

book_acm

修昊

Published
with GitBook



目錄

1. [介紹](#)
2. [当前任务](#)
3. [训练计划](#)
4. [提示](#)
 - i. [位操作](#)
 - ii. [卡题点](#)
 - iii. [奇技淫巧](#)
 - iv. [借助Python](#)
 - v. [使用Java](#)
5. [赛用cpp](#)
6. [赛用vimrc](#)
7. [库函数使用](#)
8. [分类](#)
 - i. [图论](#)
 - ii. [数论](#)
 - iii. [基础数据结构](#)
 - iv. [搜索](#)
 - v. [计算几何](#)
 - vi. [博弈](#)
 - vii. [串处理](#)
 - viii. [动态规划](#)
 - ix. [概率](#)
 - x. [排序](#)
 - xi. [STL模板](#)
 - i. [STL中map用法详解（转）](#)
 - ii. [STL之set集合容器（转）](#)
 - iii. [STL中的set容器的一点总结（转）](#)
 - xii. [哈希](#)
 - xiii. [杂题，水题](#)
 - xiv. [组合数学](#)
 - xv. [网络流](#)
9. [解题报告](#)
10. [相关学习资料](#)

ACM Template

ACM开源模板

主要来自于平时的总结，学习。非常欢迎提issue来一起完善这个模板。

从ACM_template转移到这个项目中 -- 依然可以通过简单的方式更新这个项目，但是添加了一条更加方便的阅读途径。

http://svtter.gitbooks.io/book_acm/

此外，你还可以通过这条途径下载到本模板的电子版本，供打印使用

目录

- [当前任务](#)
- [训练计划](#)
- [提示](#)
 - [位操作](#)
 - [卡题点](#)
 - [奇技淫巧](#)
 - [借助Python](#)
 - [使用Java](#)
- [赛用cpp](#)
- [赛用vimrc](#)
- [库函数使用](#)
- [分类](#)
 - [图论](#)
 - [数论](#)
 - [基础数据结构](#)
 - [搜索](#)
 - [计算几何](#)
 - [博弈](#)
 - [串处理](#)
 - [动态规划](#)
 - [概率](#)
 - [排序](#)
 - [STL模板](#)
 - [哈希](#)
 - [杂题，水题](#)
 - [组合数学](#)
 - [网络流](#)
- [解题报告](#)
- [相关学习资料](#)

LICENSE

GPL

ACM分类任务

- ☐ 添加关于二分图的内容
- ☐ 数论的相关内容
- ☐ 网络流的相关内容
 - ☒ 基础网络流
 - ☐ 有上下界的网络流
- ☐ 卡特兰数
- ☐ 补充其他相关内容
- ☐ 矩阵的连乘

大二上学期ACM训练计划

第二周，第三周

1. YT第十二章（应用最小生成树算法编程）-相关题库（5道题，最少做三道）
2. YT第十三章（Warshall和floy-warshall的实验范例）
3. YT第十三章（Dijkstra算法的实验范例和bellman-ford）
4. YT第十三章（SPFA）
5. YT第十三章（应用最佳路径算法编程）-相关题库（6道题，最少做三道）
6. YT第十四章（二分匹配图的实验范例）
7. YT第十四章（计算网络最大流的实验范例）-必做题
8. YT第十四章（应用特殊图的经典算法编程）(8道题，最少做四道)

第四周一第八周

第四周：

1. 高精度（HDU: 1023 1715 1297 HLG:1159 1624 POJ: 1001 1503）
2. 枚举和二分 (HLG: 1019 1039 1118 1215 1293 1408 1429 POJ: 1753 2965 2456 1064)
3. 贪心，构造，随机化算法,分治法
 - (贪心：HDU: 2037 4296 POJ：1328 2109 2586)
 - (构造：POJ：3295 HDU：4982)
 - (随机化：POJ：3318 5454)
 - (分治法：POJ 2524)

讲解：RMQ,字典树

第五周：

1. RMQ,LCA问题 (ZOJ :2859 POJ: 3264 3368 1330)
2. 字典树 (POJ：1056 2001 2503 3630)
3. 递归,递推，找规律 (HDU: 2041~2050 2065 2067 HLG: 1038 1099 1124 1191 1200 1267 1280 1298 1327 1328 1613 1431)
4. 背包（01背包，完全背包，多重背包）(HDU: 2602 2191 1114 HLG:1053 1333 1627) 讲解：基础DP（区间DP等典型DP：LIS,LCS等）

第六周：

1. 区间DP，LIS,LCS等 (HLG: 1004 1038 1053 1116 1147 1284 1331 1426 1558)
2. 线段树，树状数组 (HDU: 1166 1754 1394 1698 1540 POJ:2828 2886 3468 2528 3667)
3. 记忆化搜索 (POJ: 3373 1691)

讲解：树形DP，状态DP

第七周

1. 树形DP, 状态DP (POJ: 3254 1185 HLG: 1475 1473 HDU: 4628 1011 1520 ZOJ: 3662)
2. 最短路径算法 (POJ: 1860 3259 1062 2253 1125 2240)
3. 差分约束系统的建立和求解 (POJ: 1201 2983 6159 1275 1364)
讲解: DP优化

第八周:

1. DP优化相关题目 (POJ: 2823 1160 1180 3709 HDU: 3507 4258 1729)
2. 基础组合数学 (组合, 母函数, 错排, 卡特兰数, 置换) (HDU: 1028 1134 1342 2067 1465 1398 4248 2049 FZU: 1570 POJ: 1942 1664 3270 1026 2369)
3. 数论基础 (HLG: 1036 1076 1077 1115 1151~1154 1328 1353 1660 POJ: 2635 3292 1845 2115)

讲解: 计算几何基础知识 (差积, 线段相交, 凸包等基础知识)

第九周—第十二周

第九周:

1. 快速幂.矩阵乘法 (HDU: 2035 HLG: 1126 1251 1375)
2. 博弈论 (POJ: 3317 1085)
3. 计算几何学 (坐标离散化, 扫描线算法, 多边形的核等) (坐标离散化: POJ 1151) (扫描线算法: POJ 1765 1151 2280 3004 3277) (多边形的核: POJ 3130 3335)

讲解: 二分图二分图 (匈牙利算法, **KM**算法), 二分图判断, 二分图模型

第十周:

1. 匈牙利算法, KM算法 (HDU: 2063 3395 POJ: 2195 2400 2239 1469 3686)
2. 二分图的最大匹配 (POJ: 3041 3020)
3. 二分图模型 (HDU: 2444 3605 POJ: 3041 2771 HLG: 1492)

讲解: 网络流, 最大流, 最小割

第十一周:

1. 网络流模型 (POJ: 3469 2112 HLG: 1521 1061 HDU: 4292)
2. 最大流的增广路算法 (POJ: 1459 3436)
3. 最小割模型 (POJ: 3308 3155)

讲解: 连通图 (强连通, 双连通, 2-SAT)

第十二周:

1. 欧拉路径和欧拉回路 (HLG: 1658 HDU: 1116 1878 POJ: 2513)
2. 强连通分支及其缩点 (POJ: 2186 3592 3114 3352 1523 HDU: 1269 3594)

3. 2-SAT问题 (POJ: 3207 3678 3683 3648 2723 2749)
4. 图的割边和割点 (POJ: 3352)

**讲解：费马大小定理，扩展欧几里得（求逆元），欧拉函数，容斥原理

第十三周—第十四周

第十三周：

1. GCD,扩展欧几里得（求逆元）
2. 费马大小定理 (HDU: 4704 POJ: 2407 1061 2115 1845 HLG: 1176 1005 1807)
3. 组合数的取余 (FZU: 2020 HDU: 3037 3944 3398)
4. 欧拉函数,容斥原理 (HDU: 1695 2824 3501 1286 POJ: 3090 2478)

讲解：高斯消元，概率DP

第十四周：

1. 高斯消元 (POJ: 2947 1487 2065 1166 1222)
2. 概率 (POJ: 3071 3440 2096 2151 HDU: 4405 3853 4035 4576 ZOJ: 3329)
3. POLYA定理，置换群 (POJ: 1286 2409 3270 1026)

讲解：搜索剪枝，A*，IDA

第十五到第十八周，主要学习较难的图算法和数据结构

1. 度限制最小生成树和第K最短路，分数规划 (poj1639, poj3621, poj2976 poj3255, poj2513, poj2449)
2. 最短路，最小生成树，二分图，最大流问题的相关理论
(poj3155,poj2112,poj1966,poj3281,poj1087,poj2289,poj3216,poj2446)
3. 最优比率生成树 (poj2728 (0/1分数规划应用))
4. 最小树形图 (poj3164(朱-刘算法))
5. 次小生成树 (poj1679(存在 $O(n^2)$ 的DP解法))
6. 无向图，有向图的最小环 (poj1734(floyd扩展))
7. Trie图的建立和应用,DFA (hdu2222 poj2778, poj3691)
8. 双端队列及其应用 (poj2823)
9. 左偏树 (poj3666, poj3016)
10. 后缀树，后缀数组 (poj3415,poj3294, poj2774,poj2758)
11. 四边形不等式理论，斜率优化 (poj1160, poj1180, poj3709)
12. 较难的状态DP，插头DP (poj3133,poj1739,poj2411、poj1763)
13. 计算几何学（半平面求交，可视图的建立，点集最小圆覆盖，对踵点）(poj3384,poj2540, poj2966, zju1450, poj2079)

做题技巧

- 卡题点
- 生成测试数据
- 位操作
- int,long,long long 类型范围

浮点数eps

判断两个浮点数误差的时候，使用 `fabs(a-b) < eps`，一般的eps为 `1e-9`。如果有可能，尽量避免浮点运算，做整数的转换。

计算一元二次方程解的时候，可以进行如此运算，例如 `((sqrt(8.0*n+1)-1)/2-eps)+1`

熟用log函数

多多利用log函数，用以减小数量级。

利用矩阵

矩阵的乘积

有向面积

通过有向面积判断点是否在图形内部

通过行列式的三个点求有向面积

例如：

$$2A = \begin{vmatrix} x_0 & y_0 & 1 \\ x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \end{vmatrix} = x_0y_1 + x_2y_0 + x_1y_2 - x_2y_1 - x_0y_2 - x_1y_0$$

两倍的三角形面积 方法是构建齐次坐标，如果逆时针，有向面积为正，顺时针，有向面积为负。

类型范围

类型	范围
unsigned int	0~4294967295
int	2147483648~2147483647
unsigned long	0~4294967295
long	2147483648~2147483647

long long的最大值：	9223372036854775807
long long的最小值：	-9223372036854775808
unsigned long long的最大值：	18446744073709551615
__int64的最大值：	9223372036854775807
__int64的最小值：	-9223372036854775808
unsigned __int64的最大值：	18446744073709551615

生成测试数据

比赛的时候出现了100 * 100组数据的情况，但是当时使用freopen忘记了具体的步骤， 特意重新写一下，也是属于基础的内容。

生成一百行数据，每行100个数据，每个数据为100。

```
#include <stdio.h>
#include <iostream>`enter code here`
using namespace std;

int main()
{
    freopen("input", "r", stdin);
    freopen("output", "w", stdout);

    for(int i = 0; i < 100; i++)
        for(int j = 0; j < 100; j++)
            printf("%d%c", 100, j == 99? '\n' : ' ');

    fclose(stdin);
    fclose(stdout);
    return 0;
}
```

运行过后生成的数据（本来应该输出在屏幕上，此时不会输出到屏幕，而是输出到文件）会保存到output文件中。 如果需要使用直接更改output的文件名，再使用一次freopen('r')即可。

C++按位操作符

bitset

```
// N为位数  
bitset <N> b;  
  
// 全部清1  
b.set();  
  
// 全部清0  
b.reset();  
  
// 为b的某一位赋值  
b[pos] = 1 || 0;
```

C语言中的位域

尚未添加

卡题

每次卡题都会记录在这里

基本问题

1. fgets函数和cin.getline函数的使用
2. 01背包问题动归的解决
3. $a*b/c$ 使用过程中溢出。。

如果保证 $a\%c==1$ 则可以 $a/c*b$ ，防止数论中的溢出

数论

1. 上下标

图论

1. 初始化的准确性
2. 点的出度入度
3. 输出路径的顺序
4. 图是否连通

一些奇技淫巧

- 位操作
- 利用python

利用def来控制编译是否使用freopen

编译的时候加上 `-DDEBUG` 选项就可以控制 `freopen()`

```
#include <stdio.h>
#include <string.h>
#include <math.h>

int main()
{
#ifdef DEBUG
    freopen("input", "r", stdin);
#endif
    return 0;
}
```

使用const定义常量，而非def

使用const好处在于，可以确定变量的类型，而不是单纯的一段代码。

段错误和栈溢出

使用 `size` 命令来查看执行文件中段大小

text	data	bss	dec	hex	filename
1320	280	4	1604	644	3-8

分别是正文段，数据段和bss段。

此外，局部变量也是存放在堆栈段的，所以栈溢出不见得是递归调用过多，也可能是局部变量太大（Stack Overflow）。

ulimit 查看栈大小

Linux的栈大小通过ulimit命令修改，不过在比赛中似乎是不允许的。使用 `ulimit -s` 命令查看栈大小，单位是KB

INF的取值

一般是0x3f3f3f3f, 可以用memset来赋值

assert.h

利用assert.h调试, 通过检测变量值来确定程序运行过程中的问题

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

int main()
{
    int x = -1;
    assert(x > 0);
    return 0;
}
```

printf的进制利用

x, o, d

```
// 大写小写16进制
printf("%X", d);
printf("%x", d);

// 效果不同
printf("%02X", d);
printf("%.2X", d);
```

python科学计算不是没有依据的==特别大型的数据需要好好计算一下,但是一般的验证,生成测试数据神马的用python真是再方便不过了.

进制转换

使用Python内置函数：`bin()`、`oct()`、`int()`、`hex()` 可实现进制转换。

其中 `int([number | string[, base]])` 可以转换其他进制.比如8进制转换到二进制,我们这样 `bin(int(x,8))` .

着实有用.

Math库的使用

我们依据素数定理求素数个数 -- 一般要使用这个定理的时候int值会比较大.

```
import math

# 求lnx的值
print math.log(x, math.e)

# 求pi
print math.pi
print math.acos(-1.0)
```

日后使用到的现在没有想到的,会陆续添加.

datetime日期库的使用

```
# 1000天以后
import datetime
a = datetime.datetime(2015,02,19)
a += datetime.timedelta(1000)
print a
```

Java相关技巧

- [Java一般操作](#)
- [Java针对大数的操作总结](#)
- [Java中的数据结构](#)

Java一般操作

1. 提交格式

```
import java.util.*;
public class Main{
    public static void main(String[] args){
        Scanner cin = new Scanner(System.in));
    }
}
```

1. 输入

```
cin.hasNext() 或 cin.hasNextInt() 或 cin.hasNextDouble()    // 判断是否有下一个输入可以用
int n = cin.nextInt();
String s = cin.next();
double t = cin.nextDouble();
String s = cin.nextLine();
```

1. 浮点数输出

```
DecimalFormat g = new DecimalFormat("0.00");
double a = 123.45678;
System.out.println(g.format(a));    // 输出 123.46
```

1. 简单大数

```
import java.math.*;

BigInteger a = BigInteger.valueOf(100000);
BigInteger b = BigInteger.valueOf(10000);

System.out.println(a.add(b));
System.out.println(a.subtract(b));
System.out.println(a.multiply(b));
System.out.println(a.divide(b));
System.out.println(a.mod(b));
System.out.println(a.compareTo(b));    // 返回值 -1, 0, 1
```

1. 字符串


```
s.substring(a,b);
String a = "0123456789";
System.out.println(a.substring(2,5));           // 234
```

1. 数组

```
int[] a = new int[123];
Arrays.fill(a,123);           // 相当于memset
System.out.println(a[3]);      // 输出 123
```

Java针对大数的操作总结

- BigInteger：针对大的整数，是以字符串方式进行传入的

```
import java.math.BigInteger ;
public class BigIntegerDemo01{
    public static void main(String args[]){
        BigInteger bi1 = new BigInteger("123456789") ; // 声明BigInteger对象
        BigInteger bi2 = new BigInteger("987654321") ; // 声明BigInteger对象
        System.out.println("加法操作：" + bi2.add(bi1)) ;    // 加法操作
        System.out.println("减法操作：" + bi2.subtract(bi1)) ;    // 减法操作
        System.out.println("乘法操作：" + bi2.multiply(bi1)) ;    // 乘法操作
        System.out.println("除法操作：" + bi2.divide(bi1)) ;    // 除法操作
        System.out.println("最大数：" + bi2.max(bi1)) ;    // 求出最大数
        System.out.println("最小数：" + bi2.min(bi1)) ;    // 求出最小数
        BigInteger result[] = bi2.divideAndRemainder(bi1) ; // 求出余数的除法操作
        System.out.println("商是：" + result[0] +
            "；余数是：" + result[1]) ;
    }
};
```

执行结果：

加法操作：1111111110

减法操作：864197532

乘法操作：121932631112635269

除法操作：8

最大数：987654321

最小数：123456789

商是：8；余数是：9

- BigDecimal：针对大的浮点数，是以字符串的方式传入的

```
import java.math.* ;
class MyMath{
    public static double add(double d1,double d2){           // 进行加法计算
        BigDecimal b1 = new BigDecimal(d1) ;
        BigDecimal b2 = new BigDecimal(d2) ;
        return b1.add(b2).doubleValue() ;
    }
    public static double sub(double d1,double d2){           // 进行减法计算
```

```

        BigDecimal b1 = new BigDecimal(d1) ;
        BigDecimal b2 = new BigDecimal(d2) ;
        return b1.subtract(b2).doubleValue() ;
    }
    public static double mul(double d1,double d2){        // 进行乘法计算
        BigDecimal b1 = new BigDecimal(d1) ;
        BigDecimal b2 = new BigDecimal(d2) ;
        return b1.multiply(b2).doubleValue() ;
    }
    public static double div(double d1,double d2,int len){        // 进行乘法计算
        BigDecimal b1 = new BigDecimal(d1) ;
        BigDecimal b2 = new BigDecimal(d2) ;
        return b1.divide(b2,len,BigDecimal.ROUND_HALF_UP).doubleValue() ;
    }
    public static double round(double d,int len){        // 进行四舍五入
        BigDecimal b1 = new BigDecimal(d) ;
        BigDecimal b2 = new BigDecimal(1) ;
        return b1.divide(b2,len,BigDecimal.ROUND_HALF_UP).doubleValue() ;
    }
};

public class BigDecimalDemo01{
    public static void main(String args[]){
        System.out.println("加法运算：" + MyMath.round(MyMath.add(10.345,3.333),1)) ;
        System.out.println("减法运算：" + MyMath.round(MyMath.sub(10.345,3.333),3)) ;
        System.out.println("乘法运算：" + MyMath.round(MyMath.mul(10.345,3.333),2)) ;
        System.out.println("除法运算：" + MyMath.div(10.345,3.333,3)) ;
    }
};
```


``java

 执行结果：

 加法运算：13.7

 减法运算：7.012

 乘法运算：34.48

 除法运算：3.104


```

总结：针对大数以字符串的方式传入，再利用提供的加减乘除方法来操作数据，而不是以'+ - \* /'来操作数据

## Java中的数据结构

好文推荐:

1. <http://blog.csdn.net/meyoung01/article/details/15812895>
2. <http://www.cnblogs.com/eflylab/archive/2007/01/20/625164.html>

### 1. Set

Set是接口，此外，还有诸多类，时间问题日后补充。

```

import java.util.HashSet;

Set set = new HashSet();
set.clear();
set.contains(element);
set.insert(element);

```

```
set.empty(element);
```

```

#include <bits/stdc++.h>
using namespace std;

#define INF 0x3f3f3f3f
#define ll long long int
#define MEM(a) memset(a, 0, sizeof(a))
#define MEMM(a) memset(a, -1, sizeof(a))
#define DEB(x, n) \
 cout << (x) << " " << (n) << endl
#define FOR(i, s, e) \
 for(int (i) = (s); (i) < (e); (i)++)
#define CR printf("\n")
#define println(format, ...) ({\
 printf(format "\n", __VA_ARGS__); })
const double PI = acos(-1.0);

int main()
{
#ifdef DEBUG
 // freopen("input", "r", stdin); //从input文件中读入
 // freopen("output", "w", stdout); //输出到output文件
#endif

 return 0;
}

```

## 库函数

- [math.h](#)
- [ctype](#)
- [string.h](#)

## math.h

### 1. 三角函数

```
double sin (double);
double cos (double);
double tan (double);
```

### 1. 反三角函数

```
double asin (double); 结果介于 $[-\pi/2, \pi/2]$
double acos (double); 结果介于 $[0, \pi]$
double atan (double); 反正切(主值), 结果介于 $[-\pi/2, \pi/2]$
double atan2 (double, double); 反正切(整圆值), 结果介于 $[-\pi/2, \pi/2]$
```

### 1. 双曲三角函数

```
double sinh (double);
double cosh (double);
double tanh (double);
```

### 1. 指数与对数

```
double exp (double);
double pow (double, double);
double sqrt (double);
double log (double); 以e为底的对数
double log10 (double); c++中自然对数函数： $\log(N)$ 以10为底： $\log_{10}(N)$ 但没有以2为底的函数但是可以用
```

### 1. 取整

```
double ceil (double); 取上整
double floor (double); 取下整
```

### 1. 绝对值

```
double fabs (double);
double abs (int);
```

1. 标准化浮点数

```
double frexp (double f, int *p); 标准化浮点数, $f = x * 2^p$, 已知f求x, p (x介于[0.5, 1])
double ldexp (double x, int p); 与frexp相反, 已知x, p求f
```

1. 取整与取余

```
double modf (double, double*); 将参数的整数部分通过指针回传, 返回小数部分
double fmod (double, double); 返回两参数相除的余数
```

1. 平方根

```
sqrt(int)
```

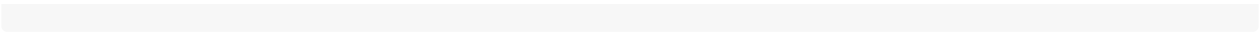
# cctype

| 函数名称       | 返回值                                                |
|------------|----------------------------------------------------|
| isalnum()  | 如果参数是字母数字, 即字母或数字, 该函数返回true                       |
| isalpha()  | 如果参数是字母, 该函数返回真                                    |
| isblank()  | 如果参数是空格或水平制表符, 该函数返回true                           |
| iscntrl()  | 如果参数是控制字符, 该函数返回true                               |
| isdigit()  | 如果参数是数字 (0~9), 该函数返回true                           |
| isgraph()  | 如果参数是除空格之外的打印字符, 该函数返回true                         |
| islower()  | 如果参数是小写字母, 该函数返回true                               |
| isprint()  | 如果参数是打印字符 (包括空格), 该函数返回true                        |
| ispunct()  | 如果参数是标点符号, 该函数返回true                               |
| isspace()  | 如果参数是标准空白字符, 如空格、进纸、换行符、回车、水平制表符或者垂直制表符, 该函数返回true |
| isupper()  | 如果参数是大写字母, 该函数返回true                               |
| isxdigit() | 如果参数是十六进制的数字, 即0~9、a~f、A~F, 该函数返回true              |
| tolower()  | 如果参数是大写字符, 则返回其小写, 否则返回该参数                         |
| toupper()  | 如果参数是小写字母, 则返回其大写, 否则返回该参数                         |

# string.h

寻找

```
char *strstr(char *str1, char *str2);
```



# 分 类

---



## 图论

---

- 使用vector来实现，或者使用new开辟新的数组
- [code](#)
- [无根树有根树转换](#)
- [邻接表存节点](#)
- [欧拉回路](#)
- [连通分量](#)
- [二分图判定](#)
- [无向图的割顶和桥](#)
- [无向图的双连通分量](#)
- [最短路Dijkstra](#)
- [负权最短路Bellman-Ford](#)
- [所有点最短路Floyd](#)

## 无根树有根树转换

---

```
// u为起点, f为父节点, d为深度
void dfs(int u, int f, int d)
{
 fa[u] = f;
 int nc = g[u].size();
 // 叶子节点插入
 if(nc == 1 && d > k) nodes[d].push_back(u);
 for(int i = 0; i < nc; i++) {
 int v = g[u][i];
 if(v != f) dfs(v, u, d+1);
 }
}
```

## 邻接表存节点

---

```
#include <iostream>
#include <vector>
#include <cstdio>
using namespace std;
//If G is a tree
//half of n is enough;

#define MAXN 1000
// N 为节点个数
typedef vector<int> vint;
vector<vint> G(MAXN);
//遍历使用clear()函数, 清空使用的空间

void Insert(int a, int i)
{
 G[a].push_back(i);
```

```

}

void DFS(int v)
{
 for(int i = 0; i < G[v].size(); i++)
 DFS(G[v][i]);
}

int main()
{
 G.clear();
 return 0;
}

//2.

// untest
const int maxn = 1000;
const int maxm = 1000; // 最大边数
int n, m;
int first[maxn];
int u[maxm], v[maxm], w[maxm], next[maxm];
void read_graph()
{
 scanf("%d%d", &n, &m);
 for(int i = 0; i < n; i++) first[i] = -1;
 for(int e = 0; e < m; e++)
 {
 scanf("%d%d%d", &u[e], &v[e], &w[e]);
 next[e] = first[u[e]];
 first[u[e]] = e;
 }
}

```

## 欧拉回路

判断这个图是否是一个能够遍历完所有的边而没有重复。

```

void euler(int u)
{
 for(int v = 0; v < n; v++) if(G[u][v] && !vis[u][v])
 {
 vis[u][v] = vis[v][u] = 1;
 euler(v);
 printf("%d %d\n", v, u);
 }
}

```

白书给的是printf u v, 需要逆序压栈。当然, 我们只需要输出就可以了。

## 连通分量

还可以使用并查集搞定, 不会爆栈

```

int current_cc;

void dfs(int u)
{
 vis[u] = 1;
 // PREvisit
 cc[u] = current_cc;
 int d = G[u].size();
 for(int i = 0; i < d; i++)
 {
 int v = G[u][i];
 if(!vis[v]) dfs(v);
 }
}

// 判断连通分量的个数
void find_cc()
{
 current_cc = 0;
 memset(vis, 0, sizeof(vis));
 for(int i = 0; i < n; i++)
 {
 current_cc++;
 dfs(i);
 }
}

```

## 二分图判定

二分图判定并且着色。把图分成不相关的两个部分

常与最大独立集共同

- untest

```

// 二分图的判定
// 初始化0, 黑色1, 白色2
const int maxn = 1000;
int color[maxn];
bool bipartite(int u) {
 for(int i = 0; i < G[u].size(); i++) {
 int v = G[u][i];
 if(color[v] == color[u]) return false; // 着色冲突
 if(!color[v]) {
 color[v] = 3 - color[u]; // 给结点v着相反颜色
 if(!bipartite(v)) return false;
 }
 }
}

```

## 独立集

最大独立集定义：从 $V$ 个点中选出 $k$ 个点，使得这 $k$ 个点互不相邻。那么最大的 $k$ 就是这个图的最大独立数。

1. 二分匹配得到最大匹配数（即最小覆盖数）
2. 最大独立集 = 顶点数 - 最小覆盖数

## 无向图的割顶和桥

- untest

无向图 $G$ ，删除点 $u$ ，连通分量数目增加，则 $u$ 为图的关节点（割顶）。对于连通图，割顶删除后图就不连通

iscut 为判定是否割点

```
// 无向图的割顶和桥

int dfs_clock;

void init()
{
 dfs_clock = 0;
 memset(pre, 0, sizeof(pre));
}

int dfs(int u, int fa) {
 int lowu = pre[u] = ++dfs_clock;
 int child = 0;
 for(int i = 0; i < G[u].size(); i++) {
 int v = G[u][i];
 if(!pre[v]) {
 child++;
 int lowv = dfs(v, u);
 lowu = min(lowu, lowv);
 if(lowv >= pre[u]) {
 iscut[u] = true;
 }
 }
 else if(pre[v] < pre[u] && v != fa) {
 lowu = min(lowu, pre[v]);
 }
 }
 if(fa < 0 && child == 1) iscut[u] = 0;
 low[u] = lowu;
 return lowu;
}
```

## 无向图的双连通分量

- untest

```
int pre[maxn], iscut[maxn], bccno[maxn], dfs_clock, bcc_cnt;
vector<int> G[maxn], bcc[maxn];
```

```

stack <Edge> s;

int dfs(int u, int fa)
{
 int lowu = pre[u] = ++dfs_clock;
 int child = 0;
 for(int i = 0; i < n ; i++)
 {
 int v = G[u][i];
 Edge e = (Edge){u, v};
 if(!pre[u]) { // 未访问
 S.push(e);
 child++;
 int lowv = dfs(v, u);
 lowu = min(lowu, lowv);
 if(lowv >= pre[u]) {
 iscut[u] = true;
 bcc_cnt++; bcc[bcc_cnt].clear();
 for(;;) {
 Edge x = S.top(); S.pop();
 if(bccno[x.u] != bcc_cnt) {
 bcc[bcc_cnt].push_back(x.u);
 bccno[x.u] = bcc_cnt;
 }
 if(bccno[x.v] != bcc_cnt) {
 bcc[bcc_cnt].push_back(x.v);
 bccno[x.v] = bcc_cnt;
 }
 if(x.u == u && x.v == v) break;
 }
 }
 }
 else if(pre[v] < pre[u] && v != fa)
 {
 S.push(e);
 lowu = min(lowu, pre[v]);
 }

 if(fa < 0 && child == 1) iscut[u] = 0;
 return lowu;
 }
}

// Init
void find_cc(int n)
{
 memset(pre, 0, sizeof(pre));
 memset(iscut, 0, sizeof(iscut));
 memset(bccno, 0, sizeof(bccno));
 dfs_clock = bcc_cnt = 0;
 for(int i = 0; i < n; i++)
 if(!pre[i]) dfs(i, -1);
}

```

## 最短路Dijkstra

- 2.untest

```
//1.
const int MAXN = 100010*3;
const int INF = 1 << 30;

struct HeapNode
{
 int d, u;
 HeapNode() { }
 HeapNode(int _d, int _u): d(_d), u(_u) { }
 bool operator<(const HeapNode& rhs) const
 {
 return d > rhs.d;
 }
};

struct Edge
{
 int from, to, dist;
 Edge() { }
 Edge(int f, int t, int d) : from(f), to(t), dist(d) { }
};

struct Dijkstra
{
 int n, m;
 vector<Edge> edges;
 vector<int> G[MAXN];
 bool done[MAXN];
 int d[MAXN], p[MAXN];

 void init(int n)
 {
 this->n = n;
 for (int i = 0; i <= n; ++i) G[i].clear();
 edges.clear();
 return;
 }

 void AddEdge(int from, int to, int dist)
 {
 edges.push_back(Edge(from, to, dist));
 m = edges.size();
 G[from].push_back(m - 1);
 return;
 }

 void dijkstra(int s)
 {
 priority_queue<HeapNode> Q;
 for (int i = 0; i <= n; ++i) d[i] = INF;
 d[s] = 0;
 memset(done, 0, sizeof(done));
 Q.push(HeapNode(0, s));
 while (!Q.empty())
 {
```

```

 HeapNode x = Q.top();
 Q.pop();
 int u = x.u;
 if (done[u]) continue;
 done[u] = true;
 for (int i = 0; i < (int)G[u].size(); ++i)
 {
 Edge& e = edges[G[u][i]];
 if (d[e.to] > d[u] + e.dist)
 {
 d[e.to] = d[u] + e.dist;
 p[e.to] = G[u][i];
 Q.push(HeapNode(d[e.to], e.to));
 }
 }
 }
 return;
}

};

//2.

const int maxn = 1000;

typedef pair<int, int> pii;

struct cmp
{
 bool operator() (const int a, const int b)
 {
 return a % 10 > b % 10;
 }
};

priority_queue <int, vector<int>, cmp> q;

void Dijkstra()
{
 bool done[maxn];
 for(int i = 0; i < n; i++) d[i] = (i == 0? 0:INF);
 memset(done, 0, sizeof(done));
 q.push(make_pair(d[0], 0));
 while(!q.empty())
 {
 pii u = q.top(); q.pop();
 int x = u.second;
 if(done[x]) continue;
 done[x] = true;
 for(int e = first[x]; e != -1; e = next[e]) if(d[v[e]] > d[x] + w[e])
 {
 d[v[e]] = d[x] + w[e];
 q.push(make_pair(d[v[e]], v[e]));
 }
 }
}

```

## 负权最短路Bellman-Ford

- untest

```
// 未使用优先队列的bellman
void bellman()
{
 for(int i = 0; i < n ;i++) d[i] = INF;
 d[0] = 0;
 for(int k = 0; k < n-1; k++)
 {
 for(int i = 0; i < m; i++)
 {
 int x= u[i], y= v[i];
 if(d[x] < INF) d[y] = min(d[y], d[x] + w[i]);
 }
 }
}

//使用有限队列的bellman
void bellman()
{
 queue <int> q;
 bool inq[maxn];
 for(int i = 0; i < n;i ++) d[i] = (i == 0? 0: INF);
 memset(inq, 0, sizeof(inq));
 q.push(0);
 while(!q.empty())
 {
 int x= q.front(); q.pop();
 inq[x] = false;
 for(int e = first[x]; e != -1; e = next[e]) if(d[v[e]] > d[x] + w[e])
 {
 d[v[e]] = d[x] + w[e];d
 if(!inq[v[e]])
 {
 inq[v[e]] = true;
 q.push(v[e]);
 }
 }
 }
}
```

## 所有点最短路Floyd



## 总纲

- [素数定理](#)
- [欧几里得公约数](#)
- [线性筛素数](#)
- [模算术](#)
- [欧拉函数筛法](#)
- [进制转换](#)
- [线性筛求莫比乌斯反演](#)
- [勾股数](#)
- [开方法](#)
- [中国剩余定理](#)

## 特殊数列

| 名称   | 数列                           |
|------|------------------------------|
| 欧拉函数 | 1, 1, 2, 2, 4, 2, 6, 4, 6, 4 |

## 素数定理

$\pi(x) \sim x/\ln x$  含义为不超过x的素数的个数，是一个近似值，比实际值要小

## 公约数

直接采用白书的算法了..

```

11n gcd(11n a, 11n b)
{
 return b == 0? a : gcd(b, a%b);
}

// ax + by = gcd(a, b)
// 无论d的初始值，d为gcd(a, b)
void gcd(11n a, 11n b, 11n &d, 11n &x, 11n &y)
{
 if(!b) { d = a; x = 1; y = 0; }
 else { gcd(b, a%b, d, y, x); y -= x*(a/b); }
}

```

## 线性筛素数

推荐使用2，代码相对清晰。

```
// 1.
```

```

#define N 10000
int prime[N];
bool isprime[N];

void makeprime()
{
 // 线性筛素数
 memset(isprime, 1, sizeof(isprime));
 int i, j;
 int num = 0;
 for(i = 2 ; i <= N; i++)
 {
 if(isprime[i])
 prime[num++] = i;
 for(j = 0; j < num && i * prime[j] <= N; j++)
 {
 isprime[i * prime[j]] = 0;
 if(i % prime[j] == 0) break;
 }
 }
}

// 2.
const int maxn = 10000000 + 10;
const int maxp = 700000;

int vis[maxn];
int prime[maxp];

// 筛素数
void sieve(int n)
{
 int m = (int) sqrt(n+0.5);
 memset(vis, 0, sizeof(vis));
 for(int i = 2; i <= m; i++) if(!vis[i])
 for(int j = i*i; j <= n; j+=i) vis[j] = 1;
}

// 生成素数表
int gen_primes(int n)
{
 sieve(n);
 int c = 0;
 for(int i = 2; i <= n; i++) if(!vis[i])
 prime[c++] = i;
 return c;
}

// 3. bitset速度打
const int N = 10000010;

bitset<N> prime;
LL phi[N];
LL f[N];
int p[N];
int k;

void isprime()

```

```

{
 k = 0;
 prime.set();
 for(int i=2; i<N; i++)
 {
 if(prime[i])
 {
 p[k++] = i;
 for(int j=i+i; j<N; j+=i)
 prime[j] = false;
 }
 }
}

```

## 模算术

### 1. 快速幂模

```

ll pow_mod(ll x, ll k) {
 ll ans = 1;
 x %= p;
 while (k) {
 if (k&1) ans = ans * x % p;
 x = x * x % p;
 k >>= 1;
 }
 return ans;
}

```

### 1. 另一种快速，含义相同

```

typedef long long ll;

ll mul_mod(ll a, ll b, int n)
{
 return a*b%n;
}

ll pow_mod(ll a, ll p, ll n)
{
 if(p == 0) return 1;
 ll ans = pow_mod(a, p/2, n);
 ans = ans * ans % n;
 if(p % 2 == 1) ans = ans * a % n;
 return ans;
}

```

### 1. 快速求前三位

更改2即可修改快速其他位数

```

int solve(int n, int k)

```

```
{
 double d = log10(n*1.0);
 int ans = (int)pow(10, 2 + fmod(k*d, 1));
 return ans;
}
```

## 整数位数

```
// 整数位数
int cal(int a)
{
 if(a == 0) return 0;
 return (int) log10((double) a)+1;
}
```

## 进制转换

除了以下方法以外，还有

```
printf("%x"); // 16
printf("%o"); // 8
printf("%d"); // 10
```

这些方法。更改X即可输出大写字母，增加ll或者l64就可以使用大范围。

## b进制转换为10进制

```
/**
 * c为数字，b为进制
 */
void trans(char *c, int b)
{
 // 输入数字不正确的可能处理没有添加
 int len = strlen(c);
 int j, temp, tt;
 int res = 0;
 temp = pow(b, len-1);
 for(j = 0; j < len; j++)
 {
 tt = c[j] - '0';
 res += tt * temp;
 temp /= b;
 }

 printf("%d\n", res);
}
```

## 十进制转换为b进制

```

/**
 * 模拟手算
 * @param b, c
 * b为int型进制, c为大数, 返回的数值为余数
 * @return rest
 */
int divide(int b, char *c)
{
 int len = strlen(c);
 int i;
 int temp, rest;
 rest = 0;
 for(i = 0; i < len; i++)
 {
 temp = c[i] - '0';
 rest = rest * 10 + temp;
 temp = rest / b;
 c[i] = temp + '0';
 rest = rest % b;
 }
 return rest;
}

// 判断是否为0
int eval(char *c, char b)
{
 int i = 0;
 int len = strlen(c);
 for(i = 0; i < len; i++)
 {
 if(c[i] != b)
 return 0;
 }
 return 1;
}

void transCHAR(int b, char *c)
{
 int a[100];
 int i = 0;
 /* test(); */
 /* output(c); */

 while(!eval(c, '0'))
 {
 a[i] = divide(b, c);
 i++;
 }

 /* printf("i: %d\n", i); */
 /* printf("b: %d\n", b); */
 /* printf("s: %s\n", c); */

 int j;
 for(j = i-1; j >= 0; j--)

```

```

 printf("%d", a[j]);
 printf("\n");
 }

```

## 欧拉函数筛法

```

// 计算单个phi
int euler_phi(int n)
{
 int m = (int) sqrt(n+0.5);
 int ans = n;
 for(int i = 2; i <= m; i++) if(n % i == 0) {
 ans = ans / i * (i-1);
 while(n % i == 0) n /= i;
 }
 if(n > 1) ans = ans / n * (n-1);
 return ans;
}

// 筛欧拉函数
const int maxn = 100000;
int phi[maxn];
void phi_table(int n)
{
 for(int i = 2; i <= n; i++) phi[i] = 0;
 phi[1] = 1;
 for(int i = 2; i <= n; i++) if(!phi[i])
 for(int j = i; j <= n; j += i) {
 if(!phi[j]) phi[j] = j;
 phi[j] = phi[j] / i * (i-1);
 }
}

```

## 线性筛求莫比乌斯反演

莫比乌斯反演:

1.  $F(n) = \sum_{d|n} f(d)$
2.  $f(n) = \sum_{d|n} u(d)F(n/d)$

$u(d)$  就是莫比乌斯反演函数

$u(d)$ 的特性:

1.  $d = 1, u(d) = 1$
2.  $d = p_1 p_2 p_3 \dots p_k$ ,  $p_i$ 均为素数, 那么 $u(d) = (-1)^k$
3. 其他情况 $u(d) = 0$
4.  $\sum_{d|n} u(d) = \{1, n=1; 0, n>1\}$
5.  $\sum_{d|n} u(d)/d = \text{euler}(n)/n$

```

void Init()
{
 memset(vis, 0, sizeof(vis));
 mu[1] = 1;
 cnt = 0;
 for(int i=2; i<N; i++)
 {
 if(!vis[i])
 {
 prime[cnt++] = i;
 mu[i] = -1;
 }
 for(int j=0; j<cnt&& i*prime[j]<N; j++)
 {
 vis[i*prime[j]] = 1;
 if(i%prime[j]) mu[i*prime[j]] = -mu[i];
 else
 {
 mu[i*prime[j]] = 0;
 break;
 }
 }
 }
}

```

## 勾股数

給一個任意數對(X,Y)，用以下公式代替

$A = X^2 - Y^2$   $B = 2XY$   $C = X^2 + Y^2$  得出的A,B,C就是一組勾股數。

若 (X,Y) 恰好互質而且一奇一偶，那麼會得到一組(A,B,C)互質的勾股數。

## 开方法

$(5/x+x)/2 = x$

循环判断x是否重复即可。

## 中国剩余定理

```

// mi两两互素
ll china(int n, int *a, int *m)
{
 ll M = 1, d, y, x = 0;
 for(int i = 0; i < n; i++) M *= m[i];
 for(int i = 0; i < n; i++)
 {
 ll w = M / m[i];
 gcd(m[i], w, d, d, y);
 x = (x + y*w*a[i]) % M;
 }
}

```

```
 }

 return (x+M)%M;
}
```



## 基础数据结构

- 大数
- 并查集
- 树状数组

## 大数

```
// 最大大小
const int MAXN = 100;

// 每个
const int DLEN = 100;

class BigNum
{
private:
 int a[12]; //可以控制大数的位数
 int len; //大数长度
public:
 BigNum(){ len = 1; memset(a,0,sizeof(a)); } //构造函数
 BigNum(const int); //将一个int类型的变量转化为大数
 BigNum(const char*); //将一个字符串类型的变量转化为大数
 BigNum(const BigNum &); //拷贝构造函数
 BigNum &operator=(const BigNum &); //重载赋值运算符，大数之间进行赋值运算

 friend istream& operator>>(istream&, BigNum&); //重载输入运算符
 friend ostream& operator<<(ostream&, BigNum&); //重载输出运算符

 BigNum operator+(const BigNum &) const; //重载加法运算符，两个大数之间的相加运算
 BigNum operator-(const BigNum &) const; //重载减法运算符，两个大数之间的相减运算
 BigNum operator*(const BigNum &) const; //重载乘法运算符，两个大数之间的相乘运算
 BigNum operator/(const int &) const; //重载除法运算符，大数对一个整数进行相除运算

 BigNum operator^(const int &) const; //大数的n次方运算
 int operator%(const int &) const; //大数对一个int类型的变量进行取模运算
 bool operator>(const BigNum & T) const; //大数和另一个大数的大小比较
 bool operator>(const int & t) const; //大数和一个int类型的变量的大小比较

 void print(); //输出大数
 int count(int n);
};

BigNum::BigNum(const int b) //将一个int类型的变量转化为大数
{
 int c, d = b;
 len = 0;
 memset(a,0,sizeof(a));
 while(d > MAXN)
 {
 c = d - (d / (MAXN + 1)) * (MAXN + 1);
 d = d / (MAXN + 1);
 a[len++] = c;
 }
}
```

```

 }
 a[len++] = d;
}

BigNum::BigNum(const char*s) //将一个字符串类型的变量转化为大数
{
 int t,k,index,l,i;
 memset(a,0,sizeof(a));
 l=strlen(s);
 len=l/DLEN;
 if(l%DLEN)
 len++;
 index=0;
 for(i=l-1;i>=0;i-=DLEN)
 {
 t=0;
 k=i-DLEN+1;
 if(k<0)
 k=0;
 for(int j=k;j<=i;j++)
 t=t*10+s[j]-'0';
 a[index++]=t;
 }
}

BigNum::BigNum(const BigNum & T) : len(T.len) //拷贝构造函数
{
 int i;
 memset(a,0,sizeof(a));
 for(i = 0 ; i < len ; i++)
 a[i] = T.a[i];
}

BigNum & BigNum::operator=(const BigNum & n) //重载赋值运算符，大数之间进行赋值运算
{
 int i;
 len = n.len;
 memset(a,0,sizeof(a));
 for(i = 0 ; i < len ; i++)
 a[i] = n.a[i];
 return *this;
}

istream& operator>>(istream & in, BigNum & b) //重载输入运算符
{
 char ch[MAXSIZE*4];
 int i = -1;
 in>>ch;
 int l=strlen(ch);
 int count=0,sum=0;
 for(i=l-1;i>=0;)
 {
 sum = 0;
 int t=1;
 for(int j=0;j<4&& i>=0;j++,i--,t*=10)
 {
 sum+=(ch[i]-'0')*t;
 }
 b.a[count]=sum;
 }
}

```

```

 count++;
 }
 b.len = count++;
 return in;
}

ostream& operator<<(ostream& out, BigNum& b) //重载输出运算符
{
 int i;
 cout << b.a[b.len - 1];
 for(i = b.len - 2 ; i >= 0 ; i--)
 {
 cout.width(DLEN);
 cout.fill('0');
 cout << b.a[i];
 }
 return out;
}

BigNum BigNum::operator+(const BigNum & T) const //两个大数之间的相加运算
{
 BigNum t(*this);
 int i, big; //位数
 big = T.len > len ? T.len : len;
 for(i = 0 ; i < big ; i++)
 {
 t.a[i] += T.a[i];
 if(t.a[i] > MAXN)
 {
 t.a[i + 1]++;
 t.a[i] -= MAXN + 1;
 }
 }
 if(t.a[big] != 0)
 t.len = big + 1;
 else
 t.len = big;
 return t;
}

BigNum BigNum::operator-(const BigNum & T) const //两个大数之间的相减运算
{
 int i, j, big;
 bool flag;
 BigNum t1, t2;
 if(*this > T)
 {
 t1 = *this;
 t2 = T;
 flag = 0;
 }
 else
 {
 t1 = T;
 t2 = *this;
 flag = 1;
 }
 big = t1.len;

```

```

 for(i = 0 ; i < big ; i++)
 {
 if(t1.a[i] < t2.a[i])
 {
 j = i + 1;
 while(t1.a[j] == 0)
 j++;
 t1.a[j--]--;
 while(j > i)
 t1.a[j--] += MAXN;
 t1.a[i] += MAXN + 1 - t2.a[i];
 }
 else
 t1.a[i] -= t2.a[i];
 }
 t1.len = big;
 while(t1.a[t1.len - 1] == 0 && t1.len > 1)
 {
 t1.len--;
 big--;
 }
 if(flag)
 t1.a[big-1] = 0 - t1.a[big-1];
 return t1;
}

BigNum BigNum::operator*(const BigNum & T) const //两个大数之间的相乘运算
{
 BigNum ret;
 int i,j,up;
 int temp,temp1;
 for(i = 0 ; i < len ; i++)
 {
 up = 0;
 for(j = 0 ; j < T.len ; j++)
 {
 temp = a[i] * T.a[j] + ret.a[i + j] + up;
 if(temp > MAXN)
 {
 temp1 = temp - temp / (MAXN + 1) * (MAXN + 1);
 up = temp / (MAXN + 1);
 ret.a[i + j] = temp1;
 }
 else
 {
 up = 0;
 ret.a[i + j] = temp;
 }
 }
 if(up != 0)
 ret.a[i + j] = up;
 }
 ret.len = i + j;
 while(ret.a[ret.len - 1] == 0 && ret.len > 1)
 ret.len--;
 return ret;
}

BigNum BigNum::operator/(const int & b) const //大数对一个整数进行相除运算

```

```

{
 BigNum ret;
 int i, down = 0;
 for(i = len - 1; i >= 0; i--)
 {
 ret.a[i] = (a[i] + down * (MAXN + 1)) / b;
 down = a[i] + down * (MAXN + 1) - ret.a[i] * b;
 }
 ret.len = len;
 while(ret.a[ret.len - 1] == 0 && ret.len > 1)
 ret.len--;
 return ret;
}

int BigNum::operator %(const int & b) const //大数对一个int类型的变量进行取模运算
{
 int i, d=0;
 for (i = len-1; i>=0; i--)
 {
 d = ((d * (MAXN+1))% b + a[i])% b;
 }
 return d;
}

BigNum BigNum::operator^(const int & n) const //大数的n次方运算
{
 BigNum t, ret(1);
 int i;
 if(n<0)
 exit(-1);
 if(n==0)
 return 1;
 if(n==1)
 return *this;
 int m=n;
 while(m>1)
 {
 t=*this;
 for(i=1; i<=m; i++)
 {
 t=t*t;
 }
 m-=i;
 ret=ret*t;
 if(m==1)
 ret=ret*(t);
 }
 return ret;
}

bool BigNum::operator>(const BigNum & T) const //大数和另一个大数的大小比较
{
 int ln;
 if(len > T.len)
 return true;
 else if(len == T.len)
 {
 ln = len - 1;
 while(a[ln] == T.a[ln] && ln >= 0)

```

```

 ln--;
 if(ln >= 0 && a[ln] > T.a[ln])
 return true;
 else
 return false;
 }
 else
 return false;
}

bool BigNum::operator >(const int & t) const //大数和一个int类型的变量的大小比较
{
 BigNum b(t);
 return *this>b;
}

void BigNum::print() //输出大数
{
 int i;
 cout << a[len - 1];
 for(i = len - 2 ; i >= 0 ; i--)
 {
 cout.width(DLEN);
 cout.fill('0');
 cout << a[i];
 }
 cout << endl;
}

int cc(int n)
{
 int a = 0;
 if(n / 1000 == 5)
 a++;
 n -= n/1000 * 1000;
 if(n / 100 == 5)
 a++;
 n -= n/100 * 100;
 if(n / 10 == 5)
 a++;
 n -= n/10 * 10;
 if(n == 5)
 a++;
 return a;
}

// 计算大数某个数字的个数
int BigNum::count(int n)
{
 int i;
 int sum;
 sum = cc(a[len-1]);
 for(i = len-2; i >= 0; i--)
 {
 sum += cc(a[i]);
 }
 cout << sum;
 return sum;
}

```

## 并查集

判断是否同一集合的数据结构，不过可以使用set来替代 -- 不过似乎就违背了初衷了

- 代码

```
/* from freecode */

#define MAXN 1010 //并查集的大小

int par[MAXN]; //par[x]表示x的父节点
int n;

void Init() //初始化
{
 int i;
 for(i=0;i<=n;i++)
 par[i] = i;
}

int Find(int x) //查询x的根节点并路径压缩
{
 if(par[x]!=x)
 par[x] = Find(par[x]);
 return par[x];
}

void Union(int x,int y) //合并x和y所在集合
{
 par[Find(x)] = Find(y);
}
```

## 树状数组

```
/* from freecode */
#define MAXN 100010

int N;
int c[MAXN];

int lowbit(int x)
{
 return x & (-x);
}

void add(int d,int x) //将编号为d的数加x
{
 while(d<=N){
 c[d] += x;
 d += lowbit(d);
 }
}
```

```

int sum(int d) //求出区间[1, d]元素的和
{
 int ans = 0;
 while(d >= 1){
 ans += c[d];
 d -= lowbit(d);
 }
 return ans;
}

```

## Karatsuba快速相乘算法

- 用于计算两个超大数字的快速相乘法则
- 来自 <http://blog.csdn.net/sunnyyoona/article/details/43234889>

```

/*****
 * 日期：2015-01-29
 * 作者：SJF0115
 * 题目：Karatsuba快速相乘算法
 * 博客：
 *****/
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;

// given two unequal sized bit strings, converts them to
// same length by adding leading 0s in the smaller string. Returns the
// the new length
int MakeSameLen(string& num1, string& num2){
 int len1 = num1.length();
 int len2 = num2.length();
 if(len1 < len2){
 for(int i = 0; i < len2 - len1; ++i){
 num1 = "0" + num1;
 }
 return len2;
 }
 else{
 for(int i = 0; i < len1 - len2; ++i){
 num2 = "0" + num2;
 }
 return len1;
 }
}

// big number minus function
string MinusString(string num1, string num2) {
 int len1 = num1.length();
 int len2 = num2.length();
 // 相等
 if(num1 == num2){
 return "0";
 }
 // 正负
 bool positive = true;

```



```

 if(len1 < len2 || (len1 == len2 && num1 < num2)){
 positive = false;
 // 交换使之num1 > num2
 string tmp = num1;
 num1 = num2;
 num2 = tmp;
 int temp = len1;
 len1 = len2;
 len2 = temp;
 }//if
 string result;
 int i = len1 - 1, j = len2 - 1;
 int a, b, sum, carray = 0;
 // 从低位到高位对位做减法
 while(i >= 0 || j >= 0){
 a = i >= 0 ? num1[i] - '0' : 0;
 b = j >= 0 ? num2[j] - '0' : 0;
 sum = a - b + carray;
 carray = 0;
 // 不够减
 if(sum < 0){
 sum += 10;
 carray = -1;
 }//if
 result.insert(result.begin(), sum + '0');
 --i;
 --j;
 }//while
 // 删除前导0
 string::iterator it = result.begin();
 while(it != result.end() && *it == '0'){
 ++it;
 }//while
 result.erase(result.begin(), it);
 return positive ? result : "-" + result;
}
// big number add function
string AddString(string num1, string num2){
 int len1 = num1.length();
 int len2 = num2.length();
 // 容错处理
 if(len1 <= 0){
 return num2;
 }//if
 if(len2 <= 0){
 return num1;
 }
 string result;
 int i = len1 - 1, j = len2 - 1;
 int a, b, sum, carry = 0;
 // 倒序相加
 while(i >= 0 || j >= 0 || carry > 0){
 a = i >= 0 ? num1[i] - '0' : 0;
 b = j >= 0 ? num2[j] - '0' : 0;
 // 按位相加并加上进位
 sum = a + b + carry;
 // 进位
 carry = sum / 10;
 result.insert(result.begin(), sum % 10 + '0');
 }
}

```

```

 --i;
 --j;
 }//while
 return result;
}
// 移位
string ShiftString(string num,int len){
 if(num == "0"){
 return num;
 }//if
 for(int i = 0;i < len;++i){
 num += "0";
 }//for
 return num;
}
// Karatsuba快速相乘算法
string KaratsubaMultiply(string num1, string num2) {
 int len = MakeSameLen(num1,num2);
 if(len == 0){
 return 0;
 }//if
 // all digit are one
 if(len == 1){
 return to_string((num1[0] - '0')*(num2[0] - '0'));
 }//if
 int mid = len / 2;
 // Find the first half and second half of first string.
 string x1 = num1.substr(0,mid);
 string x0 = num1.substr(mid,len - mid);
 // Find the first half and second half of second string
 string y1 = num2.substr(0,mid);
 string y0 = num2.substr(mid,len - mid);
 // Recursively computer
 string z0 = KaratsubaMultiply(x0,y0);
 string z1 = KaratsubaMultiply(AddString(x1,x0),AddString(y1,y0));
 string z2 = KaratsubaMultiply(x1,y1);
 // (z2*10^(2*m)) + ((z1-z2-z0)*10^(m)) + (z0)
 // z2*10^(2*m)
 string r1 = ShiftString(z2,2*(len - mid));
 // (z1-z2-z0)*10^(m)
 string r2 = ShiftString(MinusString(MinusString(z1,z2),z0),len - mid);
 return AddString(AddString(r1,r2),z0);
}

int main(){
 string num1("12345001");
 string num2("1006789");
 string result = KaratsubaMultiply(num1,num2);
 // 输出
 cout<<result<<endl;
 return 0;
}

```

## 搜索的相关知识

---

- 深搜DFS
- 广搜BFS
- 搜索树的应用

## 计算几何

- [二维几何基础](#)
- [圆和球](#)
- [欧拉四面体公式](#)
- [扩展模板](#)
  - [计算几何经典模板](#)
  - [计算几何模板](#)

## 二维几何基础

```

/*=====*\
| 计算几何基础函数 |
| 1.点和向量的定义 |
| 2.向量的基本运算 |
| 3.点积 |
| 4.向量长度 |
| 5.两向量角度 |
| 6.叉积 (2向量/3点) |
| 7.向量旋转 |
| 8.向量的单位法线 |
| 9.求两点距离 |
| 10.直线 (射线) 交点 |
| 11.点到直线的距离 |
| 12.点到线段的距离 |
| 13.点在直线上的投影 |
| 14.线段相交判定 (规范相交)|
| 15.点是否在一条线段上 |
| 16.求多边形面积 |
| 17.判断点是否在多边形内 |
| 18.求凸包 |
| 19.求凸包周长 |
| 20.求多边形面积 |
=====/

#include <iostream>
#include <cmath>
using namespace std;
#define eps 1e-10

/***** 定义点 *****/
struct Point{
 double x,y;
 Point(double x=0,double y=0):x(x),y(y) {}
};

/***** 定义向量 *****/
typedef Point Vector;
/***** 向量 + 向量 = 向量 *****/
Vector operator + (Vector a,Vector b)
{
 return Vector(a.x+b.x,a.y+b.y);
}

```

```

/***** 点 - 点 = 向量 *****/
Vector operator - (Point a,Point b)
{
 return Vector(a.x-b.x,a.y-b.y);
}
/***** 向量 * 数 = 向量 *****/
Vector operator * (Vector a,double b)
{
 return Vector(a.x*b,a.y*b);
}
/***** 向量 / 数 = 向量 *****/
Vector operator / (Vector a,double b)
{
 return Vector(a.x/b,a.y/b);
}
bool operator < (const Point& a,const Point& b)
{
 return a.x<b.x || (a.x==b.x && a.y<b.y);
}
int dcmp(double x) //减少精度问题
{
 if(fabs(x)<eps)
 return 0;
 else
 return x<0?-1:1;
}
bool operator == (const Point &a,const Point &b) //判断两点是否相等
{
 return dcmp(a.x-b.x)==0 && dcmp(a.y-b.y)==0;
}
/***** 向量点积 *****/
double Dot(Vector a,Vector b)
{
 return a.x*b.x+a.y*b.y;
}
/***** 向量长度 *****/
double Length(Vector A)
{
 return sqrt(Dot(A,A));
}
/***** 两向量角度 *****/
double Angle(Vector A,Vector B)
{
 return acos(Dot(A,B)/Length(A)/Length(B));
}

/***** 2向量求叉积 *****/
double Cross(Vector a,Vector b)
{
 return a.x*b.y-b.x*a.y;
}
/***** 3点求叉积 *****/
double Cross(Point a,Point b,Point c)
{
 return (c.x-a.x)*(b.y-a.y) - (c.y-a.y)*(b.x-a.x);
}
/***** 向量旋转 *****/
Vector Rotate(Vector A,double rad)
{

```

```

 return Vector(A.x*cos(rad)-A.y*sin(rad),A.x*sin(rad)+A.y*cos(rad));
 }
 /***** 向量的单位法线 *****/
 Vector Normal(Vector A)
 {
 double L = Length(A);
 return Vector(-A.y/L,A.x/L);
 }
 /***** 点和直线 *****/
 /***** 求两点间距离 *****/
 double dist(Point a,Point b)
 {
 return sqrt((a.x-b.x)*(a.x-b.x) + (a.y-b.y)*(a.y-b.y));
 }
 /***** 直线交点 *****/
 Point GetLineIntersection(Point P,Vector v,Point Q,Vector w)
 {
 Vector u = P-Q;
 double t = Cross(w,u) / Cross(v,w);
 return P+v*t;
 }
 /***** 点到直线的距离 *****/
 double DistanceToLine(Point P,Point A,Point B)
 {
 Vector v1 = B-A,v2 = P-A;
 return fabs(Cross(v1,v2)) / Length(v1); //如果不取绝对值，得到的是有向距离
 }
 /***** 点到线段的距离 *****/
 double DistanceToSegment(Point P,Point A,Point B)
 {
 if(A==B) return Length(P-A);
 Vector v1 = B-A,v2 = P-A,v3 = P-B;
 if(dcmp(Dot(v1,v2))<0) return Length(v2);
 else if(dcmp(Dot(v1,v3)) > 0) return Length(v3);
 else return fabs(Cross(v1,v2)) / Length(v1);
 }
 /***** 点在直线上的投影 *****/
 Point GetLineProjection(Point P,Point A,Point B)
 {
 Vector v = B-A;
 return A+v*(Dot(v,P-A) / Dot(v,v));
 }
 /***** 线段相交判定（规范相交） *****/
 bool SegmentProperIntersection(Point a1,Point a2,Point b1,Point b2)
 {
 double c1 = Cross(a2-a1,b1-a1),c2 = Cross(a2-a1,b2-a1),
 c3 = Cross(b2-b1,a1-b1),c4 = Cross(b2-b1,a2-b1);
 return dcmp(c1)*dcmp(c2)<0 && dcmp(c3)*dcmp(c4)<0;
 }
 /***** 点是否在一条线段上 *****/
 bool OnSegment(Point p,Point a1,Point a2)
 {
 return dcmp(Cross(a1-p,a2-p)) == 0 && dcmp(Dot(a1-p,a2-p)) <0 ;
 }
 /***** 求多边形面积 *****/
 double ConvexPolygonArea(Point* p,int n)
 {
 double area = 0;
 for(int i=1;i<n-1;i++)

```

```

 area += Cross(p[i]-p[0],p[i+1]-p[0]);
 }
 return area/2;
}

/***** 判断点是否在多边形内 *****/
//判断点q是否在多边形内
//任意凸或者凹多边形
//顶点集合p[]按逆时针或者顺时针顺序存储(1..pointnum)
struct Point{
 double x,y;
};

struct Line{
 Point p1,p2;
};

double xmulti(Point p1,Point p2,Point p0) //求p1p0和p2p0的叉积,如果大于0,则p1在p2的顺时针方
{
 return (p1.x-p0.x)*(p2.y-p0.y)-(p2.x-p0.x)*(p1.y-p0.y);
}
double Max(double a,double b)
{
 return a>b?a:b;
}
double Min(double a,double b)
{
 return a<b?a:b;
}
bool ponls(Point q,Line l) //判断点q是否在线段l上
{
 if(q.x > Max(l.p1.x,l.p2.x) || q.x < Min(l.p1.x,l.p2.x)
 || q.y > Max(l.p1.y,l.p2.y) || q.y < Min(l.p1.y,l.p2.y))
 return false;
 if(xmulti(l.p1,l.p2,q)==0) //点q不在l的延长线或者反向延长线上,如果叉积再为0,则确定点q在线段
 return true;
 else
 return false;
}
bool pinplg(int pointnum,Point p[],Point q)
{
 Line s;
 int c = 0;
 for(int i=1;i<=pointnum;i++){ //多边形的每条边s
 if(i==pointnum)
 s.p1 = p[pointnum],s.p2 = p[1];
 else
 s.p1 = p[i],s.p2 = p[i+1];
 if(ponls(q,s)) //点q在边s上
 return true;
 if(s.p1.y != s.p2.y){ //s不是水平的
 Point t;
 t.x = q.x - 1,t.y = q.y;
 if((s.p1.y == q.y && s.p1.x <=q.x) || (s.p2.y == q.y && s.p2.x <= q.x)){
 int tt;
 if(s.p1.y == q.y)
 tt = 1;
 else if(s.p2.y == q.y)

```

```

 tt = 2;
 int maxx;
 if(s.p1.y > s.p2.y)
 maxx = 1;
 else
 maxx = 2;
 if(tt == maxx) //如果这个端点的纵坐标较大的那个端点
 c++;
 }
 else if(xmulti(s.p1,t,q)*xmulti(s.p2,t,q) <= 0){ //L和边s相交
 Point lowp,higp;
 if(s.p1.y > s.p2.y)
 lowp.x = s.p2.x,lowp.y = s.p2.y,higp.x = s.p1.x,higp.y = s.p1.y;
 else
 lowp.x = s.p1.x,lowp.y = s.p1.y,higp.x = s.p2.x,higp.y = s.p2.y;
 if(xmulti(q,higp,lowp)>=0)
 c++;
 }
}
}
if(c%2==0)
 return false;
else
 return true;
}
/***** 求凸包 *****/
struct Point{
 double x,y;
};
double dis(Point p1,Point p2)
{
 return sqrt((p1.x-p2.x)*(p1.x-p2.x)+(p1.y-p2.y)*(p1.y-p2.y));
}
double xmulti(Point p1,Point p2,Point p0) //求p1p0和p2p0的叉积,如果大于0,则p1在p2的顺时针方
{
 return (p1.x-p0.x)*(p2.y-p0.y)-(p2.x-p0.x)*(p1.y-p0.y);
}
int graham(Point p[],int n,int p1[]) //点集,点的个数,凸包顶点集
{
 int p1[10005];
 //找到纵坐标(y)最小的那个点,作第一个点
 int t = 1;
 for(int i=1;i<=n;i++)
 if(p[i].y < p[t].y)
 t = i;
 p1[1] = t;
 //顺时针找到凸包点的顺序,记录在 int p1[]
 int num = 1; //凸包点的数量
 do{ //已确定凸包上num个点
 num++; //该确定第 num+1 个点了
 t = p1[num-1]+1;
 if(t>n) t = 1;
 for(int i=1;i<=n;i++){ //核心代码。根据叉积确定凸包下一个点。
 double x = xmulti(p[i],p[t],p[p1[num-1]]);
 if(x<0) t = i;
 }
 p1[num] = t;
 } while(p1[num]!=p1[1]);
 return num-1; //凸包顶点数
}

```



```

}
/***** 求凸包周长 *****/
struct Point{
 double x,y;
};
double dis(Point p1,Point p2)
{
 return sqrt((p1.x-p2.x)*(p1.x-p2.x)+(p1.y-p2.y)*(p1.y-p2.y));
}
double xmulti(Point p1,Point p2,Point p0) //求p1p0和p2p0的叉积,如果大于0,则p1在p2的顺时针方
{
 return (p1.x-p0.x)*(p2.y-p0.y)-(p2.x-p0.x)*(p1.y-p0.y);
}
double graham(Point p[],int n) //点集和点的个数
{
 int pl[10005];
 //找到纵坐标 (y) 最小的那个点, 作第一个点
 int t = 1;
 for(int i=1;i<=n;i++)
 if(p[i].y < p[t].y)
 t = i;
 pl[1] = t;
 //顺时针找到凸包点的顺序, 记录在 int pl[]
 int num = 1; //凸包点的数量
 do{ //已确定凸包上num个点
 num++; //该确定第 num+1 个点了
 t = pl[num-1]+1;
 if(t>n) t = 1;
 for(int i=1;i<=n;i++){ //核心代码。根据叉积确定凸包下一个点。
 double x = xmulti(p[i],p[t],p[pl[num-1]]);
 if(x<0) t = i;
 }
 pl[num] = t;
 } while(pl[num]!=pl[1]);
 //计算凸包周长
 double sum = 0;
 for(int i=1;i<num;i++)
 sum += dis(p[pl[i]],p[pl[i+1]]);
 return sum;
}
/***** 求多边形面积 *****/
struct Point{ //定义点结构
 double x,y;
};
double getS(Point a,Point b,Point c) //返回三角形面积
{
 return ((b.x - a.x) * (c.y - a.y) - (b.y - a.y)*(c.x - a.x))/2;
}
double getPS(Point p[],int n) //返回多边形面积。必须确保 n>=3, 且多边形是凸多边形
{
 double sumS=0;
 for(int i=1;i<=n-1;i++)
 sumS+=getS(p[1],p[i],p[i+1]);
 return sumS;
}

```

## 圆和球

```

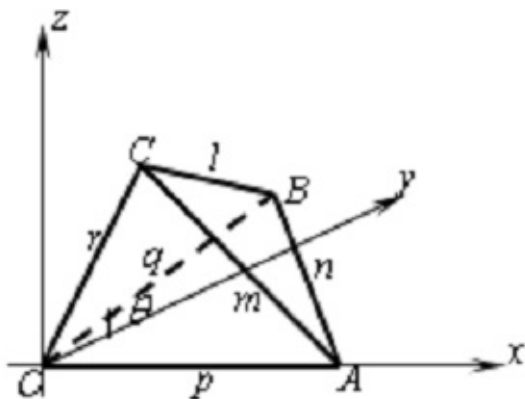
#include <iostream>
#include <cmath>
using namespace std;
#define eps 1e-10
/***** 定义点 *****/
struct Point{
 double x,y;
 Point(double x=0,double y=0):x(x),y(y) {}
};
/***** 定义三维点 *****/
struct Point3{
 double x,y,z;
 Point3(double x=0,double y=0,double z=0):x(x),y(y),z(z) {}
};
/***** 定义圆 *****/
struct Circle{
 Point c;
 double r;
 Circle(Point c,double r):c(c),r(r){}
 Point point(double a){
 return Point(c.x + cos(a)*r,c.y + sin(a)*r);
 }
};
/***** 三维点距离 *****/
double dis3(Point3 A,Point3 B)
{
 return sqrt((A.x-B.x)*(A.x-B.x)+(A.y-B.y)*(A.y-B.y)+(A.z-B.z)*(A.z-B.z));
}
/***** 球面 *****/
/***** 角度转换成弧度 *****/
double torad(double deg)
{
 return deg/180 * acos(-1); //acos(-1)就是PI
}
/***** 经纬度（角度）转化为空间坐标 *****/
void get_coord(double R,double lat,double lng,double &x ,double &y,double &z)
{
 lat = torad(lat); //纬度
 lng = torad(lng); //经度
 x = R*cos(lat)*cos(lng);
 y = R*cos(lat)*sin(lng);
 z = R*sin(lat);
}
/***** 两点的球面距离 *****/
double disA2B(double R,Point3 A,Point3 B)
{
 //将球面距离看成求点A, B和半径R构成的扇形的弧长
 double d = dis3(A,B); //弦长
 double a = 2*asin(d/2/R); //圆心角
 double l = a*R; //弧长
 return l;
}

```

## 欧拉四面体公式

引用声明，来自：<http://blog.csdn.net/archibaldyangfan/article/details/8035332>

1, 建立x, y, z直角坐标系。设A、B、C点的坐标分别为 $(a_1, b_1, c_1), (a_2, b_2, c_2), (a_3, b_3, c_3)$ , 四面体O-ABC的六条棱长分别为l, m, n, p, q, r;



2, 四面体的体积为, 由于现在不知道向量怎么打出来, 我就插张图片了,

$$V = \frac{1}{6} (\overrightarrow{OA} \times \overrightarrow{OB}) \cdot \overrightarrow{OC} = \frac{1}{6} \begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix}$$

将这个式子平方后得到:

$$\begin{aligned} V^2 &= \frac{1}{36} \begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix} \begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix} \\ &= \frac{1}{36} \begin{vmatrix} a_1^2 + b_1^2 + c_1^2 & a_1 a_2 + b_1 b_2 + c_1 c_2 & a_1 a_3 + b_1 b_3 + c_1 c_3 \\ a_1 a_2 + b_1 b_2 + c_1 c_2 & a_2^2 + b_2^2 + c_2^2 & a_2 a_3 + b_2 b_3 + c_2 c_3 \\ a_1 a_3 + b_1 b_3 + c_1 c_3 & a_2 a_3 + b_2 b_3 + c_2 c_3 & a_3^2 + b_3^2 + c_3^2 \end{vmatrix}. (1) \end{aligned}$$

3, 根据矢量数量积的坐标表达式及数量积的定义得

$$a_1^2 + b_1^2 + c_1^2 = \overrightarrow{OA} \cdot \overrightarrow{OA} = |\overrightarrow{OA}|^2 = p^2,$$

$$a_2^2 + b_2^2 + c_2^2 = \overrightarrow{OB} \cdot \overrightarrow{OB} = |\overrightarrow{OB}|^2 = q^2,$$

$$a_3^2 + b_3^2 + c_3^2 = \overrightarrow{OC} \cdot \overrightarrow{OC} = |\overrightarrow{OC}|^2 = r^2,$$

又根据余弦定理得

$$a_1a_2 + b_1b_2 + c_1c_2 = \overrightarrow{OA} \cdot \overrightarrow{OB} = p \cdot q \cdot \cos(\hat{p}, q) = \frac{p^2 + q^2 - n^2}{2},$$

$$a_1a_3 + b_1b_3 + c_1c_3 = \overrightarrow{OA} \cdot \overrightarrow{OC} = p \cdot r \cdot \cos(\hat{p}, r) = \frac{p^2 + r^2 - m^2}{2},$$

$$a_2a_3 + b_2b_3 + c_2c_3 = \overrightarrow{OB} \cdot \overrightarrow{OC} = q \cdot r \cdot \cos(\hat{q}, r) = \frac{q^2 + r^2 - l^2}{2}.$$

4, 将上述的式子带入 (1), 就得到了传说中的欧拉四面体公式

$$V^2 = \frac{1}{36} \begin{vmatrix} p^2 & \frac{p^2 + q^2 - n^2}{2} & \frac{p^2 + r^2 - m^2}{2} \\ \frac{p^2 + q^2 - n^2}{2} & q^2 & \frac{q^2 + r^2 - l^2}{2} \\ \frac{p^2 + r^2 - m^2}{2} & \frac{q^2 + r^2 - l^2}{2} & r^2 \end{vmatrix}$$

## 博弈问题

### 最常见的SG博弈：


1. 二人博弈
2. 对于同一局面，两者可以操作的集合完全相同
3. 游戏总可以在有限步数内结束。
4. 假定2人都"足够聪明"
5. 游戏无法操作时，不能操作的一方为负。也可能有不能操作为胜的局面

### 取石子问题：

1. 取石子1: 有一堆 $n$ 个石子，两个人轮流取，每次可以取 $1..m$ 个，没得取判负。
2. 取石子2: 有 $k$ 堆石子，每堆 $n_i$ 个，两个轮流取，每次可以从某一堆中取 $1..m$ 个 没得取判负。

### N局面和P局面：

- 必须在有限步内结束
- 双方都猜取最佳策略的前提下，任一局面不是先手胜利就是后手胜利。
- 规定：先手胜为N局面，后手胜利为P局面
- 此类问题一般问某一个局面时先手必胜还是先手必败。
  - 附加问题:先手必胜时，输出当前一步可以选步
- 最终局面都是P局面
- 对于一个局面，至少有一种操作是他变成P局面，则他是N局面
- 对于一个局面，无论如何操作都使他变成N局面，则他是P局面
- 例如：取石子问题
  - 有一堆 $n$ 个石子，两人轮流取，每次可以去 $1..m$ 个，没得取的判负
  - 在游戏中， $n$ 不断变化， $m$ 是定植
  - 一个局面可以用当前的石子数 $n$ 来表示
  - $n = 0$ 必败  $n = 1..m$ 都是必胜局面  $n = m+1$  是必败局面  $n = m+2 .. 2m+1$  是必胜局面 将博弈游戏看作图

- 将游戏的中间状态看作顶点
  - 性质2：对于同一局面，两个游戏者可操作的集合完全相同。点对当前的执行者是无差别的。
- 将状态转移看作边
  - 性质三：游戏总可以在有限步之内结束 这是一个有向无环图 

顶点必胜必败态 --- 每个顶点都会对应N局面必胜或者P局面必败

- N点（必胜点）或者P点（必败点）

- 如果顶点有边指向P点，则该点为N点.

- 如果顶点没有边指向P点，则该点为P点。

## 判断先手必胜还是必败

- 图是[有向无环图]
- 通过拓扑排序对图的每个点进行染色，从而确定每个点的状态
  - 必胜时要输出一个可行解：寻找该状态对应的N节点指向P节点的那条边。
- 算法复杂度和状态数有关
- 一堆石子的状态数为N
- K堆石子的状态是一个K维向量
  - 状态数为  $n_1 * n_2 * \dots * n_k$
  - 过多。

## SG函数

- 对顶点赋一个SG值
- 设顶点V，有  $V \rightarrow v_1, v \rightarrow v_2, \dots v \rightarrow v_t$
- $sg(v) = \min(N - \{sg(v_1), sg(v_2), \dots sg(v_t)\})$
- $sg(v)$  定义为没有出现在  $\{sg(v_1) \dots sg(v_t)\}$  中的最小自然数
- 同样可以用拓扑序对节点计算sg值

## SG函数与NP局面的关系

- 初始节点（必须是P节点）没有出边，  $sg = 0$
- 若V有边指向某  $sg(v_i) = 0$  的  $v_i$ ，则  $sg(v) > 0$
- 若v没有边指向某  $sg(v_i) = 0$  的  $v_i$ ，则  $sg(v) = 0$
- 所有P节点  $sg=0$
- 所有N节点  $sg > 0$

## 取石子游戏

- 一维：只有一堆石子
- n先手必胜当且仅当  $sg(n) > 0$
- 二维的石子游戏：两堆石子  $n_1, n_2$
- 状态： $\setminus$
- $sg(\setminus) = ?$
- 断言： $sg(\setminus) = sg(n_1) \wedge sg(n_2)$ 
  - 两个一维sg函数的异或

## $sg() = sg(n_1) \wedge sg(n_2)$

- 证明要点： $sg(\setminus) = sg(n_1) \wedge sg(n_2)$  满足sg函数的性质
- 若  $sg(\setminus) = x$ ，则存在操作使得  $\setminus \rightarrow \setminus$  且  $sg(\setminus) = 0 \dots x-1$
- 若  $sg(\setminus) = x$  不存在操作使得  $\setminus \rightarrow \setminus$  且  $sg(\setminus) = x$



## 字符串处理

- 比较字典序
- 字符串反转
- 获取整行
- KMP

### 比较字典序

```
template <class T>
bool lexicographicallySmaller(vector <T> a, vector <T> b)
{
 int n = a.size();
 int m = b.size();
 int i;
 for(i = 0; i < n && i < m; i++)
 {
 if(a[i] < b[i]) return true;
 else if(a[i] > b[i]) return false;
 }
 return (i == n && i < m);
}
```

### 字符串反转

主要涉及了对一个数字的反转

```
int b;
sscanf(num, "%d", &b);
for(int i = 0; i < len/2 ;i ++)
{
 char t = num[i];
 num[i] = num[len-1-i];
 num[len-1-i] = t;
}
sscanf(num, "%d", &x);
```

### 获取整行

- 使用 `cin.getline(str, num, endchar)` 函数
- 另外，如果不加上endchar则默认为'\n'

如果使用 `fgets(str, num, stdin)` 的话：

1. 没有读到\n是不会换行的



2. 没有读完当前行是不会换行的，即\$当前行的字符数 > num\$
3. 此外，这个函数会将\n读入到str中

## KMP

用的是白书kmp，此外P为短串，T为长串，f为next数组

KMP 本质也是自动机，此处为MP算法，处理失效函数才是KMP

```
// 生成next数组，即为f数组
void getFail(char *P, int *f)
{
 int m = strlen(P);
 f[0] = 0; f[1] = 0;
 for(int i = 1; i < m; i++)
 {
 int j = f[i];
 while(j && P[i] != P[j]) j = f[j];
 f[i+1] = P[i] == P[j] ? j+1 : 0;
 }
}

int find(char *T, char *P, int *f)
{
 int n = strlen(T), m = strlen(P);
 getFail(P, f);
 int j = 0;
 for(int i = 0; i < n; i++) {
 while(j && P[j] != T[i]) j = f[j];
 if(P[i] == T[i]) j++;
 if(j == m) return i-m+1; // 找到了
 }
 return -1; // 没找到
}
```

## 动态规划

各种初步动态规划的复杂度，以及状态转移方程

## 数字三角形

```
// d(i, j) = a(i, j) + max{d(i, j), d(i+1, j+1)}
void cal(int n)
{
 for(int i = n-1; i >= 0; i--)
 for(int j = 0; j <= i; j++)
 dp[i][j] = a[i][j] + max(dp[i+1][j], dp[i+1][j+1]);
}
```

## 嵌套矩形

## 硬币问题

## 01背包问题

思路：使用最大价值的子集，尽可能的填满空集，求最大价值

dp方程（经过滚动优化）：

$$dp[v] = \max(dp[v], dp[v - w[i]] + v[i])$$

## 点集配对问题

## 最长上升子序列问题（LIS）

- $d[i]$ 表示以下标 $i$ 结尾的最长上升子序列的长度
- $g[i]$ 表示 $d$ 值为 $i$ 的最小状态编号 如此一来，可以通过下列方法更新求出最大值.  $g$ 中会把数列 $A$ 中的小数不停的排在 $g$ 里面，换个思路：

将  $A$ 数列中的数字放在 $g$ 中，如果大放在最后面，如果不大就代替掉前面的数字，每次返回的 $k$ 是上升子序列的长度，每次都与最大值 $ans$ 进行判断，算法的复杂度为 $\$n\log n\$$ 。

```
for(int i = 1; i <= n; i++) g[i] = INF;
int ans = 0;
for(int i = 0; i < n; i++) {
 int k = lower_bound(g+1, g+n+1, S[i]) - g;
 d[i] = k;
 g[k] = S[i];
 ans = max(ans, k);
}
```

```
}
```

## 最长公共子序列问题（LCS）

---

## 货担郎问题(TSP)

---

## 矩阵链乘(MCM)

---

$$f(i,j) = \max\{f(i,k) + f(k,j) + p_{i-1}p_kp_j\}$$

需要按照j-i递增的方式递推。记忆化搜索没有问题。

## 最有排序二叉树问题（OBST）

---

本文出自[Svitter](#)的blog

本文出自[Svitter的blog](#)

- 排列

## STL的相关模板

---

- [set](#)
- [map](#)
- [queue](#)
- [stack](#)
- [sort](#)
- [unique](#)
- [二分查找](#)
- [next\\_permutation\(\)](#)

### set

---

- [STL之set集合容器（转）](#)
- [STL中的set容器的一点总结（转）](#)

### map

---

- [STL中map用法详解（转）](#)

### queue

---

```
queue <vector> q;

q.push();
q.pop();
```

### set

---

```
set <vector> s;

// 清空
s.clear();

// 寻找v
s.count(v);

// 添加
s.insert(v);

// 示例
#include <set>
#include <iterator>
#include <iostream>
```

```

using namespace std;
int main()
{
 set<int>eg1;
 //插入
 eg1.insert(1);
 eg1.insert(100);
 eg1.insert(5);
 eg1.insert(1); //元素1因为已经存在所以set中不会再次插入1
 eg1.insert(10);
 eg1.insert(9);
 //遍历set, 可以发现元素是有序的
 set<int>::iterator set_iter=eg1.begin();
 cout<<"Set named eg1:"<<endl;
 for(;set_iter!=eg1.end();set_iter++) cout<<*set_iter<<" ";
 cout<<endl;
 //使用size()函数可以获得当前元素个数
 cout<<"Now there are "<<eg1.size()<<" elements in the set eg1"<<endl;
 if(eg1.find(200)==eg1.end())//find()函数可以查找元素是否存在
 cout<<"200 isn't in the set eg1"<<endl;

 set<int>eg2;
 for(int i=6;i<15;i++)
 eg2.insert(i);
 cout<<"Set named eg2:"<<endl;
 for(set_iter=eg2.begin();set_iter!=eg2.end();set_iter++)
 cout<<*set_iter<<" ";
 cout<<endl;
 //获得两个set的并
 set<int>eg3;
 cout<<"Union:";
 set_union(eg1.begin(),eg1.end(),eg2.begin(),eg2.end(),insert_iterator<set<int>>(eg3,
 copy(eg3.begin(),eg3.end(),ostream_iterator<int>(cout," ")));
 cout<<endl;
 //获得两个set的交, 注意进行集合操作之前接收结果的set要调用clear()函数清空一下
 eg3.clear();
 set_intersection(eg1.begin(),eg1.end(),eg2.begin(),eg2.end(),insert_iterator<set<int>>
 cout<<"Intersection:";
 copy(eg3.begin(),eg3.end(),ostream_iterator<int>(cout," "));
 cout<<endl;
 //获得两个set的差
 eg3.clear();
 set_difference(eg1.begin(),eg1.end(),eg2.begin(),eg2.end(),insert_iterator<set<int>> >
 cout<<"Difference:";
 copy(eg3.begin(),eg3.end(),ostream_iterator<int>(cout," "));
 cout<<endl;
 //获得两个set的对称差, 也就是假设两个集合分别为A和B那么对称差为AUB-A∩B

```

## Sort

qsort中cmp的写法

```

int cmp_char(const void * _a, const void * _b)
{

```

```

 char * a = (char*) _a;
 char * b = (char*) _b;
 return *a - *b;
}

int cmp_string(const void *_a, const void *_b)
{
 char * a = (char*) _a;
 char * b = (char*) _b;
 return strcmp(a, b);
}

```

sort中直接放数据类型即可

## unique

去重函数。返回last元素。事实上并没有删除，仅仅是将重复元素移到了数组的末端

```
unique(a, a+n);
```

## 二分查找

```

// STL 二分查找数组中>=value的第一个元素的位置，如果没有，返回n
// 前闭后开[a, n)
lower_bound(a, a+n, value)

// 反之，upper_bound
upper_bound(a, a+n, value)

```

串处理函数



# STL中map用法详解

引用声明，来自：[http://blog.csdn.net/it\\_yuan/article/details/22697205](http://blog.csdn.net/it_yuan/article/details/22697205)

Map是STL的一个关联容器，它提供一对一（其中第一个可以称为关键字，每个关键字只能在map中出现一次，第二个可能称为该关键字的值）的数据处理能力，由于这个特性，它完成有可能在我们处理一对一数据的时候，在编程上提供快速通道。这里说下map内部数据的组织，map内部自建一颗红黑树(一种非严格意义上的平衡二叉树)，这颗树具有对数据自动排序的功能，所以在map内部所有的数据都是有序的，后边我们会见识到有序的好处。下面举例说明什么是一一对一的数据映射。比如一个班级中，每个学生的学号跟他的姓名就存在着——映射的关系，这个模型用map可能轻易描述，很明显学号用int描述，姓名用字符串描述(本篇文章中不用char \*来描述字符串，而是采用STL中string来描述),下面给出map描述代码：Map mapStudent;

## 1. map的构造函数

map共提供了6个构造函数，这块涉及到内存分配器这些东西，略过不表，在下面我们将接触到一些map的构造方法，这里要说下的就是，我们通常用如下方法构造一个map：

```
Map<int, string> mapStudent;
```

## 2. 数据的插入

在构造map容器后，我们就可以往里面插入数据了。这里讲三种插入数据的方法：第一种：用insert函数插入pair数据，下面举例说明(以下代码虽然是随手写的，应该可以在VC和GCC下编译通过，大家可以运行下看什么效果，在VC下请加入这条语句，屏蔽4786警告 #pragma warning (disable:4786) )

```
#include <map>
#include <string>
#include <iostream>
Using namespace std;
Int main()
{
 Map<int, string> mapStudent;
 mapStudent.insert(pair<int, string>(1, "student_one"));
 mapStudent.insert(pair<int, string>(2, "student_two"));
 mapStudent.insert(pair<int, string>(3, "student_three"));
 map<int, string>::iterator iter;
 for(iter = mapStudent.begin(); iter != mapStudent.end(); iter++)
 {
 Cout<<iter->first<<" "<<iter->second<<end;
 }
}
```

第二种：用insert函数插入value\_type数据，下面举例说明

```

#include <map>
#include <string>
#include <iostream>
Using namespace std;
Int main()
{
 Map<int, string> mapStudent;
 mapStudent.insert(map<int, string>::value_type (1, "student_one"));
 mapStudent.insert(map<int, string>::value_type (2, "student_two"));
 mapStudent.insert(map<int, string>::value_type (3, "student_three"));
 map<int, string>::iterator iter;
 for(iter = mapStudent.begin(); iter != mapStudent.end(); iter++)
 {
 Cout<<iter->first<<" "<<iter->second<<end;
 }
}

```

第三种：用数组方式插入数据，下面举例说明

```

#include <map>
#include <string>
#include <iostream>
Using namespace std;
Int main()
{
 Map<int, string> mapStudent;
 mapStudent[1] = "student_one";
 mapStudent[2] = "student_two";
 mapStudent[3] = "student_three";
 map<int, string>::iterator iter;
 for(iter = mapStudent.begin(); iter != mapStudent.end(); iter++)
 {
 Cout<<iter->first<<" "<<iter->second<<end;
 }
}

```

以上三种用法，虽然都可以实现数据的插入，但是它们是有区别的，当然了第一种和第二种在效果上是完成一样的，用insert函数插入数据，在数据的插入上涉及到集合的唯一性这个概念，即当map中有这个关键字时，insert操作是插入数据不了的，但是用数组方式就不同了，它可以覆盖以前该关键字对应的值，用程序说明

```

mapStudent.insert(map<int, string>::value_type (1, "student_one"));
mapStudent.insert(map<int, string>::value_type (1, "student_two"));

```

上面这两条语句执行后，map中1这个关键字对应的值是“student\_one”，第二条语句并没有生效，那么这就涉及到我们怎么知道insert语句是否插入成功的问题了，可以用pair来获得是否插入成功，程序如下

```

Pair<map<int, string>::iterator, bool> Insert_Pair;
Insert_Pair = mapStudent.insert(map<int, string>::value_type (1, "student_one"));

```

我们通过pair的第二个变量来知道是否插入成功，它的第一个变量返回的是一个map的迭代器，如果插入成功的话 Insert\_Pair.second应该是true的，否则为false。下面给出完成代码，演示插入成功与否问题

```
#include <map>
#include <string>
#include <iostream>
Using namespace std;
Int main()
{
 Map<int, string> mapStudent;
 Pair<map<int, string>::iterator, bool> Insert_Pair;
 Insert_Pair=mapStudent.insert(pair<int, string>(1, "student_one"));
 If(Insert_Pair.second == true)
 {
 Cout<<"Insert Successfully"<<endl;
 }
 Else
 {
 Cout<<"Insert Failure"<<endl;
 }
 Insert_Pair=mapStudent.insert(pair<int, string>(1, "student_two"));
 If(Insert_Pair.second == true)
 {
 Cout<<"Insert Successfully"<<endl;
 }
 Else
 {
 Cout<<"Insert Failure"<<endl;
 }
 map<int, string>::iterator iter;
 for(iter = mapStudent.begin(); iter != mapStudent.end(); iter++)
 {
 Cout<<iter->first<<" "<<iter->second<<endl;
 }
}
```

大家可以用如下程序，看下用数组插入在数据覆盖上的效果

```
#include <map>
#include <string>
#include <iostream>
Using namespace std;
Int main()
{
 Map<int, string> mapStudent;
 mapStudent[1] = "student_one";
 mapStudent[1] = "student_two";
 mapStudent[2] = "student_three";
 map<int, string>::iterator iter;
 for(iter = mapStudent.begin(); iter != mapStudent.end(); iter++)
 {
 Cout<<iter->first<<" "<<iter->second<<endl;
 }
}
```

### 3. map的大小

在往map里面插入了数据，我们怎么知道当前已经插入了多少数据呢，可以用size函数，用法如下：

```
Int nSize = mapStudent.size();
```

### 4. 数据的遍历

这里也提供三种方法，对map进行遍历 第一种：应用前向迭代器，上面举例程序中到处都是了，略过不表 第二种：应用反相迭代器，下面举例说明，要体会效果，请自个动手运行程序

```
#include <map>
#include <map>
#include <string>
#include <iostream>
Using namespace std;
Int main()
{
 Map<int, string> mapStudent;
 mapStudent.insert(pair<int, string>(1, "student_one"));
 mapStudent.insert(pair<int, string>(2, "student_two"));
 mapStudent.insert(pair<int, string>(3, "student_three"));
 map<int, string>::reverse_iterator iter;
 for(iter = mapStudent.rbegin(); iter != mapStudent.rend(); iter++)
 {
 Cout<<iter->first<<" "<<iter->second<<end;
 }
}
```

第三种：用数组方式，程序说明如下

```
#include <map>
#include <string>
#include <iostream>
Using namespace std;
Int main()
{
 Map<int, string> mapStudent;
 mapStudent.insert(pair<int, string>(1, "student_one"));
 mapStudent.insert(pair<int, string>(2, "student_two"));
 mapStudent.insert(pair<int, string>(3, "student_three"));
 int nSize = mapStudent.size()
 //此处有误，应该是 for(int nIndex = 1; nIndex <= nSize; nIndex++)
 //by rainfish
 for(int nIndex = 0; nIndex < nSize; nIndex++)
 {
 Cout<<mapStudent[nIndex]<<end;
 }
}
```

## 5. 数据的查找（包括判定这个关键字是否在map中出现）

在这里我们将体会，map在数据插入时保证有序的好处。要判定一个数据（关键字）是否在map中出现的方法比较多，这里标题虽然是数据的查找，在这里将穿插着大量的map基本用法。这里给出三种数据查找方法 第一种：用count函数来判定关键字是否出现，其缺点是无法定位数据出现位置,由于map的特性，一对一的映射关系，就决定了count函数的返回值只有两个，要么是0，要么是1，出现的情况，当然是返回1了 第二种：用find函数来定位数据出现位置，它返回的一个迭代器，当数据出现时，它返回数据所在位置的迭代器，如果map中没有要查找的数据，它返回的迭代器等于end函数返回的迭代器，程序说明

```
#include <map>
#include <string>
#include <iostream>
Using namespace std;
Int main()
{
 Map<int, string> mapStudent;
 mapStudent.insert(pair<int, string>(1, "student_one"));
 mapStudent.insert(pair<int, string>(2, "student_two"));
 mapStudent.insert(pair<int, string>(3, "student_three"));
 map<int, string>::iterator iter;
 iter = mapStudent.find(1);
 if(iter != mapStudent.end())
 {
 Cout<<"Find, the value is "<<iter->second<<endl;
 }
 Else
 {
 Cout<<"Do not Find"<<endl;
 }
}
```

第三种：这个方法用来判定数据是否出现，是显得笨了点，但是，我打算在这里讲解 Lower\_bound函数用法，这个函数用来返回要查找关键字的下界(是一个迭代器) Upper\_bound函数用法，这个函数用来返回要查找关键字的上界(是一个迭代器) 例如：map中已经插入了1, 2, 3, 4的话，如果lower\_bound(2)的话，返回的2，而upper-bound（2）的话，返回的就是3 Equal\_range函数返回一个pair，pair里面第一个变量是Lower\_bound返回的迭代器，pair里面第二个迭代器是 Upper\_bound返回的迭代器，如果这两个迭代器相等的话，则说明map中不出现这个关键字，程序说明

```
#include <map>
#include <string>
#include <iostream>
Using namespace std;
Int main()
{
 Map<int, string> mapStudent;
 mapStudent[1] = "student_one";
 mapStudent[3] = "student_three";
 mapStudent[5] = "student_five";
 map<int, string>::iterator iter;
 iter = mapStudent.lower_bound(2);
 {
 //返回的是下界3的迭代器
 }
```

```

 Cout<<iter->second<<endl;
 }
 iter = mapStudent.lower_bound(3);
 {
 //返回的是下界3的迭代器
 Cout<<iter->second<<endl;
 }
 iter = mapStudent.upper_bound(2);
 {
 //返回的是上界3的迭代器
 Cout<<iter->second<<endl;
 }
 iter = mapStudent.upper_bound(3);
 {
 //返回的是上界5的迭代器
 Cout<<iter->second<<endl;
 }
 Pair<map<int, string>::iterator, map<int, string>::iterator> mapPair;
 mapPair = mapStudent.equal_range(2);
 if(mapPair.first == mapPair.second)
 {
 cout<<"Do not Find"<<endl;
 }
 Else
 {
 Cout<<"Find"<<endl;
 }
 mapPair = mapStudent.equal_range(3);
 if(mapPair.first == mapPair.second)
 {
 cout<<"Do not Find"<<endl;
 }
 Else
 {
 Cout<<"Find"<<endl;
 }
}

```

## 6. 数据的清空与判空

清空map中的数据可以用clear()函数，判定map中是否有数据可以用empty()函数，它返回true则说明是空map

## 7. 数据的删除

这里要用到erase函数，它有三个重载了的函数，下面在例子中详细说明它们的用法

```

#include <map>
#include <string>
#include <iostream>
Using namespace std;
Int main()
{
 Map<int, string> mapStudent;

```

```

 mapStudent.insert(pair<int, string>(1, "student_one"));
 mapStudent.insert(pair<int, string>(2, "student_two"));
 mapStudent.insert(pair<int, string>(3, "student_three"));
//如果你要演示输出效果, 请选择以下的一种, 你看到的效果会比较好
//如果要删除1, 用迭代器删除
map<int, string>::iterator iter;
iter = mapStudent.find(1);
mapStudent.erase(iter);
//如果要删除1, 用关键字删除
Int n = mapStudent.erase(1); //如果删除了会返回1, 否则返回0
//用迭代器, 成片的删除
//一下代码把整个map清空
mapStudent.erase(mapStudent.begin(), mapStudent.end());
//成片删除要注意的是, 也是STL的特性, 删除区间是一个前闭后开的集合
//自个加上遍历代码, 打印输出吧
}

```

## 8. 其他一些函数用法

这里有swap, key\_comp, value\_comp, get\_allocator等函数, 感觉到这些函数在编程用的不是很多, 略过不表, 有兴趣的话可以自个研究

## 9. 排序

这里要讲的是一点比较高深的用法了, 排序问题, STL中默认是采用小于号来排序的, 以上代码在排序上是不存在任何问题的, 因为上面的关键字是int型, 它本身支持小于号运算, 在一些特殊情况, 比如关键字是一个结构体, 涉及到排序就会出现问題, 因为它没有小于号操作, insert等函数在编译的时候过不去, 下面给出两个方法解决这个问题 第一种: 小于号重载, 程序举例

```

#include <map>
#include <string>
Using namespace std;
typedef struct tagStudentInfo
{
 Int nID;
 String strName;
}StudentInfo, *PStudentInfo; //学生信息
Int main()
{
 int nSize;
 //用学生信息映射分数
 map<StudentInfo, int>mapStudent;
 map<StudentInfo, int>::iterator iter;
 StudentInfo studentInfo;
 studentInfo.nID = 1;
 studentInfo.strName = "student_one";
 mapStudent.insert(pair<StudentInfo, int>(studentInfo, 90));
 studentInfo.nID = 2;
 studentInfo.strName = "student_two";
 mapStudent.insert(pair<StudentInfo, int>(studentInfo, 80));
 for (iter=mapStudent.begin(); iter!=mapStudent.end(); iter++)
 cout<<iter->first.nID<<endl<<iter->first.strName<<endl<<iter->second<<endl;
}

```

以上程序是无法编译通过的，只要重载小于号，就OK了，如下：

```

Typedef struct tagStudentInfo
{
 Int nID;
 String strName;
 Bool operator < (tagStudentInfo const& _A) const
 {
 //这个函数指定排序策略，按nID排序，如果nID相等的话，按strName排序
 If(nID < _A.nID) return true;
 If(nID == _A.nID) return strName.compare(_A.strName) < 0;
 Return false;
 }
}StudentInfo, *PStudentInfo; //学生信息

```

第二种：仿函数的应用，这个时候结构体中没有直接的小于号重载，程序说明

```

#include <map>
#include <string>
Using namespace std;
Typedef struct tagStudentInfo
{
 Int nID;
 String strName;
}StudentInfo, *PStudentInfo; //学生信息
Classs sort
{
 Public:
 Bool operator() (StudentInfo const &_A, StudentInfo const &_B) const
 {
 If(_A.nID < _B.nID) return true;
 If(_A.nID == _B.nID) return _A.strName.compare(_B.strName) < 0;
 Return false;
 }
};
int main()
{
 //用学生信息映射分数
 Map<StudentInfo, int, sort>mapStudent;
 StudentInfo studentInfo;
 studentInfo.nID = 1;
 studentInfo.strName = "student_one";
 mapStudent.insert(pair<StudentInfo, int>(studentInfo, 90));
 studentInfo.nID = 2;
 studentInfo.strName = "student_two";
 mapStudent.insert(pair<StudentInfo, int>(studentInfo, 80));
}

```

## 10. 另外

由于STL是一个统一的整体，map的很多用法都和STL中其它的东西结合在一起，比如在排序上，这里默认用的是小于号，即less<>，如果要从大到小排序呢，这里涉及到的东西很多，在此无法一一加以说明。还



要说明的是，map中由于它内部有序，由红黑树保证，因此很多函数执行的时间复杂度都是 $\log_2 N$ 的，如果用map函数可以实现的功能，而STL Algorithm也可以完成该功能，建议用map自带函数，效率高一些。下面说下，map在空间上的特性，否则，估计你用起来会有时候表现的比较郁闷，由于map的每个数据对应红黑树上的一个节点，这个节点在不保存你的数据时，是占用16个字节的，一个父节点指针，左右孩子指针，还有一个枚举值（标示红黑的，相当于平衡二叉树中的平衡因子），我想大家应该知道，这些地方很费内存了吧，不说了.....

# STL之set集合容器

引用声明，来自：<http://blog.csdn.net/lyhvoyage/article/details/22989659>

stl set集合容器实现了红黑树（Red-Black Tree）的平衡二叉检索树的数据结构，在插入元素时，它会自动调整二叉树的排列，把该元素放到适当的位置，以确保每个子树根节点的键值大于左子树所有节点的键值，而小于右子树所有节点的键值；另外，还得确保根节点的左子树的高度与右子树的高度相等，这样，二叉树的高度最小，从而检索速度最快。要注意的是，它不会重复插入相同键值的元素，而采取忽略处理。平衡二叉检索树的检索使用中序遍历算法，检索效率高于vector、deque、和list的容器。另外，采用中序遍历算法可将键值由小到大遍历出来，所以，可以理解为平衡二叉检索树在插入元素时，就会自动将元素按键值从小到大的顺序排列。构造set集合的主要目的是为了快速检索，使用set前，需要在程序头文件中包含声明“#include”。

## 1.创建set集合对象

创建set对象时，需要指定元素的类型，这一点和其他容器一样。

```
#include<iostream>
#include<set>
using namespace std;
int main()
{
 set<int> s;
 return 0;
}
```

## 2.元素的插入与中序遍历

采用insert()方法把元素插入到集合中，插入规则在默认的比较规则下，是按元素值从小到大插入，如果自己指定了比较规则函数，则按自定义比较规则函数插入。使用前向迭代器对集合中序遍历，结果正好是元素排序后的结果。

```
#include<iostream>
#include<set>
using namespace std;
int main()
{
 set<int> s;
 s.insert(5); //第一次插入5，可以插入
 s.insert(1);
 s.insert(6);
 s.insert(3);
 s.insert(5); //第二次插入5，重复元素，不会插入
 set<int>::iterator it; //定义前向迭代器
 //中序遍历集合中的所有元素
 for(it = s.begin(); it != s.end(); it++)
 {
 cout << *it << " ";
 }
}
```

```

 }
 cout << endl;
 return 0;
}
//运行结果:1 3 5 6

```

### 3.元素的方向遍历

使用反向迭代器reverse\_iterator可以反向遍历集合，输出的结果正好是集合元素的反向排序结果。它需要用到rbegin()和rend()两个方法，它们分别给出了反向遍历的开始位置和结束位置。

```

#include<iostream>
#include<set>
using namespace std;
int main()
{
 set<int> s;
 s.insert(5); //第一次插入5，可以插入
 s.insert(1);
 s.insert(6);
 s.insert(3);
 s.insert(5); //第二次插入5，重复元素，不会插入
 set<int>::reverse_iterator rit; //定义反向迭代器
 //反向遍历集合中的所有元素
 for(rit = s.rbegin(); rit != s.rend(); rit++)
 {
 cout << *rit << " ";
 }
 cout << endl;
 return 0;
}
//运行结果:6 5 3 1

```

### 4.元素的删除

与插入元素的处理一样，集合具有高效的删除处理功能，并自动重新调整内部的红黑树的平衡。删除的对象可以是某个迭代器位置上的元素、等于某键值的元素、一个区间上的元素和清空集合。

```

#include<iostream>
#include<set>
using namespace std;
int main()
{
 set<int> s;
 s.insert(5); //第一次插入5，可以插入
 s.insert(1);
 s.insert(6);
 s.insert(3);
 s.insert(5); //第二次插入5，重复元素，不会插入
 s.erase(6); //删除键值为6的元素
 set<int>::reverse_iterator rit; //定义反向迭代器
 //反向遍历集合中的所有元素

```

```

 for(rit = s.rbegin(); rit != s.rend(); rit++)
 {
 cout << *rit << " ";
 }
 cout << endl;
 set<int>::iterator it;

 it = s.begin();
 for(int i = 0; i < 2; i++)
 it = s.erase(it);
 for(it = s.begin(); it != s.end(); it++)
 cout << *it << " ";
 cout << endl;

 s.clear();
 cout << s.size() << endl;

 return 0;
}
/*
运行结果：
5 3 1
5
0
*/

```

## 5.元素的检索

使用find()方法对集合进行检索，如果找到查找的键值，则返回该键值的迭代器位置；否则，返回集合最后一个元素后面的一个位置，即end()。

```

#include<iostream>
#include<set>
using namespace std;
int main()
{
 set<int> s;
 s.insert(5); //第一次插入5，可以插入
 s.insert(1);
 s.insert(6);
 s.insert(3);
 s.insert(5); //第二次插入5，重复元素，不会插入
 set<int>::iterator it;
 it = s.find(6); //查找键值为6的元素
 if(it != s.end())
 cout << *it << endl;
 else
 cout << "not find it" << endl;
 it = s.find(20);
 if(it != s.end())
 cout << *it << endl;
 else
 cout << "not find it" << endl;
 return 0;
}
/*

```

```

运行结果：
6
not find it
*/

```

下面这种方法也能判断一个数是否在集合中：

```

#include <cstdio>
#include <set>
using namespace std;
int main() {
 set<int> s;
 int a;
 for(int i = 0; i < 10; i++)
 s.insert(i);
 for(int i = 0; i < 5; i++) {
 scanf("%d", &a);
 if(!s.count(a)) //不存在
 printf("does not exist\n");
 else
 printf("exist\n");
 }
 return 0;
}

```

## 6.自定义比较函数

使用insert将元素插入到集合中去的时候，集合会根据设定的比较函数将该元素放到该放的节点上去。在定义集合的时候，如果没有指定比较函数，那么采用默认的比较函数，即按键值从小到大的顺序插入元素。但在很多情况下，需要自己编写比较函数。编写比较函数有两种方法。(1)如果元素不是结构体，那么可以编写比较函数。下面的程序比较规则为按键值从大到小的顺序插入到集合中。

```

#include<iostream>
#include<set>
using namespace std;
struct mycomp
{ //自定义比较函数，重载“<”操作符
 bool operator() (const int &a, const int &b)
 {
 if(a != b)
 return a > b;
 else
 return a > b;
 }
};
int main()
{
 set<int, mycomp> s; //采用比较函数mycomp
 s.insert(5); //第一次插入5，可以插入
 s.insert(1);
 s.insert(6);
 s.insert(3);
 s.insert(5); //第二次插入5，重复元素，不会插入
}

```

```

 set<int,mycomp>::iterator it;
 for(it = s.begin(); it != s.end(); it++)
 cout << *it << " ";
 cout << endl;
 return 0;
}
/*
运行结果: 6 5 3 1
*/

```

(2)如果元素是结构体，那么可以直接把比较函数写在结构体内。

```

#include<iostream>
#include<set>
#include<string>
using namespace std;
struct Info
{
 string name;
 double score;
 bool operator < (const Info &a) const // 重载"<"操作符，自定义排序规则
 {
 //按score由大到小排序。如果要由小到大排序，使用">"即可。
 return a.score < score;
 }
};
int main()
{
 set<Info> s;
 Info info;

 //插入三个元素
 info.name = "Jack";
 info.score = 80;
 s.insert(info);
 info.name = "Tom";
 info.score = 99;
 s.insert(info);
 info.name = "Steaven";
 info.score = 60;
 s.insert(info);

 set<Info>::iterator it;
 for(it = s.begin(); it != s.end(); it++)
 cout << (*it).name << " : " << (*it).score << endl;
 return 0;
}
/*
运行结果:
Tom : 99
Jack : 80
Steaven : 60
*/

```

# STL中的set容器的一点总结

引用声明，来自：<http://www.cnblogs.com/BeyondAnyTime/archive/2012/08/13/2636375.html>

## 1.关于set

C++ STL之所以得到广泛的赞誉，也被很多人使用，不只是提供了像vector, string, list等方便的容器，更重要的是STL封装了许多复杂的数据结构算法和大量常用数据结构操作。vector封装数组，list封装了链表，map和set封装了二叉树等，在封装这些数据结构的时候，STL按照程序员的使用习惯，以成员函数方式提供的常用操作，如：插入、排序、删除、查找等。让用户在STL使用过程中，并不会感到陌生。

关于set，必须说明的是set关联式容器。set作为一个容器也是用来存储同一数据类型的数据类型，并且能从一个数据集合中取出数据，在set中每个元素的值都唯一，而且系统能根据元素的值自动进行排序。应该注意的是set中数元素的值不能直接被改变。C++ STL中标准关联容器set, multiset, map, multimap内部采用的就是一种非常高效的平衡检索二叉树：红黑树，也成为RB树(Red-Black Tree)。RB树的统计性能要好于一般平衡二叉树，所以被STL选择作为关联容器的内部结构。

关于set有下面几个问题：

### (1) 为何map和set的插入删除效率比用其他序列容器高？

大部分人说，很简单，因为对于关联容器来说，不需要做内存拷贝和内存移动。说对了，确实如此。set容器内所有元素都是以节点的方式来存储，其节点结构和链表差不多，指向父节点和子节点。结构图可能如下：

```

A
 / \
B C
 / \ \
D E F G

```

因此插入的时候只需要稍做变换，把节点的指针指向新的节点就可以了。删除的时候类似，稍做变换后把指向删除节点的指针指向其他节点也OK了。这里的一切操作就是指针换来换去，和内存移动没有关系。

### (2) 为何每次insert之后，以前保存的iterator不会失效？

iterator这里就相当于指向节点的指针，内存没有变，指向内存的指针怎么会失效呢(当然被删除的那个元素本身已经失效了)。相对于vector来说，每一次删除和插入，指针都有可能失效，调用push\_back在尾部插入也是如此。因为为了保证内部数据的连续存放，iterator指向的那块内存存在删除和插入过程中可能已经被其他内存覆盖或者内存已经被释放了。即使push\_back的时候，容器内部空间可能不够，需要一块新的更大的内存，只有把以前的内存释放，申请新的更大的内存，复制已有的数据元素到新的内存，最后把需要插入的元素放到最后，那么以前的内存指针自然就不可用了。特别时在和find等算法在一起使用的时候，

牢记这个原则：不要使用过期的iterator。

### （3）当数据元素增多时，set的插入和搜索速度变化如何？

如果你知道log2的关系你应该就彻底了解这个答案。在set中查找是使用二分查找，也就是说，如果有16个元素，最多需要比较4次就能找到结果，有32个元素，最多比较5次。那么有10000个呢？最多比较的次数为log10000，最多为14次，如果是20000个元素呢？最多不过15次。看见了吧，当数据量增大一倍的时候，搜索次数只不过多了1次，多了1/14的搜索时间而已。你明白这个道理后，就可以安心往里面放入元素了。

## 2.set中常用的方法

**begin()** ,返回set容器的第一个元素

**end()** ,返回set容器的最后一个元素

**clear()** ,删除set容器中的所有的元素

**empty()** ,判断set容器是否为空

**max\_size()** ,返回set容器可能包含的元素最大个数

**size()** ,返回当前set容器中的元素个数

**rbegin** ,返回的值和end()相同

**rend()** ,返回的值和rbegin()相同

写一个程序练一练这几个简单操作吧：

```
1 #include <iostream>
2 #include <set>
3
4 using namespace std;
5
6 int main()
7 {
8 set<int> s;
9 s.insert(1);
10 s.insert(2);
11 s.insert(3);
12 s.insert(1);
13 cout<<"set 的 size 值为 : "<<s.size()<<endl;
14 cout<<"set 的 maxsize的值为 : "<<s.max_size()<<endl;
15 cout<<"set 中的第一个元素是 : "<<*s.begin()<<endl;
16 cout<<"set 中的最后一个元素是:"<<*s.end()<<endl;
17 s.clear();
18 if(s.empty())
19 {
20 cout<<"set 为空 !!! "<<endl;
21 }
```

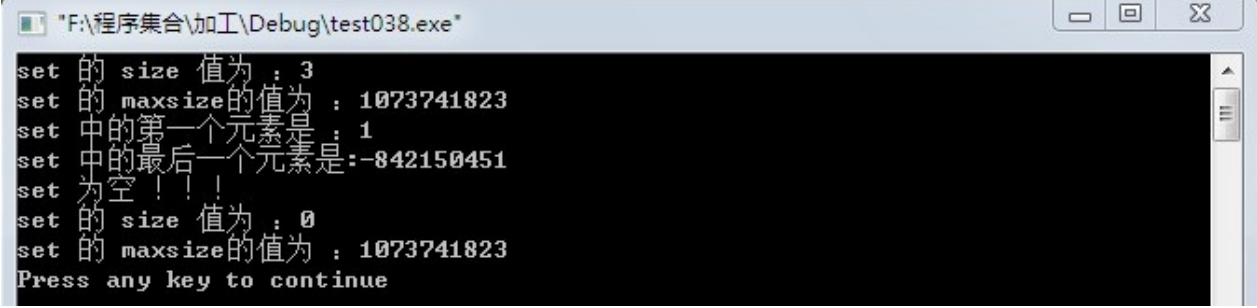


```

22 cout<<"set 的 size 值为 : "<<s.size()<<endl;
23 cout<<"set 的 maxsize的值为 : "<<s.max_size()<<endl;
24 return 0;
25 }

```

运行结果：



```

"F:\程序集合\加工\Debug\test038.exe"
set 的 size 值为 : 3
set 的 maxsize的值为 : 1073741823
set 中的第一个元素是 : 1
set 中的最后一个元素是 : 3
set 为空 !!!
set 的 size 值为 : 0
set 的 maxsize的值为 : 1073741823
Press any key to continue

```

小结：插入3之后虽然插入了一个1，但是我们发现set中最后一个值仍然是3哈，这就是set。还要注意begin() 和 end()函数是不检查set是否为空的，使用前最好使用empty()检验一下set是否为空。

**count()** 用来查找set中某个键值出现的次数。

这个函数在set并不是很实用，因为一个键值在set只可能出现0或1次，这样就变成了判断某一键值是否在set出现过了。

示例代码：

```

1 #include <iostream>
2 #include <set>
3
4 using namespace std;
5
6 int main()
7 {
8 set<int> s;
9 s.insert(1);
10 s.insert(2);
11 s.insert(3);
12 s.insert(1);
13 cout<<"set 中 1 出现的次数是 : "<<s.count(1)<<endl;
14 cout<<"set 中 4 出现的次数是 : "<<s.count(4)<<endl;
15 return 0;
16 }

```

运行结果：



```

"F:\程序集合\加工\Debug\test038.exe"
set 中 1 出现的次数是 : 1
set 中 4 出现的次数是 : 0
Press any key to continue

```

**equal\_range()**

返回一对定位器，分别表示第一个大于或等于给定关键值的元素和 第一个大于给定关键值的元素，这个返回值是一个pair类型，如果这一对定位器中哪个返回失败，就会等于end()的值。具体这个有什么用途我还没遇到过~~~

示例代码：

```

1 #include <iostream>
2 #include <set>
3
4 using namespace std;
5
6 int main()
7 {
8 set<int> s;
9 set<int>::iterator iter;
10 for(int i = 1 ; i <= 5; ++i)
11 {
12 s.insert(i);
13 }
14 for(iter = s.begin() ; iter != s.end() ; ++iter)
15 {
16 cout<<*iter<<" ";
17 }
18 cout<<endl;
19 pair<set<int>::const_iterator, set<int>::const_iterator> pr;
20 pr = s.equal_range(3);
21 cout<<"第一个大于等于 3 的数是 : "<<*pr.first<<endl;
22 cout<<"第一个大于 3的数是 : " <<*pr.second<<endl;
23 return 0;
24 }

```

运行结果：



```

F:\程序集合\加工\Debug\test039.exe
1 2 3 4 5
第一个大于等于 3 的数是 : 3
第一个大于 3的数是 : 4
Press any key to continue

```

**erase(iterator)** ,删除定位器**iterator**指向的值

**erase(first,second)**,删除定位器**first**和**second**之间的值

**erase(key\_value)**,删除键值**key\_value**的值

看看程序吧：

```

1 #include <iostream>
2 #include <set>
3
4 using namespace std;
5
6 int main()

```

```

7 {
8 set<int> s;
9 set<int>::const_iterator iter;
10 set<int>::iterator first;
11 set<int>::iterator second;
12 for(int i = 1 ; i <= 10 ; ++i)
13 {
14 s.insert(i);
15 }
16 //第一种删除
17 s.erase(s.begin());
18 //第二种删除
19 first = s.begin();
20 second = s.begin();
21 second++;
22 second++;
23 s.erase(first,second);
24 //第三种删除
25 s.erase(8);
26 cout<<"删除后 set 中元素是 :";
27 for(iter = s.begin() ; iter != s.end() ; ++iter)
28 {
29 cout<<*iter<<" ";
30 }
31 cout<<endl;
32 return 0;
33 }

```

运行结果：



```

F:\程序集合\加工\Debug\test040.exe
删除后 set 中元素是 : 4 5 6 7 9 10
Press any key to continue

```

小结：set中的删除操作是不进行任何的错误检查的，比如定位器的是否合法等等，所以用的时候自己一定要注意。

**find()**，返回给定值值得定位器，如果没找到则返回**end()**。

示例代码：

```

1 #include <iostream>
2 #include <set>
3
4 using namespace std;
5
6 int main()
7 {
8 int a[] = {1,2,3};
9 set<int> s(a,a+3);
10 set<int>::iterator iter;
11 if((iter = s.find(2)) != s.end())
12 {
13 cout<<*iter<<endl;

```

```

14 }
15 return 0;
16 }

```

### insert(key\_value);

将key\_value插入到set中，返回值是pair::iterator,bool>，bool标志着插入是否成功，而iterator代表插入的位置，若key\_value已经在set中，则iterator表示的key\_value在set中的位置。

### inset(first,second);

将定位器first到second之间的元素插入到set中，返回值是void.

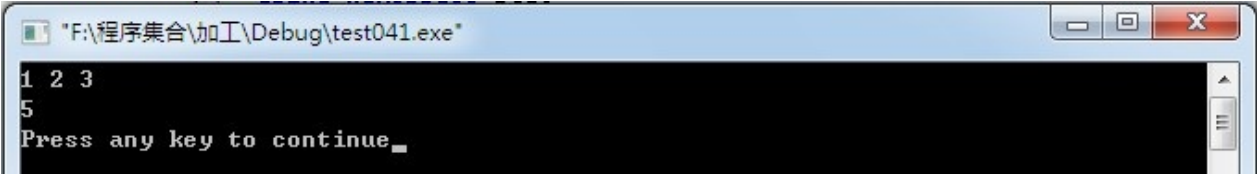
示例代码：

```

1 #include <iostream>
2 #include <set>
3
4 using namespace std;
5
6 int main()
7 {
8 int a[] = {1,2,3};
9 set<int> s;
10 set<int>::iterator iter;
11 s.insert(a,a+3);
12 for(iter = s.begin() ; iter != s.end() ; ++iter)
13 {
14 cout<<*iter<<" ";
15 }
16 cout<<endl;
17 pair<set<int>::iterator,bool> pr;
18 pr = s.insert(5);
19 if(pr.second)
20 {
21 cout<<*pr.first<<endl;
22 }
23 return 0;
24 }

```

运行结果：



**lower\_bound(key\_value)**，返回第一个大于等于key\_value的定位器

**upper\_bound(key\_value)**，返回最后一个大于等于key\_value的定位器

示例代码：

```
1 #include <iostream>
2 #include <set>
3
4 using namespace std;
5
6 int main()
7 {
8 set<int> s;
9 s.insert(1);
10 s.insert(3);
11 s.insert(4);
12 cout<<*s.lower_bound(2)<<endl;
13 cout<<*s.lower_bound(3)<<endl;
14 cout<<*s.upper_bound(3)<<endl;
15 return 0;
16 }
```

运行结果：



```
"F:\程序集合\加工\Debug\test042.exe"
3
3
4
Press any key to continue
```

## 哈希

---

- [使用set](#)
- [哈希数组表示法](#)
- [哈希指针表示法](#)

## set

---

```
set <int> a;
// n 为检查的数字
a.count(n);
// 加入n到n
a.add(n);
```

## 哈希数组表示法

---

- [代码](#)

```
#define MAXN 1010
#define MOD 1009
typedef ElemType int;

ElemType point[MAXN];
int hash[MAXN+1], next[MAXN];
int n;

//get hashcode
int hashcode(ElemType &a)
{
 //cal hashcode
 return abs(a.x+ a.y) % MOD;
}

void insertHash(int i)
{
 //头插入，拉链法
 int key = hashcode(point[i]);
 next[i] = hash[key];
 hash[key] = i;
}

bool search(ElemType &t)
{
 int key = hashcode(t);
 int i = hash[key];
 while(i != -1)
 {
 if(point[i] == t)
 return true;
 i = next[i];
 }
```

```

 }
 return false;
}

```

## 哈希指针表示法

- 代码

```

#define ElemType int
const int M = 1007;

struct Node
{
 ElemType d;
 Node *next;
};

Node *pnd[M+1];
Node nd[M+1];
int n_cnt; //count
int a_cnt; //count
int a[M+10];

int hashcode(int i)
{
 return i % M;
}

bool search(int i)
{
 bool found = false;
 int p = hashcode(i);
 Node *pt = pnd[p];
 while(pt)
 {
 if(pt->d == i)
 return true;
 pt = pt->next;
 }
}

void Insert_hash(int i)
{
 int p = hashcode(i);
 nd[n_cnt].d = i;
 nd[n_cnt].next = pnd[p];
 pnd[p] = &nd[n_cnt];
 n_cnt++;
 a[a_cnt++] = i;
}

```

## 水题不水

---

- [Floyd判圈算法](#)

## Floyd判圈算法

---

用于判断循环，可以达到线性的目的

可以抽象成为两个小孩跑步，但是一个小孩的速度是另一个小孩的两倍，当跑的快的小孩追上跑的慢的小孩的时候，算法结束。此外，时间复杂度为  $O(1)$  也是格外炫酷。

一个next一次，另一个next两次，很快便可以追上。

可能提出疑问：

既然肯定又重复的地方，为啥不让第一个小孩就在那里等呢？因为算法运行结果不一定是在小孩等的地方结束，可能还没有跑进圈里！



## 组合数学

- [公式](#)
- [特殊的数列](#)
- [全排列](#)
- [排列组合数字](#)
- [斯特林数stirling](#)
- [错排问题](#)

## 公式

1.  $C(n, 0) = C(n, n) = 1$
2.  $C(n, k) = C(n, n-k)$
3.  $C(n, k) + C(n, k+1) = C(n+1, k+1)$
4.  $C(n, k+1) = C(n, k) * (n-k) / (k+1)$

## 特殊的数列

| 名称     | 数列                                                    |
|--------|-------------------------------------------------------|
| 斐波那契数列 | 1 1 2 3 5 8                                           |
| 卡特兰数   | 1 2 5 14 42 132 429 1430 4862 16796 (\$4n-6/n*f(n)\$) |

## 全排列

可以使用next\_permutation生成

```
template <class Type>
void Perm(Type list[], int k, int m)
{
 if(k > m)//list[m] > list[0];
 {
 for(int i = 0; i <= m ; i++)
 cout << list[i];
 cout << endl;
 }
 else
 {
 for(int i = k; i <= m; i++)
 {
 swap(list[k], list[i]);
 Perm(list, k+1, m);
 swap(list[k], list[i]);
 }
 }
}
```

## 排列组合数计算

```

#define ll long long

/**
 * 排列数
 * P(n, k)
 * 特别的, k = n 的时候, 返回n!
 */
ll getP(int n, int k)
{
 ll res = 1;
 int b = n-k;
 for(int i = n; i > b; i--)
 res *= i;
 return res;
}

/**
 * 组合数
 * C(n, k)
 * 直接递推, 返回需要的那一个
 * 注意溢出问题
 */

const int maxn = 1000;
ll C[maxn][maxn];

void init()
{
 for(int i = 0; i < maxn; i++)
 {
 C[i][0] = C[i][i] = 1;
 for(int j = 1; j < i; j++) C[i][j] = C[i-1][j] + C[i-1][j-1];
 }
}

ll getC(int n, int k)
{
 if(C[n][0] == 1)
 return C[n][k];

 // 组合数性质
 C[n][0] = C[n][n] = 1;
 for(int i = 1; i < n; i++)
 C[n][i] = C[n][i-1] *(n-i+1)/i;
 return C[n][k];
}

```

## 斯特林数stirling

- 第一类 stirling数  $s(n, k)$

$n$ 个人分成 $k$ 组, 组内再按特定顺序围圈

也就是分成了k组，组内就像是项链颜色一样，

1. ( {A, B}, {C, D} )
2. ( {B, A}, {C, D} )

属于一组

1. ({A}, {B, C, D})
2. ({A}, {B, D, C})

不属于一组

给定  $s(n,0)=0, s(1,1)=1$ ，有递归关系  $s(n,k)=s(n-1,k-1) + (n-1) s(n-1,k)$

- 第二类 stirling数

$S(n, k)$  是把p元素集合划分到k个不可区分的盒子里且没有空盒的划分个数。

公式:

$$S(n, n) = 1 \quad (n \geq 0)$$

$$S(n, 0) = 0 \quad (n \geq 1)$$

$$S(n,k)=k \cdot S(n-1,k)+S(n-1,k-1), \text{ (} 1 \leq k \leq n-1 \text{)}$$

## 错排问题

错排问题是组合数学中的问题之一。考虑一个有n个元素的排列，若一个排列中所有的元素都不在自己原来的位置上，那么这样的排列就称为原排列的一个错排。 -- wikipedia

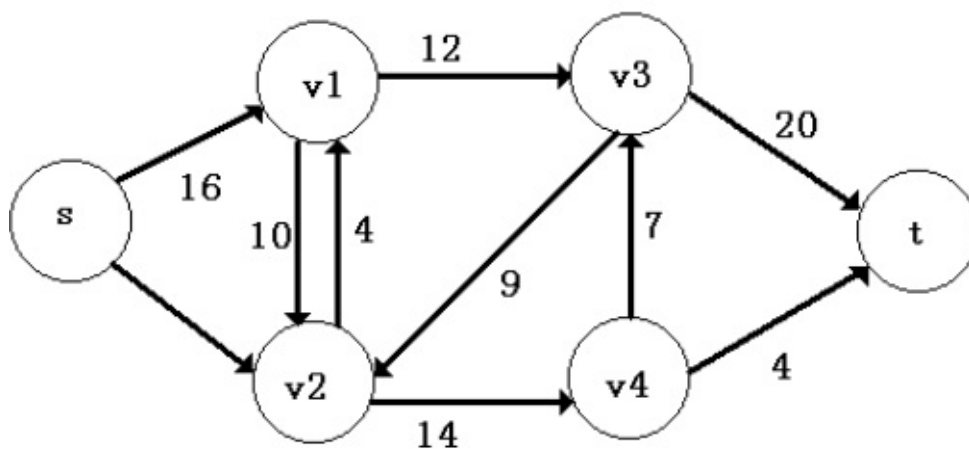
$D_n = (n-1)(D_{n-1}+D_{n-2})$  [2] 递推公式  $D_n = n! \left( \frac{1}{2!} - \frac{1}{3!} + \dots + (-1)^n \frac{1}{n!} \right)$ . [3]  
直接公式

## 关于网络流的学习

应该属于图论的部分，但是单独拿出来作为了一个分类。对于模板而言有些过于详细

## 什么是网络流？

- 外文名：Network-flows
- 使用领域范围：统筹学中的最优化问题
- 具体：一个给定的网络上寻求两点间最大运输量的问题。也可以想成最大流量的问题。
- 实例：给定一个有向图 $G=(V, E)$ ，把图中的边看做管道，每条边上有一个权值，表示该管道的流量上限。给定源点 $s$ 和汇点 $t$ ，现在假设在 $s$ 处有一个水源， $t$ 处有一个蓄水池，问从 $s$ 到 $t$ 的最大水流量是多少。



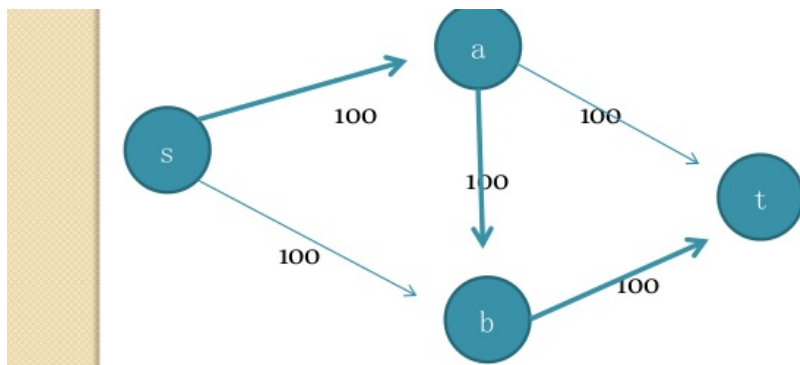
少。

- 定理
- 网络流图中，源点流出的量，等于汇点流入的量，除汇点外的任何点，其流入量之和等于流出量之和。

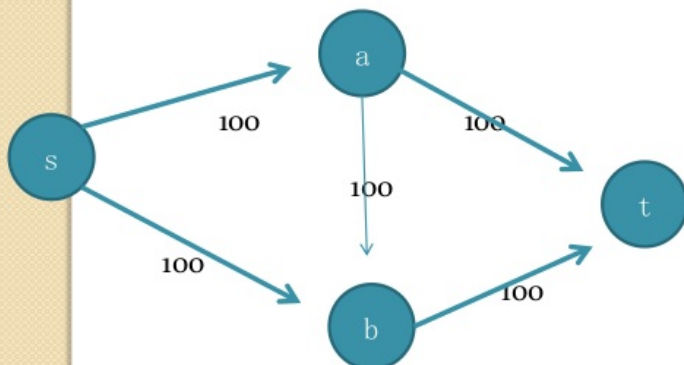
## 算法

### 解决最大流的Ford-Fulkerson算法

- 基本思路：
  1. 每次用DFS从源到汇找一条可行路径，然后把这条路塞满。这条路径上容量最小的那条边的容量，就是这次DFS所找到的流量。
  2. 然后对于路径上的每条边，其容量要减去刚才找到的流量。这样，每次DFS都可能增大流量，直到某次dfs找不到可行路径为止，最大流就求出来了。
- 这么想是否正确？
  - 问题在：过早的认为  $a \rightarrow b$  上的流量不为0，因而封锁了流量继续增大的可能。

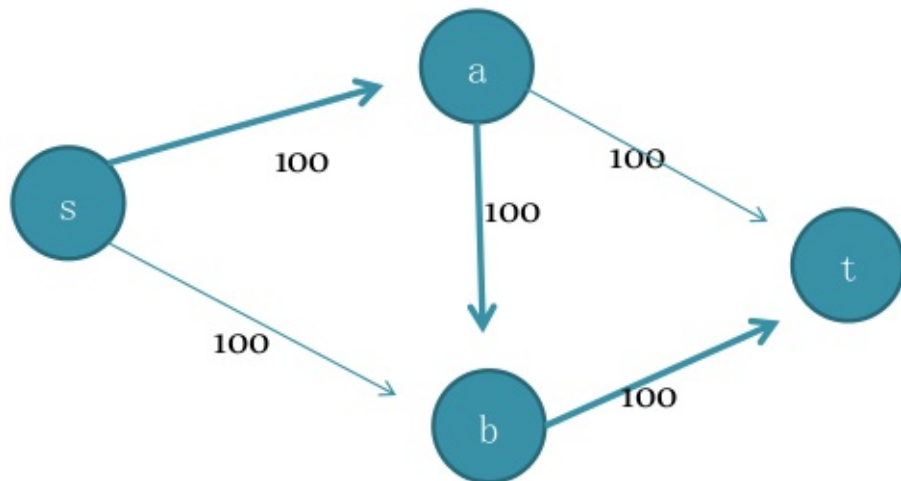


- 如果我们沿着s-a-b-t路线走 仅能得到一个100的流

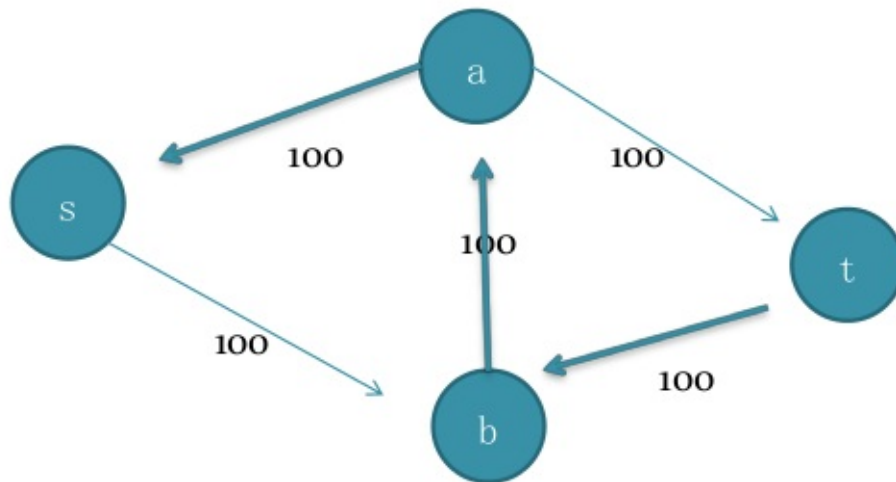


实际上此图存在流量为200的流

- 改进的思路：应该能修改已经建立的流网络，使得"不合理"的流量被删除。
- 一种实现：
  - 对上次DFS时找到的流量路径上的边，添加一条"反向"边，反向边上的容量等于上次DFS时找到的该边上的流量，再利用"反向"的容量和其他边上剩余的容量寻找路径。
    - 第一次DFS:



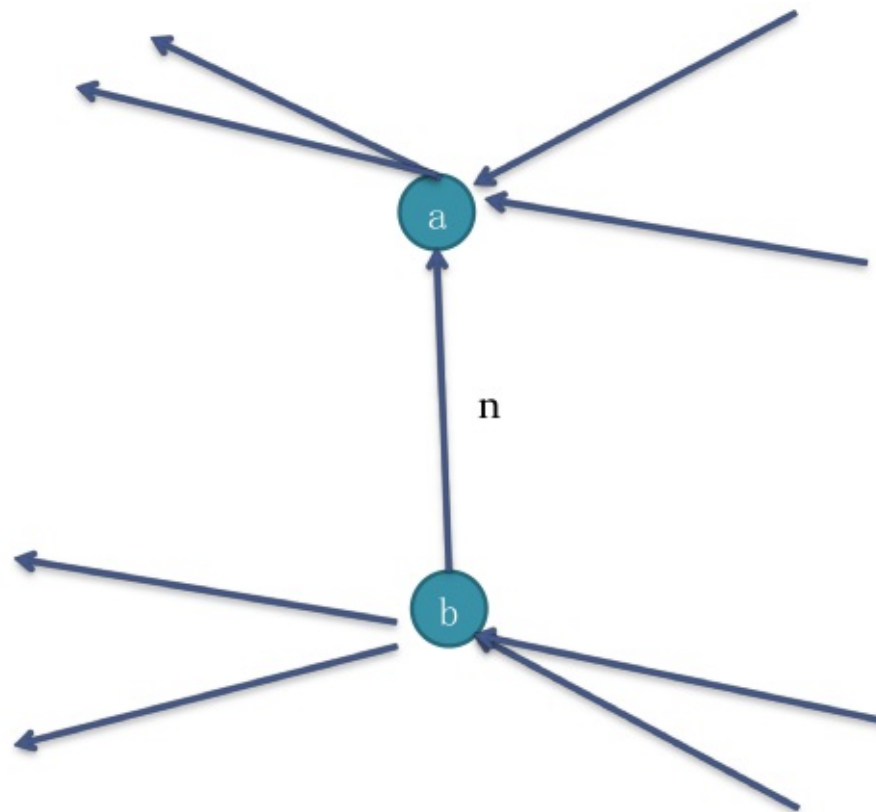
- 第二次DFS:



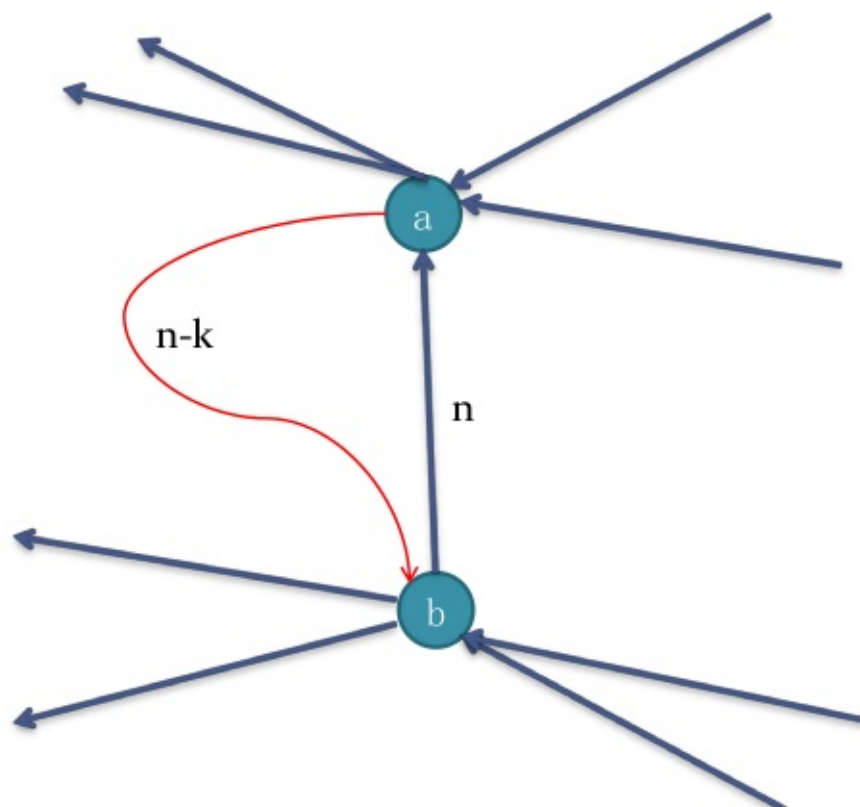
(这样做的好处是什么？为什么这么做？)

2. 为什么添加反向边（取消流）的操作是有效的？

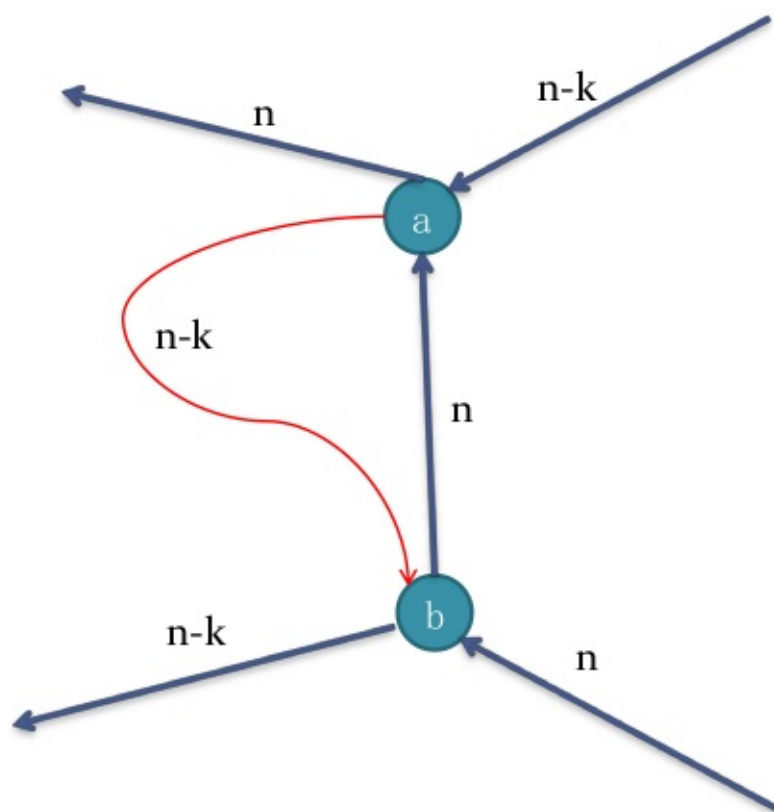
- i. 假设在第一次寻找流的时候，发现在  $b \rightarrow a$  上可以有流量来自源，到达b，再流出a后抵达汇点。



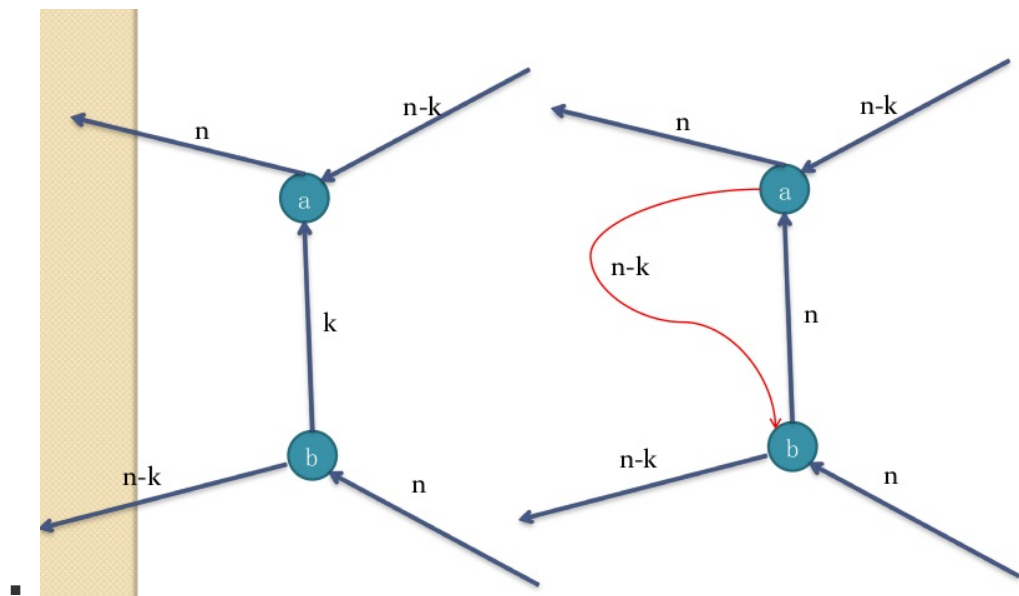
- ii. 构建残余网络时添加  $a \rightarrow b$ ，容量是n，增广的时候发现了流量n-k，即新增了 n-k 的流量。这n-k的流量，a进b出，最终留到汇



- iii. 现在要证明  $2n-k$  的流量，在原图上确实是可以从源流到汇的。  
 iv. 把流入  $b$  的边合并，看做一条，把流出  $a$  的边也合并，同理把流入  $a$  的和流出  $b$  的边也合并

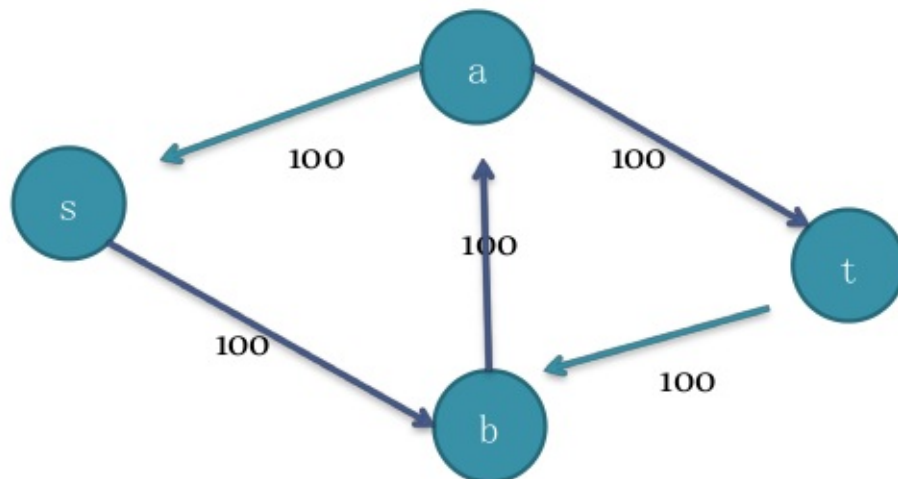


- v. 再合并



vi. 这是一个感性的认识, 并非严格证明。

3. 答：反向边的存在使得该流量没有被删除掉 这样我们第二次DFS搜索的时候就可以在新的网络里找到新的路径。 这是一个取消流的操作，也可以看作是两条路径的合并。



4. 两次搜索分别找到流量为100的流，加上第一次搜索得到的流量为100的流，总流量 上升到 200.
5. (因为反向边的存在，抵消了两次互相灌输)

## 概念

- 残余网络 在一个网络流图上，找到一条源到汇的路径（即找到了一个流量）后，对路径上所有的 边，其容量都减去此次找到的流量，对路径上所有的边，都添加一条反向边，这样得到 的新图，就称为原图的"残留网络"。
- 增广路径 每次寻找新流量并构造新残余网络的过程，就叫做寻找流量的"增广路径"， 也叫增广。 特别的，增广过后的图，就是残余网络。

## 复杂度分析

- 现在假设：每条边的容量都是整数，这个算法每次都能将流至少增加1。由于整个网络的流量最多不超



过图中所有的边的容量和 $C$ ，从而算法会结束。

- 看复杂度: 增广路径算法可以用DFS，复杂度为边数 $m$ +顶点数 $n$  DFS最多运行 $C$ 次 所以时间复杂度为  $O(C(m+n))=O(Cn^2)$
- 结论 这个算法实现很简单，但是注意到在图中， $C$ 可能Huge。比如说下图：

。

如果运气不好，这种图会让程序跑200次DFS——虽然实际上最少只要两次我们就能得到最大流。

- 那么问题就来了：
  - 学xx哪家强？
  - 如何避免上述情况发生？在每次增广的时候，选择从源到汇的具有最少边数的增广路径，即不是通过DFS寻找增广路径，而是通过BFS寻找增广路径。这就是著名的**Edmonds-Karp**最短增广路算法。
  - 已经证明这种算法的复杂度上限为  $O(nm^2)$  ( $n$ 是点数， $m$ 是边数)
  - 相对另一个复杂度为  $O(C \cdot n^2)$
- 测试题目[poj1273](#)

## Dinic 快速网络流算法

前面的网络流算法，每进行一次增广，都要做一遍BFS，十分浪费。能否少做几次BFS？

这就是Dinic算法要解决的问题

- Edmonds-Karp的提高余地在于：需要多次从 $s$ 到 $t$ 调用BFS，可以设法减少调用次数。
- 亦即：使用一种代价较小的高效的增广算法（万变不离其宗）
- 考虑：在一次增广的过程中，寻找多条增广路径
- DFS

## 具体实现

1. 首先利用 **BFS** 对残余网络分层

。

。

。一个节点的"层"数，就是源点到它最少要经过的边数。

2. 分层结束后，利用**DFS**从前一层往后一层反复寻找增广路。即：要求DFS每一步都必须走到下一层的节点。

。因此，前面在分层时，只要进行到汇点的层次数被算出即可停止，因为按照DFS的规则，和汇点同层或下一层的节点，是不可能走到汇点的。(为什么?)

。DFS过程中，要是碰到了汇点，则说明找到了一条增广路径。此时要增加总流量的值，消减路径上各边的容量，添加反向边，即所谓的进行增广。

。DFS找到一条增广路径后，并不立即结束，而是回溯后继续寻找下一个增广路径。回溯到哪一个节点呢？回溯到的节点u满足以下条件：

i. DFS搜索树的树边(u, v)上的容量已经变成0。即刚刚找到的增广路径上所增加的容量，等于(u, v)本次增广前的容量（DFS的过程中，是从u走到更下层的v的）。

ii. u是满足条件1的最上层的节点。如果回溯到源点且无法继续往下走了，DFS结束。因此，一次DFS的过程中，可以找到多条增广路径。DFS结束后，对残余网络再次分层，然后进行DFS。当残余网络的分层操作无法算出汇点的层次（即BFS到达不了汇点），算法结束，最大流求出。一般用栈实现DFS，这样就能从栈中取出增广路径。

3. 要求出最大流中每条边的流量，怎么办？

- 将原图备份，原图上的边的容量减去昨晚最大流的残余网络上的边的剩余容量，就是边的流量。

## 复杂度

---

- Dinic的复杂度是  $O(n^2 * m)$  (n是点数，m是边数)

## 题目：

---

- 测试题目
    - [poj1273](#)
    - [poj3436](#)
    - [poj2112](#)
    - [poj1149](#)
- 

## 有流量下界的网络最大流

---

## 临时的想法

---

如果使用水流法来判断最大流——即模拟过程，利用端点蓄水量来判断水流流向。

利用动态规划，单位蓄水。需要BFS分层——是否就是第二个算法。

# 解题报告

---

## 相关学习 资料

---