l<r) {
            int mid = (l+r+1)>>1;
            if(f[mid]>=a[i])
                r = mid - 1;
            else
                l = mid;
        }
        length = Math.max(length, r+1);
        f[r+1] = a[i];
    }
    return length;
}
```

最大连续子矩阵和

思路：

1.将矩阵中的每一个元素的值都转变为它同一列前 i - 1 行的元素和，那么便可以通过前缀和数组获得到任意上下区间中的元素和，也就变为了一维问题；

2.求这一个上下区间内的最大连续子段和，也就是找上下区间固定的所有子矩阵的最大连续和；

```
public static int subMaxMatrix(int[][] g){
    for(int i=1;i<=g.length;i++)
        for(int  j=1;j<=g[i].length;j++)
            g[i][j] += g[i-1][j];//前缀和初始化
    int ans = Integer.MIN_VALUE;
    for(int  start=1;start<=g.length;start++){
        for(int  end=1;end<=g[i].length;end++){//枚举start行~end行
            int dpi = 0;
            for(int col = 1;col<=g[i].length;col++){
                int ai = g[end][col] - g[start-1][col];
                dpi = Math.max(dpi+ai,ai);
            ans = Math.max(dpi,ans);13}
```

```
        }
    }
    return ans;
}
```

背包专题

01背包

题目链接：小明的背包1 - 蓝桥云课 (lanqiao.cn)

题目描述

小明有一个容量为 V 的背包。

这天他去商场购物，商场一共有 N 件物品，第 i 件物品的体积为 w_i，价值为 v_i。

小明想知道在购买的物品总体积不超过 V 的情况下所能获得的最大价值为多少，请你帮他算算。

思路：

1.用一个二维数组 dp 表示考虑前 i 件物品，在背包容量为 j 的时候能取得的最大价值，当 i 或者 j 为 0 的时候，dp 值为0；

2.每一件物品都可以有两个情况——选 or 不选，如果不选此件物品，那么 dp 的结果与考虑前 i - 1件物品在相同背包容量时能取得的最大价值相同，如果选择，那么 dp 的结果为这件物品能带来的价值 + 考虑前 i - 1 件物品，在背包容量为（放入这件物品后剩余大小）时能取到的最大价值，两个取最大值；

3.状态转移方程为：dp[i][j] = Math.max(dp[i-1][j] , v[i] + dp[i-1][j-w[i]]);

4.最终结果即为 dp[n][m]

```java
import java.util.Scanner;

public class Main {
    public static void main(String[] args)
        {Scanner sc =  new
        Scanner(System.in);int n =
        sc.nextInt();//商品数量
        int m = sc.nextInt();//背包的容量
        int[] w = new int[n+1];
        int[] v = new int[n+1];
        for(int i=1;i<=n;i++) {
            w[i] = sc.nextInt();//体积
            v[i] = sc.nextInt();//价值
        }
        int[][] dp = new int[n+1][m+1];//1 ~ n
        for(int i=0;i<=n;i++) {
            for(int j=0;j<=m;j++)
                {if(i==0 || j==0)
                    dp[i][j] = 0;
                else {
                    if(j<w[i]) {
```

```
                    }else {//j>=w[i]
                        dp[i][j] = Math.max(dp[i-1][j], v[i]+dp[i-1][j-w[i]]);
                    }
                }
            }
        }
        System.out.println(dp[n][m]);
    }
}
```

优化（二维转一维）

思路：

1.在二维数组赋值的过程中，dp[i][j] 的值只与第 i - 1 层的 dp 值有关，便可以用一维 dp 数组表示每一层在背包容量为 j 时取得的最大价值；

2.在内层循环进行处理时，会用到 dp[i-1][j-w[i]]的值，但是因为内层循环是从小到大的，所以在用到 dp[i-1][j-w[i]] 的值时，dp[j - w[i]] 已经表示为第 i 层的值，而不是第 i - 1 层的值，所以内层循环需要从大到小执行；

```
import java.util.Scanner;

public class Main {
    public static void main(String[] args)
        {     Scanner      sc      =      new
        Scanner(System.in);    int    n    =
        sc.nextInt();
        int m = sc.nextInt();
        int[] w = new int[n+1];
        int[] v = new int[n+1];
        for(int i=1;i<=n;i++){
            w[i] = sc.nextInt();
            v[i] = sc.nextInt();
        }
        int[] dp = new int[m+1];
        for(int i=0;i<n+1;i++){
            for(int j=m;j>=w[i];j--){
                dp[j] = Math.max(dp[j],dp[j-w[i]]+v[i]);
            }
        }
        System.out.println(dp[m]);
    }
```

完全背包

题目链接：小明的背包2 - 蓝桥云课 (lanqiao.cn)

题目描述

小明有一个容量为 V 的背包。

这天他去商场购物，商场一共有 N 种物品，第 i 种物品的体积为 w_i，价值为 v_i，每种物品都有无限多个。

小明想知道在购买的物品总体积不超过 V 的情况下所能获得的最大价值为多少，请你帮他算算。

思路：

1.基本原理与01背包相同；

2.每一件物品都有无限个，其实并不是无限个，因为背包的容量是有限的，即每一件物品的最多数量为 j / w[i];

3.放入第 i 件商品后，仍然可以放第 i 件物品，那么剩余体积能带来的最大价值为 dp[i][j-w[i]] ；

4.状态转移方程为：dp[i][j] = Math.max(dp[i-1][j] , dp[i]\ [j-w[i]] + p[i]);

```java
import java.util.Scanner;

public class Main {
    public static void main(String[] args)
        { Scanner scan = new
        Scanner(System.in);int n =
        scan.nextInt();
        int v = scan.nextInt();
        int dp[][] = new int[n+1][v+1];
        int w[] = new int[n+1];
        int p[] = new int[n+1];
        for(int i = 1 ; i <=n;i++) {
            w[i] = scan.nextInt();
            p[i] = scan.nextInt();
        }
        for(int i =1; i <= n; i++)
            { for(int j =1; j <=v; j++)
            {
                }else {
                if(w[i]<=j) {
                    dp[i][j]=dp[i-1][j];
                }
            }
        }
        System.out.println(dp[n][v]);
    }
}
```

多重背包

题目链接：小明的背包3 - 蓝桥云课 (lanqiao.cn)

题目描述

小明有一个容量为 V 的背包。

这天他去商场购物，商场一共有 N 种物品，第 i 种物品的体积为 w_i，价值为 v_i，数量为 s_i。

小明想知道在购买的物品总体积不超过 V 的情况下所能获得的最大价值为多少，请你帮他算算。

思路：

1.在01背包的基础上增加了每一件物品的数量，并且数量是指定的；

2.多一层循环判断第 i 件物品放多少个，并且不能超过背包容量 j ，如果放入k = 0 件，即表示不放；

```java
import java.util.*;

public class Main{
    static int n,v;
    static int[] w=new int[1010];
    static int[] c=new int[1010];
```

```
    static int[] s=new int[1010];
    static int[][] dp=new int[2020][2020];
    public static void main(String[] args) {
        Scanner sc=new Scanner(System.in);
        n=sc.nextInt();
        v=sc.nextInt();
        for(int i=1;i<=n;i++) {
            w[i]=sc.nextInt();
            c[i]=sc.nextInt();
            s[i]=sc.nextInt();
        }
        for(int i=1;i<=n;i++) {
            for(int j=0;j<=v;j++) {
                for(int k=0;k<=s[i]&&k*w[i]<=j;k++) {
                    dp[i][j]=Math.max(dp[i][j],  dp[i-1][j-k*w[i]]+k*c[i]);
                }
            }
        }
        System.out.println(dp[n][v]);
    }
}
```

第十二章 难点攻克

区间素数筛

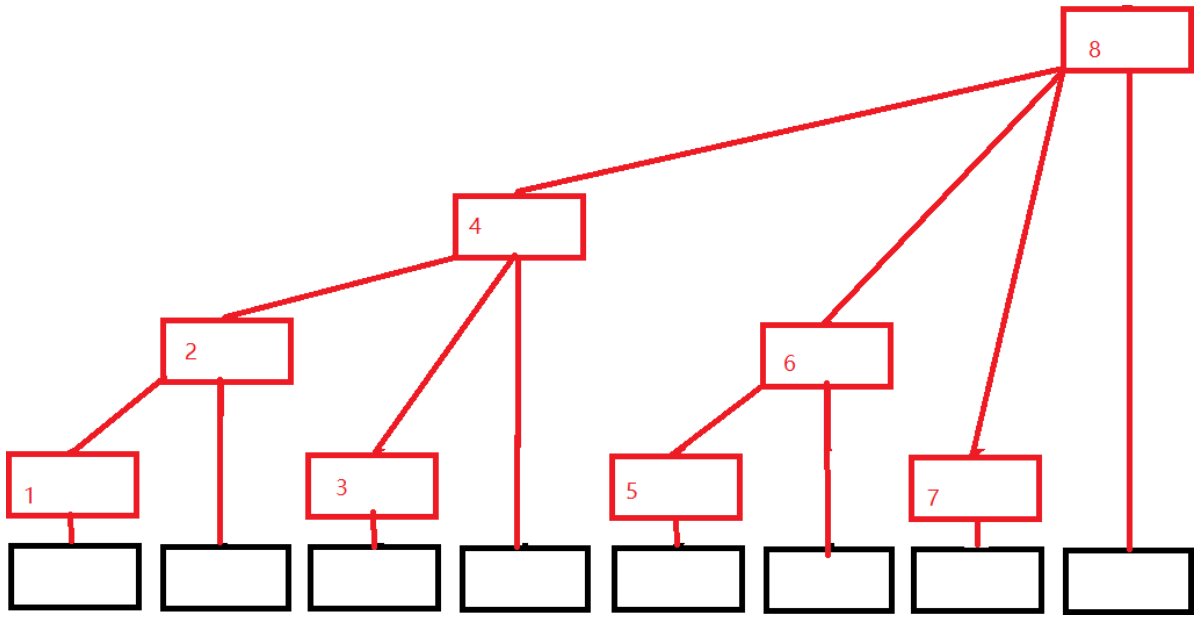
思路:

1.用埃氏筛法计算出 sqrt(b) 以内的素数，然后用这个素数去筛选 [a ， b] 合数;

```
public static int primtCount(int a,int b){
    boolean[] isprimeSmall = new boolean[(int)Math.sqrt(b)+5];
    boolean[] isprimeBig = new boolean[b-a+5];
    Arrays.fill(isprimeSmall,true);
    Arrays.fill(isprimeBig,true);
    for(int i=2;i<=b/i;i++){
        if(isprimeSmall[i] == true){
            for(int  j=2;j<isprimeSmall.length/i;j++)
                isprimeSmall[i*j] = false;
            for(int  j=Math.max(2,(a-1+i)/i)*i;j<=b;j=j+i)
        isprimeBig[j-a] = false;12     }
    }
    int count = 0;
    for(int i=0;i<(b-a);i++)
        if(isprimebig[i]==true && i+a>=2)
            count++;
    }
    return count;
}
```

树状数组

什么是树状数组



可以解决大部分基于区间上的更新以及求和问题。

lowbit (x & -x)

解释一:

x 的二进制最右边第一个出现的 1 和后面出现的 0 组成的 10 进制数:

lowbit(6) = 110 -> 2

lowbit(8) = 1000 -> 8

解释二:

x 的二进制最右边有 k 个连续的 0 ， lowbit(x) = 2^x ;

树状数组实现

```
public class TreeArray{
    int[] A;//原始数据
    int[] C;//树状数组
    int n;//数组元素个数
    public TreeArray(int[] A){
        this.A = new int[A.length];
        n = A.length;
        C = new int[n+1];
        for(int i=1;i<=A.length;i++){
        updata(i,A[i-1]);11
        }
    }
    public int lowbit(int x){
    return x & -x;15
    }
    public void updata(int i,int val){
        //原来的值A[i-1]，现在的值是val
```

```
        int data = val - A[i-1];//新旧值之间的差距
        A[i-1] = val;
        for(int pos = i;pos<=n;pos = pos+lowbit(pos)){
            C[pos] += data;
        }
    }
    public int sum(int pos){
        int sum = 0;
        for(int  i=pos;i>=1;i-=lowbit(i)){
            sum = sum + C[i];
            return sum;
        }
    }
    public int sumRange(int start,int end){
        if(start<1 || start>n || end<1 || end>n) return -1;
        else
            return sum(end) - sum(start-0);
    }
}
```

线段树

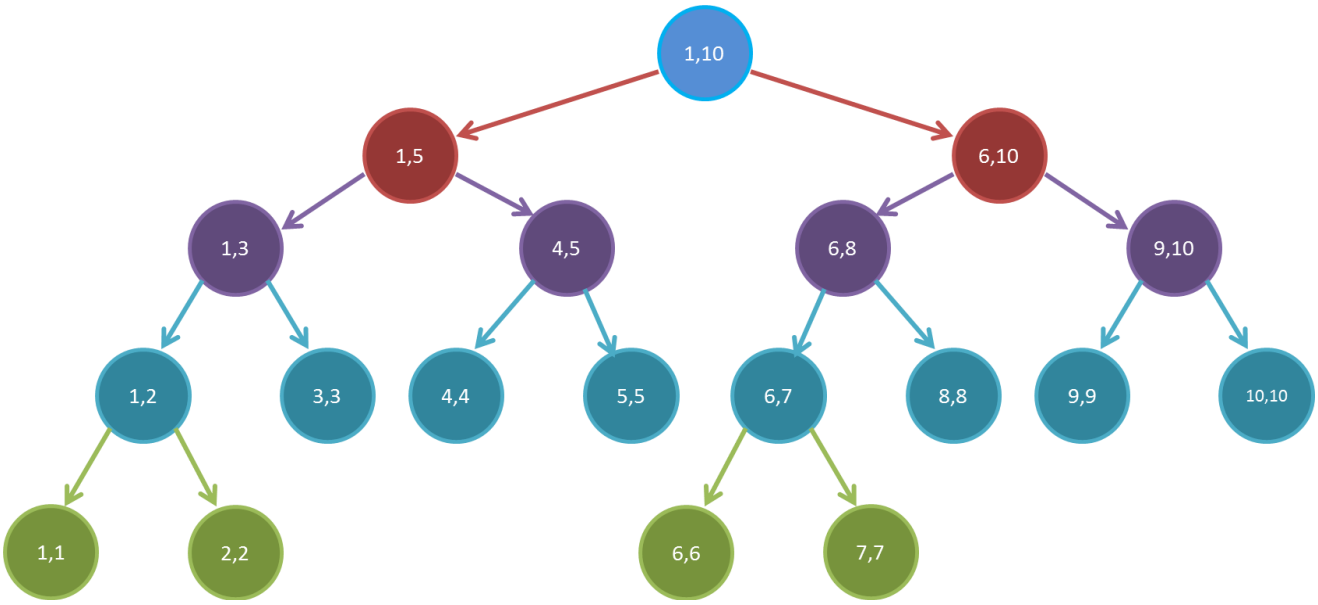
什么是线段树

线段树，是一种 二叉搜索树 。它将一段区间划分为若干 单位区间 ，每一个节点都储存着一个区间。它 功能强大 ，支持区间求和，区间最大值，区间修改，单点修改等操作。线段树的思想和分治思想很相像。线段树的每一个节点都储存着一段区间 [L . . R] 的信息，其中 叶子节点 L = R 。它的大致思想是：将一段大区间平均地划分成 2 个小区间，每一个小区间都再平均分成 2 个更小区间……以此类推，直到每一个区间的 L 等于 R （这样这个区间仅包含一个节点的信息，无法被划分）。通过对这些区间进行修改、查询，来实现对大区间的修改、查询。

这样一来，每一次单点修改、单点查询的时间复杂度都只为 O ($\log_2 n$) 。但是，可以用线段树维护的问题必须满足区间加法 ，否则是不可能将大问题划分成子问题来解决的。

线段树的原理和实现

1. 线段树主要是把一段大区间 平均地划分 成两段小区间进行维护，再用小区间的值来更新大区间。这样既能保证正确性，又能使时间保持在 log 级别（因为这棵线段树是平衡的）。也就是说，一个 [L , R] [][L,R] 的区间会被划分成 [L , （L + R）/ 2] 和 [（L + R）/ 2 + 1 , R] 这两个小区间进行维护，直到 L = R 。
2. 下图就是一棵 [1 , 10] 的线段树的分解过程（相同颜色的节点在同一层）



线段树一般需要四个函数build(root,l,n) 建树函数， modify(root,x,k) 修改函数， query(root,l,r) 查询函数，pushup(root)更新函数。我们最常用的操作就是遍历，而树的遍历需要从根节点开始，所以每一个函数都需要传入根节点root，下面对四个函数做简单介绍

build(root , l , n) 建树函数

该函数是树的初始化，通过根节点，向下初始化所有子节点。

```
public static void build(int u, int l, int r) {
    tree[u] = new node();
    if(l == r){//叶子节点
        tree[u].l = l;
        tree[u].r = r;
    }else {
        tree[u].l = l;
        tree[u].r = r;
        int mid = (l + r)/2;
        build(2*u, l, mid);
        build(2*u+1, mid + 1, r);
    }
}
```

modify(root , x , k) 修改函数

x表示要修改的位置，k表示修改的值，通过该函数对x处的值进行修改

```
public static void modify(int u, int x, long k) {
    if(tree[u].l == tree[u].r) {
        tree[u].v = k;
    }else {
        int mid = (tree[u].l + tree[u].r)/2;
        if(x <= mid) {
            modify(u*2, x, k);
        }else {
            modify(u*2+1, x, k);
        }
        pushup(u);//更新u
    }
}
```

query(root , l , r) 查询函数

查询l~r区间的区间值

```
public static long qurey(int u, int l, int r) {
    if(l <= tree[u].l && r >= tree[u].r) {
        return tree[u].v;
    }
    long res = 0;
    int mid = (tree[u].l + tree[u].r) / 2;
    if(l <= mid ) {
        res = qurey(u*2, l, r);
    }
    if(r > mid) {
        res += qurey(u*2+1, l, r);
    }
    return res;
}
```

pushup(root) 更新函数

在modify函数进行修改时，修改的是叶子节点，但叶子节点的修改也会影响父节点，通过该函数更新被修改的叶子节点的父节点的值

```
private static void pushup(int u) {
tree[u].v = tree[u*2].v + tree[u*2+1].v;3
}
```

例题：小球与盒子

题目链接：小球与盒子 - 蓝桥云课 (lanqiao.cn)

题目描述

给定 N 个空间无限大的盒子，编号分别为 $1 \sim N$。

现 M 次操作，操作有以下两种类型：

- `1 x y`，表示往编号为 x 的盒子内增加 y 个小球。
- `2 x y`，表示询问编号为 $x \sim y$ 的盒子内有多少个小球。

思路：

线段树叶子节点存储的是某个盒子中小球的个数，非叶子节点存储的是一段区间内盒子中小球数量的和，通过 modify 实现某个盒子增加小球的数量的操作，通过 query 实现查询l~r个盒子的小球总量。在 modify 直接将叶子节点的值改为了k，那么在本题中，应该是叶子节点原来的值加上 k 。

```
import java.io.BufferedReader;
import java.io.IOException;
import  java.io.InputStreamReader;
import  java.io.OutputStreamWriter;
import java.io.PrintWriter;

public class 小球与盒子 {
    public static class
        node{int l,r;
        long v;
    }
    static node tree[];
    static BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    static PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
    public static void main(String[] args) throws IOException {
```

```
        String[] s = in.readLine().split(" ");
        int n = Integer.parseInt(s[0]);
        int m = Integer.parseInt(s[1]);
        tree = new node[n*4];
        build(1,1,n);
        for (int i = 0; i < m; i++) {
            s = in.readLine().split(" ");
            int op = Integer.parseInt(s[0]);
            int x = Integer.parseInt(s[1]);
            int y = Integer.parseInt(s[2]);
            if(op == 1) {
                modify(1, x, y);
            }else {
                System.out.println(qurey(1, x, y));
            }
        }
    }
    private static void build(int u, int l, int r) {
        tree[u] = new node();
        if(l == r){
            tree[u].l = l;
            tree[u].r = r;
        }else {
            tree[u].l = l;
            tree[u].r = r;
            int mid = (l + r)/2;
            build(2*u, l, mid);
            build(2*u+1, mid + 1, r);
        }
    }
    static long qurey(int u, int l, int r) {
        if(l <= tree[u].l && r >= tree[u].r) {
            return tree[u].v;
        }
        long res = 0;
        int mid = (tree[u].l + tree[u].r) / 2;
        if(l <= mid ) {
            res = qurey(u*2, l, r);
        }
        if(r > mid) {
            res += qurey(u*2+1, l, r);
        }
        return res;
    }
    static void modify(int u, int x, long k) {
        if(tree[u].l == tree[u].r) {
            tree[u].v += k;
        }else {
            int mid = (tree[u].l + tree[u].r)/2;
            if(x <= mid) {
                modify(u*2, x, k);
            }else {
                modify(u*2+1, x, k);
            }
            pushup(u);//更新u
        }
    }
    static void pushup(int u) {
        tree[u].v = tree[u*2].v + tree[u*2+1].v;
    }
}
```

树型dp

树型 dp 即在树上进行 dp 。

树是无环图，顺序可以是从叶子到根节点，也可以从根到叶子节点。

一般树型 dp 的特征很明显，即状态可以表示为树中的节点，每个节点的状态可以由其子节点状态转移而来（从叶子到根的顺序），或是由其父亲节点转移而来（从根到叶节点的顺序），也可是两者结合。

找出状态和状态转移方程仍然是树型 dp 的关键。

例题：蓝桥舞会

题目链接：蓝桥舞会 - 蓝桥云课 (lanqiao.cn)

题目描述

蓝桥公司一共有 n 名员工，编号分别为 $1 \sim n$。

他们之间的关系就像一棵以董事长为根的树，父节点就是子节点的直接上司。

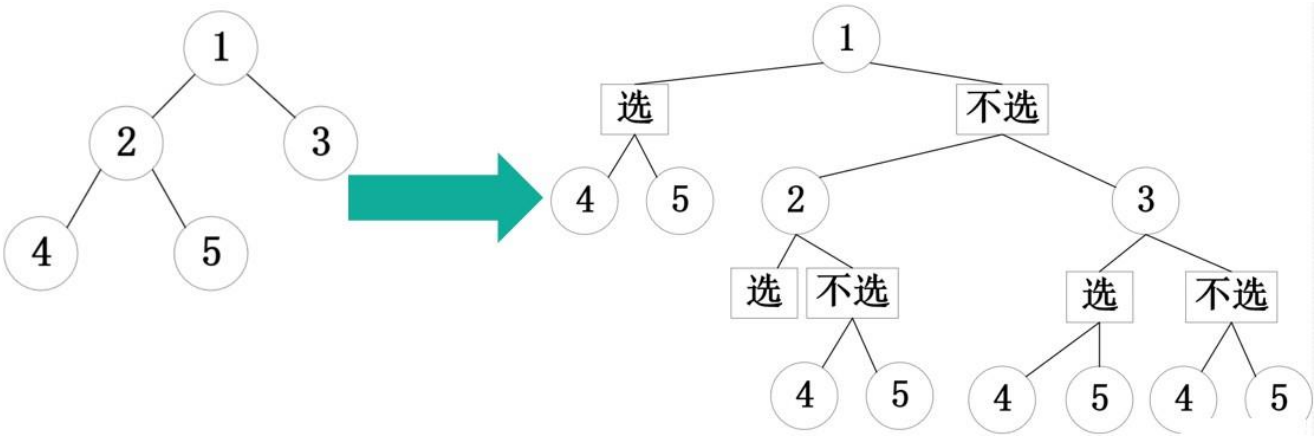
每个员工有一个快乐指数 a_i。

现蓝桥董事会决定举办一场蓝桥舞会来让员工们在工作之余享受美好时光，不过对于每个员工，他们都不愿意与自己的直接上司一起参会。

董事会希望舞会的所有参会员工的快乐指数总和最大，请你求出这个最大值。

思路：

假设题目样例也变化为左图，那么便有又图的结果：



1.首先确定 dp 数组以及数组每一维表示的意义。dp[i][j] 表示以 i 为根节点的子树，j 的取值只有 0 和 1 ，1 表示选了节点 i ，0 表示没选节点 i 。然后确定状态的转移，当选了 i 时，根据题目要求不能选择 i 的直接子节点，当没有选择 i 时，可以选择与 i 直接相连的子节点，当然，也可以不选。那么转移公式如下：

2.dp[i][0] += max(dp[u][0] , dp[u][1]) 其中 u 是与 i 直接相连的子节点，对于该子节点，我既可以选，也可以不选，取两种状态的最大值。

3.dp[i][1] += dp[u][0] 其中u是与i直接相连的子节点，对于该子节点，必然不能选择，只有一种状态。

```
import java.io.BufferedReader;
import java.io.IOException;
import  java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import  java.util.ArrayList;

public class Main {
```

```java
    static  ArrayList<Integer>[] q;//存储结点i的所有子结点
    static boolean[] visit;//判断此点是否是根节点
    static long[][] dp;
    static int[] a;//记录每一个人的快乐指数
    static BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    static PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
    public static void main(String[] args) throws IOException {
        int n = Integer.parseInt(in.readLine());
        a = new int[n + 1];
        q = new ArrayList[n+1];
        dp = new long[n + 1][2];
        visit = new boolean[n + 1];
        String[] s = in.readLine().split(" ");
        for(int i=1;i<=n;i++) {
            a[i] = Integer.parseInt(s[i-1]);
        q[i] = new ArrayList<>();25}
        for(int i=1;i<n;i++) {
            s = in.readLine().split(" ");//u的上司是v
            int u = Integer.parseInt(s[0]);
            int v = Integer.parseInt(s[1]);
            visit[u] = true;//u不是根结点
        q[v].add(u);//u是v的一个子结点（下属） 32
        }
        int root = -1;
        for(int i=1;i<=n;i++) {
            if(!visit[i]) {//只要根结点的visit值仍为false
                root = i;
                break;
            }
        }
        dfs(root);
System.out.println(Math.max(dp[root][1], dp[root][0]));//最终结果考虑根结点和不考虑根结点的最大值
    }
    static void dfs(int root) {
        dp[root][1] += a[root];//考虑根结点的初值为根结点的快乐指数
        for (int i = 0; i < q[root].size(); i++) {//处理root的每一个子结点
            int to = q[root].get(i);
            dfs(to);
            dp[root][1] += dp[to][0];//考虑根结点，那么不能考虑子结点
            dp[root][0] += Math.max(dp[to][0], dp[to][1]);//不考虑根结点，子结点可以考虑，也可以不考虑
        }
    }
}
```

区间dp

什么是区间dp

区间dp就是在区间上进行动态规划，求解一段区间上的最优解。主要是通过合并小区间的 最优解进而得出整个大区间上最优解的dp算法。

核心思路

既然求解在一个区间上的最优解，那么把这个区间分割成一个个小区间，求解每个小区间的最优解，再合并小区间得到大区间即可。所以在代码实现上，可以枚举区间长度 len 为每次分割成的小区间长度（由短到长不断合并），内层枚举该长度下可以的起点，自然终点也就明了了。然后在这个起点终点之间枚举分割点，求解这段小区间在某个分割点下的最优解。

```
for(int len = 1;len<=n;len++){//枚举长度
    for(int j = 1;j+len<=n+1;j++){//枚举起点，ends<=n
        int ends = j+len - 1;
        for(int i = j;i<ends;i++){//枚举分割点，更新小区间最优解
        dp[j][ends]  =  min(dp[j][ends],dp[j][i]+dp[i+1][ends]+something);6
        }
    }
}
```

例题：石子合并

题目链接：石子合并 - 蓝桥云课 (lanqiao.cn)

题目描述

有 n 堆石子排成一排，每堆石子有一定的数量。现在我们要将 n 堆石子并成为一堆，每次只能合并相邻的两堆石子，合并的花费为这两堆石子的总数。经过 $n-1$ 次合并后会成为一堆，求总的最小花费。

思路：

每一次都合并两个最小的石子堆，那么最后得到的总代价必然最小。其中，需要用一个 sum 数组保存不同区间石子堆的和，另外设一个 dp 二维数组,用来记录总代价，假设 i < j ,则 dp[i][j] 等于 [i , j] 区间的石子和 sum 加上 [i , j] 区间的最优化石子合并解，得到的状态方程如下：

$$dp[i][j]=\begin{cases}0, i=j\\ dp[i][k]+dp[k+1][j]+sum[j]-sum[i-1], i<k<j\end{cases}$$

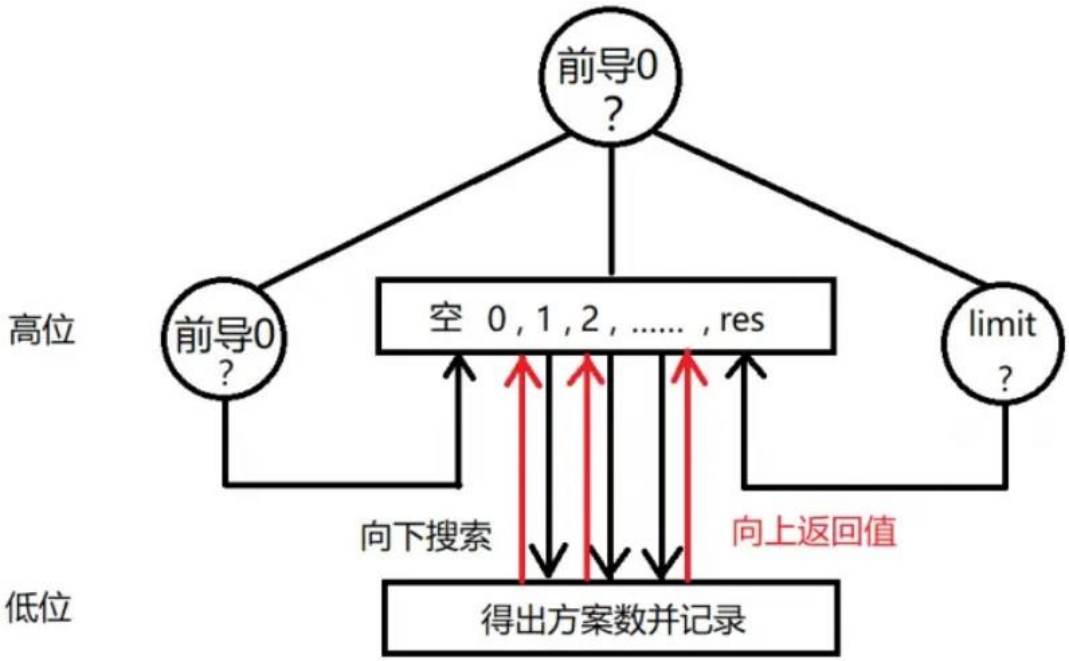
```
import java.util.Arrays;
import java.util.Scanner;

public class Main {
    static int max = 0x3f3f3f3f;
    public static void main(String[] args)
        {    Scanner     sc     =     new
        Scanner(System.in);
        while(sc.hasNext()) {
            int n = sc.nextInt();
            int[] num = new int[n+1];
            int[] sum = new int[n+1];
            int[][] dp = new int[n+1][n+1];
            for(int i=0;i<=n;i++) {
                Arrays.fill(dp[i],max);
            }
            sum[0] = 0;
            for(int i=1;i<=n;i++)
```

```
                sum[i] = num[i] + sum[i-1];
                dp[i][i] = 0;
            }
            for(int len=1; len<n; len++) {//第一层len代表区间长度
                for(int i=1; i+len<=n; i++) {//第二层i代表区间开头
                    int j = i+len;
                    for(int k=i; k<j; k++) {//第三层k代表区间分割的位置
                        dp[i][j] = Math.min(dp[i][j], dp[i][k]+dp[k+1][j]+sum[j]-sum[i-1]);
                    }
                }
            }
            System.out.println(dp[1][n]);//即代表1-n石子的最小代价
        }
    }
}
```

数位dp

数位DP往往都是这样的题型，给定一个闭区间[l ，r] ，求这个区间中满足某种条件的数的总数。

考虑人类计数的方式，最朴素的计数就是从小到大开始依次加一。但我们发现对于位数比较多的数，这样的过程中有许多重复的部分。例如，从 7000 数到 7999、从 8000 数到 8999、和从 9000 数到 9999 的过程非常相似，它们都是后三位从 000 变到 999，不一样的地方只有千位这一位，所以我们可以把这些过程归并起来，将这些过程中产生的计数答案也都存在一个通用的数组里。此数组根据题目具体要求设置状态，用递推或 DP 的方式进行状态转移。



例题：小明数：

题目链接：小明数 - 蓝桥云课 (lanqiao.cn)

题目描述

小明以他的名字命名了一种数——小明数。

对于一个十进制的数，若任意相邻两个数位的差值的绝对值不超过 K，则称其为小明数。

现小明给出一个区间 L, R，求区间 L, R 一共有多少个小明数。

思路：

1.预处理：观察给出的数据范围，指定的数最大为10的 次方，也就是最多有 位数，先通过预处理，处理出数字个数为 的所有符合要求的数字。用数组 f[i][j] 存储结果, i 表示数字的位数，j 表示有 i 位数字的数的最高位的数字为 j（注意每一位都可以取 0 ）。

2.从指定数的最高位开始遍历，且该数要小于指定数，假设求 满足相邻位数之差小于 5 的数的个数，那么就从第 位开始遍历，第 位的数字要小于 ，可以填 ，假设填 6，他们的差为 1，符合要求，那么现在第 位后面的数字无论填什么都是小于指定数的，那么当前符合要求的数的个数实际有 f[7][6] 个，就是求 7 位数且固定了第 7 位为 6 的满足要求的数的个数，也就是 f[7][6]。这个结果之前已经预处理出来了。其它同理。

3.在第二步求的仅仅是和指定数位数相同的满足要求的数的个数，为什么要先求这个呢？因为数要小于指定数，只要位数小于指定数那么它的值一定小于指定数，所以不用判断，但是位数相同时，需要逐位比较，所以需要特判。在这一步，只需要加上所有的位数小于指定数的满足要求的数就行，而它已经在预处理的时候求出来了。

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.StreamTokenizer;
import java.util.Arrays;
import java.util.Scanner;
public class Main {
    static int[][] f = new int[11][10];
    static int k;
    public static void main(String[] args) throws IOException {
        StreamTokenizer sc=new StreamTokenizer(new BufferedReader(new
InputStreamReader(System.in)));
        sc.nextToken();
        int t = (int)sc.nval;
        while(t> 0)
{       t--;
            sc.nextToken();
            k = (int)sc.nval;
            sc.nextToken();
            int l = (int)sc.nval;
            sc.nextToken();
            int r = (int)sc.nval;
            for (int i = 0; i < 11; i++) {
                Arrays.fill(f[i], 0);
            }
            init();
        System.out.println(dp(r) - dp(l - 1));
        }
        return;
    }
    private static int dp(int num)
{       if(num == 0) {
            return 0;
        }
        int res = 0;
```

```
        int last = 0;//上一个位数的数字
        int[] nu = new int[12];
        int n = 1;
        while (num > 0 ) {
            nu[n++] = num%10;
            num = num / 10;
        }
        n--;
        for (int i = n; i > 0; i--) {//遍历位数
            int x = nu[i];
            int jj;
            if(i == n) {
                jj = 1;
            }else {
                jj = 0;
            }
            for (; jj < x; jj++) {//遍历该位数上可以填的数字

                if(Math.abs(jj - last) <= k || i == n) {
                    //System.out.println("mm" + i);
                    res += f[i][jj];
                }
            }
            if(Math.abs(x-last) <= k || i == n) {
                last = x;
            }else {
                break;
            }
            if(i==1) {
                res++;
            }
        }
        //加包含前导0的，其实就是加上不是和num同位数的数字，
        for (int i = 1; i < n; i++) {
            for (int j = 1; j < 10; j++) {//从1开始
                res += f[i][j];
            }
        }
        return res;
    }
    private static void init() {
        for (int i = 0; i < 10; i++) {//初始化只有一位数字的时候，一定符合要求
            f[1][i] = 1;//注意i一定从0开始
        }
        for (int i = 2; i < 10; i++) {//初始化其它位数的数字
            for (int j = 0; j < 10; j++) {//注意，这里可以包含0
                for (int m = 0; m < 10; m++) {
                    if(Math.abs(m-j) <= k) {
                        f[i][j] += f[i-1][m];
                    }
                }
            }
        }
    }
}
```

状压dp

状态压缩动态规划，就是我们俗称的状压DP，是利用计算机二进制的性质来描述状态的一种DP方式。

很多棋盘问题都运用到了状压，同时，状压也很经常和BFS及DP连用。

状压dp其实就是将状态压缩成2进制来保存 其特征就是看起来有点像搜索，每个格子的状态只有1或0 ，是另一类非常典型的动态规划

会常常用到位运算处理：

1.判断一个数字x二进制下第i位是不是等于1。（最低第1位）

方法：if(((1<<(i−1))&x)>0) 将1左移i-1位，相当于制造了一个只有第i位 上是1，其他位上都是0的二进制数。然后与x做与运算，如果结果>0， 说明x第i位上是1，反之则是0。

2.将一个数字x二进制下第i位更改成1。

方法：x=x|(1<<(i−1)) 。

3.将一个数字x二进制下第i位更改成0。

方法：x=x&~(1<<(i−1))

4.把一个数字二进制下最靠右的第一个1去掉。

方法：x=x&(x−1)