

NOI 学习建议

——赛程及算法整理

LiCH

2024 年 9 月 27 日

前言

本文主要整理 NOI 赛程、考纲中所包含相关知识点，并提供相对应的学习建议，以供各位同学学习使用。

LiCH

2024 年 9 月 27 日

目录

第一章 赛程	1
第二章 NOIP 学习建议	2
2.1 学习编程语言	2
2.2 从排序入手	2
2.3 贪心、穷举、模拟算法	3
2.4 动态规划	3
2.5 学习简单的图论	4
2.6 常用的数据结构	4
2.7 搜索	5
2.8 重要的数学基础知识	6
第三章 C++ 语言	7
3.1 从 Hello World 开始	7
3.2 数据类型	8
3.2.1 整型	9
3.2.2 浮点型	9
3.2.3 字符型	9
3.2.4 布尔型	9
3.2.5 字符串	10

3.3	常量与变量	10
3.3.1	sizeof 关键字	11
3.4	运算	11
3.5	string 类	11
3.5.1	构造函数	11
3.5.2	赋值	12
3.5.3	求字符串的长度	12
3.5.4	字符串拼接	13
3.5.5	string 对象的比较	13
3.5.6	求 string 对象的子串	14
3.5.7	交换两个 string 对象的内容	14
3.5.8	查找子串和字符	14
3.5.9	替换子串	16
3.5.10	删除子串	16
3.5.11	插入字符串	17
3.6	指针	17
3.6.1	指针的基本概念	17
3.6.2	const 修饰指针	18
3.7	结构体	19
3.7.1	结构体数组	20
3.7.2	结构体指针	20
3.7.3	结构体做参数传递	21
3.8	引用	22
3.8.1	引用的一些作用	22
3.9	C++ 中的函数	25
3.9.1	函数的默认参数	26
3.9.2	函数占位参数	26

目 录	III
3.9.3 函数重载	27
3.10 类和对象	29
3.10.1 封装	29
第四章 排序算法	33
4.1 插入类排序	34
4.1.1 直接插入排序	34
4.1.2 希尔排序	35
4.2 交换类排序	36
4.2.1 冒泡排序	36
4.2.2 快速排序	37
4.3 选择类排序	39
4.3.1 简单选择排序	39
4.3.2 锦标赛排序	40
4.3.3 堆排序	40
4.4 归并排序	40
4.5 分配类排序	41
4.5.1 基数排序	41
4.5.2 计数排序	42
4.5.3 桶排序	43
4.6 排序算法实现	43
4.6.1 直接插入排序	43
4.6.2 希尔排序	44
4.6.3 冒泡排序	45
4.6.4 快速排序	45
4.6.5 简单选择排序	46
4.6.6 堆排序	47
4.6.7 归并排序	48

4.6.8 计数排序	49
第五章 算法基础	51
5.1 算法分析	51
5.1.1 数学基础	51
5.1.2 排列组合	52
5.1.3 运行时间的计算	54
5.1.4 运行时间中的对数	55
5.1.5 主定理	56
5.2 算法思想	57
5.2.1 蛮力法	57
5.2.2 分治法	58
5.2.3 减治法	60
5.2.4 变治法	61
5.2.5 动态规划	61
5.2.6 时空权衡	69
5.2.7 贪心算法	70
5.2.8 模拟算法	74
5.3 快速幂	75
5.3.1 一个拓展	77
5.4 高精度	80
5.4.1 四则运算	81
第六章 搜索算法	89
6.1 暴力搜索算法	89
6.1.1 深度优先搜索	91
6.1.2 宽度优先搜索	92
6.2 记忆搜索算法	94

6.2.1	剪枝策略	98
6.3	双向搜索	103
6.3.1	双向宽度优先搜索	104
6.3.2	双向迭代加深	104
第七章	数据结构	109
7.1	栈	109
7.1.1	单调栈	109
7.2	队列	117
7.2.1	单调队列	117
7.3	链表	121
7.4	树	121
7.4.1	预备知识	121
7.4.2	线段树	121
7.4.3	zkw 线段树	130
7.4.4	动态开点线段树	136
7.4.5	权值线段树	137
7.4.6	可持久化线段树	139
7.4.7	珂朵莉树	151
7.4.8	划分树	162
7.5	图	167
7.5.1	图的存储	167
7.5.2	链式前向星	169
7.6	堆	173
7.7	哈希表	177
7.7.1	位运算	177
7.7.2	集合与集合	178
7.7.3	集合与元素	179

7.7.4 遍历集合	180
7.7.5 枚举集合	180
7.8 并查集	183
7.8.1 并查集引入	183
7.8.2 并查集实现	185
7.8.3 路经压缩	186
7.8.4 按秩合并	188
7.8.5 种类并查集	196
7.9 树状数组	205
7.9.1 树状数组的引入	205
7.9.2 树状数组的实现	207
7.10 ST 表	214
7.11 分块	218
第八章 字符串	227
8.1 KMP 算法	227
8.1.1 拓展 KMP 算法	235
8.2 字典树	239
8.2.1 01 字典树	249
8.3 Manacher 算法	256
8.4 后缀数组	264
8.5 确定有限状态自动机	270
8.5.1 形式化定义	272
8.6 后缀自动机	276
8.7 AC 自动机	283
第九章 树上的问题	289
9.1 最近公共祖先问题	289

9.1.1 Tarjan 算法	297
9.2 重链剖分	300
9.2.1 LCA 问题	302
9.2.2 结合数据结构	306
9.3 树的重心	309
9.4 点分治	319
9.5 长链剖分	328
第十章 数论	336
10.1 欧几里得算法	336
10.2 拓展欧几里得算法	337
10.3 逆元	341
10.3.1 拓展欧几里得法求解	342
10.3.2 费马小定理求解	343
10.3.3 线性递推法	343
10.4 中国剩余定理	344
10.5 素数筛	350
10.6 欧拉函数	353
第十一章 图论	356
11.1 最短路问题	356
11.1.1 Floyd 算法	356
11.1.2 Bellman-Ford 算法	359
11.1.3 SPFA 算法	361
11.1.4 Dijkstra 算法	366
11.1.5 打印路径	372
11.2 匈牙利算法	373
11.2.1 最大匹配问题	374

11.2.2 最小点覆盖问题	376
11.3 图的连通性	380
11.3.1 有向图的强连通分量	380
11.3.2 无向图的双连通分量	394
11.4 负环问题	414
11.5 差分约束系统	424
11.6 最小生成树	435
11.6.1 Prim 算法	436
11.6.2 Kruskal 算法	439
11.7 拓扑排序	441
11.8 单源次短路问题	448
11.8.1 边不可重复经过的次短路问题	449
11.8.2 边可重复经过的次短路问题	453
11.9 基环树	458
11.9.1 找环	458
11.9.2 断环法	460
11.9.3 二次 DP 法	466
11.10 欧拉路径与欧拉回路	469
第十二章 动态规划	471
12.1 动态规划基础	471
12.2 线性 DP	472
12.3 背包 DP	479
12.3.1 0-1 背包问题	480
12.3.2 完全背包问题	486
12.3.3 多重背包问题	489
12.3.4 混合背包	492
12.3.5 二维费用背包	495

目 录	IX
12.3.6 分组背包	499
12.4 区间 DP	501
12.5 数位统计 DP	508
12.6 计数类 DP	516
12.7 状态压缩 DP	518
12.8 树形 DP	524
第十三章 计算几何	537
13.1 计算几何基础	537
13.1.1 几何对象	537
13.1.2 常用常量	538
13.1.3 基础操作	538
13.1.4 向量	540
13.1.5 直线	541
13.1.6 线段	542
13.1.7 几何对象之间的关系	543
13.1.8 距离	545
13.1.9 平移和旋转	546
13.1.10 对称	547
13.1.11 交点	548
13.1.12 三角形的四心	549
第十四章 组合数学	551
14.1 生成函数	551
第十五章 多项式	560
15.1 快速傅里叶变换	560
15.1.1 离散傅里叶变换	560
15.1.2 快速傅里叶变换	561

第十六章 线性代数	563
16.1 线性基	563
第十七章 杂项	566
17.1 摩尔投票法	566
17.1.1 摩尔投票法的拓展	567
17.2 集合论	569
17.2.1 集合的基本定义	569
17.2.2 集合的运算	570
17.3 离散化	571
第十八章 初赛	576
18.1 计算机概述	576
18.1.1 发展历史	576
18.1.2 计算机结构与硬件	577
18.1.3 计算机语言	581
18.1.4 信息编码	583
18.1.5 机器数与真值	584
18.1.6 计算机网络	585
18.1.7 计算机网络安全	589
18.2 排列组合	591
18.2.1 排列	591
18.2.2 组合	592
18.2.3 抽屉原理	592
18.3 概率与期望	594
18.4 杂项	596
18.4.1 图片与视频的大小计算	596
18.4.2 音频的大小计算	596

第一章 赛程

信息学竞赛的常见赛程如表1.1所示。其中，省级联赛 (NOIP) 分为普及组和提高组，普及组针对初中生，提高组主要针对高中生，但也允许水平高的初中生参加。IOI 和 APIO 均为 ACM 赛制，即在比赛途中可以及时知道自己的成绩。

表 1.1: 信息学竞赛常见赛程

竞赛名称	竞赛日期	备注
CSP 初赛	9月	考计算机基础知识、算法基础知识、写出程序运行结果、程序补充填空。
CSP 复赛	10月	程序编写，根据测试用例覆盖情况得分。
NOIP	11月	——
NOI 省选	5月	——
NOI	7月	全国决赛，共 2 天，每天 3 题 5 小时，前 50 名选手成为国家集训队队员。
CCF 冬令营	1/2 月	国家集训队 50 进 15，即预备队选拔赛。
CTSC	4/5 月	国家集训队 15 进 4，即国家队选拔赛。
IOI	夏季	——
*APIO	5月	亚洲及太平洋地区比赛，ACM 赛制。

通常信息学竞赛的流程为 CSP 初赛 →CSP 复赛 →NOIP→NOI 省选 →NOI。

第二章 NOIP 学习建议

2.1 学习编程语言

编程语言有很多，曾经 NOIP 官方指定语言包括 C++，Pascal，C 三种，然而，自 2020 年开始，NOI 系列其他赛事（包括 CCF 冬令营、CTSC、APIO、NOI）将不再支持 Pascal 语言和 C 语言，且自 2022 年开始，NOIP 竞赛也将不再支持 Pascal 语言，因此需要各位同学尝试学习并熟悉使用 C++ 语言。

2.2 从排序入手

排序是基础中的基础，常见的排序有：插入排序、希尔排序、选择排序、冒泡排序、快速排序、堆排序、基数排序、计数排序、桶排序、归并排序十种。快速排序是必备本领，最简单的掌握方法就是背下来，其他排序也需要熟悉其执行过程、时间空间复杂度。多关键字排序和稳定排序也是必须掌握的排序知识。

2.3 贪心、穷举、模拟算法

想得奖，必须掌握贪心和穷举以及模拟，虽然不能让你得满分，但可以给你拿到 30-60 分。它们是你想不出更好算法时的救命稻草。

贪心算法（又称贪婪算法）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，他所做出的是在某种意义上的局部最优解。但是贪心是可以得分的。

穷举（枚举）算法是指，通过暴力穷举，列举出所有可能的取值，从中找出最优解。

模拟算法是指，通过逐步进行操作、逐步判断来推断是否符合题目中所给出的情况。非常耗时，一般不可能得到最优解，但是可以得到部分分数。

2.4 动态规划

动态规划比较难，对思维的周密程度和逻辑要求非常高。可以用来训练思维，对于学习时间短的同学，动态规划可以帮助你迅速进入编程状态，也有助于帮你发现题目背后可能隐藏的更简便的算法。动态规划主要的思考规律应该如下：

定义函数（状态转移方程） → 建立方程 → 确定初值和边界

需要提醒的是：如果考场上想不到状态转移方程，请选择贪心、枚举或模拟等方法来获得部分分数。动态规划最后得出的答案不正确时，也不要耗费大量时间来找出错误，因为这非常难，也非常耗时间，得不偿失。

2.5 学习简单的图论

图论部分核心是两个数据结构：（单源或多源）最短路和（最小）生成树，及对应的相关算法。

针对最短路中需要学习 Dijkstra 算法和 Floyd 算法。

注：近年来图论题目越来越难，知识点也越来越多，如果学习的时间不够，请至少掌握这两种。

最小生成树部分需要掌握 Prim 算法和 Kruskal 算法。前者适用于稠密图，后者适用于疏密图。两者可以比较学习，看到它们的优点和不足。

2.6 常用的数据结构

合理地使用数据结构可以使程序的执行速度更快并节省空间，NOI 大纲内要求的数据结构如表2.1、表2.2所示，往年试题中，最常用到的是堆（优先队列）、并查集以及树状数组堆。

表 2.1: 数据结构

分级	大类	具体内容
提高级	线性结构	双端栈、双端队列、有序队列、优先队列、倍增表 (ST 表)
提高级	集合与森林	等价类、并查集、树与二叉树的转化 (孩子兄弟表示法)
提高级	特殊树	线段树与树状数组、字典树 (trie 树)、笛卡尔树、AVL 树 (treap、splay 等)、基环树
提高级	常见图	稀疏图、二分图、欧拉图、有向无环图、连通图与强连通图、重连通图

表 2.2: 数据结构

分级	大类	具体内容
入门级	线性表	链表：单链表、双向链表、循环链表 栈、队列
入门级	简单树	树的定义及相关概念、树的父亲表示法 二叉树的定义及基本性质 二叉树的孩子表示法 二叉树的先序、中序、后序遍历
入门级	特殊树	完全二叉树的定义及基本性质、数组表示法 Huffman 树的定义、构造、遍历 二叉排序树的定义、构造、遍历
入门级	简单图	图的定义和相关概念 图的邻接矩阵存储 图的邻接表存储

堆：只关注“直系亲属关系”，不关注“旁系”。常配合贪心使用。

并查集：快速判断两个元素是否有关联，增加其他算法，还可判断元素间关系。

树状数组堆：平衡查询和修改的操作复杂度的一种算法，常用于解决需要查询和修改的问题。

2.7 搜索

搜索和枚举很像，主要是深度优先搜索和广度优先搜索。

深度优先搜索：一条路走到底。

广度优先搜索：每一步将下一步的可能性放入队列中，然后按照队列顺序来探测。

2.8 重要的数学基础知识

快速幂、高精度、筛法选素数、辗转相除法。

第三章 C++ 语言

如前文所述，NOI 的大多数赛事只能选用 C++ 语言，因而需要同学对 C++ 语言有一定的掌握。本章将会对 C++ 语言的基础部分进行一定的介绍，对于熟悉 C++ 的同学可以直接从下一章开始。

3.1 从 Hello World 开始

在屏幕上输出“hello world”已经成为了学习一种新的语言传统，同样的，我们本次也开始尝试在屏幕上使用 C++ 语言输出“Hello Wordl”。

```
#include<iostream> //导入库文件

using namespace std; //使用标准命名空间

int main() { //主函数
    cout << "hello world" << endl //输出
    return 0; //结束函数
}
```

上面所示就是 C++ 语言输出“hello world”的程序，接下来我们就介绍一下图中的相关部分：

- (1) **注释**，在 C++ 程序中使用“//”进行注释用以解释代码的功能，“//”后的内容将被编译器忽略，不会参与程序的编译运行，此外，还可以使用“/*”和“*/”进行多行注释。

(2) #include<iostream>, 在程序中调用一个库函数, 包含该函数原型的头文件就必须要有, 一般写在程序开头。该程序中使用了 cout 进行输出, 它的声明在 iostream 中, 因而需要在头文件中事先声明。

(3) using namespace std, 即使用 std(标准) 命名空间, C++ 标准库中的所有函数和对象都在 std 命名空间中定义, 在交代使用的命名空间后标准库中的 cin、cout 等输入输出语句才能直接使用。

(4) int main() 为主函数的声明, 主函数是所有程序的入口, 是所有函数中第一个被执行的, main 后总跟着一对圆括号 () 用以表示它是一个函数, 括号中可以包含函数所需要的参数, 也可以什么都没有, 但即使没有参数也不能省略。int 用以表示函数的函数值是整型, 函数主体包含在一对花括号 {} 内。

(5) cout « "hello world" « endl, cout 是输出语句, 告诉计算机将引号间的字符串输出, endl 是 C++ 语言的换行控制符, 表示输出后换行输出后续内容。

(6) return 0, 主函数的返回语句, 返回语句用以结束该函数, 只要遇到 return 就会结束执行, 即使后面还有其他语句也不再执行, return 后的 0 表示顺利结束, 其他数表示有异常。

在 C++ 中, 语句间以“;”进行分隔, 需要注意的是, 除注释、字符串外所有的符号都需要在英文状态 (半角情况) 下输入, 且 C++ 语言严格区分大小写, 同时为了方便阅读, 尽量让一个语句占一行。

3.2 数据类型

通常, 数据类型会分为两大类:

基本类型 整型、字符型、指针类型、空类型

构造类型 枚举型、数组、结构体、共用体

3.2.1 整型

整型又可以分为整型和长整型两种，整型用关键字 int 表示，占 32 位，即 4 个字节，其取值范围 $[2^{31}, 2^{31} - 1]$ ；

长整型用关键字 long long 表示，占 64 位，即 8 个字节，其取值范围 $[2^{63}, 2^{63} - 1]$ 。

此外，整型是可以有符号的，有符号的整型用关键字 signed 表示，其最左位的数字代表符号，其余位表示数值，当符号位为 1 则代表数值为负数，符号位为 0 则代表数值为正数；无符号整型用关键字 unsigned 表示，所有位均代表数值。

3.2.2 浮点型

浮点型同样分为单精度型和双精度型两种，单精度型用关键字 float 表示，占 32 位，其取值范围 $[-3.4 \times 10^{38}, 3.4 \times 10^{38}]$ ；

双精度型则用关键字 double 表示，占 64 位，其取值范围 $[-1.7 \times 10^{308}, 1.7 \times 10^{308}]$ 。

3.2.3 字符型

字符型通常用来表示单个字符，表现形式为单引号引起的字符，如'A'、'0'、'z'，用关键字 char 表示，字符型所占 8 位，取值范围 $[-128, 127]$ 。

此外，ASCII 码表有 128 个字符，可以用 char 关键字的所有正数表示 ASCII 码值，其中：48-57 代表 0-9 十个阿拉伯数字，65-90 代表 26 个大写英文字母，97-122 代表 26 个小写英文字母。

3.2.4 布尔型

布尔型用关键字 bool 表示，只占 1 位，表示真、假两种情况，真可以用 true 和 1 表示，假可以用 false 和 0 表示。

3.2.5 字符串

字符串在 C++ 语言中有两种风格：

C 语言风格的字符串，char 变量名 [] = "字符串值"；

C++ 风格字符串，String 变量名 = "字符串值"；

这里我们先看 C 语言风格的字符串：

```
char str1[8] = "program";
char str2[8] = {'p', 'r', 'o', 'g', 'r', 'a', 'm'};
char str3[] = "program";
```

3.3 常量与变量

常量代表在程序运行过程中不能改变的量，在 C++ 语言中常常用需要自己定义符号常量，其有两种定义方式，具体如下：

```
//使用宏定义
#define E 2.71828182
//使用符号常量定义
const double pi = 3.1415926535; // 分号结尾
```

变量则代表在程序运行时可以改变的量，变量主要由变量类型、变量名、变量值三个部分组成，其定义方式如下：

```
int a;
float b = 1.0;
char c = 'c';
bool d = true;
```

变量必须先定义再使用，变量名用于标识存放数据的存储单元，如果定义变量时没有初始值可以省略。变量在命名时需要规避关键字，必须以字母或下划线开头。

3.3.1 sizeof 关键字

利用 sizeof 关键字可以统计数据类型所占内存大小。

语法: sizeof(数据类型/变量)。

```
int temp = 1;  
sizeof(temp);  
sizeof(int); // 两种写法求得的值相同
```

3.4 运算

基本的运算符包含: + 加法, - 减法, * 乘法, / 除法, % 取余 (取模) 五种, 其中 +, - 除加减运算外还可以表示数值的正负, 如需进行乘法运算, * 不可省略。

除法运算时会根据数据类型产生不同的结果, 例如两个整型 $3 / 2$ 得到的结果是 1 而非 1.5, $9 / 4.0$ 的结果则为 2.25。

取模运算即求余运算, 所操作的数值是整型数, 其结果也是整型, 例如 $8 \% 6$ 的结果是 2。

除上述基本运算外, 还有 ++ 自增和 -- 自减运算, 以自增运算为例, $x = i++$ 代表先将 i 的值赋给 x 然后 i 增加 1, $x = ++i$ 则代表先将 i 增加 1 然后将增加后的 i 的值赋给 x 。

3.5 string 类

3.5.1 构造函数

string 类有多个构造函数, 具体用法如下:

```
string s1(); // s1 = ""  
string s2("Hello"); // s2 = "Hello"  
string s3(4, 'K'); // s3 = "KKKK"
```

```
string s4("12345", 1, 3); //s4 = "234", 即 "12345" 的从下标 1  
开始, 长度为 3 的子串
```

string 类没有接收一个整型参数或一个字符型参数的构造函数。下面的两种写法是错误的：

```
string s1('K');  
string s2(123);
```

3.5.2 赋值

可以用char*类型的变量、常量，以及char类型的变量、常量对 string 对象进行赋值。例如：

```
string s1;  
s1 = "Hello"; // s1 = "Hello"  
s2 = 'K'; // s2 = "K"
```

string 类还有assign成员函数，可以用来对 string 对象赋值。assign 成员函数返回对象自身的引用。例如：

```
string s1("12345"), s2;  
s3.assign(s1); // s3 = s1  
s2.assign(s1, 1, 2); // s2 = "23", 即 s1 的子串(1, 2)  
s2.assign(4, 'K'); // s2 = "KKKK"  
s2.assign("abcde", 2, 3); // s2 = "cde", 即 "abcde" 的子串(2, 3)
```

3.5.3 求字符串的长度

s.length()和s.size()都返回字符串的长度。

3.5.4 字符串拼接

除了可以使用+和+=运算符对 string 对象执行字符串的连接操作外，string 类还有append成员函数，可以用来向字符串后面添加内容。append 成员函数返回对象自身的引用。例如：

```
string s1("123"), s2("abc");
s1.append(s2); // s1 = "123abc"
s1.append(s2, 1, 2); // s1 = "123abcbc"
s1.append(3, 'K'); // s1 = "123abcbcKKK"
s1.append("ABCDE", 2, 3); // s1 = "123abcbcKKCDE", 添加 "ABCDE" 的子串 (2, 3)
```

3.5.5 string 对象的比较

除了可以用<、<=、==、!=、>=、>运算符比较 string 对象外，string 类还有compare成员函数，可用于比较字符串。compare成员函数有以下返回值：小于 0，表示当前的字符串小；

等于 0，表示两个字符串相等；

大于 0，表示另一个字符串小。

几个例子：

```
string s1("hello"), s2("hello, world");
int n = s1.compare(s2);
n = s1.compare(1, 2, s2, 0, 3); // 比较 s1 的子串 (1,2) 和 s2 的子串 (0,3)
n = s1.compare(0, 2, s2); // 比较 s1 的子串 (0,2) 和 s2
n = s1.compare("Hello");
n = s1.compare(1, 2, "Hello"); // 比较 s1 的子串 (1,2) 和 "Hello"
n = s1.compare(1, 2, "Hello", 1, 2); // 比较 s1 的子串 (1,2) 和 "Hello" 的子串 (1,2)
```

3.5.6 求 string 对象的子串

`substr`成员函数可以用于求子串 (n, m)，原型如下：

```
string substr(int n = 0, int m = string::npos) const;
```

调用时，如果省略 m 或 m 超过了字符串的长度，则求出来的子串就是从下标 n 开始一直到字符串结束的部分。例如：

```
string s1 = "this is ok";
string s2 = s1.substr(2, 4); // s2 = "is i"
s2 = s1.substr(2); // s2 = "is is ok"
```

3.5.7 交换两个 string 对象的内容

`swap`成员函数可以交换两个 `string` 对象的内容。例如：

```
string s1("West"), s2("East");
s1.swap(s2); // s1 = "East", s2 = "West"
```

3.5.8 查找子串和字符

`string` 类有一些查找子串和字符的成员函数，它们的返回值都是子串或字符在 `string` 对象字符串中的位置（即下标）。如果查不到，则返回`string::npos`。`string::npos`是在 `string` 类中定义的一个静态常量。这些函数如下：

`find`: 从前往后查找子串或字符出现的位置。

`rfind`: 从后往前查找子串或字符出现的位置。

`find_first_of`: 从前往后查找何处出现另一个字符串中包含的字符。例如：`s1.find_first_of("abc")`; 是查找 `s1` 中第一次出现"abc" 中任一字符的位置。

`find_last_of`: 从后往前查找何处出现另一个字符串中包含的字符。

`find_first_not_of`: 从前往后查找何处出现另一个字符串中没有包含的字符。

`find_last_not_of`: 从后往前查找何处出现另一个字符串中没有包含的字符。

下面是 `string` 类的查找成员函数的示例程序：

```
#include <iostream>
#include <string>
using namespace std;
int main() {
    string s1("Source Code");
    int n;
    if ((n = s1.find('u')) != string::npos) //查找 u 出现的位置
        cout << "1) " << n << ", " << s1.substr(n) << endl;
    //输出 1) 2, urce Code
    if ((n = s1.find("Source", 3)) == string::npos)
        //从下标3开始查找 "Source", 找不到
        cout << "2) " << "Not Found" << endl; //输出 2) Not Found
    if ((n = s1.find("Co")) != string::npos)
        //查找子串 "Co"。能找到, 返回 "Co" 的位置
        cout << "3) " << n << ", " << s1.substr(n) << endl;
    //输出 3) 7, Code
    if ((n = s1.find_first_of("ceo")) != string::npos)
        //查找第一次出现或 'c'、'e' 或 'o' 的位置
        cout << "4) " << n << ", " << s1.substr(n) << endl;
    //输出 4) 1, ource Code
    if ((n = s1.find_last_of('e')) != string::npos)
        //查找最后一个 'e' 的位置
        cout << "5) " << n << ", " << s1.substr(n) << endl; //输出 5) 10, e
    if ((n = s1.find_first_not_of("eou", 1)) != string::npos)
        //从下标1开始查找第一次出现非 'e'、'o' 或 'u' 字符的位置
        cout << "6) " << n << ", " << s1.substr(n) << endl;
```

```
//输出 6) 3, rce Code  
return 0;  
}
```

3.5.9 替换子串

`replace`成员函数可以对 `string` 对象中的子串进行替换，返回值为对象自身的引用。例如：

```
string s1("Real Steel");  
s1.replace(1, 3, "123456", 2, 4); //用 "123456" 的子串 (2,4)  
    替换 s1 的子串 (1,3)  
cout << s1 << endl; //输出 R3456 Steel  
string s2("Harry Potter");  
s2.replace(2, 3, 5, '0'); //用 5 个 '0' 替换子串 (2,3)  
cout << s2 << endl; //输出 Ha00000 Potter  
int n = s2.find("00000"); //查找子串 "00000" 的位置, n=2  
s2.replace(n, 5, "XXX"); //将子串 (n,5) 替换为 "XXX"  
cout << s2 << endl; //输出 HaXXX Potter
```

3.5.10 删除子串

`erase`成员函数可以删除 `string` 对象中的子串，返回值为对象自身的引用。例如：

```
string s1("Real Steel");  
s1.erase(1, 3); //删除子串 (1, 3), 此后 s1 = "R Steel"  
s1.erase(5); //删除下标5及其后面的所有字符, 此后 s1 = "R Ste"
```

3.5.11 插入字符串

`insert`成员函数可以在 `string` 对象中插入另一个字符串，返回值为对象自身的引用。例如：

```
string s1("Limitless"), s2("00");
s1.insert(2, "123"); //在下标 2 处插入字符串 "123", s1 = "
    Li123mitless"
s1.insert(3, s2); //在下标 2 处插入 s2 , s1 = "Li10023mitless
    "
s1.insert(3, 5, 'X'); //在下标 3 处插入 5 个 'X', s1 = "
    Li1XXXX0023mitless"
```

3.6 指针

3.6.1 指针的基本概念

我们在先前定义变量`int a = 10;`时，相当于在内存中开辟了一块4字节大小的内存空间来存储10这个数据，由于每个空间都有一个属于他自己的地址编号，我们假设这个空间的地址编号为`0x1234`。我们可以用这个变量的变量名`a`来操作这个数据，原因是如果有很多个变量，要记忆所有的地址编号较为困难，因为内存空间开辟不一定连续，因而我们才利用变量名来操作存储的数据。

但数据既然存储在内存中，那就意味着当我们知道地址编号的时候也通过地址可以直接访问这串数据。所以我们同样可以设计一种变量来存储地址，这种变量就叫做指针，可以理解为`p = 0x1234;`。

内存是有地址编号的，通常用十六进制数字表示，从0开始记录，我们可以通过指针来间接访问内存，也就是用指针变量来保存地址编号。简单来说，**指针就是地址**。

指针的简单使用如下：

```
// 定义指针
int a = 10;
int * p; // 语法： 数据类型 * 指针变量名；
p = &a; // & 取地址符，用来获取 a 的地址

// 使用指针，通过解引用的方式来找到指针指向的内存
*p = 20; // 指针变量前加 * 表示解引用
```

既然指针也是数据类型，那么也会占用内存空间，在 **32 位操作系统** 下，指针占用 **4** 个字节，在 **64 位操作系统** 下，指针占用 **8** 个字节。

在指针定义时，有两个概念需要注意一下：**空指针**和**野指针**。

所谓空指针，即为指针变量指向内存中编号为 **0** 的空间，一般用空指针来初始化指针变量。需要注意的是，空指针指向的内存是不可访问的，其原因在于**0-255**间的内存编号被系统占用。

```
int * p = NULL; // NULL 必须大写
```

所谓野指针，即为指针变量指向非法内存的指针。空指针和野指针都不是我们申请的空间，因此不要去访问。

3.6.2 const 修饰指针

前文提到过只要一个变量前用**const**来修饰，就意味着该变量里的数据只能被访问，而不能被修改，也就是意味着“只读”(readonly)。**const**修饰指针有三种情况：

1.**const**修饰指针 (常量指针): `const int * p = &a;`

常量指针的特点：指针**指向**可以修改，指针**指向的值**不可以修改。

```
*p = 10; // 错误，指针指向的值不能修改
p = &b; // 正确，指针指向可以修改
```

2.**const**修饰常量 (指针常量): `int * const p = &a;`

指针常量的特点：指针指向的值可以修改，指针指向不可以修改。

```
*p = 10; // 正确，指针指向的值可以修改  
p = &b; // 错误，指针指向不可以修改
```

3. const既修饰指针也修饰常量，`const int * const p = &a;` 其特点为指针的值和指针指向均不可修改。

3.7 结构体

结构体属于用户自定的数据类型，允许用户存储不同的数据类型。

语法：`struct 结构体名 { 结构体成员列表; }`

```
// 创建学生数据类型，学生包括：姓名，年龄，学号。  
struct Student {  
    string name;  
    int age;  
    int number;  
};
```

结构体创建有如下三种形式：

- `struct 结构体名 变量名`
- `struct 结构体名 变量名 = {成员, 成员2...}`
- 定义结构体时创建变量

```
// 方式1创建  
struct Student student1;  
student1.name = "Jerry";  
student1.age = 15;  
student1.number = 002;  
  
// 方式2创建  
struct Student student2 = {"Tom", 15, 001};
```

```
// 方式3创建
struct Student {
    string name;
    int age;
    int number;
} student3;
```

值得注意的是，在结构体进行定义时，`struct`关键字不可以省略，但在结构体变量进行创建时，`struct`关键字可以省略。

3.7.1 结构体数组

我们同样可以将结构体放入到数组中，其主要语法如下：

```
struct 结构体名 数组名[元素个数] = { {}, {}, {} ... }
```

我们以上述的 `Student` 结构体为例创建一个数组：

```
struct Student arrange[3] = {
    {"Tom", 15, 001}, // 逗号间隔
    {"Jerry", 15, 002},
    {"Spike", 15, 003}
}
//进行修改
arrange[2].age = 19;
```

3.7.2 结构体指针

我们同样可以通过指针访问结构体中的成员，利用操作符`->`通过指针访问结构体的属性。

```
// 创建结构体变量
struct student = {"Tom", 15, 001};
// 创建指针并指向结构体变量
```

```

struct student * p = &student;
// 通过指针访问结构体变量中的数据
p -> age = 19; // 效果等价于 (*p).age = 19;

```

3.7.3 结构体做参数传递

结构体同样可以作为参数进行传递，其传递方式主要有两种，我们仍以学生数据类型为例，具体如下所示：

```

// 方法一：值传递
void printStudent(struct Student student) {
    cout << student.name << " " << student.age << " "
        << student.number << endl;
}

// 方法二：地址传递
void printStudent(struct Student * p) {
    cout << p -> name << " " << p -> age << " "
        << p -> number << endl;
}
int main() {
    Student s;
    printStudent(s); // 值传递
    printStudent(&s); // 地址传递
}

```

我们需要注意一点，由于地址传递，传递过程中根据地址可以改变结构体元素的值，如果我们不想修改，可以在传参时用**const**关键字修饰，具体如下：

```

void printStudent(const struct Student * p) {
    cout << p -> name << " " << p -> age << " "
        << p -> number << endl;
}

```

3.8 引用

我们先来看引用的语法：数据类型 &变量名 = 原变量名；例如下面这个例子：

```
int a = 10;  
int &b = a;
```

此时，我们无论是修改a还是修改b，内存空间中保存的10都会发生改变，也就是两个变量指向了同一块内存区域。

因此，我们可以发现，**引用**的作用在于将内存空间的操作权限赋予另一个变量，通俗来说就是给原变量起个别名。使用引用时需要注意的是：**引用必须初始化，且引用经过初始化后就不可以更改。**

```
int a = 10;  
int b = 20;  
int &c; // 错误，引用必须有指向  
c = b; // 此时为赋值操作
```

3.8.1 引用的一些作用

如果我们要交换两个数，我们通常想的是利用指针直接根据地址修改内存空间的值，而这里，我们同样可以用**引用作为参数传递**来实现交换：

```
// 指针实现  
void swap1(int * a, int * b) {  
    int temp = *a;  
    *a = *b;  
    *b = temp;  
}  
  
// 引用实现  
void swap2(int & a, int & b) {  
    int temp = a;
```

```
a = b;
b = temp;
}

int main() {
    int a = 10;
    int b = 20;
    swap1(&a, &b); // 传入地址
    swap2(a, b);
}
```

总结来就是，利用引用技术，可以让形参修饰实参，从而简化指针修改实参的操作。

此外，引用同样可以作为**返回值**类型存在，但是需要注意引用作为返回值时**不要返回局部变量的引用**：

```
int& test() {
    int a = 10;
    return a;
}

int main() {
    int &temp = test();
    cout << temp << endl; // 能正确输出 10
    cout << temp << endl; // 不能正确输出 10
    return 0;
}
```

其原因在于，编译器会暂时保留局部变量，第二次错误输出是因为局部变量a的内存已经被释放。另一方面，引用作返回值时可以**作为左值**，具体如下：

```
int& test() {
    static int a = 10; // 静态变量，相当于全局变量
    return a;
}
```

```

int main() {
    int &temp = test();
    cout << temp << endl; // 输出 10
    test() = 1000;
    cout << temp << endl; // 输出 1000
    return 0;
}

```

其原因在于**test()**返回的相当于是全局变量**a**的地址，对此进行修改也就是对全局变量**a**进行修改。

所以，总结一下，引用在 C++ 的内部实现其实就是一个指针常量¹，即引用就是指针的简化操作。

```

// 发现是引用，转换为 int * const ref = &a;
void func(int &ref) {
    ref = 100; // ref 是引用，转换为 * ref = 100
}

int main() {
    int a = 10;
    // 自动转换为 int * const ref = &a; 指针常量是指针指向不可修改，也解释了引用不可更改
    int &ref = a;
    ref = 20; // 内部发行 ref 是引用，自动转化为：* ref = 20;
    cout << a << endl;
    cout << ref << endl; // 只要用到了引用，就会自动视为指针解引用
    func(a);

    return 0;
}

```

¹即指针指向不可修改，指针指向的值是可以改动的，int * const p = &a;

```
const int &ref = 10; // 引用必须指向合法的内存空间
// 加上const后，编译器会将代码修改，使得这个操作相当于：
int temp = 10;
const int & ref = temp;
int &ref = 10 // 是不允许这么做的，因为10作为字面量，存放在常
量区，不属于合法的内存空间
```

此外，引用前也可以用关键字**const**来限定，目的是防止误操作改变值，此操作称之为**常量引用**。

```
void show(const int &val) {
    val = 1000; // const修饰，不允许修改，因此此行错误
    cout << val << endl;
}
int main() {
    int a = 100;
    show(a)
}
```

这里需要提一下关键字**new**, **new**的操作如下：

```
int * p = new int(10);
int * arr = new int[10];
```

上面代码中**new int(10)**返回的是一个**地址**，需要**指针类型**来接收，通过关键字**new**来声明数组也是一样的，因而，我们通过关键字**new**产生什么样的数据类型，就需要用什么样的数据类型来接收。

3.9 C++ 中的函数

在 C++ 中函数有着一些特殊的用法，我们可以逐步来了解一下。

3.9.1 函数的默认参数

在 C++ 中，函数的形参列表是可以有默认值的，大致如下：

```
int sum(int a, int b = 10, int c = 10) {  
    return a + b + c;  
}  
  
int function(int a = 0, int b = 0); // 函数声明  
int function(int a, int b) { // 函数实现  
    return a + b;  
}
```

其基本语法为：返回值类型 函数名(参数 = 默认值) {}，接下来有几点需要注意一下：

- 默认值仅在**未指定变量值**时产生作用，如若指定变量的值，将使用指定的值执行程序；
- 如果某个位置有了**默认参数**，那么**从这个位置开始向右到最后，从左向右都必须要有默认值**；
- 函数声明和实现可以分开写，但如果**函数声明时有了默认参数，则实现时不能有参数**，反之亦然。

3.9.2 函数占位参数

C++ 中函数的形参列表里可以有占位参数，用来占位，调用函数时必须填补该位置，语法为：返回值类型 函数名(数据类型) {}，具体如下所示：

```
int example01(int a, int) {  
    cout << "test01" << endl;  
}  
  
// 占位参数也可以有默认值  
int example02(int a, int = 10) {  
    cout << "test02" << endl;
```

```
}

int main() {
    example01(10, 10); // 占位参数必须填补
    example02(10);    // 有默认值则无需填补
    return 0;
}
```

3.9.3 函数重载

C++ 中允许存在同名函数，其目的在于**提高复用性**，需要注意的是发生函数重载的条件为：在同一个作用域中，函数的名称相同，但函数**参数的类型不同或个数不同或顺序不同**，具体如下所示：

```
/***
 * 1、同一作用域中，函数的名称相同
 * 2、参数的 类型、 个数、 顺序 不同
 */
void show() {
    cout << "show 的调用" << endl;
}

void show(int a) {
    cout << "show(int a) 的调用" << endl;
}

void show(float a) {
    cout << "show(float a) 的调用" << endl;
}

void show(int a, float b) {
    cout << "show(int a, float b) 的调用" << endl;
}

void show(float a, int b) {
    cout << "show(float a, int b) 的调用" << endl;
}
```

特别提醒，函数的返回值不可以作为函数重载的条件，也就是：

```
void show() { ... }  
int show() { ... }
```

这样的代码是不合法的，编译器无法重载仅按返回值区分的函数。

函数重载存在一些注意事项，我们来依次看一下：

1. 引用作为函数重载的条件，例如我们来看一个例子：

```
void function(int &a) { // 函数1  
    cout << "function(int &a)" << endl;  
}  
  
void function(const int &a) { // 函数2  
    cout << "function(const int &a)" << endl;  
}  
  
int main() {  
    int a = 10;  
    function(a); // 调用函数1，原因在于a是变量，变量本身  
                // 可读可写，而const修饰变量会限制其为只读状态，因而  
                // 传入变量会走可读可写版本。  
    function(10); // 调用函数2，直接传入10相当于const int  
                  // &a = 10。  
}
```

2. 函数重载碰到默认参数，我们再来看一个例子：

```
void function(int a, int b = 10) { ... }  
void function(int a) { ... }
```

上述代码会存在语法错误，原因在于当函数重载碰到默认参数，可能出现**二义性**，产生语法错误，这种情况需要避免，即尽量在函数重载时不要给默认参数。

3.10 类和对象

C++ 作为面向对象语言，必然也具备面向对象的三大特性：封装、继承、多态。接下来我们来了解一下 C++ 作为面向对象语言的特性。

类和对象可以理解为：类是一系列特性的抽象概括，对象是类的具体实现。

3.10.1 封装

封装是 C++ 面向对象的三大特性之一，封装的意义在于：

- 将属性和行为作为一个整体囊括；
- 将属性和行为加以权限控制。

我们实现封装的方式就是将程序的属性和功能写到一个类，例如我们现在封装一个求圆周长的功能：

```
#include "iostream"

using namespace std;

const double Pi = 3.1415926;

class Cricle { // class 代表声明的是类，class 类名 {};
public: // 访问权限
    int radius; // radius, 半径，作为圆的属性

    double calculate() { // 行为，即函数实现
        return 2 * Pi * radius;
    }
};
```

有了类的抽象，我们就可以根据这个类来实现具体的圆对象，并为其属性赋值、调用该对象的方法：

```
int main() {  
    Cricle cricle; // 创建具体的圆对象 类名 对象名；  
    cricle.radius = 10; // 为该圆对象的属性赋值  
    cout << cricle.calculate() << endl; // 调用该圆对象的方法  
}
```

有几点需要注意一下：

- 类中的属性和行为统一称之为成员，例如成员属性（变量）、成员函数（成员方法）；
- 通过一个类创建对象称之为**实例化**的过程，即给这个类实现一个具体的例子。

刚刚在我们创建类的过程中用到了public这个访问权限，C++ 中访问权限共有三种：

- public，公共权限，类中可以访问，类外也可以访问；
- protect，保护权限，类中可以访问，类外不可以访问；
- private，私有权限，类中可以访问，类外不可以访问。

其中，protect和private的区别主要体现在当子类继承父类时，private修饰的属性和方法**不可以**被子类访问到，但protect修饰的内容**可以**被子类访问。

```
class Person {  
    // 公共权限  
public:  
    string name;  
    // 保护权限
```

```
protected:  
    int idCode;  
    // 私有权限  
  
private:  
    int secretCode;  
};  
  
int main() {  
    Person p;  
    p.name = "Tom"; // 正确，公共权限，类外可以访问  
    p.idCode = "1111"; // 错误，保护权限，类外不可以访问  
    p.secretCode = "2222" // 错误，保护权限，类外不可以访问  
    return 0;  
}
```

有了权限的概念之后，我们在定义类的时候，通常会将属性设置为私有，其目的在于避免属性被修改：

```
class Person {  
  
private:  
    string name;  
  
public:  
    void setName(string name) {  
        this->name = name;  
    }  
    string getName() {  
        return name;  
    }  
};
```

由于私有权限下类外不可被访问，因而我们需要对外提供设置私有属性的方法。此外，上面代码中我们用到了一个关键字this，this只能用在类的内部，通过this可以访问类的所有成员，且this是一个指针，要用->来访问成员变量或成员函数。

现在我们初步了解了一下类 (`class`) 的用法，我们会发现这个好像和我们现前了解到的结构体 (`struct`) 很像，其实二者几乎可以相互替换，但是二者之间还是存在区别的：类 (`class`) 的默认访问权限是私有权限，而结构体 (`struct`) 的默认访问权限是公共权限。

第四章 排序算法

排序算法作为数据结构的重要组成部分，是必须掌握的知识，表4.1所示是常见十大排序算法的时间复杂度和空间复杂度。

表 4.1: 排序算法的时间与空间复杂度

排序名称	平均时间复杂度	最坏情况	最好情况	空间复杂度
插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$
希尔排序	$O(nlogn)$	$O(nlog^2n)$	$O(nlog^2n)$	$O(1)$
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
堆排序	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	$O(1)$
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$
快速排序	$O(nlogn)$	$O(n^2)$	$O(nlogn)$	$O(logn)$
基数排序	$O(n \times k)$	$O(n \times k)$	$O(n \times k)$	$O(n + k)$
计数排序	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$
桶排序	$O(n + k)$	$O(n^2)$	$O(n + k)$	$O(n + k)$
归并排序	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	$O(n)$

在进行数据处理时，经常需要对数据进行查找，为了查的更快，通常要求数据有序排列，接下来将以排序算法的类型、以默认升序对排序算法进行讲解。

4.1 插入类排序

4.1.1 直接插入排序

直接插入排序是一种最基本的插入排序方法，基本操作是将第*i*个记录插入到前面*i* - 1个已经排好序的记录中。

例如，我们对序列48, 62, 35, 77, 55, 14, 35, 98进行排序，其大致流程如下：

48	62	35	77	55	14	<u>35</u>	98
48	62	35	77	55	14	<u>35</u>	98
35	48	62	77	55	14	<u>35</u>	98
35	48	62	77	55	14	<u>35</u>	98
35	48	55	62	77	14	<u>35</u>	98
14	35	48	55	62	77	<u>35</u>	98
14	35	35	48	55	62	77	98
14	35	<u>35</u>	48	55	62	77	98

首先，将数据分为俩队，一队是已经排好序的数据，另外一队为还没有进行排序的数据。一开始假设只有第一个数字是有序的，则只有第一个数字 48 在已经排好序的队列里。接下来取数据中的第二个数字 62，因为 62 大于 48，所以应该排在 48 后面，与原先相同故不需要进行交换。此时 48, 62 在已经排好序的队列中，且数据在队列中按升序排好。再取第三个数据 35，35 小于 48，需要排在 48 前面，故将 48, 62, 向后移动一位空出第一位让 35 插入……重复流程直至将所有的数据排好。

注意到，直接插入排序是与已排序好的序列进行比较并插入，因而直

接插入排序是稳定的。

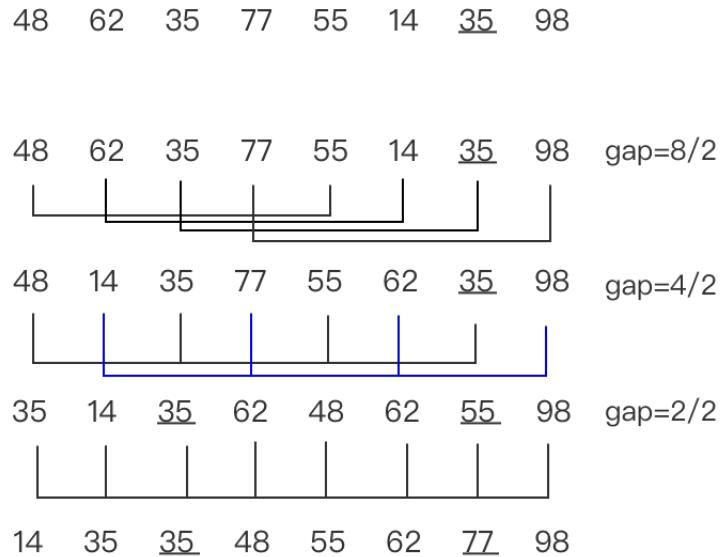
4.1.2 希尔排序

希尔排序是插入排序的一种改进。观察刚刚的直接插入排序，如果数组的最大值刚好是在第一位，要将它挪到正确的位置就需要 $n - 1$ 次移动。也就是说，原数组的一个元素如果距离它正确的位置很远的话，则需要与相邻元素交换很多次才能到达正确的位置。

希尔排序就是为了加快速度简单地改进了插入排序，交换不相邻的元素以对数组的局部进行排序。

希尔排序的思想是在插入排序的基础上，先让数组中任意间隔为 h 的元素有序，刚开始 h 的大小可以是 $h = \frac{n}{2}$ ，接着让 $h = \frac{n}{4}$ ，让 h 一直缩小，当 $h = 1$ 时，也就是此时数组中任意间隔为 1 的元素有序，此时的数组就是有序的了。

我们仍以序列 48, 62, 35, 77, 55, 14, 35, 98 为例，其大致流程如下：



从结果来看两个 35 的先后顺序没有调换，但实际上希尔排序是**不稳定**的，可以举一个简单的反例 [2, 4, 1, 2]。

4.2 交换类排序

4.2.1 冒泡排序

冒泡排序是一种较为简单的排序方法，它是通过对相邻的数据元素进行比较交换，逐步将待排序序列变成有序序列的过程。通过反复扫描待排序记录序列，在扫描过程中顺次比较相邻两个元素的大小，若逆序就交换。

仍以序48, 62, 35, 77, 55, 14, 35, 98为例, 第一轮排序的过程如下:

48	62	35	77	55	14	<u>35</u>	98
48	62	<u>35</u>	77	55	14	<u>35</u>	98
48	35	62	<u>77</u>	55	14	<u>35</u>	98
48	35	62	<u>77</u>	<u>55</u>	14	<u>35</u>	98
48	35	62	55	<u>77</u>	<u>14</u>	<u>35</u>	98
48	35	62	55	14	<u>77</u>	<u>35</u>	98
48	35	62	55	14	<u>35</u>	<u>77</u>	<u>98</u>
48	35	62	55	14	<u>35</u>	77	98

需要注意的是, 在第一趟冒泡排序完成后, 最后一个元素将是序列中最大的元素, 因而那么下一次冒泡排序时只需要排前 $n - 1$ 个元素。且冒泡排序是稳定的。

初始序列:	48	62	35	77	55	14	<u>35</u>	98
第一次排序:	48	35	62	55	14	<u>35</u>	77	98
第二次排序:	35	48	55	14	<u>35</u>	62	77	98
第三次排序:	35	48	14	<u>35</u>	55	62	77	98
第四次排序:	35	14	<u>35</u>	48	55	62	77	98
第五次排序:	14	35	<u>35</u>	48	55	62	77	98

4.2.2 快速排序

快速排序是冒泡排序的改进方法, 大致思想为: 从待排序的数据中选取一个元素为**基准值**, 然后将**小于**基准值的元素放到基准值的**前面**, 将**大于等于**基准值的元素放到基准值的**后面**。这样就将待排序的数据分为两个子表, 将这个过程称为一趟快速排序。对分割后的子表继续按照上面的方

法进行操作，直至所有子表的表长不超过 1 为止，此时待排序数据序列就变成了一个有序表。

现有初始序列：48_{begin} 62 35 77 55 14 35 98_{end}，本次选取最左侧数为基准值，具体流程如下：

```

48begin 62 35 77 55 14 35 98end
48begin 62 35 77 55 14 35end 98
48 62begin 35 77 55 14 35end 98
48 35begin 35 77 55 14 62end 98
48 35begin 35 77 55 14end 62 98
48 35 35begin 77 55 14end 62 98
48 35 35 77begin 55 14end 62 98
48 35 35 14begin 55 77end 62 98
48 35 35 14begin 55end 77 62 98
48 35 35 14beginend 55 77 62 98
14 35 35 48 55 77 62 98

```

即我们先让指针end从后向前找比基准值小的元素，如果找到了就停止，然后再让指针begin从前向后找比基准值大的元素，找到了就停止，如果指针begin和end没有相遇，则将begin和end指向的元素进行交换。在经过交换之后，继续重复上方法直至指针begin和end相遇，最后将基准值和指针begin和end同时指向的元素进行交换。

上述所示为一次快速排序的流程，在进行一次快速排序之后，按先前的基准值位置可以将序列分为两部分，从而在左右两边继续执行快速排序，相较于每次需要扫描序列的冒泡排序，可以降低扫描次数。同样的，快速排

序**不稳定**。

4.3 选择类排序

4.3.1 简单选择排序

简单选择排序会在每一趟选出较大或较小的值，将它与排好序后位于位置的值进行交换。每一趟在 $n - i + 1$ 个元素中选取关键字最小（最大）的元素作为有序序列中第 i 个记录。

现在我们以最小值为例，对序列48，62，35，77，55，14，35，98进行一次简单选择排序：

初始序列： 48 62 35 77 55 14_{min} 35 98

第 1 次排序： 14 62 35_{min} 77 55 48 35 98

第 2 次排序： 14 35 62 77 55 48 35_{min} 98

第 3 次排序： 14 35 35 77 55 48_{min} 62 98

第 4 次排序： 14 35 35 48 55_{min} 77 62 98

第 5 次排序： 14 35 35 48 55 77 62_{min} 98

第 6 次排序： 14 35 35 48 55 62 77_{min} 98

第 7 次排序： 14 35 35 48 55 62 77 98_{min}

最终结果： 14 35 35 48 55 62 77 98

每次都选择最小的值放到最左边组成有序序列，然后搜索剩余的序列重复执行。

需要注意的是，尽管从举的例子来看选择排序看似是稳定的，但其实简单选择排序是**不稳定的**，反例： [3, 3, 2]。

4.3.2 锦标赛排序

锦标赛排序，也就是树形选择排序，是简单选择排序的改进算法。在简单选择排序中，首先从n个元素中选择关键字最小的元素需要 $n - 1$ 次比较，接着在 $n - 1$ 个元素中选择关键字最小的元素需要 $n - 2$ 次比较，如此反复，每次比较都可以视作一次独立的排序。所以排序的时间复杂度为 $O(n^2)$ 。

为了改进简单选择排序而提出了树形选择排序，主要思想为：先把待排序的n个元素两两进行比较，取出较小者，然后在 $\frac{n}{2}$ 个较小者中采取同样的方法进行比较，选出较小者，如此反复，直至选出最小的元素为止。

这个过程可以使用一颗满二叉树来表示，不满时用无穷大的数补充，选出的最小元素就是这棵树的根结点。将根结点输出后，叶子结点为最小值的变为无穷大的数，然后从该叶子结点和其兄弟结点的关键字比较，修改该叶子结点到根结点路径上各结点的值，则根结点的值为次小的元素。

时间复杂度： $O(n \log n)$ ，空间复杂度： $O(n)$ ，稳定性：稳定。

4.3.3 堆排序

堆排序是树形选择排序的改进算法，弥补树形选择排序占用太多空间的缺陷。采用堆排序，只需要一个记录大小的辅助空间。

主要思想为：把待排序的元素放在堆中，每个结点表示一个元素。建立初堆后将根结点与堆的最后一个结点交换，之后重建堆的对象为 $n - 1$ 个，再把次大的结点与倒数第二个结点交换，之后重建堆的对象为 $n - 2$ 个，…，以此类推，直至重建堆的对象为 0。此时将堆输出即可得到排好序的序列。

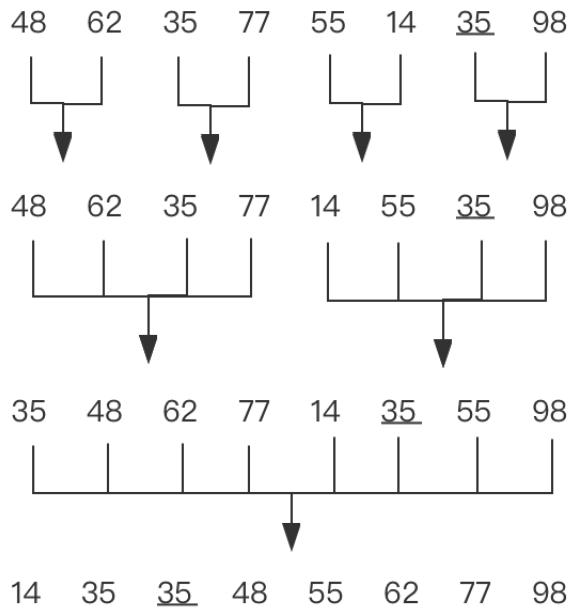
注：堆排序是不稳定的。

4.4 归并排序

归并排序的核心思想主要是将多个有序序列进行合并，我们以二路归并为例，假设初始序列有n个元素，首先将这n个元素看成n个子序列，两两

归并为有序的序列，此时序列的长度为 2，再两两归并为有序的序列，此时序列的长度为 4，…，一直归并直至得到一个长为n的序列。

仍以序列48, 62, 35, 77, 55, 14, 35, 98为例，流程如下：



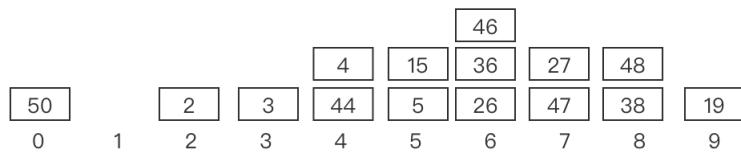
由于归并排序本质上还是比较相邻的数字进行交换，因而归并排序也是稳定的。

4.5 分配类排序

4.5.1 基数排序

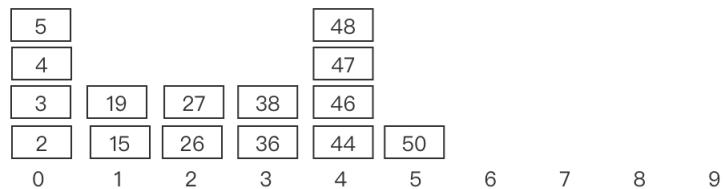
基数排序的核心是按照数的位数进行排序，即从小到大，先按个位数字大小进行排序，再按十位数字大小进行排序…直至最后形成有序序列。以3, 44, 38, 5, 47, 15, 36, 26, 27, 2, 46, 4, 19, 50, 48为例，首先比较个位数字：

50 2 3 4 44 5 15 46 26 36 27 47 38 48 19



紧接着比较十位数字大小：

2 3 4 5 15 19 26 27 36 38 44 46 47 48 50



最终即可得到有序序列，且基数排序是稳定的。

4.5.2 计数排序

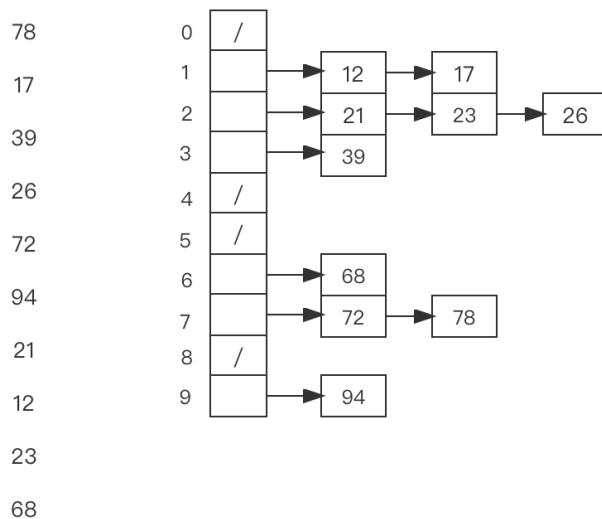
计数排序是一种适合于最大值和最小值的差值不是很大的排序。其基本思想就是把序列元素作为**数组的下标**，然后用一个临时数组统计该元素出现的次数，例如 $\text{temp}[i] = m$ ，表示元素*i*一共出现了*m*次。最后再把临时数组统计的数据从小到大汇总起来，此时汇总起来是数据是有序的，然后遍历计数数组，为0则不输出，为n则代表输出n次。

计数排序是稳定的。

4.5.3 桶排序

桶排序就是把最大值和最小值之间的数进行瓜分，例如分成 10 个区间，10 个区间对应 10 个桶，我们把各元素放到对应区间的桶中去，再对每个桶中的数进行排序，可以采用归并排序，也可以采用快速排序之类的。之后每个桶里面的数据就是有序的了，我们在进行合并汇总。

例如，我们考察序列 78, 17, 39, 26, 72, 94, 21, 12, 23, 68，桶排序的流程则是：



随后按顺序输出即可得到有序队列，同样，桶排序也是稳定的。

4.6 排序算法实现

4.6.1 直接插入排序

```
//数组 a 为待排序的数据，n 为数据的个数
void InsertSort(int* a, int n) {
    for (int i = 1; i < n - 1; i++) {
```

```
int temp=a[i];
for (int j = i-1; j >=0; j--) {
    if (a[j] > temp) {
        a[j + 1] = a[j];
        a[j] = temp;
    }
    else {
        break;
    }
}
//打印数组
for (int i = 0; i < n; i++) {
    printf("%d ", a[i]);
}
printf("\n");
}
```

4.6.2 希尔排序

```
// 希尔排序
void ShellSort(int* a, int n) {
    for (int gap = n / 2; gap >= 1;) {
        for (int i = gap; i < n ; i++) {
            int temp = a[i];
            for (int j = i-gap; j>=0; j-=gap) {
                if (a[j] > temp) {
                    a[j + gap] = a[j];
                    a[j] = temp;
                }
            }
            else {
                break;
            }
        }
    }
}
```

```
        }
    }
}

gap = gap / 2;
}

for (int i = 0; i < n; i++) {
    printf("%d ", a[i]);
}
printf("\n");
}
```

4.6.3 冒泡排序

```
void BubbleSort(int* a, int n){
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (a[j] > a[j + 1]) {
                int temp = a[j+1];
                a[j + 1] = a[j];
                a[j] = temp;
            }
        }
    }
    for (int i = 0; i < n; i++) {
        printf("%d ", a[i]);
    }
}
```

4.6.4 快速排序

```
int PartSort(int* a, int left, int right) {
```

```
int begin = left;
int end = right-1;
int key = a[left];//关键字的取值为a[dir]即a[begin], 最左侧的
元素
while (begin < end) {
    while (begin<end&&a[end] >= key) {
        end--;
    }
    while (begin<end&&a[begin] <=key) {
        begin++;
    }
    if (begin != end) {
        int temp = a[begin];
        a[begin] = a[end];
        a[end] = temp;
    }
}
int tem = a[begin];
a[begin] = a[left];
a[left]= tem;
return begin;//返回关键字的下标
}
```

4.6.5 简单选择排序

```
// 选择排序(升序)
void SelectSort(int* a, int n) {
    for (int i = 0; i < n-1; i++) {
        int min = i;
        for (int j = i+1; j < n; j++) {
            if (a[min] > a[j]) {
                min = j;
            }
        }
        if (min != i) {
            int temp = a[i];
            a[i] = a[min];
            a[min] = temp;
        }
    }
}
```

```
        }
    }

    int temp = a[min];
    a[min] = a[i];
    a[i] = temp;
}

for (int i = 0; i < n; i++) {
    printf("%d ", a[i]);
}
printf("\n");
}
```

4.6.6 堆排序

```
// 堆排序 (大堆: 升序)

void AdjustDwon(int* a, int n, int root) {
    int parent = root;
    int child = parent*2+1;
    while (child<n) {
        if (child + 1 < n && a[child + 1] > a[child]) {
            child += 1;
        }
        if (a[child] > a[parent]) {
            int temp = a[parent];
            a[parent] = a[child];
            a[child] = temp;

            parent=child;
            child = parent * 2 + 1;
        }
        else {
            break;
        }
    }
}
```

```
        }
    }
}

void HeapSort(int* a, int n) {
    if (a == NULL)
        return;
    for (int i = (n - 2) / 2; i >= 0; i--) {
        AdjustDwon(a, n, i);
    }
    for (int i = 0; i < n - 1; i++) {
        int temp = a[n - i - 1];
        a[n - i - 1] = a[0];
        a[0] = temp;
        AdjustDwon(a, n - i - 1, 0);
    }
    for (int i = 0; i < n; i++) {
        printf("%d ", a[i]);
    }
    printf("\n");
}
```

4.6.7 归并排序

```
// 归并排序非递归实现
void MergeSortNonR(int* a, int n) {
    int* temp = (int*)malloc(sizeof(a[0]) * n);
    if (NULL == temp) {
        assert(0);
        return;
    }
    int gap = 1;
```

```
while ( gap<n) {  
    for (int i = 0; i < n; i+=2*gap) {  
        int left = i;  
        int mid = left + gap;  
        int right = mid + gap;  
        if (mid >= n)  
            mid = n;  
        if (right >= n)  
            right = n;  
        //前面递归中也使用了此函数，已给出  
        MergeDate(a, left, mid, right, temp);  
    }  
    memcpy(a, temp, n * sizeof(a[0]));  
    gap *= 2;  
}  
free(temp);  
for (int i = 0; i < n; i++) {  
    printf("%d ", a[i]);  
}  
printf("\n");  
}
```

4.6.8 计数排序

```
void CountSort(int* a, int n) {  
    //统计数据出现的次数  
    int count[10] = {0}; //数据集中在9个元素，根据实际情况的不同  
    //需要更改  
    for (int i = 0; i < n; i++) {  
        count[a[i]]++;  
    }  
    //排序
```

```
int index = 0;  
for (int i = 0; i < 10; i++) {  
    while (count[i] > 0) {  
        a[index++] = i;  
        count[i]--;  
    }  
}  
for (int i = 0; i < n; i++) {  
    printf("%d ", a[i]);  
}  
printf("\n");  
}
```

第五章 算法基础

5.1 算法分析

5.1.1 数学基础

指数

$$X^A X^B = X^{A+B}$$

$$\frac{X^A}{X^B} = X^{A-B}$$

$$(X^A)^B = X^{AB}$$

$$X^N + X^N = 2X^N \neq X^{2N}$$

$$2^N + 2^N = 2^{N+1}$$

对数

定义 5.1.1. $X^A = B$ 当且仅当 $\log_X B = A$ 。

定理 5.1.2. $\log_A B = \frac{\log_C B}{\log_C A}$, 其中 $A, B, C > 0$, $A \neq 1$ 。

定理 5.1.3. $\log AB = \log A + \log B$, $A, B > 0$; $\log \frac{A}{B} = \log A - \log B$;
 $\log(A^B) = B \log A$; 且 $\log X < X$ 对所有 $X > 0$ 成立。

级数

$$\sum_{i=0}^n 2^i = 2^{n+1} - 1;$$

$$\sum_{i=0}^n A^i = \frac{A^{n+1} - 1}{A - 1}$$

在第二个公式中，如果 $A \in (0, 1)$ ，则有 $\sum_{i=0}^n A^i \leq \frac{1}{1-A}$ 。以及两个不常用的级数：

$$\sum_{i=1}^n n^2 = \frac{n(n+1)(2n+1)}{6} \approx \frac{n^3}{3}$$

$$\sum_{i=1}^n n^k \approx \frac{n^{k+1}}{|k+1|}, k \neq -1$$

需要注意的是 $k = -1$ 时，该级数为自然数倒数和，称作调和级数，调和级数是发散的。

5.1.2 排列组合

排列组合是组合数学中的基础，是排列与组合的并称。排列是指从给定个数的元素中取出指定个数的元素进行排序，组合是指从给定个数的元素中仅仅取出指定个数的元素，需要注意的是组合并不考虑内部的顺序。排列组合的中心问题是研究给定要求的排列和组合可能出现的情况总数，排列组合通常用于古典概率论。

在讨论排列组合之前需要先提两个定理：

定理 5.1.4 (加法原理). 做一件事情，完成它有 n 类方式，第一类方式有 m_1 种方法，第二类方式有 m_2 种方法， \dots ，第 n 类方式有 m_n 种方法，那么完成这件事情共有 $m_1 + m_2 + \dots + m_n$ 种方法。

定理 5.1.5 (乘法原理). 做一件事，完成它需要分成 n 个步骤，做第一步有 m_1 种不同的方法，做第二步有 m_2 种不同的方法， \dots ，做第 n 步有 m_n 种不同的方法，那么完成这件事共有 $N = m_1 \times m_2 \times m_3 \times \dots \times m_n$ 种不同的方法。

排列数

排列数主要是从 n 个不同元素中任取 m 个元素按照一定的顺序排成一列， $m \leq n$ ，其中 m 与 n 均为自然数，通常用符号 A_n^m 表示。计算公式为：

$$A_n^m = n(n-1)(n-2) \cdots [n-(m-1)] = \frac{n!}{(n-m)!}$$

其中 $n!$ 代表阶乘， $n! = n \times (n-1) \times (n-2) \times \dots \times 1$ 。

此外，排列数存在一个特殊情况，即全排列 A_n^n ，根据上述公式不难发现 $A_n^n = n!$ ，其含义为全选 n 个数字并拍成一排。

组合数

组合数即从 n 个不同元素中，任取 $m \leq n$ 个元素组成一个集合，在初高中阶段用符号 C_n^m 来表示。计算公式为：

$$C_n^m = \frac{A_n^m}{m!} = \frac{n!}{m!(n-m)!}$$

根据当前公式，不难发现 $C_n^m \times m! = A_n^m$ ，即为从 n 个元素中选出 m 个不同的元素并将这 m 个元素全排列。需要强调的是，在数学界普遍采用 $\binom{n}{m}$ 的记号代表组合数， $C_n^m = \binom{n}{m}$ 。此外，规定当 $m > n$ 时， $A_n^m = \binom{n}{m} = 0$ 。

5.1.3 运行时间的计算

我们以一个简单的求立方和函数为例：

```
int sum(int n) {  
    int partialSum;  
    partialSum = 0; // 1  
  
    for (int i = 1; i <= n; i++) { // 2  
        partialSum += i * i * i; // 3  
    }  
    return partialSum; // 4  
}
```

对这个程序段的分析是简单的。所有的声明均不计时间，1 处和 4 处各占一个时间单元。3 处每执行一次占用 4 个时间单元（两次乘法，一次加法和一次赋值），而执行 N 次共占用 $4N$ 个时间单元。第 2 行在初始化 i 、测试 $i \leq N$ 和对 i 的自增运算隐含着开销。所有这些的总开销是初始化 1 个单元时间，所有的测试为 $N+1$ 个单元时间，而所有的自增运算为 N 个单元时间，共 $2N+2$ 个时间单元。我们忽略调用方法和返回值的开销，得到总量是 $6N+4$ 个时间单元。因此我们可以得到 $T(N) = 6N + 4$ ，用大 O 表示法即为 $O(N) = N$ 。

如果每次分析一个程序都要演示所有这些工作，那么这项任务很快就会变成不可行的负担。幸运的是，由于我们有了大 O 的结果，因此就存在许多可以采取的捷径并且不影响最后的结果。例如，3 处（每次执行时）显然是 $O(1)$ 语句，因此精确计算它究竟是 2、3 还是 4 个时间单元是无关紧要的。1 处与 for 循环相比显然是不重要的，所以在这里花费时间也是不明智的。这使我们得到若干一般法则。

法则 1——for 循环：一个 for 循环的运行时间最多是该 for 循环内部语句的运行时间乘迭代次数。

法则 2——嵌套的 for 循环：嵌套的 for 循环需要从里向外分析，一

组嵌套循环内部的一条语句的运行时间为该语句的运行时间乘该组所有 for 循环的大小的乘积。

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < m; j++)
        k++;
```

根据前文的例子来看，内部的 for 循环执行 $2M + 2M + 1$ 个时间单元，外部的 for 循环执行 $2N + 1$ 个时间单元，因而该例子的时间复杂度结果为 $T = (4M + 1) \times (2N + 1)$ ，同样我们取大 O 表示为 $O(M \times N)$ 。

法则 3——顺序语句：顺序语句则计算每个片段的运行时间分别求和即可，其中求得的最大值就是所得到的运行时间。

法则 4——if-else 语句：对于程序片段：

```
if (condition)
    S1
else
    S2
```

一个 if-else 语句的运行时间不会超过判断的运行时间加上 S1 和 S2 中运行时间较长者的总运行时间。

5.1.4 运行时间中的对数

依旧来看简单的片段：

```
int i = 1;
while(i < n) {
    i = i * 2;
}
```

我们容易发现第 k 次执行时 i 的值为 2^k ，要使 $2^i < n$ ，两边同取 2 为底的对数得到 $i < \log_2 n$ ，可以得到最多执行 $\log_2 n$ 次就会停止。所以我们得到

时间复杂度为 $O(\log_2 n)$, 根据换底公式有 $\log_2 n = \frac{\log n}{\log 2}$, 考虑到 $\log 2$ 是常数, 所以最终的时间复杂度为 $O(\log n)$ 。

```
for(m=1; m<n; m++) {
    i = 1;
    while(i < n) {
        i = i * 2;
    }
}
```

同样的, 上面的片段我们不难计算出其时间复杂度为 $O(n \log n)$

5.1.5 主定理

假设有递推关系式 $T(n) = aT(\frac{n}{b}) + f(n)$, 其中 n 为问题规模, a 为递推的子问题数量, $\frac{n}{b}$ 为每个子问题的规模 (假设每个子问题的规模基本一样), $f(n)$ 为递推以外进行的计算工作。

如果我们有 $a \geq 1, b > 1$, $f(n)$ 为 n 的函数, $T(n)$ 为非负整数, 则有以下结果:

1. 若 $f(n) = O(n^{\log_b a - \epsilon}), \epsilon > 0$, 那么 $T(n) = \Theta(n^{\log_b a})$;
2. 若 $f(n) = \Theta(n^{\log_b a} \cdot \log^k n), k \geq 0$, 那么 $T(n) = \Theta(n^{\log_b a} \cdot \log^{k+1} n)$;
3. 若 $f(n) = \Omega(n^{\log_b a + \epsilon}), \epsilon > 0$, 且 $\exists C \in (0, 1)$ 并对于所有充分大的 n 有 $af(\frac{n}{b}) \leq cf(n)$, 那么 $T(n) = \Theta(f(n))$ 。

例 5.1.6 (Master01). 求解 $T(n) = 9T(\frac{n}{3}) + n$ 的时间复杂度。

在例题5.1.6中, $a = 9, b = 3$, 则我们有 $n^{\log_b a} = n^{\log_3 9} = n^2$, 此时 $f(n) = n$, 于是我们可以发现:

$$n^{\log_3 9 - 6} = n = f(n)$$

因此，本题时间复杂度为 $T(n) = O(n^2)$ 。

例 5.1.7 (Master02). 求解 $T(n) = T(\frac{2n}{3}) + 1$ 的时间复杂度。

在例题5.1.7中， $a = 1, b = \frac{3}{2}$ ，则我们有 $n^{\log_b a} = n^0$ ，此时 $f(n) = 1$ ，则我们有：

$$f(n) = \Theta(1 \times \log^{0+1} n)$$

因此，本题的时间复杂度为 $T(n) = O(\log n)$

例 5.1.8 (Master03). 求解 $T(n) = 3T(\frac{n}{4}) + n \log n$ 的时间复杂度。

在例题5.1.8中， $a = 3, b = 4$ ，则我们有 $n^{\log_b a} = n^{\log_4 3}$ ，此时 $f(n) = n \log n$ ，显然 $f(n)$ 对时间复杂度的影响更大，同时，我们有：

$$af\left(\frac{n}{b}\right) = 3 \frac{n}{4} \log \frac{n}{4} \leq \frac{3}{4} n \log n = cf(n)$$

即 $c = \frac{3}{4} < 1$ ，因此，本题的时间复杂度为 $T(n) = n \log n$ 。

5.2 算法思想

经过上一章节对排序算法的了解，本小节则是算法思想的简单介绍，在对排序算法有一定了解的基础上我们将根据经典的排序算法来阐述算法设计的一些主要思想。

5.2.1 蛮力法

蛮力法，即暴力求解，是一种简单直接地解决问题的方法，直接基于问题的描述和涉及的概念定义。

蛮力法可以说是不容忽视的算法设计策略，是唯一一个能够应用于所有问题的策略。由于蛮力法肯定会给出一种对应的解，不受数据规模的限制，可能比设计一个好的算法需要更少的时间。

常见的枚举法、穷举法、暴力解法都是蛮力思想，例如**直接选择排序**¹和**冒泡排序**²。

5.2.2 分治法

分治法的核心思想是将问题实例划分为同一个问题的几个较小的实例（最好拥有相同的规模），对这些较小的实例求解，最后合并这些较小问题实例的解得到原问题的解。

能够使用分治策略的问题一般具有如下特质：问题规模（时间上、空间上）缩小到一定程度就很容易解决，问题能够划分为多个相互独立的子问题。

快速排序是分治法的一个实例，通过基准值将序列分割，在分割后的序列再次进行排序。前文提到过时间复杂度，在这里，我们也来探讨一下快速排序的时间复杂度。

首先我们考虑**最佳情况**，也就是选择的中轴恰好是每一个数组的中值点，那么我们可以得到其时间复杂度 $C_{best}(n) = 2C_{best}(\frac{n}{2}) + C_{partition}$ 。由于第一次分割需要遍历整个序列，也就是做 n 次比较，因而上式为 $C_{best}(n) = 2C_{best}(\frac{n}{2}) + n$ 。将这个式子递推下去：

$$\begin{aligned}
 C_{best}(n) &= 2C_{best}(\frac{n}{2}) + n = 2^1 C_{best}(\frac{n}{2}) + n \\
 &= 2(2C_{best}(\frac{n}{4}) + \frac{n}{2}) + n = 2^2 C_{best}(\frac{n}{4}) + 2n \\
 &= 4(2C_{best}(\frac{n}{8}) + \frac{n}{4}) + 2n = 2^3 C_{best}(\frac{n}{8}) + 3n \\
 &\dots \\
 &= nC_{best}(1) + \log_2 n \times n = 2^{\log_2 n} C_{best}(1) + n \log_2 n
 \end{aligned}$$

¹遍历列表，找到最大或最小的与第一个没有排序好的交换

²遍历列表，每次都把与当前元素与下一个逆序的元素交换，把最大或最小的元素排到最后

当分割到最后大小为 1 时, $C_{best}(1) = 0$, 因而最终的时间复杂度 $O(n) = n \log n$ 。

这样, 我们以此为例可以得到**顺序算法**通用时间复杂度公式: $T_{sequence}(n) = aT_s(\frac{n}{b}) + f(n)$, 其中 b 代表分解份数, 其中 a 份需要求解, $f(n)$ 表示需要合并的时间。如果是**并行执行**, 则 $T_{parallel}(n) = T_p(\frac{n}{b}) + f(n)$ 。

如果是**最坏情况**, 分区并不能带来效率上的提升, 即不满足递推公式, 最不满足的情况是原来就是升序的, 每次只会从第一个位置将数组分割, 那么执行 k 次后剩下的元素为 $n - k$, 需要比较的次数是 $n - k$, 则时间复杂度为 $C_{worst}(n) = \sum_{i=0}^{n-1} (n - i) = \frac{(n-1)n}{2}$, $O(n) = n^2$ 。

最后我们来考察快速排序的**平均**时间复杂度, 所谓平均, 也就是任意一种划分情况出现的概率都相等。首先我们设 $D(n) = n - 1$ 为一趟快排需要比较的次数, 可以得到:

$$\begin{aligned} C_{ave}(n) &= D(n) + \frac{1}{n} \sum_{i=0}^{n-1} (C_{ave}(i) + C_{ave}(n - i)) \\ &= D(n) + \frac{2}{n} \sum_{i=0}^{n-1} C_{ave}(i) \end{aligned}$$

$$C_{ave}(n - 1) = D(n - 1) + \frac{2}{n - 1} \sum_{i=0}^{n-2} C_{ave}(i)$$

在这里我们使得 $nC_{ave}(n) - (n - 1)C_{ave}(n - 1)$ 可以得到:

$$\begin{aligned} &= nD(n) + 2 \sum_{i=0}^{n-1} C_{ave}(i) - (n - 1)D(n - 1) - 2 \sum_{i=0}^{n-2} C_{ave}(i) \\ &= nD(n) - (n - 1)D(n - 1) + 2C_{ave}(n - 1) \\ &= 2(n - 1) + 2C_{ave}(n - 1) \end{aligned}$$

移项合并后得到 $nC_{ave}(n) = (n + 1)C_{ave}(n - 1) + 2(n - 1)$, 两边同除 $n(n + 1)$

得到 $\frac{C_{ave}(n)}{n+1} = \frac{C_{ave}(n-1)}{n} + \frac{2(n-1)}{n(n+1)}$, 此时我们令 $Temp(n) = \frac{C_{ave}(n)}{n+1}$, 则上式变成 $Temp(n) = Temp(n-1) + \frac{2(n-1)}{n(n+1)}$, 接下来递归求解:

$$\begin{aligned} Temp(n) &= Temp(n-1) + \frac{2(n-1)}{n(n+1)} \\ &= Temp(n-2) + \frac{2(n-2)}{(n-1)n} + \frac{2(n-1)}{n(n+1)} \\ &\dots \\ &= Temp(1) + \sum_{i=1}^n \frac{2(i-1)}{i(i+1)} \\ &= \sum_{i=1}^n \frac{2(i+1)-4}{i(i+1)} \\ &= \sum_{i=1}^n \left(\frac{2}{i} - \frac{4}{i(i+1)} \right) \end{aligned}$$

其中, $\sum_{i=1}^n \frac{1}{i}$ 是调和级数, 是发散的, 其值约为 $\log n + \gamma$, $\gamma \approx 0.57721566$ 为误差值; 另一方面, $\sum_{i=1}^n \frac{1}{i(i+1)} = \sum_{i=1}^n \left(\frac{1}{i} - \frac{1}{i+1} \right) = 1 - \frac{1}{n+1}$, 所以我们可以得到 $Temp(n) = 2 \log n + 2\gamma - \frac{4n}{n+1}$, 那么最终得到结果为:

$$C_{ave} = 2(n+1) \log n + 2(n+1)\gamma - 4n \in O(n \log n)$$

所以, 快速排序的平均时间复杂度为 $O(n \log n)$ 。

5.2.3 减治法

减治法的核心思想是利用一个问题给定实例的解和同样问题较小实例的解之间的某种关系, 将一个大规模的问题逐步化简为一个小规模的问题。

减治法与分治法不同在于: **分治法**是把一个大问题划分为若干个子问题, 分别求解各个子问题, 然后再把子问题的解进行合并得到原问题的解。**减治法**同样是把一个大问题划分为若干个子问题, 但是这些子问题不需要

分别求解，只需求解其中的一个子问题，因而也无需对子问题的解进行合并。所以我们可以将减治法视为一种退化了的分治法，时间复杂度一般是 $O(\log n)$ 。

5.2.4 变治法

变治法，也就是通过转换问题使得原问题更容易求解，通常使用的手法有：

- (1) **实例化简**：还是原来的问题，只是进行了一些中间操作，使得问题求解变得容易。
- (2) **改变表现**：主要是改变使用的数据结构。
- (3) **问题化简**：将给定的问题变换为一个新的问题，对新的问题求解。

5.2.5 动态规划

所谓动态规划，简单来说就是利用**历史记录**来避免重复计算，这些历史记录往往需要一些变量来存储，通常是一维数组和二维数组。面对动态规划主要是三个步骤：

- (1) **定义初始数组**：如上所述，我们会用一个变量来存储历史数据，例如我们定义一个一维数组 `int dp[length]`，那么就需要明确这个数组元素的具体用意，例如 `dp[i]` 代表什么。
- (2) **找出元素间的关系**：当我们计算 `dp[i]` 时，可以利用 `dp[i-1] … dp[1]`，也就是所谓的历史记录。

例如我们计算斐波拉契数列 `dp[i] = dp[i - 1] + dp[i - 2]` 就是一个历史记录的应用。

- (3) **定义初始值**：当我们找到递推关系之后，就需要通过初始值来进行运算，仍以计算斐波拉契数列第 n 项为例，`dp[i] = dp[i - 1] + dp[i - 2]`，继续向下递推，最终会得到 `dp[3] = dp[2] + dp[1]`，`dp[2]` 和 `dp[1]` 是不可再分的，因而我们需要知道 `dp[2]` 和 `dp[1]` 的值。

总的来说，在完成对 $dp[length]$ 数组的定义后，通过初始值和递推关系来解决问题，就是一次动态规划的应用。接下来，我们看几道例题。

例 5.2.1 (跳台阶). 一只青蛙一次可以跳上 1 级台阶，也可以跳上 2 级。求该青蛙跳上一个 n 级的台阶总共有多少种跳法。

例题5.2.1是一个很简单的动态规划问题，我们用上述的三步法来解决：

(1) 定义初始数组：根据问题“青蛙跳上 n 级台阶有多少种跳法”，我们可以很直接的定义 $dp[i]$ 为跳上 i 级台阶有 $dp[i]$ 种跳法。

(2) 找出元素间的关系：找出元素间的递推关系是动态规划最为困难的一步，我们使用动态规划的目的是将大问题分解为多个规模小的问题，是一种分治的思想。例如 $dp[n]$ 的规模是 n ，那么 $n - 1, n - 2 \dots$ 的规模就会更小，因此，我们主要目的是要找到 $dp[n]$ 和 $dp[n - 1], dp[n - 2] \dots$ 之间存在的联系。

回到本题，青蛙要跳到第 n 层就有两种方法：从第 $n - 1$ 层跳 1 层上去；从第 $n - 2$ 层跳 2 层上去。题目的要求是要我们求出所有的可能，那么就需要一起考虑，也就是有 $dp[n] = dp[n - 1] + dp[n - 2]$ 。

(3) 定义初始值：当递推到了 $dp[1]$ 时， $dp[1] = dp[0] + dp[-1]$ ，但数组的下标从 0 开始，也就是说我们必须给出 $dp[1]$ 的值。同样， $dp[0] = 0$ ，即当台阶数为 0 时没有跳法。

但是这么定义初始值存在问题，根据递推 $dp[2] = dp[1] + dp[0]$ ，也就是说 $dp[2] = 1$ ，显然这是错误的，所以我们需要定义 $dp[2]$ 的初始值。

因此，本题的初始值 $dp[2] = 2; dp[1] = 1;$

有了上述的三步分析，本题也就迎刃而解：

```
int jump(int n) {
    if(n <= 1) {
        return n;
    }
    int dp[n + 1];
```

```

dp[1] = 1;
dp[2] = 2;

for (int i = 3; i < n + 1; ++i) {
    dp[i] = dp[i - 1] + dp[i - 2];
}
return dp[n];
}

```

例 5.2.2 (不同路径). 一个机器人位于一个 $m \times n$ 网格的左上角，机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角，共有多少条路径。

例题5.2.2则是一个经典的用二维数组来存储历史记录的问题，我们同样用三步法来解决问题：

(1) 定义初始数组：由于我们的目的是从左上角到右下角一共有多少种路径，那我们就定义 $dp[i][j]$ 的含义为：当机器人从左上角走到 (i, j) 位置时，一共有 $dp[i][j]$ 种路径。

那么， $dp[m-1][n-1]$ 就是定义的初始数组。

(2) 找出元素间的关系：现在我们的目的是找出递推关系，假设我们现在的目的是走到 (i, j) 这个位置，由于只能向右或者向下走，所以只有两个方法：(1) 从 $(i-1, j)$ 这个位置向右走，(2) 从 $(i, j-1)$ 这个位置向下走。因为是计算所有可能步骤，所以得到 $dp[i][j] = dp[i - 1][j] + dp[i][j - 1]$ 。

(3) 定义初始值：显然，当 $dp[i][j]$ 中，如果 i 或者 j 有一个为 0 时，代表在边界。所以，此时我们的初始值就是计算出所有的 $dp[0][0 \dots n-1]$ 和所有的 $dp[0 \dots m-1][0]$ 。因此初始值如下：

```

dp[0][0...n-1] = 1; // 相当于最上面一行，机器人只能一直往右走
dp[0...m-1][0] = 1; // 相当于最左面一列，机器人只能一直往下走

```

步骤都完成了，我们将之体现为代码：

```
int android(int m, int n) {
    if(m <= 0 || n <= 0)
        return 0;
    // 定义初始数组
    int dp[m][n];
    // 初始化
    for (int i = 0; i < m; ++i)
        dp[i][0] = 1;
    for (int i = 0; i < n; ++i)
        dp[0][i] = 1;
    // 递推关系
    for (int i = 1; i < m; ++i) {
        for (int j = 1; j < n; ++j) {
            dp[i][j] = dp[i-1][j] + dp[i][j-1];
        }
    }
    return dp[m - 1][n - 1];
}
```

例 5.2.3 (最小路径和). 给定一个包含非负整数的 $m \times n$ 网格，找出一条从左上角到右下角的路径，使得路径上的数字总和为最小，每次只能向下或者向右移动一步。

例题5.2.3与例题5.2.2类似，我们直接来看题目：

(1) 定义初始数组：由于我们的目的是从左上角到右下角找一条最小路径和，那我们就定义 $dp[i][j]$ 的含义为：当机器人从左上角走到 (i, j) 这个位置时，最下的路径和是 $dp[i][j]$ 。那么， $dp[m-1][n-1]$ 即是我们需要的 dp 数组。

(2) 找出元素间的关系：与上题相似，由于只能向下走或者向右走，所以有两种方式到达：从 $(i - 1, j)$ 这个位置走一步到达、从 $(i, j - 1)$ 这个位置走一步到达。

由于本次计算哪一个路径和是最小的，那么我们要从这两种方式中，选择一种，使得 $dp[i][j]$ 的值是最小的，显然有：

```
dp[i][j] = min(dp[i - 1][j], dp[i][j - 1]) + arr[i][j];
```

在这里，我们用 $arr[i][j]$ 表示网格中的值。

(3) 定义初始值：与例题5.2.2相似，当 $dp[i][j]$ 中，如果*i*或者*j*有一个为0时，关系式便不能再递推。所以初始值是计算出所有的 $dp[0][0...n-1]$ 和所有的 $dp[0...m-1][0]$ ，因此初始值如下：

```
dp[0][j] = arr[0][j] + dp[0][j-1];// 最上面一行，只能一直往左走
```

```
dp[i][0] = arr[i][0] + dp[i][0];// 最左面一列，只能一直往下走
```

例 5.2.4 (编辑距离). 给定两个单词 *word1* 和 *word2*，计算出将 *word1* 转换成 *word2* 所使用的最少操作数。

你可以对一个单词进行如下三种操作：插入一个字符，删除一个字符，替换一个字符。

示例：输入: *word1* = "horse", *word2* = "ros"

输出: 3

解释: (1)horse -> rorse (将'h' 替换为'r'); (2)rorse -> rose (删除'r'); (3)rose -> ros (删除'e')。

(1) 定义初始数组：由于我们的目的求将 *word1* 转换成 *word2* 所使用的最少操作数。那我们就定义 $dp[i][j]$ 的含义为：当字符串 *word1* 的长度为 *i*，字符串 *word2* 的长度为 *j* 时，将 *word1* 转化为 *word2* 所使用的最少操作次数为 $dp[i][j]$ 。

(2) 找出元素间的关系：对于这道题，我们可以对 *word1* 进行三种操作：插入一个字符，删除一个字符，替换一个字符。

由于我们是要让操作的次数最小，所以我们要寻找最佳操作。那么有如下关系式：

(1) 如果我们 *word1*[*i*] 与 *word2*[*j*] 相等，这个时候不需要进行任何操作，显然有 $dp[i][j] = dp[i-1][j-1]$ 。

(2) 如果我们 $\text{word1}[i]$ 与 $\text{word2}[j]$ 不相等, 这个时候我们就必须进行调整, 而调整的操作有 3 种, 我们要选择一种。三种操作对应的关系式如下:

1、如果把字符 $\text{word1}[i]$ 替换成与 $\text{word2}[j]$ 相等, 则有:

$$\text{dp}[i][j] = \text{dp}[i-1][j-1] + 1;$$

2、如果在字符串 word1 插入一个与 $\text{word2}[j]$ 相等的字符, 则有:

$$\text{dp}[i][j] = \text{dp}[i][j-1] + 1;$$

3、如果把字符 $\text{word1}[i]$ 删除, 则有:

$$\text{dp}[i][j] = \text{dp}[i-1][j] + 1;$$

那么我们应该选择一种操作, 使得 $\text{dp}[i][j]$ 的值最小, 显然有:

$$\text{dp}[i][j] = \min(\text{dp}[i-1][j-1], \text{dp}[i][j-1], \text{dp}[[i-1][j]]) + 1;$$

(3) 定义初始值: 显然, 当 $\text{dp}[i][j]$ 中, 如果 i 或者 j 有一个为 0 则代表到了数组边界。所以我们的初始值是计算出所有的 $\text{dp}[0][0\dots n]$ 和所有的 $\text{dp}[0\dots m][0]$ 。这个还是非常容易计算的, 因为当有一个字符串的长度为 0 时, 转化为另外一个字符串, 那就只能一直进行插入或者删除操作了。

$$\text{dp}[0][j] = \text{dp}[0][j - 1] + 1; // \text{dp}[0][0\dots n2] 的初始值$$

$$\text{dp}[i][0] = \text{dp}[i - 1][0] + 1; // \text{dp}[0\dots n1][0] 的初始值$$

我们看了上面四道例题, 这四道题是动态规划相对简单的题目, 我们根据例题可以再来总结一下什么是动态规划:

定义 5.2.5 (动态规划). 由于任何数学递推公式都可以直接转换成递推算法, 但是基本现实是编译器常常不能正确对待递归算法, 结果导致程序低效。当怀疑可能是这种情况时, 我们必须再给编译器提供一些帮助, 将递归算法重新写成非递归算法, 让后者把那些子问题的答案系统地记录在一个表内。利用这种方法的一种技巧叫作动态规划 (*dynamic programming*)。

我们继续通过一些题目回顾动态规划:

例 5.2.6 (最长回文子串). 给定一个字符串 s , 找到 s 中的最长回文子串。

示例: 输入: $s = \text{"babad"}$

输出: “bab”

例题5.2.6要求找到字符串中的最长回文子串，我们先考虑回文子串本身，单个字符本身是回文的，且如果一个字符串是回文串，即 $s[1\dots n]$ 是回文串，那么 $s[2\dots n-1]$ 也将是回文串。

于是我们定义 $dp[i][j]$ 代表 $s[i\dots j]$ 是回文串。现在我们找递推关系，如之前分析， $s[1\dots n]$ 是回文串，那么 $s[2\dots n-1]$ 也必是回文串，反过来思考就是 $s[i\dots j]$ 是回文子串的要求就是 $s[(i+1)\dots(j-1)]$ 是回文串且 $s[i] == s[j]$ 。于是我们得到递推关系：

$$dp[i][j] = dp[i+1][j-1] \&& s[i] == s[j];$$

我们来考虑初始条件，若仅有单个字符，那么其本身必然是回文串，即 $dp[i][i] = \text{true}$ ，如果有两个字母 $dp[i][j] = s[i] == s[j]$ 。

例 5.2.7 (打家劫舍 I). 你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。

给定一个代表每个房屋存放金额的非负整数数组，计算你不触动警报装置的情况下，一夜之内能够偷窃到的最高金额。

例 5.2.8 (打家劫舍 II). 你是一个专业的小偷，计划偷窃沿街的房屋，每间房内都藏有一定的现金。这个地方所有的房屋都围成一圈，这意味着第一个房屋和最后一个房屋是紧挨着的。同时，相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。

给定一个代表每个房屋存放金额的非负整数数组，计算你在不触动警报装置的情况下，今晚能够偷窃到的最高金额。

例 5.2.9 (打家劫舍 III). 小偷又发现了一个新的可行窃的地区。这个地区只有一个入口，我们称之为 root 。

除了 root 之外，每栋房子有且只有一个“父”房子与之相连。一番侦察之

后，聪明的小偷意识到“这个地方的所有房屋的排列类似于一棵二叉树”。如果两个直接相连的房子在同一天晚上被打劫，房屋将自动报警。

给定二叉树的root。返回在不触动警报的情况下，小偷能够盗取的最高金额。

最长公共子序列

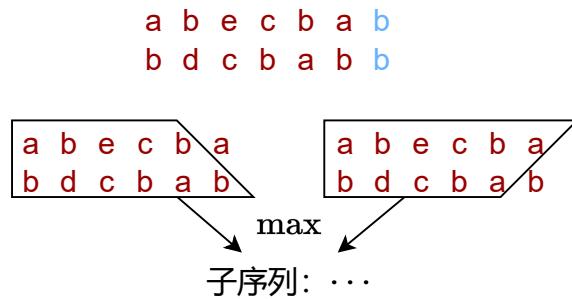
最长公共子序列 (Longest Common Subsequence, LCS) 是动态规划中的经典问题，目的是求两个序列最长的公共子序列 (可以不连续)。

在本文中，我们规定用 $s_{[-1]}$ 表示序列 s 的最后一个元素，用 $s_{[:-1]}$ 表示 s 去掉最后一个元素后的子序列， $LCS(s_1, s_2)$ 表示 s_1 和 s_2 的 LCS 的长度。现在，假如我们有 $abdcbab$ 和 $bdcabb$ 两个字符串，记为 s_1 和 s_2 ，我们要如何求它们的 LCS 呢？

既然是动态规划问题，我们就考虑状态转移。这里将会有两种情形，第一种：如果两个序列最后一个元素相同，如 $abecbab$ 和 $bdcabb$ ，最后一个元素相同，所以它们的 LCS 长度就是 $abecba$ 与 $bdcbab$ 的 LCS 长度加上 1。这是因为， s_1 和 s_2 的任何公共子序列 S 去掉最后一个元素后，都是 $s_{1[:-1]}$ 和 $s_{2[:-1]}$ 的公共子序列，所以 S 的长度不会大于 $LCS(s_{1[:-1]}, s_{2[:-1]}) + 1$ 。



如果最后一个元素不同，考虑到这两个序列的最后一个元素不会同时出现在 LCS 中，所以 LCS 长度为 $\max(LCS(s_{1[:-1]}, s_2), LCS(s_1, s_{2[:-1]}))$ 。



最后，我们还需要处理一下边界条件，即当 s_1 和 s_2 中至少有一个为空时，这时 LCS 的长度为 0。于是综合一下，我们可以得到以下式子：

$$LCS(s_1, s_2) = \begin{cases} 0, & \text{if } s_1 \text{ or } s_2 \text{ is empty} \\ LCS(s_{1[-1]}, s_{2[-1]}) + 1, & \text{if } s_{1[-1]} = s_{2[-1]} \\ \max(LCS(s_{1[-1]}, s_2), LCS(s_1, s_{2[-1]})), & \text{otherwise} \end{cases}$$

然而，如果直接翻译成递归程序，必然引起时间、空间复杂度的爆炸。于是我们可以用**记忆化**的方法，并且传参时传索引而不是序列本身。

5.2.6 时空权衡

通常在算法设计中，为了优化某些算法，我们通常采取两种方式：**存储空间换取运行时间，运行时间换取存储空间**。但用运行时间来换取空间的情况较少，往往只在资源真的有限的情况下才会使用。

需要注意的是，不是一定要用什么换什么，可能更少的存储空间带来更快的处理效率。

时空权衡思想：

输入增强：对问题的部分或全部输入做预处理，然后对获得的额外信息进行存储，以加速后面问题的求解。例如：计数排序——分布排序。

预构造：使用额外空间实现更快和（或）更方便的数据存取。例如散列

法、B树。

5.2.7 贪心算法

贪心算法，又名贪婪法，是寻找最优解问题的常用方法，这种方法模式一般将求解过程分成若干个步骤，但每个步骤都应用贪心原则，选取当前状态下最好/最优的选择（局部最有利的选择），并以此希望最后堆叠出的结果也是最好/最优的解。贪心算法对每个子问题的解决方案都做出选择，**不能回退**。

贪心算法的基本思路是从问题的某一个初始解出发一步一步地进行，根据某个优化测度，**每一步都要确保能获得局部最优解**。每一步只考虑一个数据，他的选取应该满足局部优化的条件。**若下一个数据和部分最优解连在一起不再是可行解时，就不把该数据添加到部分解中**，直到把所有数据枚举完，或者不能再添加算法停止。

但实际上，贪心算法的**适用情况并不多**。一般对一个问题分析是否适用于贪心算法，可以先选择该问题下的几个实际数据进行分析，就可以做出判断。总的来说，该算法存在以下几个问题：

1. 不能保证求得的最后解是最佳的；
2. 不能用来求最大值或最小值的问题；
3. 只能求满足某些约束条件的可行解的范围。

前文所提到的**选择排序**其实就是采用的贪心策略。它所采用的贪心策略即为每次从未排序的数据中**选取最小值**，并把最小值放在未排序数据的**起始位置**，直到未排序的数据为 0，则结束排序。

除此之外，我们再来看一道例题：

例 5.2.10 (平衡字符串). 在一个平衡字符串中，'L' 和 'R' 字符的数量是相同的。给你一个平衡字符串 s，请你将它分割成尽可能多的平衡字符串。

注意：分割得到的每个字符串都必须是平衡字符串，且分割得到的平衡字符串是原平衡字符串的连续子串。

返回可以通过分割得到的平衡字符串的最大数量。

输入示例：

```
s = "RLRRLLRLRL"
s = "LLLLRRRR"
s = "RLRRRLLRLL"
```

输出示例：

```
4 // s 可以分割为 "RL"、 "RRL"、 "RL"、 "RL"， 每个子字符串
   中都包含相同数量的 'L' 和 'R'
1 // s 只能保持原样 "LLLLRRRR"
2 // s 可以分割为 "RL"、 "RRRLLRLL"， 每个子字符串中都包含
   相同数量的 'L' 和 'R'
```

根据例题5.2.10的题意，对于一个平衡字符串s，若s能从中间某处分割成左右两个子串，若其中一个是平衡字符串，则另一个的L和R字符的数量必然是相同的，所以也一定是平衡字符串。

为了最大化分割数量，我们可以不断循环，每次从s中分割出一个最短的平衡前缀，由于剩余部分也是平衡字符串，我们可以将其当作s继续分割，直至s为空时，结束循环。

代码实现中，可以在遍历s时用一个变量balance维护L和R字符的数量之差，当balance == 0时就说明找到了一个平衡字符串，将答案加一。

```
int balancedStringSplit(string s) {
    int ans = 0;
    int balance = 0;
    for(int i = 0; i < s.length(); i++){
        if(s[i] == 'L')
            balance--;
        if(s[i] == 'R')
            balance++;
        if(balance == 0)
            ans++;
```

```

        ans++;
    }
    return ans;
}

```

例 5.2.11 (最大数对和的最小值). 一个数对 (a, b) 的数对和等于 $a + b$ 。最 大数对和是一个数对数组中最大的数对和。比方说，如果我们有数对 $(1, 5)$ ， $(2, 3)$ 和 $(4, 4)$ ，最大数对和为 $\max(1+5, 2+3, 4+4) = \max(6, 5, 8) = 8$ 。

给你一个长度为偶数 n 的数组 nums ，请你将 nums 中的元素分成 $\frac{n}{2}$ 个数对，使得：

nums 中每个元素恰好在一个数对中，且最大数对和的值最小。

请你在最优数对划分的方案下，返回最小的最大数对和。

输入示例：

```

nums = [3, 5, 2, 3]
nums = [3, 5, 4, 2, 4, 6]

```

输出示例：

```

7 // 数组中的元素可以分为数对(3,3)和(5,2)，最大数对和为
   max(3+3, 5+2) = max(6, 7) = 7。
8 // 数组中的元素可以分为数对(3,5), (4,4)和(6,2)，最大数对
   和为 max(3+5, 4+4, 6+2) = max(8, 8, 8) = 8。

```

例题5.2.11的含义为：给定一个大小为 N 的数组，按一定的组合方式得到 $\frac{N}{2}$ 个数对，然后这个组合中最大的数对和要是所有组合方式中最小的。

数组内只有两个数的情况是平凡³的，我们可以考虑数组中只有四个数 $x_1 \leq x_2 \leq x_3 \leq x_4$ 的情况。此时 $(x_1, x_4), (x_2, x_3)$ 的拆分方法对应的最大数对和一定是最小的。那么，对于 n 个数的情况(n 为偶数)，上述的条件下

³平凡就是指最简单的情形，或者说是容易证明的、容易看到的

述的条件对应的拆分方法，即第 k 大与第 k 小组成的 $\frac{n}{2}$ 个数对，同样可以使得最大数对和最小。

所以，总的来说是想让数对和都趋于平均然后返回最大数对和，因此本题的做法也很简单：先排序，然后每次从首和尾取值，记录最大值就是答案。

例 5.2.12 (跳跃游戏). 给定一个非负整数数组 `nums`，你最初位于数组的第一个下标。数组中的每个元素代表你在该位置可以跳跃的最大长度。

判断你是否能够到达最后一个下标。

输入示例：

```
nums = [2,3,1,1,4]  
nums = [3,2,1,0,4]
```

输出示例：

```
true    // 可以先跳 1 步，从下标 0 到达下标 1，然后再从下标  
        1 跳 3 步到达最后一个下标。  
false   // 无论怎样，总会到达下标为 3 的位置。但该下标的最大  
        跳跃长度是 0， 所以永远不可能到达最后一个下标。
```

我们来看例题5.2.12，设想一下，对于数组中的任意一个位置 y ，我们如何判断它是否可以到达？

根据题目的描述，只要存在一个位置 x ，它本身可以到达，并且它跳跃的最大长度为 $x + \text{nums}[x]$ ，这个值大于等于 y ，即 $x + \text{nums}[x] \geq y$ ，那么位置 y 也可以到达。

这样一来，我们依次遍历数组中的每一个位置，并实时维护最远可以到达的位置。对于当前遍历到的位置 x ，如果它在最远可以到达的位置的范围内，那么我们就可以从起点通过若干次跳跃到达该位置，因此我们可以根据 $x + \text{nums}[x]$ 更新最远可以到达的位置。

在遍历的过程中，如果最远可以到达的位置大于等于数组中的最后一个位置，那就说明最后一个位置可达，我们就可以直接返回 `true` 作为答案。

反之，如果在遍历结束后，最后一个位置仍然不可达，我们就返回false作为答案。

```
public boolean canJump(int[] nums) {  
    int pos = nums.length - 1;  
    int rightmost = 0;  
    for (int i = 0; i <= pos; ++i) {  
        //如果可以到达当前位置，则更新最大  
        if (i <= rightmost) {  
            //每次更新最大的位置  
            rightmost = Math.max(rightmost, i + nums[i]);  
            //如果可以到达最后一个位置，则直接返回  
            if (rightmost >= pos) {  
                return true;  
            }  
        }  
    }  
    return false;  
}
```

回顾几道例题，我们不难发现“贪心题”虽然大多数代码简单，但题目本身并不简单。因为“贪心题”重点在于证明，只AC不证明的最大弊端在于：简单的条件修改后，选手根本无法回答原做法是否还是正确。

因此“AC并证明”才是正确的、科学的做“贪心题”的态度，而不是因为不会证明或者看不懂证明而选择放弃证明。

5.2.8 模拟算法

模拟算法是一种最基本的算法思想，是对基本编程能力的一种考查，其解决方法就是根据题目给出的规则对题目要求的相关过程进行编程模拟。

在解决模拟类问题时，需要注意**字符串处理、特殊情况处理和对题目意思的理解**。在解题时，需要仔细分析题目给出的规则，要尽可能地做到全

面地考虑所有可能出现的情况，这是解模拟类问题的关键点之一。

模拟算法的例题可以参考洛谷 P1328 和洛谷 P1067。

5.3 快速幂

快速幂 (Exponentiation by squaring, 平方求幂) 是一种简单而有效的小算法，它可以以 $O(\log n)$ 的时间复杂度计算乘方。

如果我们要做一些高次幂运算时，往往不能暴力求解，这里则可以给出一个思路，例如我们要求 m^n ，则当 n 为偶数时， $m^n = (m^2)^{\frac{n}{2}}$ 。

一个简单的快速幂思路如下：

```
// 递归快速幂
int quickPow(int m, int n) {
    if (n == 0)
        return 1;
    else if (n % 2 == 1)
        return quickPow(m, n - 1) * a;
    else {
        int temp = quickPow(m, n / 2);
        return temp * temp;
    }
}
```

在实际问题中，题目常常会要求对一个大素数取模，这是因为计算结果可能会非常巨大，但是在里考察高精度又没有必要。这时我们的快速幂也应当进行取模，此时应当注意，原则是步步取模，如果 MOD 较大，还应当开 long long。

```
//递归快速幂（对大素数取模）
#define MOD 1000000007
typedef long long ll;
ll qpow(ll a, ll n) {
```

```

if (n == 0)
    return 1;
else if (n % 2 == 1)
    return qpow(a, n - 1) * a % MOD;
else {
    ll temp = qpow(a, n / 2) % MOD;
    return temp * temp % MOD;
}
}

```

递归虽然简洁，但会产生额外的空间开销。我们可以把递归改写为循环，来避免对栈空间的大量占用，也就是非递归快速幂。

我们换一个角度来引入非递归的快速幂。还是 7 的 10 次方，但这次，我们把 10 写成二进制的形式，也就是 $(1010)_2$ 。现在我们要计算 $7^{(1010)_2}$ ，我们可以很自然地想到把它拆分为 $7^{(1000)_2} \cdot 7^{(10)_2}$ 。实际上，对于任意的整数，我们都可以把它拆成若干个 $7^{(100\cdots)_2}$ 的形式相乘。而这些 $7^{(100\cdots)_2}$ ，恰好就是 $7^1, 7^2, 7^4 \dots$ 我们只需不断把底数平方就可以算出它们。

```

//非递归快速幂
int quickPow(int m, int n) {
    int ans = 1;
    while(n) {
        if(n & 1)          // 如果n的当前末位为1
            ans *= m;    // ans乘上当前的m
        m *= m;           // m自乘
        n >>= 1;          // n往右移一位
        // >>是右移，表示把二进制数往右移一位，相当于/2
    }
    return ans;
}

```

我们根据代码来推敲这个过程，最初令 ans 为 1，然后我们一位一位计算：

- $(1010)_2$ 的最后一位是 0，所以 m^1 这一位不要。然后 1010 变为 101， m 变为 m^2 。
- $(101)_2$ 的最后一位是 1，所以 m^2 这一位是需要的，乘入 ans。101 变为 10， m 再自乘。
- $(10)_2$ 的最后一位是 0，跳过，右移，自乘。
- 然后 1 的最后一位是 1，ans 再乘上 m^8 。循环结束，返回结果。

a	n_{binary}	ans
$7^{(1)_2}$	1010	1
$7^{(10)_2}$	101	$7^{(10)_2}$
$7^{(100)_2}$	10	$7^{(10)_2}$
$7^{(1000)_2}$	1	$7^{(10)_2} \cdot 7^{(1000)_2}$

5.3.1 一个拓展

上面所述的都是整数的快速幂，但其实，在算 a^n 时，只要 a 的数据类型支持乘法且满足结合律，快速幂的算法都是有效的。矩阵、高精度整数，都可以照搬这个思路。

```
//泛型的非递归快速幂
template <typename T>
T qpow(T a, ll n) {
    T ans = 1; // 赋值为乘法单位元，可能要根据构造函数修改
    while (n) {
        if (n & 1)
            ans = ans * a;
        n >>= 1;
        a = a * a;
    }
}
```

```

    }
    return ans;
}

```

这里有一点需要注意：较复杂类型的快速幂的时间复杂度不再是简单的 $O(\log n)$ ，它与底数的乘法的时间复杂度有关。

例如，我们再来求解一下斐波拉契数列，不过这里用的是矩阵来求解：

例 5.3.1 (斐波拉契数列).

$$F_n = \begin{cases} 1 & (n \leq 2) \\ F_{n-1} + F_{n-2} & (n \geq 3) \end{cases}$$

请求出 $F_n \bmod 10^9 + 7$ 的值。

我们设矩阵 $A = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$ ，我们有 $A \begin{bmatrix} F_n \\ F_{n+1} \end{bmatrix} = \begin{bmatrix} F_{n+1} \\ F_{n+1} + F_n \end{bmatrix} = \begin{bmatrix} F_{n+1} \\ F_{n+2} \end{bmatrix}$ ，于是：

$$\begin{aligned} \begin{bmatrix} F_n \\ F_{n+1} \end{bmatrix} &= A \begin{bmatrix} F_{n-1} \\ F_n \end{bmatrix} \\ &= A^2 \begin{bmatrix} F_{n-2} \\ F_{n-1} \end{bmatrix} \\ &= \dots \\ &= A^{n-1} \begin{bmatrix} F_1 \\ F_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \end{aligned}$$

这样以来，原来较为复杂的问题就转化成了求某个矩阵的幂的问题，这就可以应用快速幂求解：

```
#include <cstdio>
```

```
#define MOD 1000000007

typedef long long ll;

struct matrix {
    ll a1, a2, b1, b2;
    matrix(ll a1, ll a2, ll b1, ll b2) : a1(a1), a2(a2), b1(
        b1), b2(b2) {}
    matrix operator*(const matrix &y) {
        matrix ans((a1 * y.a1 + a2 * y.b1) % MOD,
                   (a1 * y.a2 + a2 * y.b2) % MOD,
                   (b1 * y.a1 + b2 * y.b1) % MOD,
                   (b1 * y.a2 + b2 * y.b2) % MOD);
        return ans;
    }
};

matrix qpow(matrix a, ll n) {
    matrix ans(1, 0, 0, 1); //单位矩阵
    while (n) {
        if (n & 1)
            ans = ans * a;
        a = a * a;
        n >>= 1;
    }
    return ans;
}

int main() {
    ll x;
    matrix M(0, 1, 1, 1);
    scanf("%lld", &x);
    matrix ans = qpow(M, x - 1);
    printf("%lld\n", (ans.a1 + ans.a2) % MOD);
```

```
    return 0;  
}
```

5.4 高精度

高精度计算 (Arbitrary-Precision Arithmetic)，通常也被称作**大整数** (bignum) 计算，运用了一些算法结构来支持更大整数间的运算，数字大小**超过**语言内建整型。

通常来说，高精度数字利用字符串存储，每个字符表示数字的一个十进制位，因此高精度计算可以视作一种字符串处理。

我们在读入字符串的时候，数字最高位会在字符串首，但我们更希望下标最小的位置存放的是数字的**最低位**，也就是存储反转的字符串。这么做的目的是当数字长度发生变化时，同样权值位可以始终保持对齐，即所有的个位都在下标[0]，所有的十位都在下标[1]…… 另一方面，所有的加、减、乘运算都从个位开始计算，为此我们存储的都是**反转的字符串**。

在这里，我们约定一个常数LEN = 1004，代表程序所容纳的最大长度，由此我们首先写出高精度数字的读入：

```
void clear(int a[]) {  
    for(int i = 0; i < LEN; i++)  
        a[i] = 0;  
}  
  
void read(int a[]) {  
    static char s[LEN + 1];  
    cin >> s;  
  
    clear(a);  
  
    int strLength = strlen(s); // 获取字符串s的长度  
    for(int i = 0; i < strLength; i++)
```

```
// s[i] - '0' 代表 s[i] 的数码
a[(len - 1) - i] = s[i] - '0';
}
```

对于输出，我们也要逆序输出，但通常情况下，我们并不希望输出前导零，所以我们需要从最高位开始向下寻找**第一个非零位**，从此处开始输出。需要注意的是终止条件是*i >= 1*而非*i >= 0*，目的是当整个数字都是 0 时，仍然可以输出一个字符 0。

```
void print(int a[]) {
    int i;
    for(i = LEN - 1; i >= 1; --i)
        if(a[i] != 0)
            break;
    for(; i >= 0; --i)
        putchar(a[i] + '0');
    putchar('\n');
}
```

5.4.1 四则运算

四则运算的实现难度也不相同，从最简单的**高精度加减法**开始，其次是高精度乘法，高精度乘法又分为**单精度乘法**和**高精度乘法**，最后是**高精度除法**。我们也按这个顺序实现功能。

高精度加法

高精度加法，就是**竖式加法**，从最低位开始，将两个加数对应位置上的数码相加，并判断是否不大于 10，如果达到 10 或以上，则将高一位的结果增加 1，当前位的结果减少 10。

```
void add(int a[], int b[], int c[]) {
    clear(c);
```

```

/***
 * 高精度实现中，一般令数组的最大长度LEN比可能的输入大，
 * 并略去末尾的几次循环，这样可以省去不少边界情况的处
 * 理，因为实际输入不会通常不会超过1000位。
*/
for(int i = 0; i < LEN - 1; i++) {
    // 对应位的数码相加
    c[i] += a[i] + b[i];
    if(c[i] >= 10) {      // 进位
        c[i + 1] += 1;
        c[i] -= 10;
    }
}
}

```

高精度减法

高精度减法也是竖式计算，即从个位逐位相减，遇到负数情况则向上一位“借”1，整体思路与加法一致：

```

void sub(int a[], int b[], int c[]) {
    clear(c);

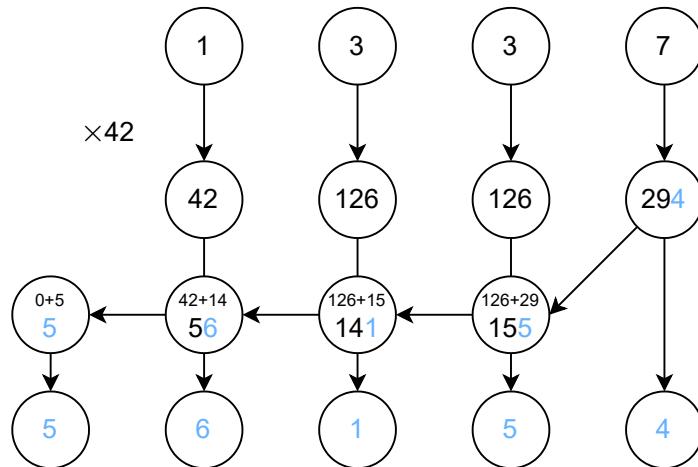
    for(int i = 0; i < LEN - 1; i++) {
        // 逐位相减
        c[i] += a[i] - b[i];
        if(c[i] < 0) {
            c[i + 1] -= 1;
            c[i] += 10;
        }
    }
}

```

显然这个代码存在问题，因为其只能处理减数 a 大于等于被减数 b 的情况，如果此时 $a < b$ ，则可以视作 $a - b = -(b - a)$ ，并以此为据调用 `sub` 函数，通过调换减数和被减数得到结果，写法为 `sub(b, a, c)`，最后在结果前面添加负号即可。

单精度乘法

通常，我们计算乘法，首先想到的也会是竖式计算，但如果乘数之一 b 是普通的 `int` 类型，我们就可以直接将 a 中的每一位数字乘以 b ，但这样需要进行一些整理。例如，我们计算 1337×42 的过程：



我们也是从个位开始逐位向上进位，但由于进位可能特别大，所以这里的进位不能简单地进行 -10 运算，而是通过对 10 取余以及进位除以 10 的商：

```
void mul_short(int a[], int b[], int c[]) {
    clear(c);

    for(int i = 0; i < LEN - 1; ++i) {
        // 直接相乘，加入结果
        c[i] += a[i] * b;
        if(c[i] >= 10) {      // 进位处理
            c[i + 1] += c[i] / 10;
            c[i] %= 10;
        }
    }
}
```

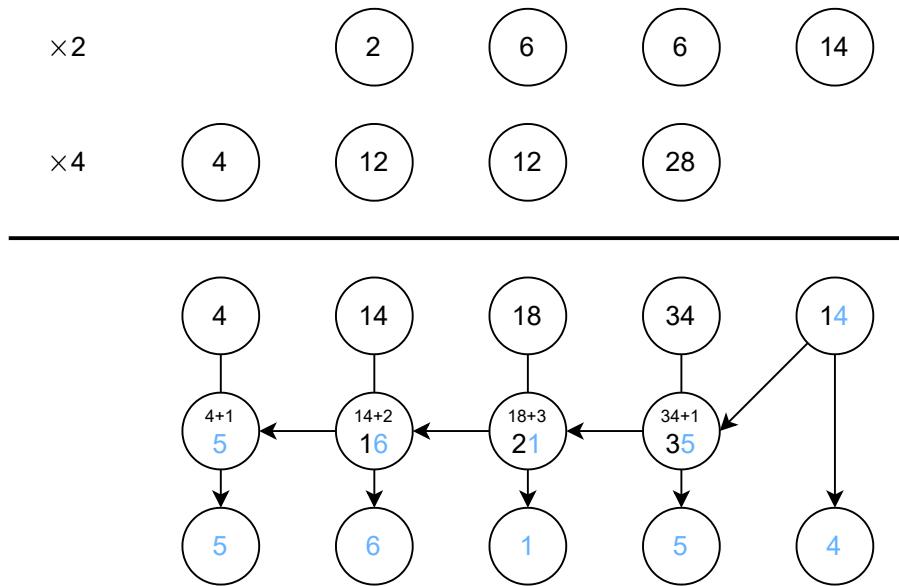
```
// c[i] / 10 得到的商是进位的量  
c[i + 1] += c[i] / 10;  
// c[i] % 10 得到的余数是在当前位留下的值  
c[i] %= 10;  
}  
}  
}
```

同样的，单精度乘法必然存在问题，其要求我们特别关注乘数 b 的取值范围，特别是当 b 和相应整型的**取值上界**属于同一量级时，就要特别注意，慎重考虑单精度乘法的使用。

高精度乘法

如果两个乘数都是高精度，那么仍旧考虑竖式计算，关注竖式乘法的运算过程，实际上是计算若干 $a \times b_i \times 10^i$ 的和，例如 1337×42 ，实际上计算的是 $1337 \times 2 \times 10^0 + 1337 \times 4 \times 10^1$ 。

于是，我们可以将 b 分解为其所有数码，得到的每个数码都是单精度数，将之分别与 a 相乘，再向左移动到各自的位置上相加即是最后的答案。当然，最后的进位也要与上述方法一致：



需要注意的是，这个过程与竖式乘法不完全相同，我们在计算时每一步乘的过程并不进位，而是将结果保留在对应的位置上留到最后统一处理，这样不会影响最后的结果。

```

void mul(int a[], int b[], int c[]) {
    clear(c);
    /**
     * 这里直接计算结果中从低到高的第i位，且一并处理了进位，
     * 第i次循环为c[i]加上了所有满足p + q = i的a[p]与b[q]的
     * 乘积之和，
     * 这样做的结果和上图运算的最后求和一样，只是更加简短。
     */
    for(int i = 0; i < LEN - 1; i++) {
        for(int j = 0; j <= i; j++)
            c[i] += a[j] * b[i - j];
        if(c[i] >= 10) {
            c[i + 1] += c[i] / 10;
            c[i] %= 10;
        }
    }
}

```

}

高精度除法

高精度除法即是竖式长除法，例如 $456 \div 12$ 的计算过程：

$$\begin{array}{r}
 & 3 \ 8 \\
 1 \ 2 & \overline{)4 \ 5 \ 6} \\
 & 3 \ 6 \\
 \hline
 & 9 \ 6 \\
 & 9 \ 6 \\
 \hline
 & 0
 \end{array}$$

竖式长除法实际上可以看作一个逐次减法的过程，例如上图中商数十位的计算可以这样理解：将 45 减去三次 12 后变得小于 12，不能再减，故此位为 3。

为了减少冗余计算，我们提前得到被除数的长度 l_a 和除数的长度 l_b ，从下标 $l_a - l_b$ 开始，从高位到低位计算商值：

```

// 被除数 a 以下标 last_dg 为最低位，是否可以再减去除数 b 而
// 保持非负
// len 是除数 b 的长度，避免反复计算
inline bool greater_eq(int a[], int b[], int last_dg, int
len) {
    // 有可能被除数剩余的部分比除数长，这个情况下最多多出 1
    // 位，故如此判断即可
    if (a[last_dg + len] != 0) return true;
    // 从高位到低位，逐位比较
    for (int i = len - 1; i >= 0; --i) {
        if (a[last_dg + i] > b[i]) return true;
        if (a[last_dg + i] < b[i]) return false;
    }
}

```

```
}

// 相等的情形下也是可行的
return true;
}

void div(int a[], int b[], int c[], int d[]) {
    clear(c);
    clear(d);

    int la, lb;
    for (la = LEN - 1; la > 0; --la)
        if (a[la - 1] != 0) break;
    for (lb = LEN - 1; lb > 0; --lb)
        if (b[lb - 1] != 0) break;
    if (lb == 0) {
        puts("> <");
        return;
    } // 除数不能为零

    // c 是商
    // d 是被除数的剩余部分，算法结束后自然成为余数
    for (int i = 0; i < la; ++i) d[i] = a[i];
    for (int i = la - lb; i >= 0; --i) {
        // 计算商的第 i 位
        while (greater_eq(d, b, i, lb)) {
            // 若可以减，则减
            // 这一段是一个高精度减法
            for (int j = 0; j < lb; ++j) {
                d[i + j] -= b[j];
                if (d[i + j] < 0) {
                    d[i + j + 1] -= 1;
                    d[i + j] += 10;
                }
            }
        }
    }
}
```

```
    }
    // 使商的这一位增加 1
    c[i] += 1;
    // 返回循环开头，重新检查
}
}

}
```

在上述程序中，我们实现了一个**greater_eq()**的函数，该函数用于判断被除数以下标**last_dg**为最低位，是否可以再减去除数而保持非负。此后对于商的每一位，不断调用**greater_eq()**，并在成立的时候用高精度减法从余数中减去除数，也即模拟了竖式除法的过程。

至此，我们完成了高精度运算的四则运算实现。

第六章 搜索算法

6.1 暴力搜索算法

暴力搜索通常会将所有可能结果都列举出来，从中寻找答案，性能不会很高。但是，能将暴力算法写出来通常就意味着你已经把题目做出来一半了。例如我们来看一个例题：

例 6.1.1 (3D 迷宫). 你被困在了一个 3D 的迷宫中，在这个迷宫内 'S' 代表起点，'E' 代表终点，'.' 代表可以走的单元，'#' 代表不可以走的单元。

现在假设每次移动都将耗费 1 分钟的时间，请你计算出逃离迷宫的最短时间，如 'Escaped in 11 minute(s).' 表示逃出迷宫的最短时间为 11 分钟。如果不可逃出的话，请输出 'Trapped!'。

输入示例： 输入第一行有 3 个数字，分别表示迷宫层数 L、每层迷宫的行数 R、每层迷宫的列数 C，随后 L 个矩阵表示各层迷宫的内部情况。

```
1 3 3
S##
#E#
###
3 4 5
S.... ##### #####
.###. ##### #####
.##. .##.## #.###
###.## ##... ####E
```

输出示例：

```
Trapped!
Escaped in 11 minute(s).
```

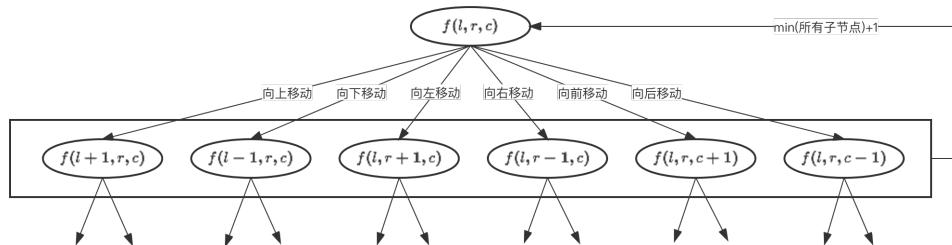
例题6.1.1可以认为是一个图的问题，我们从起点要找到一条通往终点的路最简单的想法就是暴力穷举。那么该如何穷举？首先我们需要确定状态转移方程，我们通常使用状态名称(参数列表)来表示一个状态，而使用状态转移方程来发现状态转移的规律。就本题而言，我们用l表示当前迷宫层数，r表示当前迷宫行号，c表示当前迷宫列号。因而状态有两种表达形式：

1. 用 $f(l, r, c)$ 表示从**当前位置**出发到**目的地**的最短时间，那么计算从起点到终点最短时间就可以表示为 $f(\text{起点层数}, \text{起点行号}, \text{起点列号})$
2. 用 $f(l, r, c)$ 表示从**起点**出发到**当前位置**的最短时间，那么计算从起点到终点最短时间就可以表示为 $f(\text{终点层数}, \text{终点行号}, \text{终点列号})$

上述两种状态表示可以使用同一个状态转移方程，这个方程有三个值：

```
 $f(l, r, c) = \min(f(l\pm1, r, c), f(l, r\pm1, c), f(l, r, c\pm1)) + 1$ 
=  $\infty$  // 当  $(l, r, c)$  位置不合法时
= 0 // 当  $(l, r, c)$  就是 目的地
```

这个状态转移方程代表的是**某个点到目的地的最短时间**就是向另外六个方向（上下前后左右）转移后到目的地的最短时间 +1，下图即是状态转移示意图。



6.1.1 深度优先搜索

深度优先搜索 (Depth-First-Search, DFS) 是暴力搜索的常用手段，该算法的特点是不撞南墙不回头，并且通常用递归来实现，大致如图6.1所示：

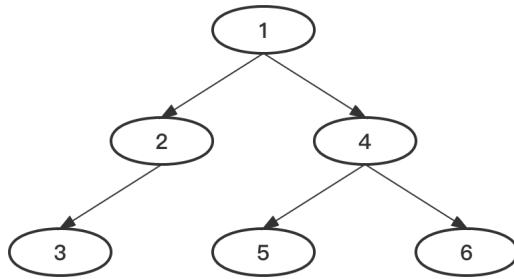


图 6.1: DFS 示意图

现在我们用 DFS 来解决问题，用 $f(l, r, c)$ 表示从当前位置出发到目的地的最短时间，为了不让搜索算法走回头路，我们需要记录搜索算法走过状态，遇到走不通的时候再回头选择其他分支，也就是回溯思想，具体实现如下：

```

int L, R, C;      // 迷宫的层数、行数、列数
char labyrinth[30][30][30]; // 迷宫具体结构
int dir[6][3] = {
    {1,0,0},{-1,0,0},{0,1,0},{0,-1,0},{0,0,1},{0,0,-1} }; // 
    // 用来存储六个方向
bool vis[30][30][30]; // 记录迷宫某个位置是否走过

int dfs(int l, int r, int c) {
    if (labyrinth[l][r][c] == 'E') {
        return 0;
    }

    int rlt = INT_MAX;
    for (int i = 0; i < 6; i++) {
  
```

```

int ll = l + dir[i][0];
int rr = r + dir[i][1];
int cc = c + dir[i][2];
if (ll < 0 || ll >= L || rr < 0 || rr >= R || cc < 0
    || cc >= C ||
    labyrinth[ll][rr][cc] == '#' || vis[ll][rr][cc])
    { // 如果这个位置已经来过了就不用进入了
    continue;
}

vis[l][r][c] = true; // 记录当前位置已经走过了
rlt = min(rlt, dfs_back(ll, rr, cc));
vis[l][r][c] = false; // 解除记录，继续走其他的分支
}
return (rlt != INT_MAX) ? rlt + 1 : INT_MAX;
}

```

6.1.2 宽度优先搜索

宽度优先搜索 (Breadth-First-Search, BFS) 也是暴力搜索的常用手段，该算法的特点是逐层深入，因此通常用队列来实现，大致如图6.2所示：

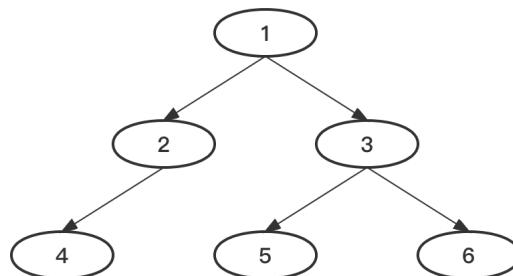


图 6.2: BFS 示意图

我们用 BFS 来解决问题，使用 $f(l, r, c)$ 表示从起点出发到当前位置的最短时间，可以写出下面的算法：

```
int L, R, C;      // 迷宫的层数、行数、列数
char maze[30][30][30]; // 迷宫具体结构
int dir[6][3] = {
    {1,0,0}, {-1,0,0}, {0,1,0}, {0,-1,0}, {0,0,1}, {0,0,-1} }; // 用来存储六个方向
bool vis[30][30][30]; // 记录迷宫某个位置是否走过

// 记录每一步的状态，val=f(l,r,c)
struct step {
    int l, r, c;
    int val;
};

int bfs(int l, int r, int c) {
    queue<step> q; // 状态队列
    q.push(step{ l,r,c,0 }); // 初始状态
    while (!q.empty()) {
        step s = q.front();
        q.pop(); // 取出队列首个状态

        for (int i = 0; i < 6; i++) {
            // 计算下一个方向并判断下一个方向可不可行
            step next = { s.l + dir[i][0], s.r + dir[i][1], s.c + dir[i][2], s.val + 1 };
            if (next.l < 0 || next.l >= L || next.r < 0 ||
                next.r >= R || next.c < 0 || next.c >= C ||
                maze[next.l][next.r][next.c] == '#' || vis[
                    next.l][next.r][next.c]) { // 使用回溯法，防止无限递归
                continue;
            }
            vis[next.l][next.r][next.c] = true;
            q.push(next);
        }
    }
}
```

```

    }

    else if (maze[next.l][next.r][next.c] == 'E') {
        // 如果到终点返回最短时间
        return next.val;
    }

    vis[next.l][next.r][next.c] = true;      // 记录
                                                下一个位置已经探测过了
    q.push(next);   // 向队列中推入下一个状态
}

return INT_MAX; // 如果不可达，返回无穷大
}

```

6.2 记忆搜索算法

暴力搜索性能较差，其原因在于存在大量重复搜索，我们仍通过一个例题来看**记忆搜索算法**：

例 6.2.1 (滑雪). *Michael* 喜欢滑雪但这并不奇怪，因为滑雪的确很刺激。可是为了获得速度，滑的区域必须向下倾斜，而且当你滑到坡底，你不得不再次走上坡或者等待升降机来载你。*Michael* 想知道载一个区域中最长底滑坡。区域由一个二维数组给出。数组的每个数字代表点的高度。例如：

1	2	3	4	5
16	17	18	19	6
15	24	25	20	7
14	23	22	21	8
13	12	11	10	9

一个人可以从某个点滑向上下左右相邻四个点之一，当且仅当高度减小。

在上面的例子中，一条可滑行的滑坡为24-17-16-1。

当然25-24-23-...-3-2-1更长。事实上，这是最长的一条。

输入：输入的第一行表示区域的行数R和列数C($1 \leq R, C \leq 100$)。下面是R行，每行有C个整数，代表高度height， $0 \leq h \leq 10000$ ，下面是一个例子：

```
5 5
1 2 3 4 5
16 17 18 19 6
15 24 25 20 7
14 23 22 21 8
13 12 11 10 9
```

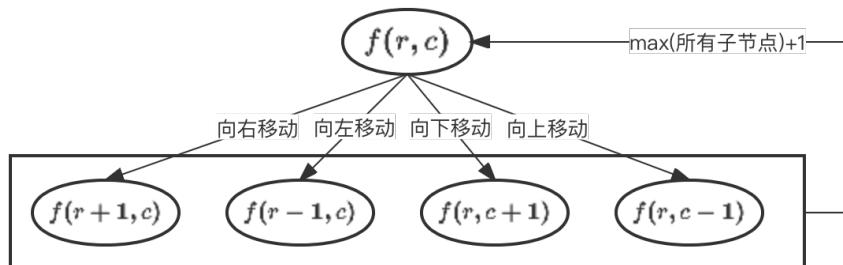
输出：

输出最长区域的长度，本题为25。

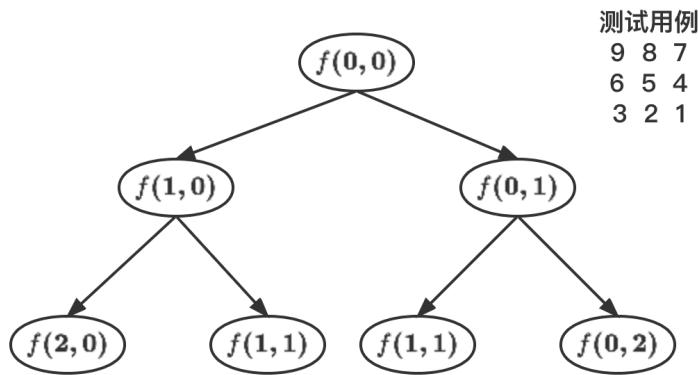
根据例题6.2.1所述，如果要找到最长的滑行距离，实际上就是计算所有位置可以滑雪的最长距离，然后从中选最长的。那么为了计算某个位置可以滑雪的最长距离，首先我们还是来分析一下本题的状态空间和状态转移方程。就本题而言，我们使用r来表示滑雪者所在的行号，c表示滑雪者所在的列号，用 $f(r, c)$ 表示从当前位置开始滑雪者可以走过的最长区域长度，状态转移方程可以表示为：

$$\begin{aligned} f(r, c) &= \max(f(r \pm 1, c), f(r, c \pm 1)) + 1 \\ &= 1 \quad // \text{当 } (r, c) \text{ 不可以再向其他方向移动时} \end{aligned}$$

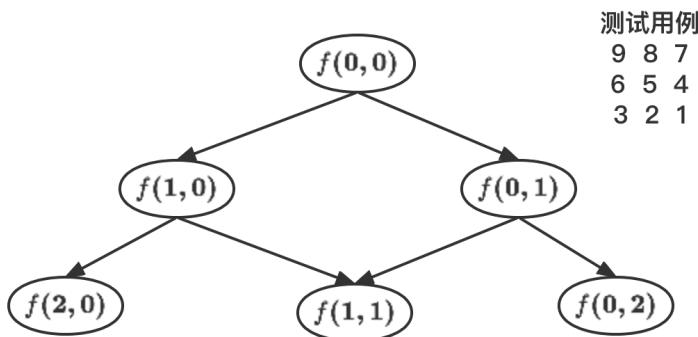
也就是某个位置可以滑雪的最长距离等于转移到周围四个位置后可以滑雪的最长距离+1，若位置不合法最长距离就是0。状态转移示意图如下所示：



接下来，让我们来观察一个测试用例，并思考一下搜索算法的前几步：



注意到，当我们从 $f(0,0)$ 状态出发后，搜索算法连续两次计算了 $f(1,1)$ ，事实上，我们完全可以保存第一次计算的结果，在第二次遇到 $f(1,1)$ 时直接使用上一次计算的结果就好了。这种记录结果再利用的搜索方法，就称之为记忆化搜索，示意图如下：



有了上述的分析，我们开始编写算法：

```

int map[100][100]; // 保存各个位置的高度
int dp[100][100]; // 保存计算过的结果，即 dp[r][c] = f(r, c)
int dir[4][2] = { {1,0}, {-1,0}, {0,1}, {0,-1} }; // 保存转移
方向
int R, C; // 保存该区域的行数和列数
    
```

```
int dfs(int r, int c) {
    // 如果已经计算过这个状态了则直接返回结果
    if (dp[r][c] != 0){
        return dp[r][c];
    }

    int rlt = 0;
    for (int i = 0; i < 4; i++) {
        // 计算下一个方向并判断下一个方向可不可行
        int rr = r + dir[i][0];
        int cc = c + dir[i][1];
        if (rr < 0 || rr >= R || cc < 0 || cc >= C || map[r][c] <= map[rr][cc]) {
            continue;
        }

        // 获取下一个状态的最长距离
        int tmp = dfs(rr, cc);
        rlt = (tmp > rlt) ? tmp : rlt;
    }

    // 注意！这里在返回语句中保存计算过的结果
    return dp[r][c] = rlt + 1;
}

int solve() {
    // 计算每个位置的可以滑雪的最长距离，并返回一个最大的
    int maxLen = 0;
    for (int i = 0; i < R; i++) {
        for (int j = 0; j < C; j++) {
            int tmp = dfs(i, j);
            maxLen = (tmp > maxLen) ? tmp : maxLen;
        }
    }
}
```

```
    }
}

return maxLen;
}
```

记忆化搜索通过记录中间状态防止重复搜索，进而大幅加快搜索速度，这与另一个著名的算法思想**动态规划**是不谋而合的，两者在性能没有显著差别。

搜索算法看到这里，我们会明显发现一个问题，即在上面的搜索算法中，我们都要遍历状态空间中所有的状态，然而很多时候，有些状态我们压根就不用计算，我们因此可以略过那些状态，例如跳过那些不能到达最终目标的分支，这也就是利用**剪枝思想**。

6.2.1 剪枝策略

所谓剪枝，也就是前文所提到的减治思想，即将问题分解为多个部分，舍弃其中不必要的部分，仅仅关注与问题有关的部分。我们仍然来看一道例题：

例 6.2.2 (拼合木棍). 乔治拿了相同长度的木棍，随机切开，直到所有零件的长度最大为 50 个单位。现在，他想将木棍恢复到原始状态，但是他忘记了原来多少木棍以及它们原本有多长。

请帮助他设计一个程序，计算出那些棍子的最小可能的原始长度。所有以单位表示的长度都是大于零的整数。

输入： 输入包含 2 行。

第一行包含切割后的木棍零件数，最多为 64 根木棍。

第二行包含被空格隔开的那些部分的长度。

```
9
```

```
5 2 1 5 2 1 5 2 1
```

```

4
1 2 3 4

```

输出：棍子的最小可能的原始长度，示例输出如下

```

6
5

```

例题6.2.2在我们通常的想法中往往是要假设一下可能的原始长度，不断尝试是否能将所有木棍零件都利用起来拼接成多根这个长度的木棍，最后在所有可行的原始长度中选一个最小的。

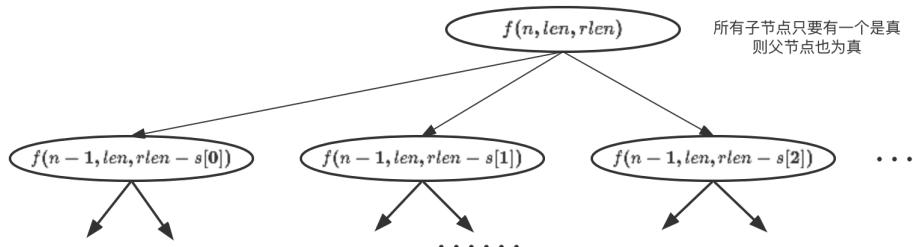
首先分析状态转移方程，我们可以使用n表示还有多少木棍零件没有使用，len表示可能的原始长度，rlen表示当前正在拼接的木棍长度与可能的原始长度相差多少个单位， $f(n, len, rlen)$ 表示在($n, len, rlen$)的条件下能否把所有的木棍都用上并且所有拼接起来的木棍的长度都是len，本题我们还需要用 $s[i]$ 来表示第i根木棍零件的长度。状态转移方程如下：

```

f(n, len, rlen) = f(n-1, len, rlen-s[0]) || f(n-1, len, rlen-s[1])
|| ...
= true      // 当 n==0 且 rlen==0 时
= false     // 当 s 中没有一个木棍零件可用

```

也就是使用n个木棍零件其中一个 $s[i]$ ，如果剩余木棍零件能够拼接出多根len的木棍和一个 $rlen-s[i]$ 的木棍，那么原问题就是可解的，需要注意的是这个搜索空间非常大，每个父节点生成的子节点个数不可估算，状态转移示意图如下：



我们回顾题目，题目中隐含了几条的信息：

- (1) 所有零件长度和一定是木棍长度的倍数，且所有木棍零件都要用到，因此木棍长度最小是所有木棍零件中的最大值。

此外题目中要求计算“那些棍子”，这意味着最终拼接的木棍不止一根，因此木棍长度最大是所有木棍零件长度和的一半。

此外题目要求找到最小的可行长度，那么只要从最小的可能木棍长度开始搜索即可，一旦搜索到可行解就不必考虑更大的木棍长度了。

- (2) 给所有木棍零件按长度从大到小排序，先用长的木棍零件可以比较容易知道方案可不可行。

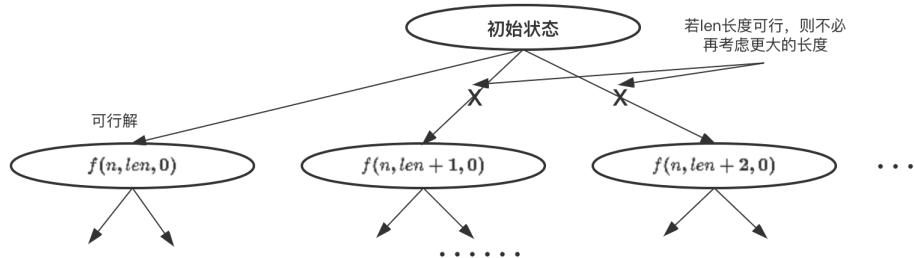
- (3) 在某次尝试中使用长度 L 的木棍零件不能得到可行解的话，那么所有长度为 L 的木棍零件都不能得到可行解。

例如一组木棍零件长度为 $[5, 5, 5, 3, 1]$ ，如果使用第一根长度为 5 的木棍零件不能得到可行解的话就不考虑使用后面两个长度为 5 的木棍零件了。

- (4) 如果使用第一个木棍零件不能得到可行解的话那么后面的情况也不用试了，因为第一个木棍零件肯定是要用上的。

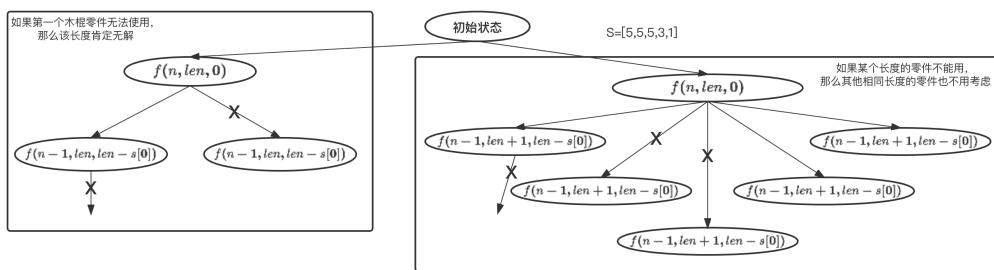
不存在更优解剪枝

在搜索状态空间的过程中，往往能找到多个可行解，但并非所有可行解都需要遍历，我们一般将这样的剪枝称为**不存在更优解剪枝**。在本题中，**隐含信息 (1)** 正是这类剪枝，当我们从小到大搜索到第一个可行解时，找到的就是最小的木棍长度了。



预测不可达剪枝

在搜索状态空间的过程中，往往能提前预测本状态是否能到达可行解，因此可以提前跳过不能到达可行解的分支，我们一般将这样的剪枝成为预测不可达剪枝。在本题中，**隐含信息(3)**和**隐含信息(4)**正是这类剪枝，示意图如下：



为了实现**隐含信息(3)**中的剪枝，我们将 $f(n, len, rlen)$ 进行修改，这里改为 $f(n, len, rlen, index)$ ，其中 $index$ 参数表示在 s 数组中开始搜索的下标，即当 $s[i]$ 不可达时，搜索算法直接跳到下一种长度的零件下标，并通过 $index$ 参数向下一层递归函数传递这个信息。

```

int N;           // 零件总数
int sum;         // 所有零件长度和
int s[64];       // 各零件长度
bool vis[64];    // 记录零件使用状态

bool dfs(int n, int len, int rlen, int index) {
    // n==0 和 rlen==0 同时满足表示找到了可行解
    if (n == 0 && rLen == 0) {
        return true;
    }

    // rlen 为 0 表示要拼接下一根木棍了
    if (rLen == 0) {
        rLen = len;
    }
}

```

```
    index = 0;
}

int pre = 0;      // 用来记录不能使用的木棍长度
for (int i = index; i < N; i++) {
    // 如果剩余长度比零件要短就跳过
    // 如果这个零件已经使用过了就跳过（回溯法）
    // 如果这个零件长度已经试过了不能用就跳过（隐含信息3）
    if (rLen < s[i] || vis[i] || s[i] == pre) {
        continue;
    }

    vis[i] = true; // 记录使用状态（回溯法）
    if (dfs(n - 1, len, rLen - s[i], i + 1)) {
        return true;
    }
    vis[i] = false; // 取消使用状态（回溯法）

    // 如果第一根棍子都不能用，那后面的都不要试了（隐含
    // 信息4）
    if (i == 0) {
        return false;
    }

    // 记录不可用的长度
    pre = s[i];
}
return false;
}

int solve() {
    sort(s); // 对s数组排序（隐含信息2）
```

```
for (int i = s[0]; i <= sum / 2; i++) {      // 木棍长度
    区间限制 (隐含信息 1)
    if (sum % i == 0) {                      // 所有零件长度和一
        定是木棍长度的倍数 (隐含信息 1)
        memset(vis, 0, N);                  // vis 数组都初始化为
        false
        if (dfs(N, i, 0, 0)) {
            return i;                      // 从小到大找, 若找到一
            个就不考虑其他的了 (隐含信息 1)
        }
    }
}
return 0;
}
```

6.3 双向搜索

我们常常会面临这样一类搜索问题：起点是已知的，终点也是已知的，需要确定能否从起点到达终点，如果可以，需要多少步。

如果我们用常规的搜索方法，无论是 DFS 还是 BFS，采用暴力搜索思想从起点开始进行，那得到的解答树可能非常庞大，这样漫无目的的搜索就像大海捞针。

这个时候我们就可以换种思路，既然终点是已知的，我们完全可以分别从起点和终点出发，看它们能否相遇，这样一来，如果原本的解答树规模是 a^n ，使用双向搜索后，规模立刻缩小到了约 $2a^{\frac{n}{2}}$ ，当 n 较大时优化非常可观。

双向搜索主要有两种：双向 BFS(双向宽度优先搜索) 和双向迭代加深。

6.3.1 双向宽度优先搜索

与普通的 BFS 不同，双向 BFS 维护两个而不是一个队列，然后轮流拓展两个队列。同时，用数组¹或哈希表记录当前的搜索情况，给从两个方向拓展的节点以不同的标记。当某点被两种标记同时标记时，搜索结束。

```
queue<T> Q[3]; // T表示通用的泛型，可能为 int, string 等
bool found = false;
Q[1].push(st); // st为起始状态
Q[2].push(ed); // ed为终止状态
for(int d = 0; d < D + 2; ++d){// D为最大深度，最后答案为 d-1
    int dir = (d & 1) + 1, sz = Q[dir].size(); // 记录一下
    // 当前的搜索方向，1为正向，2为反向
    for(int i = 0; i < sz; ++i) {
        auto x = Q[dir].front();
        Q[dir].pop();
        if (H[x] + dir == 3)// H是数组或哈希表，若 H[x]+dir
        ==3说明两个方向都搜到过这个点
            found = true;
        H[x] = dir;
        // 这里需要把当前状态能够转移到的新状态压入队列
    }
    if (found)
        // ...
}
```

6.3.2 双向迭代加深

首先简单介绍一下(单向)迭代加深。

迭代加深算法的思想和实现都很简单。就是控制 DFS 的最大深度，如果深度超过最大深度就返回。某个深度搜完后没有得到答案便令最大深度+1，

¹如果状态可以被表示为较小的整数时可以用数组进行存储

然后重新开始搜索。

迭代加深算法听起来好像效果和宽度优先搜索差不多，还重复搜索了很多次。但是，由于搜索的时间复杂度几乎完全由解答树的最后一层确定，所以它与 BFS 在时间上只有常数级别的差距，以此换来的优势是：空间占用很小，有时候方便剪枝、方便传参等。

双向迭代加深就是相应地，从两个方向迭代加深搜索。先从起点开始搜 0 层，再从终点开始搜 0 层，然后从起点开始搜 1 层，……，如此反复直到确认结果。

例 6.3.1 (洛谷 P1379-八数码难题). 在 3×3 的棋盘上，摆有八个棋子，每个棋子上标有 1 至 8 的某一数字。棋盘中留有一个空格，空格用 0 来表示。空格周围的棋子可以移到空格中。

要求解的问题是：给出一种初始布局（初始状态）和目标布局（为了使题目简单，设目标状态为 123804765），找到一种最少步骤的移动方法，实现从初始布局到目标布局的转变。

输入格式：输入初始状态，一行九个数字，空格用 0 表示

```
283104765
```

输出格式：只有一行，该行只有一个数字，表示从初始状态到目标状态需要的最少移动次数（测试数据中无特殊无法到达目标状态数据）

```
4
```

例题6.3.1的起始条件是给出的，终点也是确定的，典型的双向搜索。状态直接用 int 表示即可。考虑到九位数比较大，还是要开哈希表，可以直接使用 STL 中的unordered_map。

本题的参考实现代码如下：

```
#include <bits/stdc++.h>

using namespace std;
```

```
int e[] = {1, 10, 100, 1000, 10000, 100000, 1000000,
           10000000, 100000000, 1000000000}, D;
bool found;
unordered_map<int, int> H;

inline int at(int x, int i) { return x \% e[i + 1] / e[i]; }

inline int swap_at(int x, int p, int i) { return x - at(x, i)
    ) * e[i] + at(x, i) * e[p]; }

void dfs(int x, int p, int d, int dir) {
    if (H[x] + dir == 3)
        found = true;
    H[x] = dir;
    if (d == D)
        return;
    // p表示0的位置，这是比BFS好的一点，用BFS的话要专门开结构体储存p，或者现算
    if (p / 3)
        dfs(swap_at(x, p, p - 3), p - 3, d + 1, dir);
    if (p / 3 != 2)
        dfs(swap_at(x, p, p + 3), p + 3, d + 1, dir);
    if (p \% 3)
        dfs(swap_at(x, p, p - 1), p - 1, d + 1, dir);
    if (p \% 3 != 2)
        dfs(swap_at(x, p, p + 1), p + 1, d + 1, dir);
}

int main() {
    int st, p, ed = 123804765;
    cin >> st;
    for (p = 0; at(st, p); ++p); // 找到起始状态中0的位置
    while (1) {
```

```

dfs(st, p, 0, 1);
if (found) {
    cout << D * 2 - 1;
    break;
}
dfs(ed, 4, 0, 2);
if (found) {
    cout << D * 2;
    break;
}
D++;
}
return 0;
}

```

例 6.3.2 (洛谷 P2324-[SCOI2005] 骑士精神). 在一个 5×5 的棋盘上有 12 个白色的骑士和 12 个黑色的骑士, 且有一个空位。

在任何时候一个骑士都能按照骑士的走法 (它可以走到和它横坐标相差为 1, 纵坐标相差为 2 或者横坐标相差为 2, 纵坐标相差为 1 的格子) 移动到空位上。给定一个初始的棋盘, 怎样才能经过移动变成如下目标棋盘:

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & * & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

其中, 0 表示白色骑士, 1 表示黑色骑士, * 表示空位。为了体现出骑士精神, 他们必须以最少的步数完成任务。

输入格式:

第一行有一个正整数 $T(T \leq 10)$, 表示一共有 T 组数据。

接下来有 T 个 5×5 的矩阵, 每两组数据之间没有空行。

```
2
10110
01*11
10111
01001
00000
01011
110*1
01110
01010
00100
```

输出格式:

对于每组数据都输出一行, 如果能在 15 步以内 (包括 15 步) 到达目标状态, 则输出步数, 否则输出 -1。

```
7
-1
```

第七章 数据结构

7.1 栈

栈是一种特殊的线性表，它只能在一个表的一个**固定端**进行数据结点的插入和删除操作。栈按照**先进后出**的原则来存储数据，也就是说，先插入的数据将被压入栈底，最后插入的数据在栈顶。读出数据时，从栈顶开始逐个读出。栈中没有数据时，称为空栈。

7.1.1 单调栈

单调栈主要用于 $O(n)$ 解决 NGE(Next Greater Element) 问题，即对序列中每个元素，找到下一个(上一个)比它大(小)的元素。

这个实现也相对简单，我们维护一个栈，表示“**待确定 NGE 元素**”，随后遍历序列，每当我们碰上一个新元素，我们知道，越靠近栈顶的元素离新元素位置越近，所以我们不断比较新元素与栈顶元素，如果新元素比栈顶大，则可断定新元素就是栈顶的 NGE，于是弹出栈顶元素并继续进行比较，直到新元素不比栈顶大，再将新元素压入栈。当然，这样形成的栈是**单调递减的**。

例 7.1.1 (洛谷-P5788 【模板】单调栈). 给出项数为 n 的整数数列 $a_{1\dots n}$ 。

定义函数 $f(i)$ 代表数列中第 i 个元素之后第一个大于 a_i 的元素的下标，即 $f(i) = \min_{i < j \leq n, a_j > a_i} \{j\}$ 。若不存在，则 $f(i) = 0$ 。

求出 $f(1 \dots n)$ 。

输入格式

第一行一个正整数 n 。

第二行 n 个正整数 $a_{1\dots n}$ 。

```
5
1 4 2 3 5
```

输出格式

一行 n 个整数 $f(1 \dots n)$ 的值。

```
2 5 4 5 0
```

数据规模与约定：

对于 30% 的数据， $n \leq 100$ ；

对于 60% 的数据， $n \leq 5 \times 10^3$ ；

对于 100% 的数据， $1 \leq n \leq 3 \times 10^6$ ， $1 \leq a_i \leq 10^9$ 。

具体实现如下：

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    ios::sync_with_stdio(false);
    int n;
    cin >> n;
    vector<int> V(n + 1), ans(n + 1);
    for (int i = 1; i <= n; ++i)
        cin >> V[i];
    stack<int> S;
    for (int i = 1; i <= n; ++i) {
        while (!S.empty() && V[S.top()] < V[i]) {
            ans[S.top()] = i;
            S.pop();
        }
        S.push(i);
    }
}
```

```

    }
    S.push(i);
}
for (int i = 1; i <= n; ++i)
    cout << ans[i] << " ";
return 0;
}

```

例 7.1.2 (leetcode-42 接雨水). 给定 n 个非负整数表示每个宽度为1的柱子的高度图，计算按此排列的柱子下雨之后能接多少雨水。

输入格式:

输入的第一行包含一个整数 n ，表示一共有 n 个柱子。

第二行，包含 n 个整数，表示每个柱子的高度。

```

12
0 1 0 2 1 0 1 3 2 1 2 1

```

输出格式:

一行，包含一个数，表示接到的雨水量。

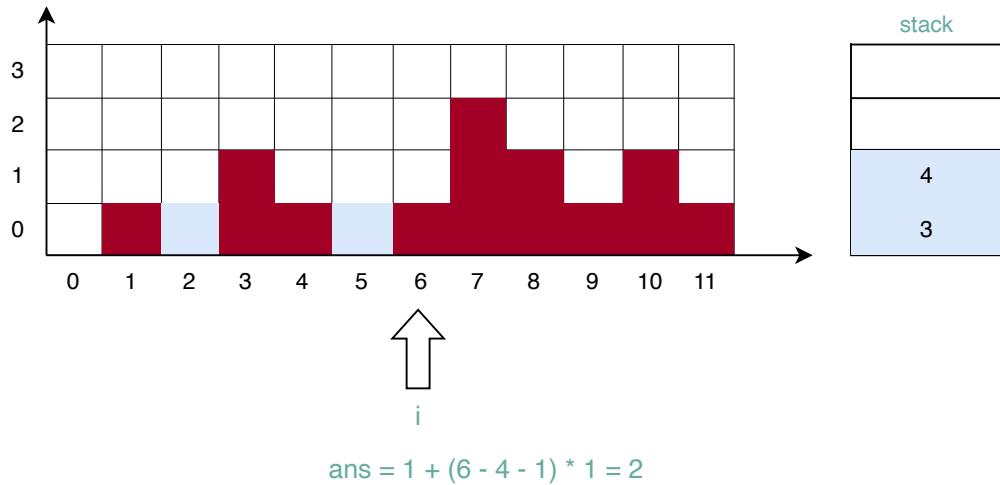
```

6

```

我们针对例题7.1.2首先维护一个单调栈，单调栈存储数组下标。我们用一个数组height[]来记录每个柱子的高度，并满足从栈底到栈顶的下标对应的数组height中的元素递减。

现在我们从左到右遍历数组，当我们遍历到下标 i 时，如果栈内至少有两个元素，记栈顶元素为 top , top 下面的那个的元素记为 $left$ ，显然有 $height[left] \geq height[top]$ ，如果 $height[i] > height[top]$ ，则可以得到一个高度是 $\min(height[left], height[i]) - height[top]$ ，宽度是 $i - left - 1$ 的一个接水的区域，根据高度和宽度就可以计算出雨水量。



为了得到 left，需要将 top 出栈，在对 top 计算能接的雨水量后，left 将变为新的 top，重复上述操作直到栈空或者下标对应的height中的元素大于或等于height[i]。

在对下标 i 处计算能接的雨水量后，将 i 入栈，继续遍历后面的下标并计算雨水量，完成遍历之后即可得到能接的雨水总量。

```
// ...
int ans = 0;
stack<int> stk;
for(int i = 0; i < n; ++i) {
    while(!stk.empty() && height[i] > height[stk.top()]) {
        int top = stk.top();
        stk.pop();
        if(stk.empty())
            break;
        int left = stk.top();
        int currentWidth = i - left - 1;
        int currentHeight = min(height[left], height[i]) -
                           height[top];
        ans += currentWidth * currentHeight;
    }
}
```

```

    stk.push(i);
}
// ...

```

例 7.1.3 (CF1313C2 Skyscrapers (hard version)). *The outskirts of the capital are being actively built up in Berland. The company "Kernel Panic" manages the construction of a residential complex of skyscrapers in New Berlskva.*

All skyscrapers are built along the highway. It is known that the company has already bought n plots along the highway and is preparing to build n skyscrapers, one skyscraper per plot.

Architects must consider several requirements when planning a skyscraper. Firstly, since the land on each plot has different properties, each skyscraper has a limit on the largest number of floors it can have. Secondly, according to the design code of the city, it is unacceptable for a skyscraper to simultaneously have higher skyscrapers both to the left and to the right of it.

Formally, let's number the plots from 1 to n . Then if the skyscraper on the i -th plot has a_i floors, it must hold that a_i is at most m_i ($1 \leq a_i \leq m_i$). Also there mustn't be integers j and k such that $j < i < k$ and $a_j > a_i < a_k$. Plots j and k are not required to be adjacent to i .

The company wants the total number of floors in the built skyscrapers to be as large as possible. Help it to choose the number of floors for each skyscraper in an optimal way, i.e. in such a way that all requirements are fulfilled, and among all such construction plans choose any plan with the maximum possible total number of floors.

输入格式:

The first line contains a single integer n ($1 \leq n \leq 500000$) —the number of plots.

The second line contains the integers m_1, m_2, \dots, m_n ($1 \leq m_i \leq 10^9$) —the limit on the number of floors for every possible number of floors for a skyscraper on each plot.

5
1 2 3 2 1

输出格式:

Print n integers a_i —the number of floors in the plan for each skyscraper, such that all requirements are met, and the total number of floors in all skyscrapers is the maximum possible.

If there are multiple answers possible, print any of them.

1 2 3 2 1

例题7.1.3属于单调栈优化 DP，非常显然的是最后形成的一定是一个**先单调增、再单调减**的序列。我们用 dpl 表示以某个点为最高点时答案数组的前缀和， dpr 表示以某个点为最高点时答案的后缀和， dp 表示以某个点为最高点时的答案，那么显然 $dp[i] = dpl[i] + dpr[i] - A[i]$ 。

怎么算 dpl 和 dpr 呢？以 dpl 为例，当 i 为最高点时，我们从这个点往左走，如果遇到比 $A[i]$ 大的，那么就必须把这个点削成 $A[i]$ ；如果遇到一个小于等于 $A[i]$ 的，那么后面的部分都可以沿用以此点为最高点的安排了。也就是说，我们要确定左侧最近的小于等于 $A[i]$ 的点的位置 $last[i]$ ，那么我们有： $dpl[i] = A[i] * (i - last[i]) + dpl[last[i]]$ 。

而左侧最近的小于等于 $A[i]$ 的点可以用单调栈算出。同理，也用单调栈，我们可以对每个 i 算出右侧最近的小于等于 $A[i]$ 的点 $next[i]$ ，那么我们有： $dpr[i] = A[i] * (next[i] - i) + dpr[next[i]]$ 。

这就 $O(n)$ 地解决了这个动态规划问题。

例 7.1.4 (洛谷-P1823 [COI2007] Patrik 音乐会的等待). n 个人正在排队进入一个音乐会。人们等得很无聊，于是他们开始转来转去，想在队伍里寻找

自己的熟人。

队列中任意两个人 a 和 b , 如果他们是相邻或他们之间没有人比 a 或 b 高, 那么他们是可以互相看得见的。

写一个程序计算出有多少对人可以互相看见。

输入格式:

输入的第一行包含一个整数 n , 表示队伍中共有 n 个人。

接下来的 n 行中, 每行包含一个整数, 表示人的高度, 以毫微米 (等于 10^{-9} 米) 为单位, 这些高度分别表示队伍中人的身高。

```
7  
2  
4  
1  
2  
2  
5  
1
```

输出格式:

输出仅有一行, 包含一个数 s , 表示队伍中共有 s 对人可以互相看见。

```
10
```

数据规模与约定:

对于全部的测试点, 保证 $1 \leq$ 每个人的高度 $< 2^{31}$, $1 \leq n \leq 5 \times 10^5$ 。

例题7.1.4体现了单调栈的一种重要应用, 即解决某些涉及到“**两元素间所有元素均(不)大/小于这两者**”的问题。

题中, 我们用一个单调栈存储那些**“可以看到当前位置的人的身高”**, 显然这是单调不增的。每当遍历到一个新的位置时, 就计算当前位置的人可以看到前面的多少人, 并更新单调栈。由于这道题有身高相等的情况, 所以需要合并相同身高的人 (这里可以用pair), 具体可以参考代码:

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
int main() {
    ios::sync_with_stdio(false);
    int n;
    cin >> n;
    ll ans = 0;
    vector<int> V(n);
    for (auto &e : V)
        cin >> e;
    stack<pair<int, int>> S; // 这里 pair 的第二个成员表示相同
                           // 元素的数量
    for (auto e : V) {
        int cnt = 0;
        while (!S.empty() && S.top().first <= e) {
            if (S.top().first == e)
                cnt = S.top().second;
            ans += S.top().second;
            S.pop();
        }
        if (!S.empty())
            ans++;
        S.emplace(e, cnt + 1);
    }
    cout << ans << endl;
    return 0;
}
```

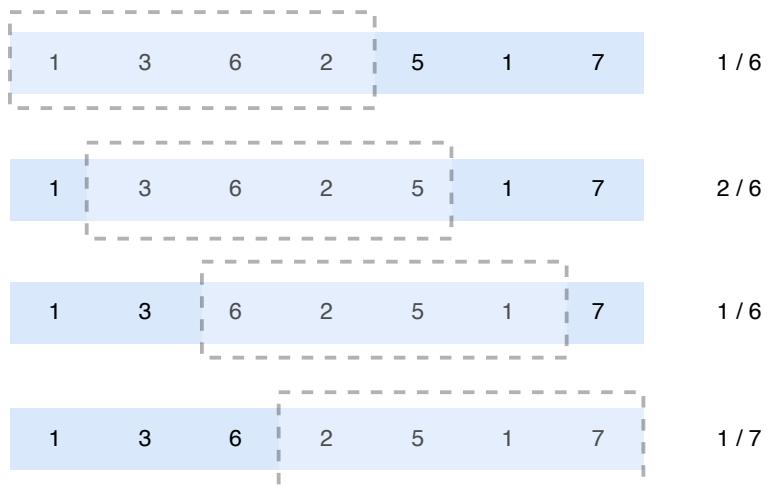
7.2 队列

队列和栈类似，也是一种特殊的线性表。和栈不同的是，**队列只允许在表的一端进行插入操作，而在另一端进行删除操作**。一般来说，进行插入操作的一端称为**队尾**，进行删除操作的一端称为**队头**。队列中没有元素时，称为空队列。

7.2.1 单调队列

“如果一个选手比你小还比你强，你就可以退役了。”——单调队列原理

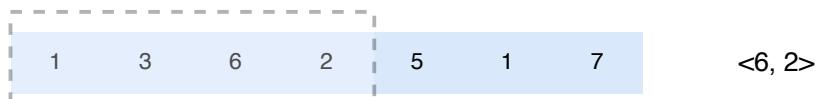
单调队列是一种主要用于解决**滑动窗口类**问题的数据结构，即在长度为 n 的序列中，求每个长度为 m 的区间的区间最值，其时间复杂度是 $O(n)$ ，在这个问题中比 $O(n \log n)$ 的 ST 表和线段树要优。



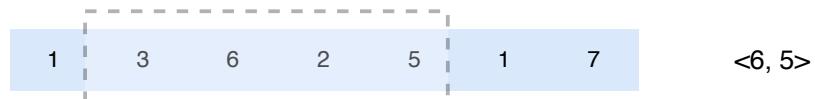
单调队列的基本思想是维护一个**双向队列** (deque)，遍历序列，仅当一个元素**可能**成为某个区间最值时才保留它。

例如，我们把上面的序列可以看成学校里各个年级的选手，数字越大代表能力越强。每个选手只能在大学四年间参赛，毕业了就没有机会了。那么，每一年的王牌选手都在哪个年级呢？

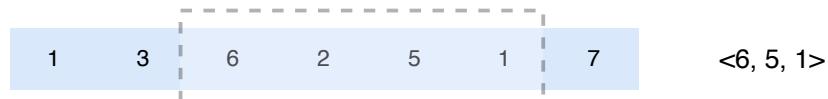
一开始的时候，大三大四的学长都比较菜，大二的最强，而大一的等大二的毕业后还有机会上位，所以队列里有两个数。



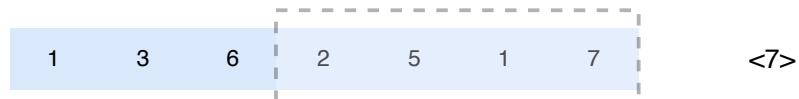
一年过去了，原本大一的成为大二，却发现新进校的新生非常强，自己再也没有机会成为最大值了，所以弹出队列。



又过了一年，新入校的新生尽管能力只有 1，但理论上只要后面的人比他还菜，还是可能成为区间最大值的，所以入队。



终于，原本的王牌毕业了，后面的人以为熬出头了，谁知道这时一个巨佬级别的新生进入了集训队，这下其他所有人都没机会了。



当然这只是比方，现实中各位选手的实力是会增长的，不符合这个模型。

总之，观察就会发现，我们维护的这个队列总是**单调递减的**。如果**维护区间最小值**，那么维护的队列就是**单调递增的**，这就是为什么叫**单调队列**。

参考代码如下：

```
// m为窗口大小, v为序列
deque<int> Q; // 存储编号
```

```

for (int i = 0; i < n; ++i) {
    if (!Q.empty() && i - Q.front() >= m) // 毕业
        Q.pop_front();
    while (!Q.empty() && V[Q.back()] < V[i]) // 比新生弱的当场退役 (求区间最小值把这里改成>即可)
        Q.pop_back();
    Q.push_back(i); // 新生入队
    if (i >= m - 1)
        cout << V[Q.front()] << " ";
}

```

接下来是一个例题：

例 7.2.1 (leetcode-239 滑动窗口最大值). 给你一个整数数组 $nums$, 有一个大小为 k 的滑动窗口从数组的最左侧移动到数组的最右侧。

你只可以看到在滑动窗口内的 k 个数字。滑动窗口每次只向右移动一位。返回滑动窗口中的最大值。

输入格式

共两行：

第一行包含两个整数 n 和 k , 分别代表数组长度和滑动窗口的长度；

第二行包含 n 个整数, 代表数组元素。

```

8 3
1 3 -1 -3 5 3 6 7

```

输出格式

一行, 包含 $n - k + 1$ 个元素, 代表每次滑动窗口中的最大值。

```

3 3 5 5 6 7

```

AC 代码如下：

```

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

```

```
int main() {
    int n, k;
    cin >> n >> k;
    vector<int> nums(n);
    for(int i = 0; i < n; i++) {
        cin >> nums[i];
    }

    deque<int> Q;
    vector<int> result;

    for (int i = 0; i < n; ++i) {
        if (!Q.empty() && i - Q.front() >= k)
            Q.pop_front();
        // 维护队列，保证队列中的元素从大到小排列
        while (!Q.empty() && nums[Q.back()] < nums[i])
            Q.pop_back();
        Q.push_back(i);
        if (i >= k - 1)
            result.push_back(nums[Q.front()]);
    }

    for(int i : result) {
        cout << i << " ";
    }

    return 0;
}
```

7.3 链表

链表是一种数据元素按照**链式存储结构**进行存储的数据结构，这种存储结构具有在物理上存在非连续的特点。

链表由一系列数据结点构成，每个数据结点包括**数据域**和**指针域**两部分。其中，**指针域**保存了数据结构中下一个元素存放的地址。链表结构中数据元素的逻辑顺序是通过**链表中的指针链接次序**来实现的。

7.4 树

树是典型的非线性结构。通常所说的树结构往往是**二叉树**，二叉树有且仅有一个根结点，该结点没有前驱结点。在二叉树结构中的其他结点都有且仅有一个前驱结点，而且可以有两个后继结点。

7.4.1 预备知识

树 (Tree) 可以用多种方式定义，定义树的一种自然的方式是递归的方式。一棵树是一些节点的集合，这个集合可以是空集；如若不是空集，则树由称作**根** (root) 的节点r以及 0 个或多个非空的子树 T_1, T_2, \dots, T_k 组成，这些子树中每棵树的根节点都被来自r的一条边 (edge) 所连结。

7.4.2 线段树

线段树 (Segment Tree) 是算法竞赛最常用的数据结构，主要用于**维护区间信息** (要求满足结合律)。与树状数组相比，它可以实现 $O(\log n)$ 的**区间修改**，还可以同时支持如加、乘等多种操作，更具通用性。

我们根据一道例题来学习线段树：

例 7.4.1 (洛谷 P3372 【模板】线段树 1). 如题，已知一个数列，你需要进行下面两种操作：

1. 将某区间每一个数加上 k

2. 求出某区间每一个数的和

输入格式:

第一行包含两个整数 n, m , 分别表示该数列数字的个数和操作的总个数。

第二行包含 n 个用空格分隔的整数, 其中第 i 个数字表示数列第 i 项的初始值。

接下来 m 行每行包含 3 或 4 个整数, 表示一个操作, 具体如下:

1. 1 x y k : 将区间 $[x, y]$ 内每个数加上 k 。

2. 2 x y : 输出区间 $[x, y]$ 内每个数的和。

```
5 5
1 5 4 2 3
2 2 4
1 2 3 2
2 3 4
1 1 5 1
2 1 4
```

输出格式:

输出包含若干行整数, 即为所有操作 2 的结果。

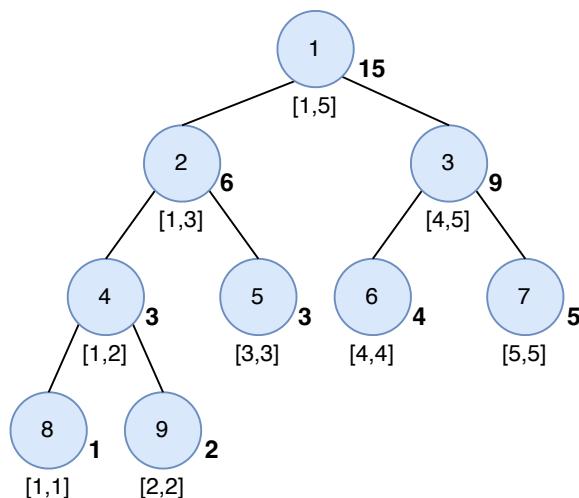
```
11
8
20
```

线段树建立

线段树是一棵**平衡二叉树**, 母结点代表整个区间的和, 越往下区间越小。需要注意的是, 线段树的每个**节点**都对应一条**线段 (区间)**, 但并不保

证所有的线段(区间)都是线段树的节点，这两者应当区分开。

我们来看线段树的构成，每个节点 p 的左右子节点的编号为 $2p$ 和 $2p+1$ ，假如节点 p 存储的是区间 $[a, b]$ 的和，设 $mid = \lfloor \frac{l+r}{2} \rfloor$ ，那么两个子节点分别存储 $[l, mid]$ 与 $[mid, r]$ 的和。我们不难发现，右节点对应的区间长度与左节点对应的区间长度相同或者恰好多 1。下图所示就是 $[1, 2, 3, 4, 5]$ 的线段树模型。



我们通过递归的方法创建线段树：

区间修改

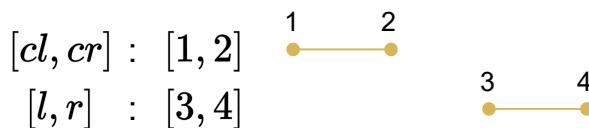
在讲区间修改前，先引入一个“**懒标记**”(或延迟标记)的概念。懒标记是线段树的关键，对于区间修改，朴素的想法是用递归的方式一层层修改，但这样的时间复杂度比较高。使用懒标记后，对于那些正好是线段树节点的区间，我们不继续递归下去，而是打上一个标记，将来要用到它的**子区间**的时候，再向下传递。

```
void update(ll l, ll r, ll d, ll p = 1, ll cl = 1, ll cr = n
) {
    if (cl > r || cr < l) // 区间无交集
        return; // 剪枝
    else if ((cl >= l && cr <= r) // 当前节点对应的区间包含
             在目标区间中
        tree[p] += (cr - cl + 1) * d; // 更新当前区间的值
        if (cr > cl) // 如果不是叶子节点
            mark[p] += d; // 给当前区间打上标记
    }
    else { // 与目标区间有交集，但不包含于其中
        ll mid = (cl + cr) / 2;
        mark[p * 2] += mark[p]; // 标记向下传递
        mark[p * 2 + 1] += mark[p];
        tree[p * 2] += mark[p] * (mid - cl + 1); // 往下更新
        // 一层
        tree[p * 2 + 1] += mark[p] * (cr - mid);
        mark[p] = 0; // 清除标记
        update(l, r, d, p * 2, cl, mid); // 递归地往下寻找
        update(l, r, d, p * 2 + 1, mid + 1, cr);
        tree[p] = tree[p * 2] + tree[p * 2 + 1]; // 根据子节
        点更新当前节点的值
    }
}
```

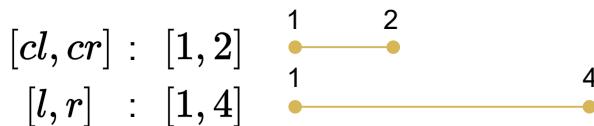
我们来解释这个过程：

更新时，我们是从最大的区间开始，递归向下处理。注意到，**任何区间都是线段树上某些节点的并集**，于是我们记目标区间为 $[l, r]$ ，当前区间为 $[cl, cr]$ ，当前节点为 p ，我们会遇到三种情况：

1. 当前区间与目标区间没有交集，则直接结束递归；



2. 当前区间被包括在目标区间里：



这时可以更新当前区间，乘上区间长度即可：

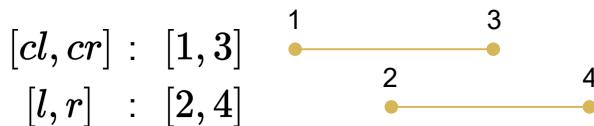
```
tree[p] += (cr - cl + 1) * d;
```

然后打上懒标记，注意，叶子节点可以不打标记，因为不会再向下传递了：

```
mark[p] += d;
```

这个标记表示“该区间上每一个点都要加上 d ”，因为原来可能存在标记，所以是 $+=$ 而不是 $=$ 。

3. 当前区间与目标区间相交，但不包含于其中：



这时需要把当前区间一分为二进行处理，如果存在懒标记，要先把懒标记传递给子节点，注意这里也是 $+=$ ，因为原来可能存在懒标记：

```
ll mid = (cl + cr) / 2;
mark[p * 2] += mark[p];
mark[p * 2 + 1] += mark[p];
```

两个子节点的值也就需要相应的更新，即乘以区间长度：

```
tree[p * 2] += mark[p] * (mid - cl + 1);
tree[p * 2 + 1] += mark[p] * (cr - mid);
```

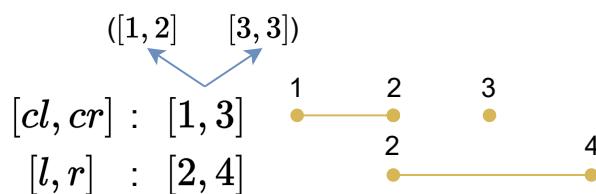
同时清除该节点的懒节点标记：

```
mark[p] = 0;
```

这个过程并不是递归的，我们只往下传递一层，以后要用再才继续传递。我们常常把这个传递过程封装成一个函数：

```
inline void push_down(ll p, ll len) {
    mark[p * 2] += mark[p];
    mark[p * 2 + 1] += mark[p];
    tree[p * 2] += mark[p] * (len - len / 2);
    tree[p * 2 + 1] += mark[p] * (len / 2); // 右边的区间可能要短一点
    mark[p] = 0;
}
```

然后在update函数中调用：push_down(p, cr - cl + 1);在传递完标记之后，再递归去处理左右两个子节点：



区间查询

区间查询的方法与区间修改完全相似：

```
ll query(ll l, ll r, ll p = 1, ll cl = 1, ll cr = n) {
    if (cl > r || cr < l)
        return 0;
    else if (cl >= l && cr <= r)
        return tree[p];
    else {
        ll mid = (cl + cr) / 2;
        push_down(p, cr - cl + 1);
        return query(l, r, p * 2, cl, mid) + query(l, r, p *
            2 + 1, mid + 1, cr);
        // 上一行拆成三行写就和区间修改格式一致了
    }
}
```

最后给出模板题的解：

```
#include <bits/stdc++.h>
#define MAXN 100005
using namespace std;
typedef long long ll;
inline ll read() {
    ll ans = 0;
    char c = getchar();
    while (!isdigit(c))
        c = getchar();
    while (isdigit(c)) {
        ans = ans * 10 + c - '0';
        c = getchar();
    }
    return ans;
}
```

```
ll n, m, A[MAXN], tree[MAXN * 4], mark[MAXN * 4]; // 经验表  
明开四倍空间不会越界  
  
inline void push_down(ll p, ll len) {  
    mark[p * 2] += mark[p];  
    mark[p * 2 + 1] += mark[p];  
    tree[p * 2] += mark[p] * (len - len / 2);  
    tree[p * 2 + 1] += mark[p] * (len / 2);  
    mark[p] = 0;  
}  
  
void build(ll l = 1, ll r = n, ll p = 1) {  
    if (l == r)  
        tree[p] = A[l];  
    else {  
        ll mid = (l + r) / 2;  
        build(l, mid, p * 2);  
        build(mid + 1, r, p * 2 + 1);  
        tree[p] = tree[p * 2] + tree[p * 2 + 1];  
    }  
}  
  
void update(ll l, ll r, ll d, ll p = 1, ll cl = 1, ll cr = n  
) {  
    if (cl > r || cr < l)  
        return;  
    else if (cl >= l && cr <= r) {  
        tree[p] += (cr - cl + 1) * d;  
        if (cr > cl)  
            mark[p] += d;  
    }  
    else {  
        ll mid = (cl + cr) / 2;  
        push_down(p, cr - cl + 1);  
        update(l, r, d, p * 2, cl, mid);  
        update(l, r, d, p * 2 + 1, mid + 1, cr);  
    }  
}
```

```
        tree[p] = tree[p * 2] + tree[p * 2 + 1];
    }
}

ll query(ll l, ll r, ll p = 1, ll cl = 1, ll cr = n) {
    if (cl > r || cr < l)
        return 0;
    else if (cl >= l && cr <= r)
        return tree[p];
    else {
        ll mid = (cl + cr) / 2;
        push_down(p, cr - cl + 1);
        return query(l, r, p * 2, cl, mid) + query(l, r, p * 2 + 1, mid + 1, cr);
    }
}

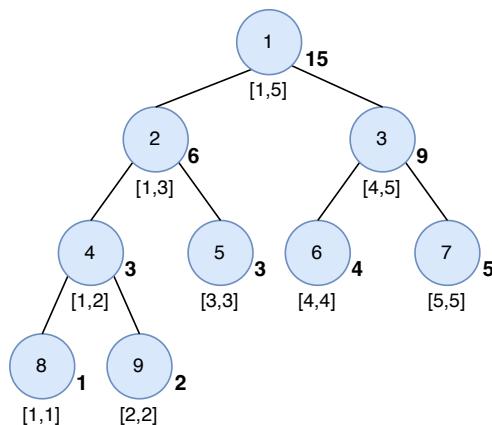
int main() {
    n = read();
    m = read();
    for (int i = 1; i <= n; ++i)
        A[i] = read();
    build();
    for (int i = 0; i < m; ++i) {
        ll opr = read(), l = read(), r = read();
        if (opr == 1) {
            ll d = read();
            update(l, r, d);
        }
        else
            printf("%lld\n", query(l, r));
    }
    return 0;
}
```

实际上线段树还可以维护区间最值、区间 gcd 等，操作除了区间加也可以是区间乘、区间赋值，需要深入了解线段树原理。

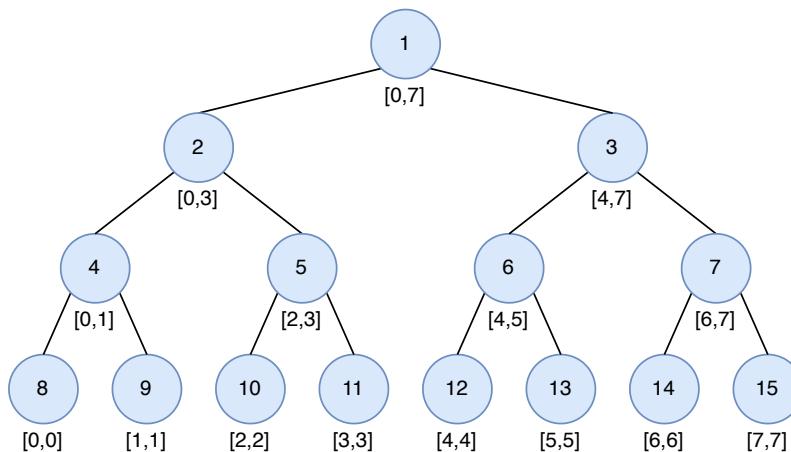
7.4.3 zkw 线段树

非递归线段树，因为张昆玮在《统计的力量》中介绍了这种数据结构，常常被称为 **zkw 线段树**，是一种代码较短、常数较小的线段树写法。

普通的线段树，是从上到下处理的，因为对于普通线段树，我们很容易定位根节点，却不容易定位叶子节点。



但存在一种特殊情况，线段树的叶子节点很好定位，那就是这棵二叉树是满二叉树时：



根据这棵树，我们发现它的叶子节点的编号很有规律，假设一共有 n 个叶子节点，若某叶子节点所代表的退化线段为 $[x, x]$ ，则其编号 $p = x + n$ 。不仅如此，这棵树还有大量很好的性质：

- 树一共有 $2n - 1$ 个节点；
- 一共有 $H = \log_2 n + 1$ 层，第 h 层有 2^{h-1} 个节点，且该层线段长度为 2^{H-h} ；
- 若某节点的编号为 p ，则其父节点的编号为 $\lfloor \frac{p}{2} \rfloor$ ，子节点的编号为 $2p$ 和 $2p + 1$ ；
- 若两节点编号分别为 p 和 q ，则它们是兄弟节点等价于 $p \wedge q = 1$ ；
- 除根结点外，编号为偶数的节点都是某个节点的左节点，编号为奇数的节点都是某个节点的右节点。

要构建出这样一棵满二叉树，需要让 n 是 2 的整数幂。如果 n 不是 2 的整数幂则需要往后补 0 使它成为 2 的整数幂即可。

我们可以写出一个建树的方法：

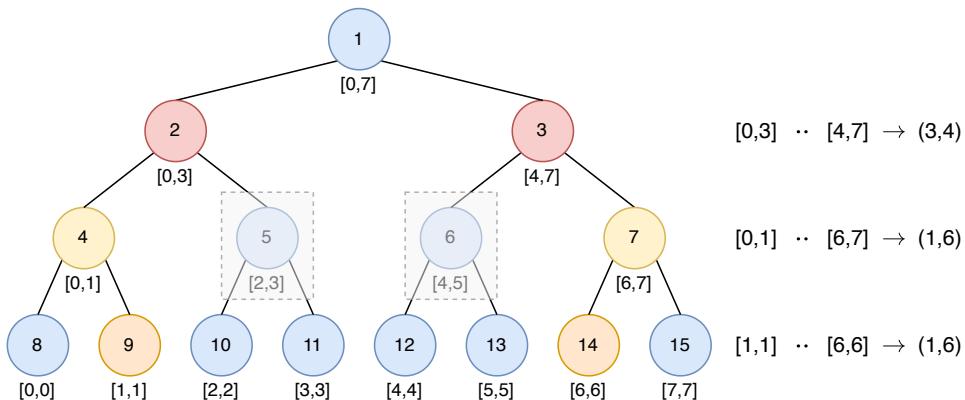
```
int N = 1 << __lg(n + 5) + 1;
vector<int> tree(N << 1);
for (int i = 1; i <= n ++i)
    tree[i + N] = A[i];
for (int i = N; i; --i)
    tree[i] = tree[i << 1] + tree[i << 1 | 1];
```

以及单点修改函数：

```
void update(int x, int d) {
    for (int i = x + N; i; i >= 1)
        tree[i] += d;
}
```

区间查询

在**区间查询**时，我们把查询 $[l, r]$ 这个闭区间的信息换成查询 $(l-1, r+1)$ 这个开区间的信息。然后从下往上，一层一层地缩小区间范围。



观察上图我们可以发现，不断上移的过程，就是不断把端点及其兄弟排除出查询区间的过程。而如果查询区间的**左端点**是父节点的**左儿子/右端点**是父节点的**右儿子**，说明它的兄弟应当是**答案的一部分**。最后在查询的区间为空区间，即两端点为兄弟节点时退出循环。

```

int query(int l, int r) {
    int ans = 0;
    for (l += N - 1, r += N + 1; l ^ r ^ 1; l >>= 1, r >>= 1) {
        if (~l & 1) ans += tree[l ^ 1]; // 左端点是左儿子
        if (r & 1) ans += tree[r ^ 1]; // 右端点是右儿子
    }
    return ans;
}

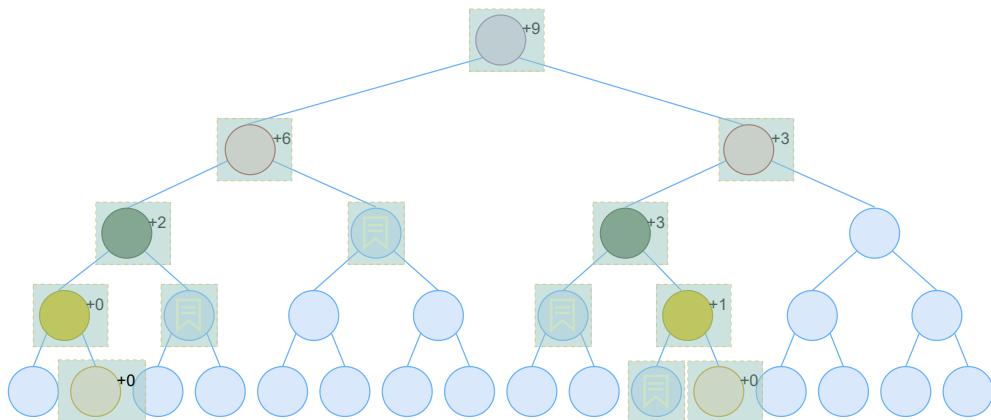
```

需要注意的是，因为我们把闭区间查询改成了开区间查询，如果原数据是 $0 - \text{index}$ 的话就会出现 -1 这样的端点，为了避免这种情况，建议原容器用 $1 - \text{index}$ 存放并稍微开大一点。

区间修改

区间修改可能稍微麻烦一点，考虑到在非递归的情况下，标记下传是比较困难的，所以我们不下传标记，而是将**标记永久化**。

具体地和上文中区间查询类似，当左端点是左儿子/右端点是右儿子时，我们对它的**兄弟**进行修改并**打上标记**，表示这棵子树中每个节点都要修改。但单纯这样是不够的，因为上述修改还会波及到这些节点的**各级祖先**。



所以我们需要在途中根据**实际修改的区间长度**来更新各级祖先的值，而且这种操作需要一路上推到根节点。

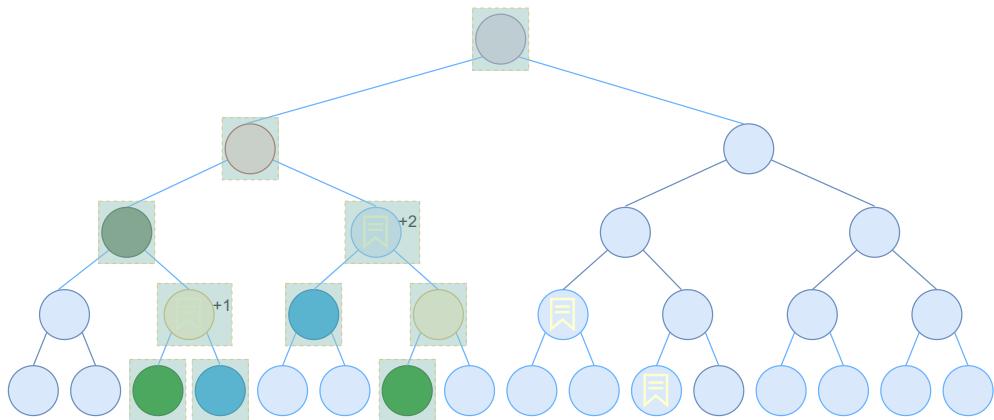
```

void update(int l, int r, int d) {
    int len = 1, cntl = 0, cntr = 0; // cntl、cntr是左、右两边分别实际修改的区间长度
    for (l += N - 1, r += N + 1; l ^ r ^ 1; l >>= 1, r >>=
        1, len <<= 1) {
        tree[l] += cntl * d, tree[r] += cntr * d;
        if (~l & 1) tree[l ^ 1] += d * len, mark[l ^ 1] += d,
            cntl += len;
        if (r & 1) tree[r ^ 1] += d * len, mark[r ^ 1] += d,
            cntr += len;
    }
    for (; l; l >>= 1, r >>= 1)
}

```

```
    tree[l] += cntl * d, tree[r] += cntr * d;
}
```

在有区间修改存在时，区间查询也需要考虑标记的影响。所以除了加上端点的兄弟节点的信息，沿途中遇到的标记也对答案有相应的贡献，依赖于 **实际查询的区间长度**，这同样也需要上推到根节点。



```
int query(int l, int r) {
    int ans = 0, len = 1, cntl = 0, cntr = 0;
    for (l += N - 1, r += N + 1; l ^ r ^ 1; l >= 1, r >=
        1, len <= 1) {
        ans += cntl * mark[l] + cntr * mark[r];
        if (~l & 1) ans += tree[l ^ 1], cntl += len;
        if (r & 1) ans += tree[r ^ 1], cntr += len;
    }
    for (; l; l >= 1, r >= 1)
        ans += cntl * mark[l] + cntr * mark[r];
    return ans;
}
```

同样的，`zkw`线段树也有着许多其他功能，在本节最后给出**区间加和区间查找最大值**的实现：

区间加

```

void update(int l, int r, int d) {
    for (l += N - 1, r += N + 1; l ^ r ^ 1; l >>= 1, r >>= 1) {
        if (l < N) tree[l] = max(tree[l << 1], tree[l << 1 | 1]) + mark[l],
            tree[r] = max(tree[r << 1], tree[r << 1 | 1]) + mark[r];
        if (~l & 1) tree[l ^ 1] += d, mark[l ^ 1] += d;
        if (r & 1) tree[r ^ 1] += d, mark[r ^ 1] += d;
    }
    for (; l; l >>= 1, r >>= 1)
        if (l < N) tree[l] = max(tree[l << 1], tree[l << 1 | 1]) + mark[l],
            tree[r] = max(tree[r << 1], tree[r << 1 | 1]) + mark[r];
}

```

区间查找最大值

```

int query(int l, int r) {
    int maxl = -INF, maxr = -INF;
    for (l += N - 1, r += N + 1; l ^ r ^ 1; l >>= 1, r >>= 1) {
        maxl += mark[l], maxr += mark[r];
        if (~l & 1) cmax(maxl, tree[l ^ 1]);
        if (r & 1) cmax(maxr, tree[r ^ 1]);
    }
    for (; l; l >>= 1, r >>= 1)
        maxl += mark[l], maxr += mark[r];
    return max(maxl, maxr);
}

```

7.4.4 动态开点线段树

通常来说，线段树占用空间是总区间长 n 的常数倍，空间复杂度是 $O(n)$ 。然而，有时候 n 很巨大，而我们又不需要使用所有的节点，这时便可以**动态开点**，即不再一次性建好树，而是一边修改、查询一边建立。我们不再用 $p * 2$ 和 $p * 2 + 1$ 代表左右儿子，而是用 ls 和 rs 记录左右儿子的编号。设总查询次数为 m ，则这样的总空间复杂度为 $O(m \log n)$ 。

比起普通线段树，动态开点线段树有一个优势：它能够处理零或负数位置。此时求 mid 不能再用 $(\text{cl} + \text{cr}) / 2$ ，而要用 $(\text{cl} + \text{cr} - 1) / 2$ 。

因为缓存命中等原因，动态开点线段树写成结构体形式速度往往更快一些。不过 $\text{tree}[\text{tree}[p].\text{ls}].\text{val}$ 之类的写法怎么看都很反人类，所以这里通常会用宏简化一下：

```
// MAXV一般能开多大开多大，例如内存限制128M时可以开到八百万
左右

#define ls(x) tree[x].ls
#define rs(x) tree[x].rs
#define val(x) tree[x].val
#define mark(x) tree[x].mark
const int MAXV = 8e6;
int L = 1, R = 1e5, cnt = 1;
struct node {
    ll val, mark;
    int ls, rs;
} tree[MAXV];
void push_down(int p, int len) {
    if (len <= 1) return;
    if (!ls(p)) ls(p) = ++cnt;
    if (!rs(p)) rs(p) = ++cnt;
    val(ls(p)) += mark(p) * (len / 2);
    mark(ls(p)) += mark(p);
    val(rs(p)) += mark(p) * (len - len / 2);
```

```

    mark(rs(p)) += mark(p);
    mark(p) = 0;
}

ll query(int l, int r, int p = 1, int cl = L, int cr = R) {
    if (cl >= l && cr <= r) return val(p);
    push_down(p, cr - cl + 1);
    ll mid = (cl + cr - 1) / 2, ans = 0;
    if (mid >= l) ans += query(l, r, ls(p), cl, mid);
    if (mid < r) ans += query(l, r, rs(p), mid + 1, cr);
    return ans;
}

void update(int l, int r, int d, int p = 1, int cl = L, int
cr = R) {
    if (cl >= l && cr <= r) return val(p) += d * (cr - cl +
1), mark(p) += d, void();
    push_down(p, cr - cl + 1);
    int mid = (cl + cr - 1) / 2;
    if (mid >= l) update(l, r, d, ls(p), cl, mid);
    if (mid < r) update(l, r, d, rs(p), mid + 1, cr);
    val(p) = val(ls(p)) + val(rs(p));
}

```

可以看到，除了在push_down中进行了新节点的创建，其他基本和普通线段树一致。动态开点线段树不需要build，通常用在没有提供初始数据的场合（例如初始全0），这时更能显示出优势。

当然，除了动态开点，其实先离散化再建树也常常能达到效果。但动态开点写起来更简单直观，而且在强制在线时只能这样做。

7.4.5 权值线段树

桶我们经常使用，例如计数排序时用`cnt[]`数组记录每个数出现的次数。**权值线段树**就是用线段树维护一个桶，它可以 $O(\log v)$ (v 为值域) 地查

询某个范围内的数出现的总次数。不仅如此，它还可以 $O(\log v)$ 地求得第 k 大的数。事实上，它常常可以代替平衡树使用。

权值线段树需要按值域开空间，当值域过大时需要离散化或动态开点。

下面是一个动态开点的权值线段树的例子：

```
void insert(int v) { // 插入
    update(v, 1);
}
void remove(int v) { // 删除
    update(v, -1);
}
int countl(int v) {
    return query(L, v - 1);
}
int countg(int v) {
    return query(v + 1, R);
}
int rank(int v) { // 求排名
    return countl(v) + 1;
}
int kth(int k, int p = 1, int cl = L, int cr = R) { // 求指定排名的数
    if (cl == cr)
        return cl;
    int mid = (cl + cr - 1) / 2;
    if (val(ls(p)) >= k)
        return kth(k, ls(p), cl, mid); // 往左搜
    else
        return kth(k - val(ls(p)), rs(p), mid + 1, cr); // 往右搜
}
int pre(int v) { // 求前驱
    int r = countl(v);
```

```

    return kth(r);
}

int suc(int v) { // 求后继
    int r = val(1) - countg(v) + 1;
    return kth(r);
}

```

代替平衡树使用时，权值线段树代码比较短，但是当值域较大、询问较多时空间占用会比较大。

7.4.6 可持久化线段树

大家应该都知道 Word 有撤销功能，这个功能可以帮助我们返回之前的某个状态，并在此基础上进行修改。实现这样的功能，便需要**可持久化**的数据结构，它保存每个历史版本，可以高效地查询和修改历史版本，且因其维持原结构的**不变性**还在函数式编程领域有很大作用。

可持久化线段树就是这样一种数据结构。想要让线段树可持久化，最朴素的实现方法是每进行一次操作都建一棵新的树，但显然这样做的时间和空间复杂度都是不可接受的。稍微思考一下会发现，每次修改操作都只会有少数的点被修改，所以大量的点是可以共用的。

要实现这一点，需要我们用**动态开点**的方法存储每个点的左右子节点。不过，对于最初版本的那棵树，我们应当一次性建好，而不必一个一个插入：

```

void build(int l = 1, int r = n, int p = 1) {
    if (l == r)
        val(p) = A[l];
    else {
        ls(p) = ++cnt, rs(p) = ++cnt; // 新建节点
        int mid = (l + r) / 2;
        build(l, mid, ls(p));
        build(mid + 1, r, rs(p));
    }
}

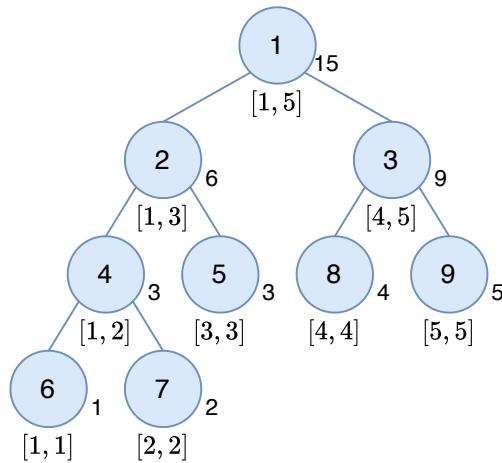
```

```

        build(mid + 1, r, rs(p));
        val(p) = val(ls(p)) + val(rs(p));
    }
}

```

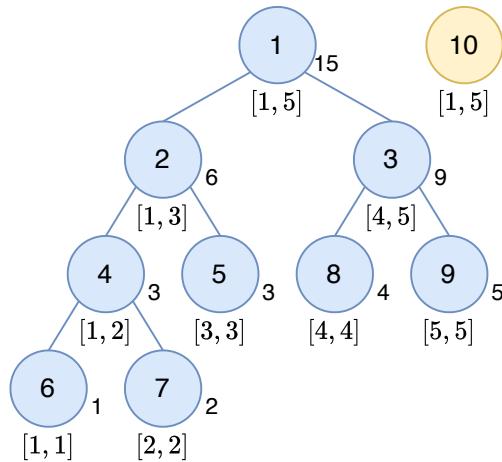
假设原始数据为 [1, 2, 3, 4, 5]，现在我们有了形如这样的一棵线段树：



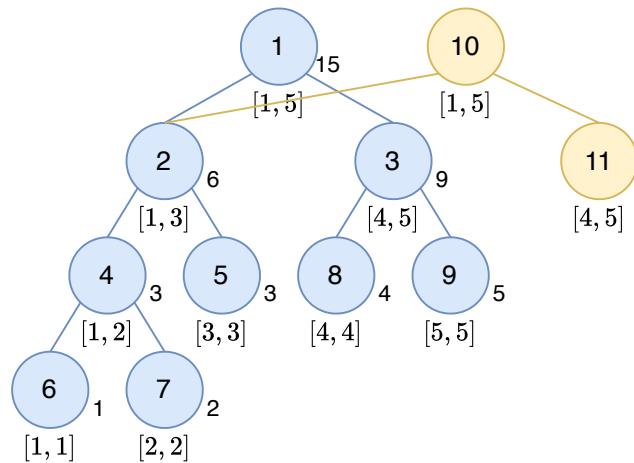
我们维持这棵树不变，添加额外的节点来代替修改操作。

单点修改

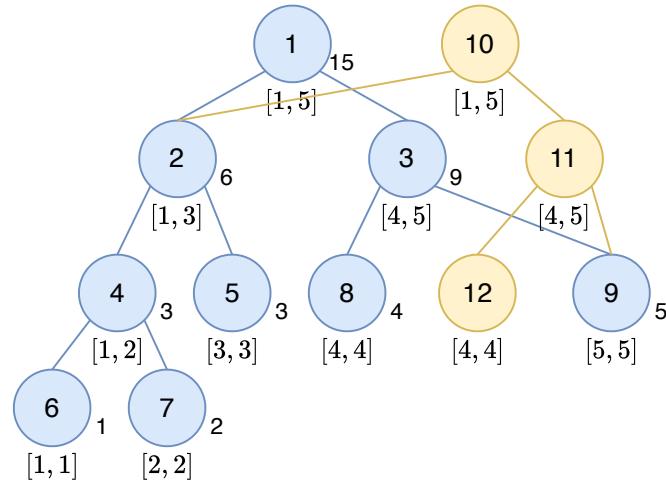
假如我们现在要令 A_4 加 2，我们先创建一个新的根节点：



要修改的 A_4 对应的叶子节点在右子树上。所以，这个新节点的左儿子应该与原节点的左儿子相同，即与之共用左子树。现在我们再创建一个新的节点作为右儿子：

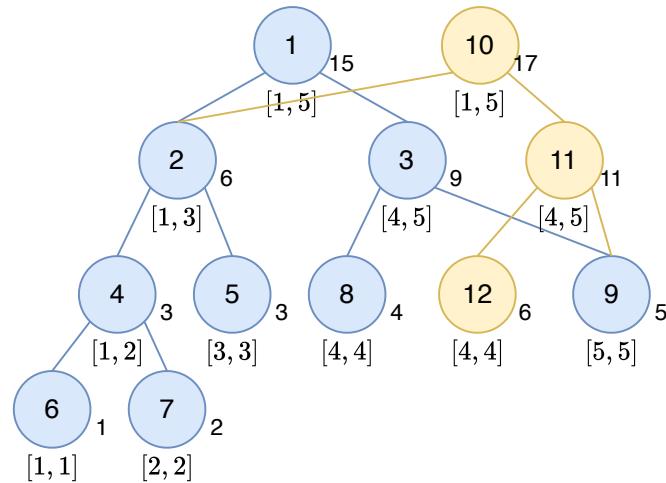


接下来处理这个新节点，显然这将是一个递归的过程。我们容易发现 A_4 应该在左子树，所以相似地，沿用原树的右儿子，然后创建一个新的节点作为左儿子：



接下来给各个节点赋值，和普通的线段树类似，叶子节点直接赋值，其它节点则利用子节点计算值。

这样我们就完成了**单点修改**，事实上我们没有修改任何东西，我们保留了原来的版本，并新增了一个修改后的版本。



```
// 令A_x加d, p表示原版本的节点, q表示新版本的节点
void update(int x, int d, int p, int q, int cl = 1, int cr =
n) {
    if (cl == cr)
        val(q) = val(p) + d; // 给叶子节点赋值
    else {
        ls(q) = ls(p), rs(q) = rs(p); // 复制节点
        int mid = (cl + cr) / 2;
        if (x <= mid)
            ls(q) = ++cnt, update(x, d, ls(p), ls(q), cl,
mid); // 创建新节点作为左儿子, 然后往左递归
        else
            rs(q) = ++cnt, update(x, d, rs(p), rs(q), mid +
1, cr); // 创建新节点作为右儿子, 然后往右递归
        val(q) = val(ls(q)) + val(rs(q)); // 根据子节点给当
前节点赋值
    }
}
```

查询操作和普通的查询操作是一样的，只不过需要指定从哪个根节点开始查询。

例 7.4.2 (洛谷-P3919 【模板】可持久化线段树 1 (可持久化数组)). 如题，你需要维护这样的一个长度为 N 的数组，支持如下几种操作

1. 在某个历史版本上修改某一个位置上的值
2. 访问某个历史版本上的某一位置的值

此外，每进行一次操作 (对于操作 2，即为生成一个完全一样的版本，不作任何改动 **)，就会生成一个新的版本。版本编号即为当前操作的编号 (从 1 开始编号，版本 0 表示初始状态数组)

数据规模：

对于 30% 的数据： $1 \leq N, M \leq 10^3$

对于 50% 的数据： $1 \leq N, M \leq 10^4$

对于 70% 的数据： $1 \leq N, M \leq 10^5$

对于 100% 的数据： $1 \leq N, M \leq 10^6, 1 \leq loc_i \leq N, 0 \leq v_i < i, -10^9 \leq a_i, value_i \leq 10^9$

经测试，正常常数的可持久化数组可以通过，请各位放心

~~数据略微凶残，请注意常数不要过大~~

~~另，此题 I/O 量较大，如果实在 TLE 请注意 I/O 优化~~

~~询问生成的版本是指你访问的那个版本的复制~~

输入格式：

输入的第一行包含两个正整数 N, M ，分别表示数组的长度和操作的个数。

第二行包含 N 个整数，依次为初始状态下数组各位的值 (依次为 a_i ， $1 \leq i \leq N$)。

接下来 M 行每行包含 3 或 4 个整数，代表两种操作之一 (i 为基于的历史版本号)：

1. 对于操作 1, 格式为 $v_i \ 1 \ loc_i \ value_i$, 即为在版本 v_i 的基础上, 将 a_{loc_i} 修改为 $value_i$
2. 对于操作 2, 格式为 $v_i \ 2 \ loc_i$, 即访问版本 v_i 中的 a_{loc_i} 的值, 生成一样版本的对象应为 v_i

```
5 10
59 46 14 87 41
0 2 1
0 1 1 14
0 1 1 57
0 1 1 88
4 2 4
0 2 5
0 2 4
4 2 1
2 2 2
1 1 5 91
```

输出格式:

输出包含若干行, 依次为每个操作 2 的结果。

```
59
87
41
87
88
46
```

样例说明:

一共 11 个版本, 编号从 0-10, 依次为:

0 : 59 46 14 87 41
1 : 59 46 14 87 41
2 : 14 46 14 87 41

```
3 : 57 46 14 87 41  
4 : 88 46 14 87 41  
5 : 88 46 14 87 41  
6 : 59 46 14 87 41  
7 : 59 46 14 87 41  
8 : 88 46 14 87 41  
9 : 14 46 14 87 41  
10 : 59 46 14 87 91
```

其实题7.4.2只需要单点修改单点查询，根本用不到和信息，用可持久化线段树有点大材小用的味道，不过确实可以用它求解。完整代码如下：

```
#include <bits/stdc++.h>  
using namespace std;  
  
int read() {  
    int ans = 0, sgn = 1;  
    char c = getchar();  
    while (!isdigit(c)) {  
        if (c == '-')  
            sgn *= -1;  
        c = getchar();  
    }  
    while (isdigit(c)) {  
        ans = ans * 10 + c - '0';  
        c = getchar();  
    }  
    return ans * sgn;  
}  
  
const int MAXV = 20000000, MAXN = 1000005;  
#define ls(x) tree[x].ls  
#define rs(x) tree[x].rs  
#define val(x) tree[x].val  
#define mark(x) tree[x].mark
```

```
struct node {
    int val, ls, rs;
} tree[MAXV];
int A[MAXN], roots[MAXN], n, m, cnt = 1; // roots记录每个历史版本的根节点
void build(int l = 1, int r = n, int p = 1) {
    if (l == r)
        val(p) = A[l];
    else {
        ls(p) = ++cnt, rs(p) = ++cnt;
        int mid = (l + r) / 2;
        build(l, mid, ls(p));
        build(mid + 1, r, rs(p));
        val(p) = val(ls(p)) + val(rs(p));
    }
}
void update(int x, int d, int p, int q, int cl = 1, int cr = n) { // 单点修改
    if (cl == cr)
        val(q) = d;
    else {
        ls(q) = ls(p), rs(q) = rs(p);
        int mid = (cl + cr) / 2;
        if (x <= mid)
            ls(q) = ++cnt, update(x, d, ls(p), ls(q), cl,
                                   mid);
        else
            rs(q) = ++cnt, update(x, d, rs(p), rs(q), mid +
                                   1, cr);
        val(q) = val(ls(q)) + val(rs(q));
    }
}
int query(int l, int r, int p, int cl = 1, int cr = n) { //
```

```
区间查询

if (cl > r || cr < l)
    return 0;
else if (cl >= l && cr <= r)
    return val(p);
else {
    int mid = (cl + cr) / 2;
    return query(l, r, ls(p), cl, mid) + query(l, r, rs(
        p), mid + 1, cr);
}
}

int main() {
    n = read(), m = read();
    for (int i = 1; i <= n; ++i)
        A[i] = read();
    build();
    roots[0] = 1;
    for (int t = 1; t <= m; ++t) {
        int v = read(), o = read();
        if (o == 1) {
            int x = read(), d = read();
            roots[t] = ++cnt; // 新建节点
            update(x, d, roots[v], roots[t]);
        }
        else {
            int x = read();
            roots[t] = roots[v]; // 复用v号版本
            printf("%d\n", query(x, x, roots[v]));
        }
    }
    return 0;
}
```

区间修改

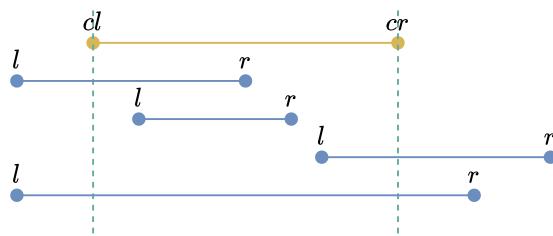
如果涉及到区间修改，会稍麻烦一些。为了不占用过多空间，我们常常使用一种叫**标记永久化**的技术。我们不再向下传递标记，相反，我们在查询时带着标记传递。

```

ll query(int l, int r, int p, int cl = 1, int cr = n, ll mk
= 0) { // 区间查询
    if (cl > r || cr < l)
        return 0;
    else if (cl >= l && cr <= r)
        return val(p) + mk * (cr - cl + 1); // 加上带的标记
    else {
        int mid = (cl + cr) / 2;
        return query(l, r, ls(p), cl, mid, mk + mark(p)) +
            query(l, r, rs(p), mid + 1, cr, mk + mark(p)); // 带着标记传递
    }
}

```

`update`函数中，因为没有传递标记，所以靠子节点给当前节点赋值可能会出现错误，我们换一种方法计算。我们考虑每次更新状态对区间 $[cl, cr]$ 的影响。



如图可以发现，不管 $[cl, cr]$ 与 $[l, r]$ 是怎样的位置关系， $[cl, cr]$ 中需要更新的区间长度都是同样的 $[\max(cl, l), \min(cr, r)]$ 。下面是区间修改和查询的实现：

```

void update(int l, int r, int d, int p, int q, int cl = 1,
int cr = n) { // 区间修改
    ls(q) = ls(p), rs(q) = rs(p), mark(q) = mark(p); // 复制
    节点
    if (cl >= l && cr <= r) {
        if (cr > cl)
            mark(q) += d;
    }
    else {
        int mid = (cl + cr) / 2;
        if (cl <= r && mid >= l) // 提前进行判断，以免新建不
        必要的节点
            ls(q) = ++cnt, update(l, r, d, ls(p), ls(q), cl,
            mid);
        if (mid + 1 <= r && cr >= l)
            rs(q) = ++cnt, update(l, r, d, rs(p), rs(q), mid
            + 1, cr);
    }
    val(q) = val(p) + (min(cr, r) - max(cl, l) + 1) * d; // 根据需要更新的区间长度计算当前节点的值
}

```

接下来简单介绍一种利用可持久化线段树求静态区间第 k 小数的方法，先建立一棵全 0 的树：

```

void build(int l = 1, int r = n, int p = 1) { // 建树
    val(p) = 0;
    if (l != r) {
        ls(p) = ++cnt, rs(p) = ++cnt;
        int mid = (l + r) / 2;
        build(l, mid, ls(p));
        build(mid + 1, r, rs(p));
    }
}

```

```
}
```

按原区间中的顺序，把离散化后的数据一个一个地插入可持久化的**权值线段树**：

```

for (int i = 0; i < n; ++i) {
    roots[i + 1] = ++cnt;
    update(L[i], 1, roots[i], roots[i + 1]); // L是离散化得到的数组
}
```

由于我们已经有了求 $[1, r]$ 内第 k 小数的方法，也就是查询相应历史版本，而求 $[l, r]$ 内第 k 小数，只需对 `kth` 函数稍作修改。注意，我们在求 `kth` 时会用到当前左儿子对应的区间和，现在要排除 $[1, l - 1]$ ，那么把这部分和减去即可：

```

int kth(int k, int p, int q, int cl = 1, int cr = n) { // 求指定排名的数
    if (cl == cr)
        return ori[cl];
    int mid = (cl + cr) / 2;
    if (val(ls(q)) - val(ls(p)) >= k)
        return kth(k, ls(p), ls(q), cl, mid); // 往左搜
    else
        return kth(k - (val(ls(q)) - val(ls(p))), rs(p), rs(q), mid + 1, cr); // 往右搜
}
```

查找时只需 `kth(k, roots[l - 1], roots[r])` 便可以 $O(\log n)$ 地求得静态区间第 k 大数。

7.4.7 珂朵莉树

珂朵莉树起源于 CodeForces 上的一道题，题目要求我们实现一种数据结构，可以较快地实现：

- 区间加
- 区间赋值
- 求区间第 k 大值
- 求区间 n 次方和

例 7.4.3 (CF896C Willem, Chtholly and Seniorious). *Seniorious is made by linking special talismans in particular order. After over 500 years, the carillon is now in bad condition, so Willem decides to examine it thoroughly.*

Seniorious has n pieces of talisman. Willem puts them in a line, the i -th of which is an integer a_i .

In order to maintain it, Willem needs to perform m operations. There are four types of operations:

- 1 l r x : *For each i such that $l \leq i \leq r$, assign $a_i + x$ to a_i .*
- 2 l r x : *For each i such that $l \leq i \leq r$, assign x to a_i .*
- 3 l r x : *Print the x -th smallest number in the index range $[l, r]$, i.e. the element at the x -th position if all the elements a_i such that $l \leq i \leq r$ are taken and sorted into an array of non-decreasing integers. It's guaranteed that $1 \leq x \leq r - l + 1$.*
- 4 l r x y : *Print the sum of the x -th power of a_i such that $l \leq i \leq r$, modulo y , i.e. $(\sum_{i=l}^r a_i^x) \bmod y$.*

输入格式:

The only line contains four integers $n, m, seed, v_{max}$ ($1 \leq n, m \leq 10^5, 0 \leq seed \leq 10^9 + 7, 1 \leq v_{max} \leq 10^9$).

The initial values and operations are generated using following pseudo code:

```
def rnd():

    ret = seed
    seed = (seed * 7 + 13) % 1000000007
    return ret

for i = 1 to n:

    a[i] = (rnd() % vmax) + 1

for i = 1 to m:

    op = (rnd() % 4) + 1
    l = (rnd() % n) + 1
    r = (rnd() % n) + 1

    if (l > r):
        swap(l, r)

    if (op == 3):
        x = (rnd() % (r - l + 1)) + 1
    else:
        x = (rnd() % vmax) + 1

    if (op == 4):
        y = (rnd() % vmax) + 1
```

Here op is the type of the operation mentioned in the legend.

10 10 7 9

输出格式:

For each operation of types 3 or 4, output a line containing the answer.

2 1 0 3

Note:

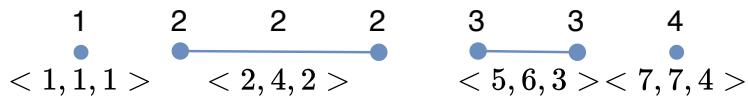
In example, the initial array is $\{8, 9, 7, 2, 3, 1, 5, 6, 4, 8\}$.

The operations are:

- 2 6 7 9
- 1 3 10 8
- 4 4 6 2 4
- 1 4 5 8
- 2 1 7 1
- 4 7 9 4 4
- 1 2 7 9
- 4 5 8 1 1
- 2 5 7 5
- 4 3 10 8 5

珂朵莉树的适用范围是有**区间赋值**操作且**数据随机**的题目。其实珂朵莉树看上去并不像是树状数据结构，但因为一般要用到`std::set`，而`std::set`是用红黑树实现的，所以也不算名不副实。在随机数据下，珂朵莉树可以达到 $O(n \log(\log n))$ 的复杂度。

珂朵莉树的思想在于随机数据下的区间赋值操作很可能让大量元素变为同一个数。所以我们以三元组 $\langle l, r, v \rangle$ 的形式保存数据（区间 $[l, r]$ 中的元素的值都是 v ）：



```

struct node {
    ll l, r;
    mutable ll v; // 这里 mutable 要写不然可能会 CE
    node(ll l, ll r, ll v) : l(l), r(r), v(v) {} // 构造函数
    bool operator<(const node &o) const { return l < o.l; }
    // 重载小于运算符
};
  
```

考虑到`set`容器进行插入、删除和查询操作的时间复杂度都是 $O(\log n)$ ，这里选用`set`容器，然后把这些三元组存储到`set`里：

```
set<node> tree;
```

要把结构体放进`set`里需要重载小于运算符，`set`会保证内部元素有序。另一方面，`mutable`使得当整个结构体为`const`时，因为可能有区间加等操作，标为`mutable`的成员仍可变。

然而我们进行区间操作时并不总是那么幸运，可能会把原本连续的区间断开。我们需要一个函数来实现“断开”的操作，把 $\langle l, r, v \rangle$ 断成 $\langle l, pos - 1, v \rangle$ 和 $\langle pos, r, v \rangle$ ：

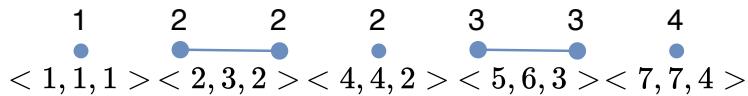
```

auto split(ll pos) {
    // 若不支持C++14, auto须改为set<node>::iterator
    auto it = tree.lower_bound(node(pos, 0, 0)); // 寻找左端
    点大于等于pos的第一个节点
    // 若不支持C++11, auto须改为set<node>::iterator
    if (it != tree.end() && it->l == pos) // 如果已经存在以
        pos为左端点的节点, 直接返回
        return it;
    it--; // 否则往前数一个节点
    ll l = it->l, r = it->r, v = it->v;
    tree.erase(it); // 删除该节点
    tree.insert(node(l, pos - 1, v)); // 插入<l, pos-1, v>和<
    pos, r, v>
    return tree.insert(node(pos, r, v)).first; // 返回以pos
    开头的那个节点的迭代器
    // insert默认返回值是一个pair, 第一个成员是我们要的
}

```

例如刚刚的情况，我们来看split(4)会发生什么：

首先lower_bound，找到左端点大于等于4的节点，即 $<5, 6, 3>$ 。它的左端点不是4，所以回退，得 $<2, 4, 2>$ 。我们把节点 $<2, 4, 2>$ 删除，然后插入 $<2, 3, 2>$ 及 $<4, 4, 2>$ 即可：



区间赋值是珂朵莉树的精髓，其实现方法也非常简单：

```

void assign(ll l, ll r, ll v) {
    auto end = split(r + 1), begin = split(l); // 顺序不能颠
    倒, 否则可能RE
    tree.erase(begin, end); // 清除一系列节点
}

```

```

tree.insert(node(l, r, v)); // 插入新的节点
}

```

也就是把范围内的节点全部删除，然后换上新的（范围较大的）节点。但要注意求end和begin的顺序不能颠倒，因为split(end)可能把begin原来所在的节点断开。

例 7.4.4 (CF915E Physical Education Lessons). *Alex* 高中毕业了，他现在是大学新生。虽然他学习编程，但他还是要上体育课，这对他来说完全是一个意外。快要期末了，但是不幸的 *Alex* 的体育学分还是零蛋！*Alex* 可不希望被开除，他想知道到期末还有多少天的工作日，这样他就能在这些日子里修体育学分。但是在这里计算工作日可不是件容易的事情：

从现在到学期结束还有 n 天（从 1 到 n 编号），他们一开始都是工作日。接下来学校的工作人员会依次发出 q 个指令，每个指令可以用三个参数 l, r, k 描述：

如果 $k = 1$ ，那么从 l 到 r （包含端点）的所有日子都变成非工作日。

如果 $k = 2$ ，那么从 l 到 r （包含端点）的所有日子都变成工作日。

帮助 *Alex* 统计每个指令下发后，剩余的工作日天数。

输入格式：

第一行一个整数 n ，第二行一个整数 q ($1 \leq n \leq 10^9$, $1 \leq q \leq 3 \cdot 10^5$)，分别是剩余的天数和指令的个数。

接下来 q 行，第 i 行有 3 个整数 l_i, r_i, k_i ，描述第 i 个指令 ($1 \leq l_i, r_i \leq n$, $1 \leq k_i \leq 2$)。

```

4
6
1 2 1
3 4 1
2 3 2
1 3 2

```

```
2 4 1  
1 4 2
```

输出格式:

输出 q 行，第 i 行表示第 i 个指令被下发后剩余的工作日天数。

```
2  
0  
2  
3  
1  
4
```

只需要在assign过程中求一下和即可，部分代码如下：

```
int sum;  
  
void assign(int l, int r, int v) {  
    int tot = 0, len = 0;  
    auto end = split(r + 1), it = split(l), begin = it;  
    for (it; it != end; it++) {  
        len += (it->r - it->l + 1);  
        tot += it->v * (it->r - it->l + 1);  
    }  
    tree.erase(begin, end);  
    tree.insert(node(l, r, v));  
    if (v == 1)  
        sum += (len - tot);  
    else  
        sum -= tot;  
}  
  
int main() {  
    int n = read(), q = read();  
    tree.insert(node(1, n, 1));  
    sum = n;  
    while (q--) {
```

```

        int l = read(), r = read(), k = read();
        assign(l, r, k == 1 ? 0 : 1);
        printf("%d\n", sum);
    }
    return 0;
}

```

根据这题的思路，我们来实现本节开头的操作，做法其实并不复杂，核心就是一个**暴力**。区间加的实现就是挨个相加：

```

void add(ll l, ll r, ll v) {
    auto end = split(r + 1);
    for (auto it = split(l); it != end; it++)
        it->v += v;
}

```

求区间 k 大值就直接扔到vector里排下序：

```

ll kth(ll l, ll r, ll k) {
    auto end = split(r + 1);
    vector<pair<ll, ll>> v; // 这个pair里存节点的值和区间长度
    for (auto it = split(l); it != end; it++)
        v.push_back(make_pair(it->v, it->r - it->l + 1));
    sort(v.begin(), v.end()); // 直接按节点的值的大小排下序
    for (int i = 0; i < v.size(); i++) { // 然后挨个丢出来，直到丢出k个元素为止
        k -= v[i].second;
        if (k <= 0)
            return v[i].first;
    }
}

```

求区间 n 次方和则用快速幂直接求：

```
ll sum_of_pow(ll l, ll r, ll x, ll y) {
    ll tot = 0;
    auto end = split(r + 1);
    for (auto it = split(l); it != end; it++)
        // qpow需要另外实现一下
        tot = (tot + qpow(it->v, x, y) * (it->r - it->l + 1)
               ) % y;
    return tot;
}
```

需要注意的是，如果题目不保证随机数据，很多时候，珂朵莉树也许只能当作一种**对拍方法或者骗分算法**。

现在我们回到例题7.4.3，这里给出参考做法：

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
inline ll read() {
    ll ans = 0;
    char c = getchar();
    while (!isdigit(c))
        c = getchar();
    while (isdigit(c)) {
        ans = ans * 10 + c - '0';
        c = getchar();
    }
    return ans;
}
struct node {
    ll l, r;
    mutable ll v;
    node(ll l, ll r, ll v) : l(l), r(r), v(v) {}
    bool operator<(const node &o) const { return l < o.l; }
}
```

```
};

set<node> tree;

auto split(ll pos) {
    auto it = tree.lower_bound(node(pos, 0, 0));
    if (it != tree.end() && it->l == pos)
        return it;
    it--;
    ll l = it->l, r = it->r, v = it->v;
    tree.erase(it);
    tree.insert(node(l, pos - 1, v));
    return tree.insert(node(pos, r, v)).first;
}

void assign(ll l, ll r, ll v) {
    auto end = split(r + 1), begin = split(l);
    tree.erase(begin, end);
    tree.insert(node(l, r, v));
}

ll qpow(ll a, ll n, ll p) {
    ll ans = 1;
    a %= p;
    while (n) {
        if (n & 1)
            ans = ans * a % p;
        n >>= 1;
        a = a * a % p;
    }
    return ans;
}

ll n, m, seed, vmax;
ll rnd() {
    ll ret = seed;
    seed = (seed * 7 + 13) % 1000000007;
    return ret;
}
```

```
}

void add(ll l, ll r, ll v) {
    auto end = split(r + 1);
    for (auto it = split(l); it != end; it++)
        it->v += v;
}

ll kth(ll l, ll r, ll k) {
    auto end = split(r + 1);
    vector<pair<ll, ll>> v;
    for (auto it = split(l); it != end; it++)
        v.push_back(make_pair(it->v, it->r - it->l + 1));
    sort(v.begin(), v.end());
    for (int i = 0; i < v.size(); i++) {
        k -= v[i].second;
        if (k <= 0)
            return v[i].first;
    }
}

ll sum_of_pow(ll l, ll r, ll x, ll y) {
    ll tot = 0;
    auto end = split(r + 1);
    for (auto it = split(l); it != end; it++)
        tot = (tot + qpow(it->v, x, y) * (it->r - it->l + 1)
               ) % y;
    return tot;
}

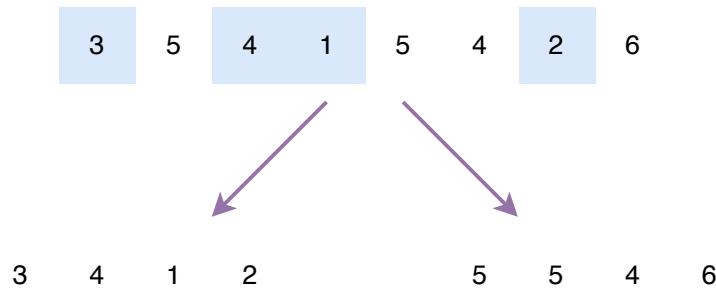
int main() {
    n = read(), m = read(), seed = read(), vmax = read();
    for (int i = 1; i <= n; ++i) {
        int r = rnd();
        tree.insert(node(i, i, r % vmax + 1));
    }
    for (int i = 1; i <= m; ++i) {
```

```
ll opr = rnd() % 4 + 1, l = rnd() % n + 1, r = rnd()
    % n + 1, x, y;
if (l > r)
    swap(l, r);
if (opr == 3)
    x = rnd() % (r - l + 1) + 1;
else
    x = rnd() % vmax + 1;
if (opr == 4)
    y = rnd() % vmax + 1;
switch (opr) {
case 1:
    add(l, r, x);
    break;
case 2:
    assign(l, r, x);
    break;
case 3:
    printf("%lld\n", kth(l, r, x));
    break;
case 4:
    printf("%lld\n", sum_of_pow(l, r, x, y));
}
}
return 0;
}
```

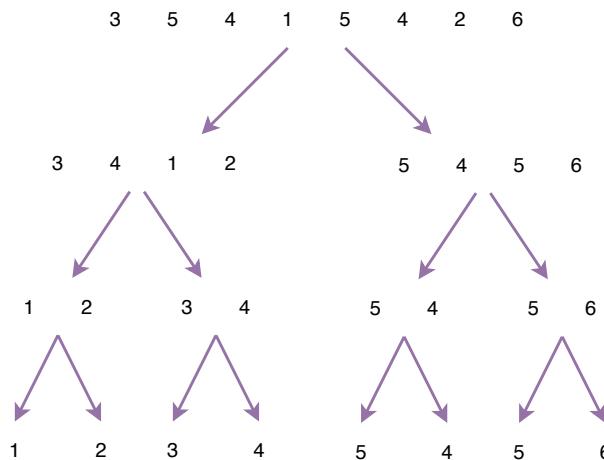
7.4.8 划分树

划分树是一种可以 $O(n \log n)$ 地求静态区间第 k 小数的数据结构。当然，主席树（即可持久化线段树）也可以实现这个需求，但是划分树的常数更小。

划分树的每一个节点是一个数组。对于一个长度大于 1 的数组 A ，我们把它划分为长度最多相差 1 的两个子数组 A_L 和 A_R ，作为子节点。其中， A_L 中任意一个数都小于等于 A_R 中任意一个数。与此同时，还保证 A_L 与 A_R 中元素的相对位置与在 A 中时相同。



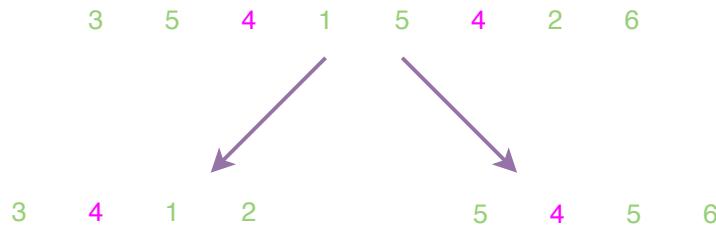
方便起见，我们把原数组补充至长度为 2^D 。这样的话， A_L 和 A_R 的长度就总是相等了。此时，划分树是一棵满二叉树，它有 $D + 1$ 层，每层的长度总和均为 2^D 。



不难发现，划分树实际上的每一层都可以看作快速排序的一个中间结果，到最后一层时，排序完全完成。

实际上建出这棵树很容易，对于每一个节点而言，我们只需要从左往右扫一遍，把小于中位数的丢在左边，把大于中位数的丢在右边，用等于中

位数的填空位即可，需要注意的是，最好不要最后再去填空位，而要提前计算好左边的空位数，以确定有多少个等于中位数的数要放到左边。



如上图所示，中位数为 4，有 3 个数比 4 小，3 个数比 4 大，剩下两个空位用 4 填。计算得出左边剩下一个空位，所以第一个 4 需要放到左边。

因此，一般来说，确定数组的中位数，需要在排序后的数组中找位于中间的那个值。不过，划分树尽管包含了大量的数组，我们不需要多次排序，而只需要排序一次。这是因为，考虑划分树和快速排序的关系，不难发现划分树上任何一个节点的排序结果，都是整个数组的排序结果在**对应位置的子数组**。

除了建出这棵树外，我们还设置一个数组 `cnt`，记录从起始位置到现在有多少节点去到了左侧，这在后面查询操作中非常重要。建树的程序如下：

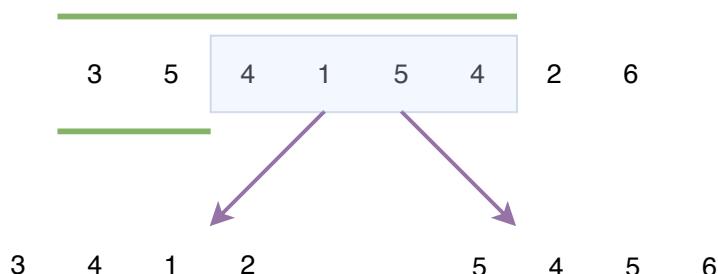
```

// 设 A 为原数组
for (int i = 1; i <= n; ++i) {
    tree[0][i] = A[i];
}
sort(A, A + N);
for (int d = 0, w = N; d < D; ++d, w >>= 1) { // 处理第 d 层
    for (int i = 0; i < N; i += w) { // 处理 [i, i + w] 这个节点
        int l = i, r = i + (w >> 1), p = A[r - 1], rest = w >>
        1; // p 为中位数
        for (int j = i; j < i + w; ++j) {
            rest -= tree[d][j] < p; // rest 为左边的空位数
        }
    }
}

```

```
for (int j = i; j < i + w; ++j) {
    // cnt其实可以看成一个前缀和
    cnt[d][j] = j == i ? 0 : cnt[d][j - 1];
    // 小于中位数，或者等于中位数且左边还有空位
    if (tree[d][j] < p || tree[d][j] == p && rest-- > 0)
        tree[d + 1][l++] = tree[d][j], cnt[d][j]++;
    else
        tree[d + 1][r++] = tree[d][j];
}
}
```

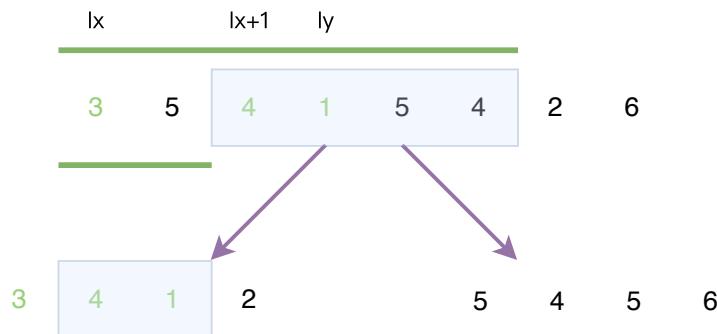
接下来考虑查询时怎么往子节点转移，不妨设下图中 3 的下标为 i ，我们要查询 $[i+2, i+5]$ 中第 1 小的数。分别计算出 $[i, i+1]$ 和 $[i, i+5]$ 中被划分到左边的数的个数 lx 、 ly ，以及被划分到右边的数的个数 rx 、 ry 。



可以发现在查询的区间中，被划分到左边的数的个数为 $ly-1x$ ，在本图中为 2，而我们要找的是第 1 小的数。根据划分树的性质，被划分到左边的数不大于被划分到右边的数，所以要找的数应该在左边。

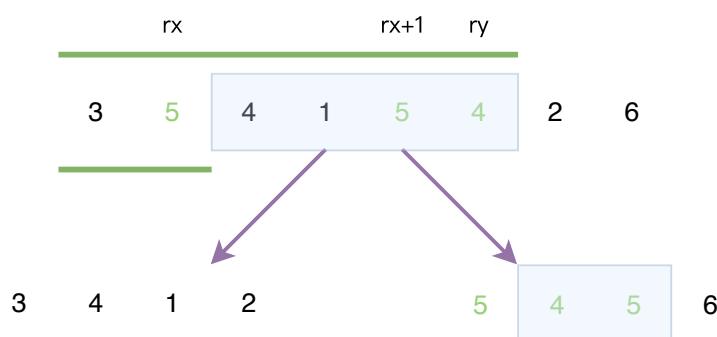
那么在转移到子节点后，原来的查询区间会有什么变化呢？因为划分树在划分元素时会**保持相对位置**，所以只需要考虑原区间里那些要去左边的数，是第几个被划分到左边的即可。

答案是：第 $l_x + 1$ 个到第 l_y 个，这对应的左子树中的下标范围是 $[i + l_x, i + l_y - 1]$ 。



上面的例子是转移到左子树，如果是转移到右子树呢？例如，现在我们要求原区间中第 3 大的数。

其实是完全一样的道理，我们考虑原区间里那些要去右边的数，是第几个被划分到右边的。显然，是第 $rx + 1$ 个到第 ry 个。假设右儿子的起始下标是 i ，那么下标范围是 $[i + rx, i + ry - 1]$ 。需要注意的是，我们已经知道了有 $ly - lx$ 个数会去左边，所以这时我们查的不再是第 k 小数，而是第 $k - (ly - lx)$ 小数。



最后，给出非递归实现的程序：

```
// 在节点 [i, i + N >> d) 中求 [x, y] 区间的第 k 小值
for (int d = 0, i = 0; d < D; d++) {
    int lx = x == i ? 0 : cnt[d][x - 1], ly = cnt[d][y];
    int rx = x - i - lx, ry = y - i - ly + 1;
    if (ly - lx >= k) {
        x = i + lx;
    }
}
```

```

y = i + ly - 1;
} else {
    // 转移到右子树，起始下标有变
    i += N >> d + 1;
    x = i + rx;
    y = i + ry - 1;
    k -= ly - lx;
}
cout << tree[D][x] << '\n';

```

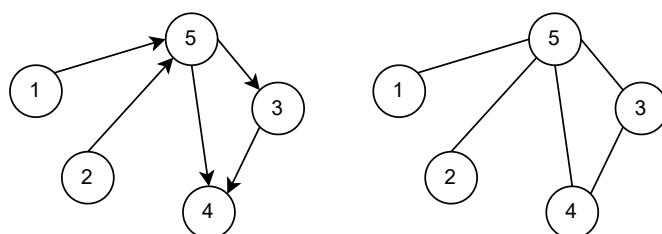
7.5 图

图是另一种非线性数据结构。在图结构中，数据结点一般称为**顶点**，而边是顶点的**有序偶对**。如果两个顶点之间存在一条边，那么就表示这两个顶点具有**相邻关系**。图可以分为**有向图**和**无向图**，有向图中的边是有方向的，而无向图的边是双向连通的。

7.5.1 图的存储

每当我们涉及到对图的问题的处理时，为了解决它们，我们首先得把图存储起来，这个过程我们称为图的存储。

我们来举两个例子，一个有向图和一个无向图，具体如下所示：



邻接矩阵

谈到存图，最朴素的想法当然是用一个二维数组`mat[] []`存储两个边的连接情况。假如从顶点 u 到顶点 v 有一条边，则令`mat[u][v] = 1`。这种建图方法称为邻接矩阵。

例如上面的那张有向图（左）和无向图（右）的邻接矩阵是：

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

这是没有边权的情况，对于有边权的图，其实只要把对应的 1 换成边权即可。

邻接矩阵的优点显而易见：简单好写，查询速度快。但缺点也很明显：**空间复杂度太高了。** n 个点对应大小为 n^2 的数组，如果点的数量巨大，这种方法就完全不可行了。

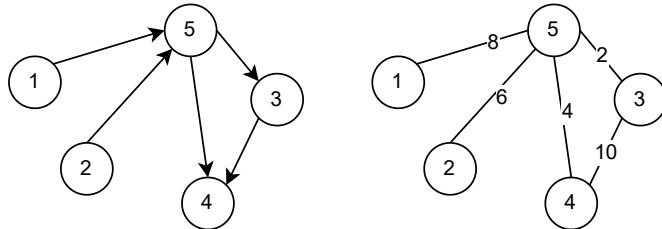
另一方面，我们可以看到，上面那两个矩阵中有大量的元素是 0，有大量空间被浪费了。这虽然使得我们可以迅速判断两个点之间是否没有边，但我们为此付出的代价太大了，我们其实更关注那些确实存在的边。我们希望，可以跳过这些 0，直达有边的地方。

邻接表

我们可以把邻接矩阵的行从数组替换为链表。我们需要用一个结构体来同时储存边的终点和权值：

```
struct Edge {
    int to, w; // 没有边权可以不使用结构体，只存储终点即可
};
```

我们把上面给出的两个图结构增加权值，结果如下所示：



我们根据图给出它的邻接表：

1	:	5	
2	:	5	
3	:	4	
4	:		
5	:	3	4

1	:	5(8)			
2	:	5(6)			
3	:	4(10)	5(2)		
4	:	3(10)	5(4)		
5	:	3(2)	4(10)	2(6)	1(8)

换句话说，邻接表存储每个顶点能够到达哪些顶点。注意这里链表的顺序是无关紧要的。

7.5.2 链式前向星

链式前向星是一种用**数组模拟链表**的存储方式，下面我们给出链式前向星的一种实现：

```

struct Edge {
    int to, w, next;
}edges[MAXM];

int first[MAXN], cnt; // cnt为当前边的编号

```

```

void add(int from, int to, int w) {
    edges[++cnt].w = w;      //新增一条编号为 cnt+1 的边，边权为 w
    edges[cnt].to = to;      //该边的终点为 to
    edges[cnt].next = first[from]; //把下一条边，设置为当前
                                   //起点的最后一条边
    first[from] = cnt; //该边成为当前起点新的边
}

```

在上述程序中，通常会使用多个数组来代替原有的结构体：

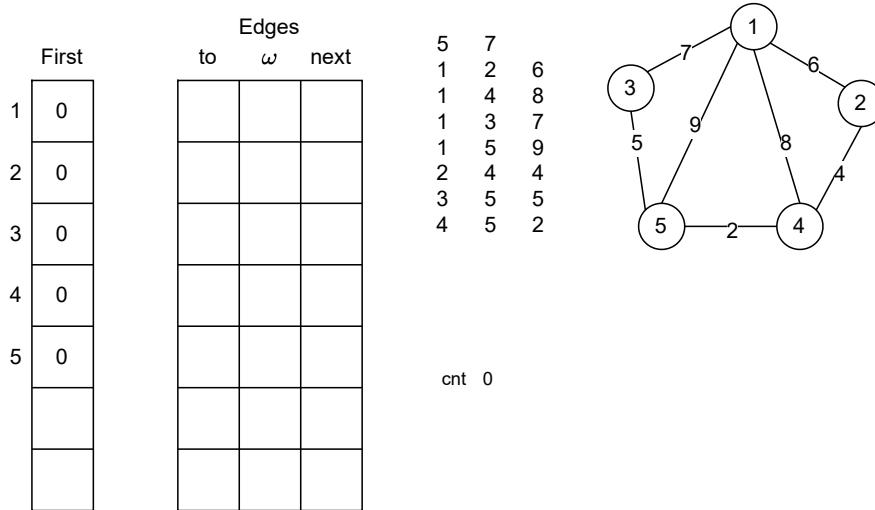
```

int head[N], edges[M], _next[M], weight[M], cnt;

void add(int from, int to, int w) {
    edges[cnt] = to;
    weight[cnt] = w;
    _next[cnt] = head[from];
    head[from] = cnt++;
}

```

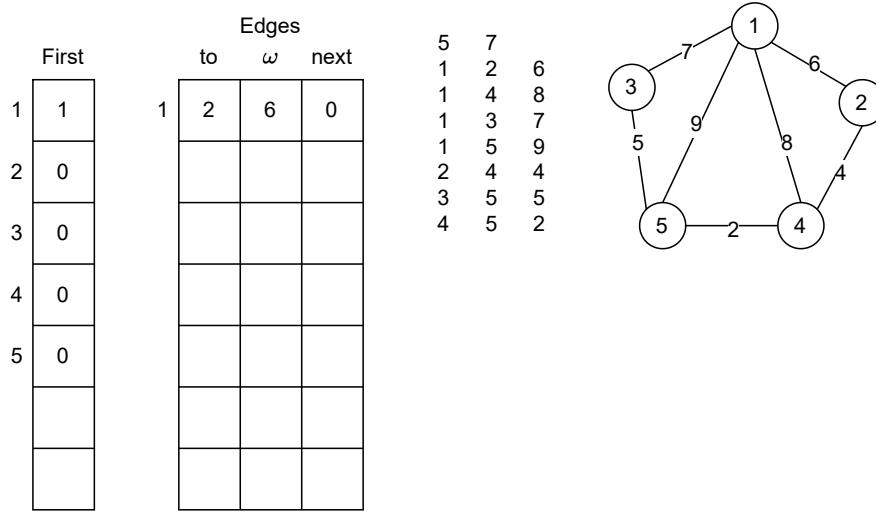
现在根据代码，我们用一张图来讲述前向星的过程：



我们为每条边额外储存一个属性next，并赋予每条边一个编号，我们

可以根据这个边的编号进行访问。`First`数组则用于储存每个起点读过的最后一条边的编号。

`First`数组的初始值均为 0，同样的`cnt`的初始值也为 0，现在我们开始读以 1 为起点，终点为 2 的边，具体如图所示：

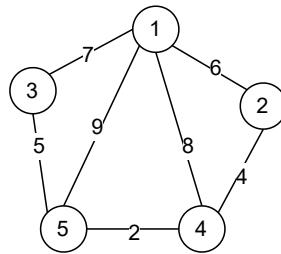


其中`to`记录的就是当前节点的终点节点， `ω` 记录的是权值，而`next`记录的是当前节点存储的上一条边的编号，由于现在读的是第一条边，所以上一条边的编号是 0。同时，由于我们读过一条边了，记录其编号为 1，则`First`数组 1 号位的值改为 1。

我们现在将以节点 1 为起点的节点读完情况展示出来：

First	Edges		
	to	ω	next
1	4	0	0
2	0	8	1
3	0	9	2
4	0	7	3
5	0		

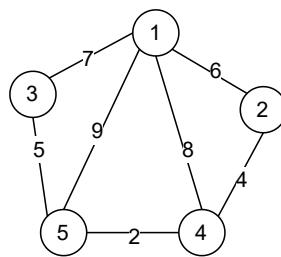
5	7	6
1	2	8
1	4	9
1	3	7
1	5	4
2	4	5
3	5	2
4	5	



现在我们读以节点 2 为起点的边，其具体如下所示：

First	Edges		
	to	ω	next
1	4	0	0
2	5	8	1
3	0	9	2
4	0	7	3
5	0	4	0

5	7	6
1	2	8
1	4	9
1	3	7
1	5	4
2	4	5
3	5	2
4	5	

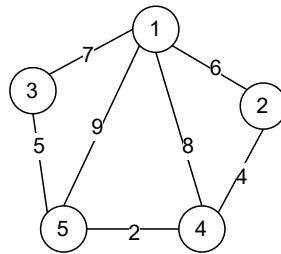


同样的，`next`记录的是当前节点存储的上一条边的编号，由于现在读的是以节点 2 为起点的第一条边，所以上一条边的编号是 0。我们需要注意`First`数组存储的是每个起点对应的最后一条边的编号，而节点 2 存储的第一条边也是最后一条边的编号是 5。

最终我们读完所有的边的结果应当是下图所示的情况：

First	Edges		
	to	ω	next
1	4	2	6
2	5	4	8
3	6	5	9
4	7	3	7
5	0	4	4
		5	5
		5	0
7	5	2	0

5	7	6
1	2	
1	4	8
1	3	7
1	5	9
2	4	4
3	5	5
4	5	2



当我们存储完所有的边后，如果我们要找以某个点为起点的边，我们就可以从First数组开始找，因为First数组存储的是每个起点对应的最后一条边的编号，例如我们找以节点 1 为起点的边，我们从First数组中可以得到最后的一条边编号是 4，根据编号 4 就可以到Edges数组中找到对应的记录，且由于Edges数组中有村有next值，next记录的是当前节点存储的上一条边的编号，即我们可以找到编号为 3 的记录，如此循环就可以得到以此节点为起点的所有边。

由于前向星结构会把新元素添加到最前面而不是最后面，因此称之为“前”向星。

7.6 堆

堆 (Heap) 是一类数据结构，它们拥有**树状结构**，且能够保证父节点比子节点大 (或小)。当根节点保存堆中最大值时，称为**大根堆**；反之，则称为**小根堆**。

二叉堆 (Binary Heap) 是最简单、常用的堆，是一棵符合堆的性质的**完全二叉树**。它可以实现 $O(\log n)$ 地插入或删除某个值，并且 $O(1)$ 地查询最大 (或最小) 值。

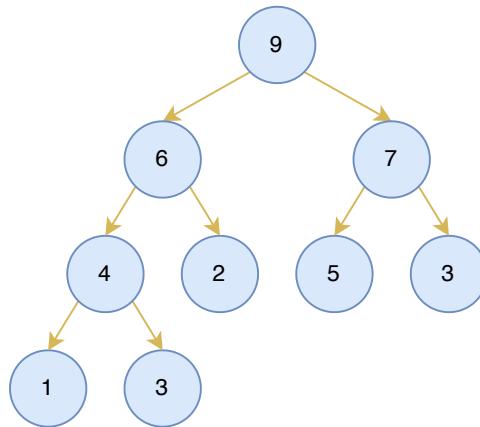


图 7.1: 大根堆示例

作为一棵完全二叉树，二叉堆完全可以用一个1-index的数组来存储，对于节点 p ， $p * 2$ 即为左儿子， $p * 2 + 1$ 即为右节点。同时，用size记录当前二叉堆中节点的个数。

9 1	6 2	7 3	4 4	2 5	5 6	3 7	1 8	3 9
--------	--------	--------	--------	--------	--------	--------	--------	--------

现在我们考虑如何保证二叉堆的性质不被破坏。实际上，对于一个破坏堆性质的节点，我们可以使其**上浮或下沉**，因为最差也不过是上浮到顶或是下沉到底，所以只需要 $O(\log n)$ 的时间就可以使其不再破坏性质。但其实，插入和删除都只需要上浮/下沉一个节点。

我们以大根堆为例，上浮过程即**当前节点不断与父节点比较**，如果比父节点大就与之交换，直到不大于父节点或成为根节点为止：

```

void swim(int n) {
    for (int i = n; i > 1 && heap[i] > heap[i / 2]; i /= 2)
        swap(heap[i], heap[i / 2]);
}
  
```

同样的，下沉过程即**不断与较大的子节点比较**，如果比它小就与之交换，直到不小于任何子节点或成为叶子节点为止。之所以要与较大的子节点比较，

是为了保证交换上来的节点比两个子节点都大：

```
int son(int n) { // 找到需要交换的那个子节点
    return n * 2 + (n * 2 + 1 <= size && heap[n * 2 + 1] >
        heap[n * 2]);
}

void sink(int n) {
    for (int i = n, t = son(i); t <= size && heap[t] > heap[i];
         i = t, t = son(i))
        swap(heap[i], heap[t]);
}
```

有了上浮功能后，我们就可以在大根堆里插入元素，实现同样也很简单，按照完全二叉树的定义在尾部直接插入、然后上浮即可：

```
void insert(int x) {
    heap[++size] = x;
    swim(size);
}
```

在删除元素时，我们可以直接将**根节点的元素与最后一个节点交换**，并让数组的**size减1**：

```
void pop() {
    swap(heap[1], heap[size--]);
    sink(1);
}
```

由于大根堆的特性，在查询序列最大值时直接返回根节点即可：

```
int top() {
    return heap[1];
}
```

最后给出一个数组 $O(n)$ 建堆的方案，其实只需复制过来然后**从底部到顶部依次下沉即可**。但实际上因为**叶子节点**不需要下沉，所以可以从 $\frac{n}{2}$ 处开始

遍历。

```
void build(int A[], int n) { // 从一个（这里是 0-index 的）数组
    // O(n) 地建立二叉堆
    memcpy(heap + 1, A, sizeof(int) * n);
    size = n;
    for (int i = n / 2; i > 0; --i)
        sink(i);
}
```

例 7.6.1 (大根堆). 有一堆石头，每块石头的重量都是正整数。每一回合，从中选出任意两块石头，然后将它们一起粉碎。假设石头的重量分别为 x 和 y ，且 $x \leq y$ 。那么粉碎的可能结果如下：

如果 $x = y$ ，那么两块石头都会被完全粉碎；

如果 $x \neq y$ ，那么重量为 x 的石头将会完全粉碎，而重量为 y 的石头新重量为 $y-x$ 。

最后，最多只会剩下一块石头，返回此石头最小的可能重量。如果没有石头剩下，就返回 0。

输入示例：

```
[2, 7, 4, 1, 8, 1]
```

输出示例：

```
1
/**
 * 组合 2 和 4，得到 2，所以数组转化为 [2, 7, 1, 8, 1]，
 * 组合 7 和 8，得到 1，所以数组转化为 [2, 1, 1, 1]，
 * 组合 2 和 1，得到 1，所以数组转化为 [1, 1, 1]，
 * 组合 1 和 1，得到 0，所以数组转化为 [1]，这就是最优值。
*/
```

提示：

```
1 <= stones.length <= 30
1 <= stones[i] <= 1000
```

例题7.6.1的做法并不难，将所有石头的重量放入最大堆中，每次依次从队列中取出最重的两块石头 a 和 b ，必有 $a \geq b$ 。如果 $a > b$ ，则将新石头 $a - b$ 放回到最大堆中；如果 $a = b$ ，两块石头完全被粉碎，因此不会产生新的石头。重复上述操作，直到剩下的石头少于 2 块。

最终可能剩下 1 块石头，该石头的重量即为最大堆中剩下的元素，返回该元素；也可能没有石头剩下，此时最大堆为空，返回。

7.7 哈希表

哈希表，也被称为散列表，源自于**散列函数 (Hash function)**，其思想是如果在结构中存在关键字和 T 相等的记录，那么必定在 $F(T)$ 的存储位置可以找到该记录，这样就可以不用进行比较操作而直接取得所查记录。

7.7.1 位运算

我们曾学习集合论 (set theory) 的相关知识，例如，包含若干整数的集合 $S = \{0, 2, 3\}$ 。在编程中，通常用哈希表 (hash table) 表示集合。

如果要编程实现“求两个哈希表的交集”，不可避免的需要一个一个地遍历哈希表中的元素。那么，有没有效率更高的做法呢？

这个时候就需要用到二进制，我们可以将集合用二进制的形式表示，二进制从低到高第 i 位为 1 表示 i 在集合中，为 0 表示 i 不在集合中。例如集合 $S = \{0, 2, 3\}$ 可以用二进制数 $1101_{(2)}$ 表示；反过来，二进制数 $1101_{(2)}$ 也对应着集合 $\{0, 2, 3\}$ 。换言之，包含非负整数的集合 S 可以用如下方式“压缩”成一个数字：

$$f(S) = \sum_{i \in S} 2^i$$

例如,集合 $\{0, 2, 3\}$ 的二进制表示为 $1101_{(2)}$,也可以压缩为 $2^0+2^2+2^3=13$ 。

位运算有着“并行计算”的特点,借此我们可以高效地做一些和集合有关的运算。

7.7.2 集合与集合

术语	集合	位运算	集合示例	位运算示例
交集	$A \cap B$	$a \& b$	$\{0, 2, 3\}$ $\cap \{0, 1, 2\}$ $= \{0, 2\}$	1101 $\& 0111$ $= 0101$
并集	$A \cup B$	$a \mid b$	$\{0, 2, 3\}$ $\cup \{0, 1, 2\}$ $= \{0, 1, 2, 3\}$	1101 $ 0111$ $= 1111$
对称差	$A \triangle B$	$a \oplus b$	$\{0, 2, 3\}$ $\triangle \{0, 1, 2\}$ $= \{0, 1, 2, 3\}$	1101 $\oplus 0111$ $= 1010$
差	$A - B$	$a \& \sim b$	$\{0, 2, 3\}$ $- \{1, 3\}$ $= \{0, 3\}$	1101 $\& 1001$ $= 1001$
差(子集)	$A - B, B \subseteq A$	$a \oplus b$	$\{0, 2, 3\}$ $- \{0, 2\}$ $= \{3\}$	1101 $\oplus 0101$ $= 1000$
包含于	$A \subseteq B$	$a \& b = a$ $a \mid b = b$	$\{0, 2\} \subseteq$ $\{0, 2, 3\}$	$0101 \& 1101$ $= 0101$ $0101 \mid 1101$ $= 1101$

其中 $\&$ 表示按位与， $|$ 表示按位或， \oplus 表示按位异或， \sim 表示按位取反。有一点需要注意：包含于的两种位运算写法是等价的，在编程时只需判断其中任意一种。

7.7.3 集合与元素

集合与元素的判断中通常会用到移位运算。其中 $<<$ 表示左移， $>>$ 表示右移。

术语	集合	位运算	集合示例	位运算示例
空集	\emptyset	0		
单元素集合	$\{i\}$	$1 << i$	$\{2\}$	$1 << 2$
全集	$U = \{0, 1, \dots, n - 1\}$	$(1 << n) - 1$	$\{0, 1, 2, 3\}$	$(1 << 4) - 1$
补集	$C_U S = U - S$	$((1 << n) - 1) \oplus s$	$U = \{0, 1, 2, 3\}$ $\{1, 2\}^C = \{0, 3\}$	1111 $\oplus 0110$ $= 1001$
属于	$i \in S$	$(s >> i) \& 1 == 1$	$2 \in \{0, 2, 3\}$	$(1101 >> 2)$ $\& 1 == 1$
不属于	$i \notin S$	$(s >> i) \& 1 == 0$	$1 \notin \{0, 2, 3\}$	$(1101 >> 1)$ $\& 1 == 0$
添加元素	$S \cup \{i\}$	$s (1 << i)$	$\{0, 3\} \cup \{2\}$	$1001 (1 << 2)$
删除元素	$S - \{i\}$	$s \& \sim(1 << i)$	$\{0, 2, 3\} - \{2\}$	$1001 \& \sim(1 << 2)$
删除元素 (一定存在)	$S - \{i\}, i \in S$	$s \oplus (1 << i)$	$\{0, 2, 3\} - \{2\}$	$1101 \oplus (1 << 2)$
删除最小 元素		$s \& (s - 1)$		

我们删除最小元素的方法是 $s \& (s - 1)$, 其原理如下:

```
s = 101100
s - 1 = 101011
s & (s - 1) = 101000
```

我们将最低位的 1 变成 0, 同时 1 右边的 0 都取反, 变成 1, 特别的: 如果 s 是 2 的幂, 那么有 $s \& (s - 1) = 0$ 。

此外, 只包含最小元素的子集, 即二进制最低 1 及其后面的 0, 也叫 `lowbit`, 可以用 $s \& -s$ 算出, 例如:

```
s = 101100
~s = 010011
(~s)+1 = 010100 // 补码的定义, -s即s的最低 1 左侧取反, 右侧不变
s & -s = 000100 // lowbit
```

7.7.4 遍历集合

遍历集合的常用方法是按位遍历, 即从低位到高位依次遍历, 设元素范围从 0 到 $n - 1$, 挨个判断每个元素是否在集合 s 中:

```
for (int i = 0; i < n; i++) {
    if ((s >> i) & 1) { // i 在 s 中
        // 处理 i 的逻辑
    }
}
```

7.7.5 枚举集合

枚举所有集合

设元素范围从 0 到 $n - 1$, 从空集 \emptyset 枚举到全集 U :

```
for (int s = 0; s < (1 << n); s++) {
```

```
// 处理 s 的逻辑  
}
```

枚举非空子集

设集合为 s , 从大到小枚举 s 的所有非空子集 sub :

```
for (int sub = s; sub; sub = (sub - 1) & s) {  
    // 处理 sub 的逻辑  
}
```

有一个问题是: 为什么要写成 $sub = (sub - 1) \& s$ 呢?

暴力做法是从 s 出发, 不断减 1, 直到 0。但这样做, 中途会遇到很多并不是 s 的子集的情况。例如 $s = 10101$ 时, 减 1 得到 10100, 这是 s 的子集, 但再减 1 得到 10011, 这并不是 s 的子集, 下一个子集应该是 10001。

把所有的合法子集按顺序列出来, 会发现我们做的相当于“压缩版”的二进制减法, 例如

$10101 \rightarrow 10100 \rightarrow 10001 \rightarrow 10000 \rightarrow 00101 \rightarrow \dots$

如果忽略掉 10101 中的两个 0, 数字的变化和二进制减法是一样的, 即:

$111 \rightarrow 110 \rightarrow 101 \rightarrow 100 \rightarrow 011 \rightarrow \dots$

如何快速跳到下一个子集呢? 比如, 怎么从 10100 跳到 10001?

- 普通的二进制减法, 是 $10100 - 1 = 10011$, 也就是把最低位的 1 变成 0, 同时把最低位的 1 右边的 0 都变成 1。
- 压缩版的二进制减法也是类似的, 对于 $10100 - 10001$, 也会把最低位的 1 变成 0, 对于最低位的 1 右边的 0, 并不是都变成 1, 只有在 $s = 10101$ 中的 1 才会变成 1。怎么做到?

减 1 后 $\&$ 10101 就行，也就是 $(10100-1) \& 10101 = 10001$ 。

枚举子集 (包含空集)

如果要从大到小枚举 s 的所有子集 sub (从 s 枚举到空集 \emptyset):

```
int sub = s;
do {
    // 处理 sub 的逻辑
    sub = (sub - 1) & s;
} while (sub != s);
```

原理是当 $sub = 0$ 时(即空集)，再减 1 就得到-1，对应的二进制为 $111\cdots 1$ ，再 $\&s$ 就得到了 s 。所以当循环到 $sub = s$ 时，说明最后一次循环的 $sub = 0$ (空集)， s 的所有子集都枚举到了，退出循环。

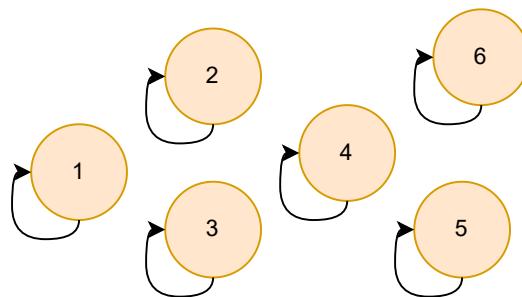
7.8 并查集

并查集可谓是最优雅而简洁的数据结构之一，主要用于解决一些元素分组的问题。我们通常用并查集处理不相交的集合，并提供两种操作：

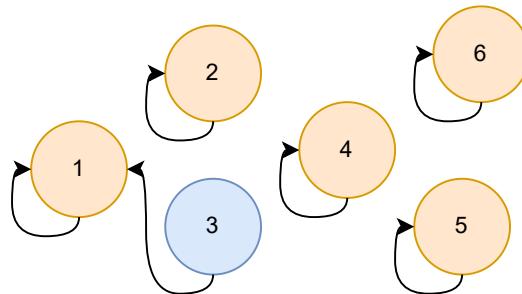
- 合并 (Union): 把两个集合合并到同一个集合中；
- 查询 (Find): 查询两个元素是否在同一个集合中。

7.8.1 并查集引入

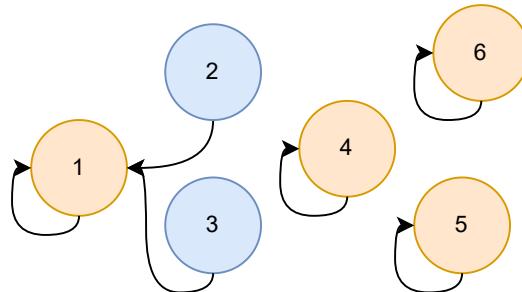
并查集的核心在于用集合中的一个元素代表集合，我们来看一个例子，现有如下 $[1, 2, 3, 4, 5, 6]$ 六个元素：



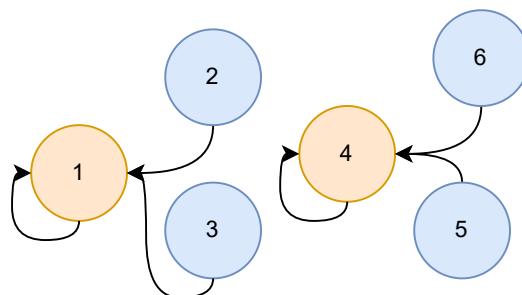
我们来了解一下并查集的运作，最开始所有的集合只有一个元素，那么代表元素也就是那个唯一的元素。现在，我们合并元素 1 和元素 3 所在的集合，元素 1 作为该集合的代表元素，结果如下：



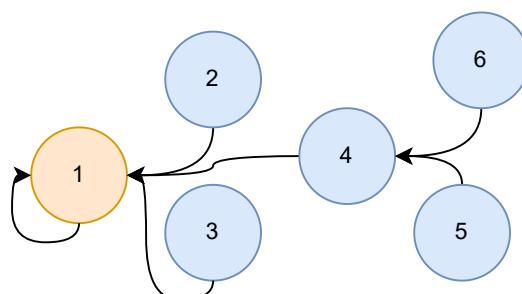
现在，我们想来合并元素 2 和元素 3 所在的集合，**合并的是代表元素**，我们假设也将元素 2 和合并到元素 3 所在的集合，其结果如图所示：



我们也将元素 4、元素 5、元素 6 进行合并，结果如下所示：

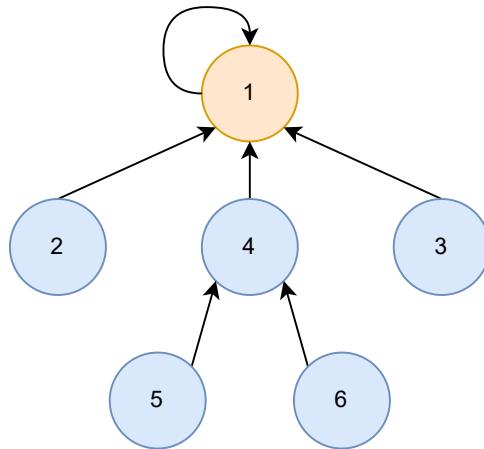


如果我们想要合并元素 2 和元素 6 所在的集合，自然也是让代表元素进行合并，代表元素合并后也就意味着这个集合合并到另一个集合下：



最终我们可以完成这个并查集的合并流程，从图上可以看出，这显然是一个**树状结构**，如果我们想根据集合中某个元素，我们要做的就是根据这个节点以及该节点的箭头指向找到他的**父节点**，并迭代这个过程直到找

到根节点，这的注意的是，根节点的父节点是其自身。现在，我们将上图画成通常的树状结构：



7.8.2 并查集实现

根据上述对并查集执行过程的描述，我们来尝试实现一个并查集。假设现有编号为1, 2, 3, ..., n的n个元素，用一个数组fa[]来存储每个元素的父节点（每个元素有且只有一个父节点）。

首先是初始化，一开始，我们先将它们的父节点设为自己：

```

int fa[MAXN];
inline void init(int n) {
    for(int i = 1; i <= n; i++)
        fa[i] = i;
}
  
```

接下来是查询，这里选择用递归实现代表元素的查询，即不断访问父节点，直至根节点（根节点的标志是其父节点是自身）：

```

int find(int x) {
    if(fa[x] == x)
        return x;
    else
  
```

```

    return find(fa[x]);
}

```

此外，若要判断两个元素是否属于同一集合，只需要看其根节点（即代表元素）是否相同即可。

最后是合并操作，在这里实现方式也很简单，即找到两个元素所在集合的代表元素，将前者的父节点设置为后者即可：

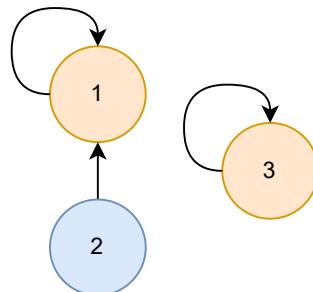
```

inline void merge(int i, int j) {
    fa[find(i)] = find(j);
}

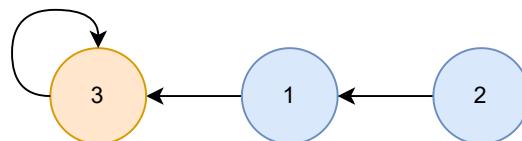
```

7.8.3 路径压缩

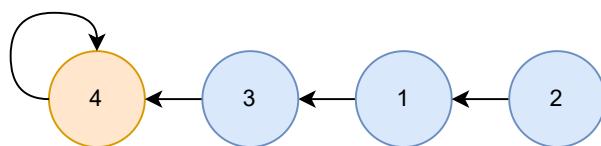
尽管我们实现了并查集，但是这样的设计存在问题，我们考虑一个极端情况，有两个点集{1,2}和{3}，具体如下：



现在我们要合并元素 2 与元素 3，即调用`merge(2, 3)`，按照我们之前的定义，当我们根据元素 2 找到根节点即元素 1 后，会将元素 1 的父节点设置为元素 3，即`fa[1] = 3`，也就是如图所示：

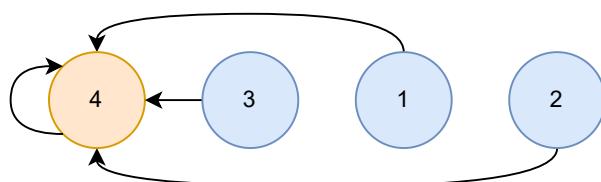


恰在此时，又来了元素 4，我们仍旧调用`merge(2, 4)`进行合并，那么也就通过迭代查找，找到此时元素 2 的根节点即元素 3，将其父节点设置为元素 4，即`fa[3] = 4`，情况如下图所示：



如此下去可能会形成一条长长的链，随着链越来越长，我们想要从底部找到根节点会变得越来越难。

为了解决这样的问题，我们可以使用**路径压缩**的方法。既然我们只关心一个元素对应的根节点，那我们希望每个元素到根节点的路径尽可能短，最好只需要一步，大致如下：



这样也很好实现，只要我们在查询的过程中，把沿途的每个节点的父节点都设为根节点即可。下一次再查询时，我们就可以省很多事。这用递归的写法很容易实现：

```

int find(int x) {
    if(x == fa[x])
        return x;
    else{
        fa[x] = find(fa[x]); //父节点设为根节点
        return fa[x];         //返回父节点
    }
}
/***
 * 通常用三元运算符写作一行：
 */

```

```

* return x == fa[x] ? x : (fa[x] = find(fa[x]));
* 但需要注意，赋值运算符=的优先级没有三元运算符?:高，因而要
    加括号
*/

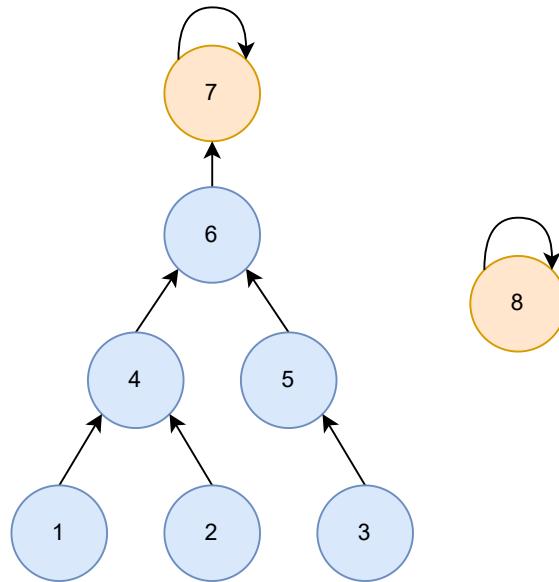
```

路径压缩优化后，并查集的时间复杂度已经比较低了，绝大多数不相交集合的合并查询问题都能够解决。

7.8.4 按秩合并

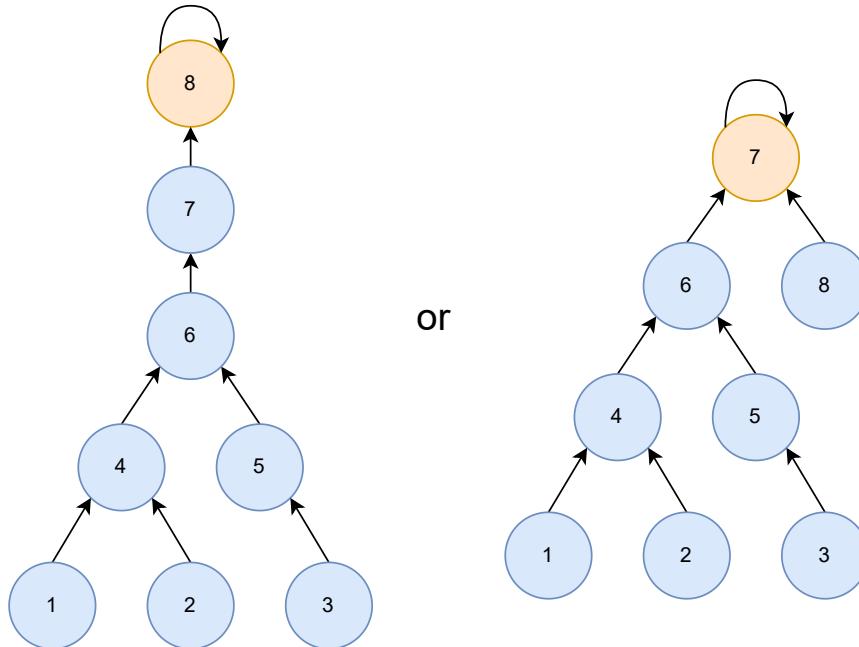
路径压缩优化后，并查集**并非都是只有两层的树**，其主要原因在于**路径压缩只在查询时进行**，也只压缩一条路径，所以并查集最终的结构仍然可能是比较复杂的。

现在我们考虑一种情况，现有一棵较为复杂的树和一个单元素两个元素集合，要合并元素 7 和元素 8，如下所示：



此时如果我们可以选择，是选择将元素 7 的父节点置为元素 8，还是选择将元素 8 的父节点置为元素 7？显然是后者更好，因为如果将元素 7 的父节点置为元素 8 必然会导致树的深度增加，每个字节点到根节点的距离

都变长了，查找根节点的路径也会变长。而将元素 8 的父节点置为元素 7，则不会有这个问题。



这个问题也告诉我们，当我们进行并查集合并的时候，我们应当**将简单的树往复杂度树中合并**，这样查找根节点的距离发生改变的节点个数较少。

在这里，可以利用一个数组 `rank[]` 记录每个**根节点对应的树的深度**，如果不是根节点，则 `rank[]` 记录的是以它为根节点的**子树**的深度。一开始将所有元素的秩置为 1，合并时比较两个根结点，把 `rank` 较小的合并到较大的树中。初始化代码如下：

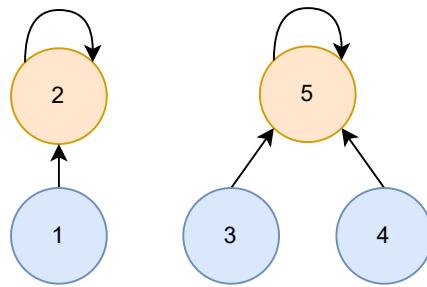
```
inline void init(int n) { // 初始化
    for (int i = 1; i <= n; ++i) {
        fa[i] = i;
        rank[i] = 1;
    }
}
inline void merge(int i, int j) { // 合并
```

```

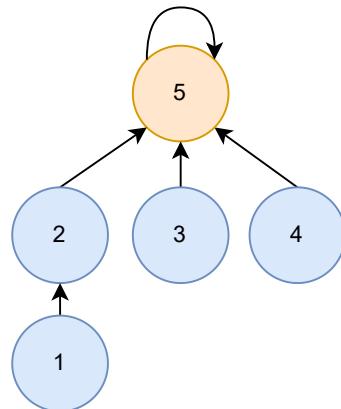
int x = find(i), y = find(j);      // 先找到两个根节点
if (rank[x] <= rank[y])
    fa[x] = y;
else
    fa[y] = x;
if (rank[x] == rank[y] &
    & x != y)
    rank[y]++;
// 如果深度相同且根节点不同，则新的
// 根节点的深度 +1
}

```

上面程序中，当深度相同时新根的节点需要加 1，这其实不难理解，例如我们现有两棵树，具体如下：



现在我们要合并元素 2 和元素 5，我们调用`merge(2, 5)`，即将元素 2 的父节点置为元素 5，显然树的深度增加了 1，结果如下：



所以现在我们来看一道例题：

例 7.8.1 (洛谷 P1551-亲戚). 题目背景：若某个家族人员过于庞大，要判断两个是否是亲戚，确实还很不容易，现在给出某个亲戚关系图，求任意给出的两个人是否具有亲戚关系。

题目描述：规定： x 和 y 是亲戚， y 和 z 是亲戚，那么 x 和 z 也是亲戚。如果 x, y 是亲戚，那么 x 的亲戚都是 y 的亲戚， y 的亲戚也都是 x 的亲戚。

输入格式：第一行：三个整数 n, m, p , ($n \leq 5000, m \leq 5000, p \leq 5000$)，分别表示有 n 个人， m 个亲戚关系，询问 p 对亲戚关系。

以下 m 行：每行两个数 M_i, M_j , $1 \leq M_i, M_j \leq N$ ，表示 M_i 和 M_j 具有亲戚关系。

接下来 p 行：每行两个数 P_i, P_j ，询问 P_i 和 P_j 是否具有亲戚关系。

输出格式： P 行，每行一个'Yes'或'No'。

表示第 i 个询问的答案为“具有”或“不具有”亲戚关系。

例题7.8.1是一个常见的问题。我们可以建立模型，把所有人划分到若干个不相交的集合中，每个集合里的人彼此是亲戚。为了判断两个人是否为亲戚，只需看它们是否属于同一个集合即可，这也就是并查集结构的一种应用场景。

```
#include <cstdio>
#define MAXN 5005
int fa[MAXN], rank[MAXN];
inline void init(int n) {
    for (int i = 1; i <= n; ++i) {
        fa[i] = i;
        rank[i] = 1;
    }
}
int find(int x) {
    return x == fa[x] ? x : (fa[x] = find(fa[x]));
}
```

```
}

inline void merge(int i, int j) {
    int x = find(i), y = find(j);
    if (rank[x] <= rank[y])
        fa[x] = y;
    else
        fa[y] = x;
    if (rank[x] == rank[y] && x != y)
        rank[y]++;
}

int main() {
    int n, m, p, x, y;
    scanf("%d%d%d", &n, &m, &p);
    init(n);
    for (int i = 0; i < m; ++i) {
        scanf("%d%d", &x, &y);
        merge(x, y);
    }
    for (int i = 0; i < p; ++i) {
        scanf("%d%d", &x, &y);
        printf("%s\n", find(x) == find(y) ? "Yes" : "No");
    }
    return 0;
}
```

例 7.8.2 (NOIP 提高组 2017 年 D2T1 洛谷 P3958 奶酪). 题目描述: 现有一块大奶酪, 它的高度为 h , 它的长度和宽度我们可以认为是无限大的, 奶酪中间有许多半径相同的球形空洞。我们可以在这块奶酪中建立空间坐标系, 在坐标系中, 奶酪的下表面为 $z = 0$, 奶酪的上表面为 $z = h$ 。

现在, 奶酪的下表面有一只小老鼠 *Jerry*, 它知道奶酪中所有空洞的球心所在的坐标。如果两个空洞相切或是相交, 则 *Jerry* 可以从其中一个空洞

跑到另一个空洞，特别地，如果一个空洞与下表面相切或是相交，Jerry 则可以从奶酪下表面跑进空洞；如果一个空洞与上表面相切或是相交，Jerry 则可以从空洞跑到奶酪上表面。

位于奶酪下表面的 Jerry 想知道，在不破坏奶酪的情况下，能否利用已有的空洞跑到奶酪的上表面去？

空间内两点 $P_1(x_1, y_1, z_1)$ 、 $P_2(x_2, y_2, z_2)$ 的距离公式如下：

$$dist(P_1, P_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$$

输入格式：每个输入文件包含多组数据。

第一行，包含一个正整数 T ，代表该输入文件中所含的数据组数。

接下来是 T 组数据，每组数据的格式如下：

第一行包含三个正整数 n, h 和 r ，两个数之间以一个空格分开，分别代表奶酪中空洞的数量，奶酪的高度和空洞的半径。

接下来的 n 行，每行包含三个整数 x, y, z ，两个数之间以一个空格分开，表示空洞球心坐标为 (x, y, z) 。

```
3
2 4 1
0 0 1
0 0 3
2 5 1
0 0 1
0 0 4
2 5 2
0 0 2
2 0 4
```

输出格式： T 行，分别对应 T 组数据的答案，

如果在第 i 组数据中，Jerry 能从下表面跑到上表面，则输出 'Yes'，如果不能，则输出 'No' (均不包含引号)。

```
Yes
```

No

Yes

例题7.8.2同样是并查集的一个题目，我们把所有空洞划分为若干个集合，一旦两个空洞相交或相切，就把它们放到同一个集合中。

我们还可以划出2个特殊元素，分别表示**底部**和**顶部**，如果一个空洞与底部接触，则把它与表示底部的元素放在同一个集合中，顶部同理。最后，只需要看顶部和底部是不是在同一个集合中即。代码如下：

```
#include <cstdio>
#include <cstring>
#define MAXN 1005
typedef long long ll;
int fa[MAXN], rank[MAXN];
ll X[MAXN], Y[MAXN], Z[MAXN];
inline bool next_to(ll x1, ll y1, ll z1, ll x2, ll y2, ll z2
    , ll r) {
    return (x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2) + (
        z1 - z2) * (z1 - z2) <= 4 * r * r;
    //判断两个空洞是否相交或相切
}
inline void init(int n) {
    for (int i = 1; i <= n; ++i) {
        fa[i] = i;
        rank[i] = 1;
    }
}
int find(int x) {
    return x == fa[x] ? x : (fa[x] = find(fa[x]));
}
inline void merge(int i, int j) {
    int x = find(i), y = find(j);
    if (rank[x] <= rank[y])
```

```

    fa[x] = y;
else
    fa[y] = x;
if (rank[x] == rank[y] && x != y)
    rank[y]++;
}

int main() {
    int T, n, h;
    ll r;
    scanf("%d", &T);
    for (int I = 0; I < T; ++I) {
        memset(X, 0, sizeof(X));
        memset(Y, 0, sizeof(Y));
        memset(Z, 0, sizeof(Z));
        scanf("%lld%lld%lld", &n, &h, &r);
        init(n);
        fa[1001] = 1001; //用1001代表底部
        fa[1002] = 1002; //用1002代表顶部
        for (int i = 1; i <= n; ++i)
            scanf("%lld%lld%lld", X + i, Y + i, Z + i);
        for (int i = 1; i <= n; ++i) {
            if (Z[i] <= r)
                merge(i, 1001); //与底部接触的空洞与底部合并
            if (Z[i] + r >= h)
                merge(i, 1002); //与顶部接触的空洞与顶部合并
        }
        for (int i = 1; i <= n; ++i) {
            for (int j = i + 1; j <= n; ++j) {
                if (next_to(X[i], Y[i], Z[i], X[j], Y[j], Z[j], r))
                    merge(i, j); //遍历所有空洞，合并相交或
                                //相切的球
            }
        }
    }
}

```

```
    }
    printf("%s\n", find(1001) == find(1002) ? "Yes" : "No");
}
return 0;
}
```

总的来说，凡是涉及到元素的分组管理问题，都可以考虑使用并查集进行维护。

7.8.5 种类并查集

接下来我们来看并查集的一个拓展——种类并查集。一般的并查集维护的是具有连通性、传递性的关系，例如**亲戚的亲戚是亲戚**。但是，有时候，我们要维护另一种关系：**敌人的敌人是朋友**。种类并查集就是为了解决这个问题而设计的。

我们先来看一道例题：

例 7.8.3 (洛谷 P1525-关押罪犯). 题目描述： S 城现有两座监狱，一共关押着 N 名罪犯，编号分别为 1 到 N 。他们之间的关系自然也极不和谐。很多罪犯之间甚至积怨已久，如果客观条件具备则随时可能爆发冲突。我们用“怨气值”(一个正整数值)来表示某两名罪犯之间的仇恨程度，怨气值越大，则这两名罪犯之间的积怨越多。如果两名怨气值为 c 的罪犯被关押在同一监狱，他们俩之间会发生摩擦，并造成影响力为 c 的冲突事件。

每年年末，警察局会将本年内监狱中的所有冲突事件按影响力从大到小排成一个列表，然后上报到 S 城 Z 市长那里。公务繁忙的 Z 市长只会去看列表中的第一个事件的影响力，如果影响很坏，他就会考虑撤换警察局长。

在详细考察了 N 名罪犯间的矛盾关系后，警察局长觉得压力巨大。他准备将罪犯们在两座监狱内重新分配，以求产生的冲突事件影响力都较小，

从而保住自己的乌纱帽。假设只要处于同一监狱内的某两个罪犯间有仇恨，那么他们一定会在每年的某个时候发生摩擦。

那么，应如何分配罪犯，才能使 Z 市长看到的那个冲突事件的影响力最小？这个最小值是多少？

输入格式

每行中两个数之间用一个空格隔开。

第一行为两个正整数 N, M ，分别表示罪犯的数目以及存在仇恨的罪犯对数。

接下来的 M 行每行为三个正整数 a_i, b_i, c_i ，表示 a_i 号和 b_i 号罪犯之间存在仇恨，其怨气值为 c_i 。

其中 $1 < a_j \leq b_j \leq N$ ， $0 < c_j \leq 10^9$ ，且每对罪犯组合只出现一次。

输出格式

共 1 行，为 Z 市长看到的那个冲突事件的影响力。如果本年内监狱中未发生任何冲突事件，请输出 0。

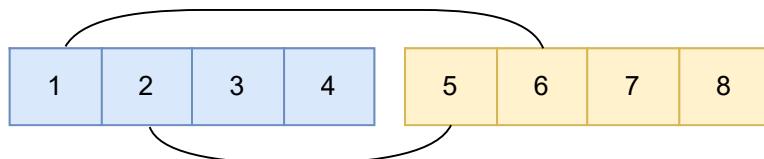
我们看例题7.8.3，很容易想到可以用**贪心算法**，首先把所有矛盾关系从大到小排个序，然后尽可能地把矛盾大的犯人关到不同的监狱里，直到不能这么做为止。这看上去可以用并查集维护，但是却存在一个问题：我们得到的信息，不是哪些人应该在相同的监狱，而是**哪些人应该在不同的监狱**。

我们先来看种类并查集，我们开一个两倍大小的并查集。例如，假如我们要维护 4 个元素的并查集，我们改为开 8 个单位的空间：

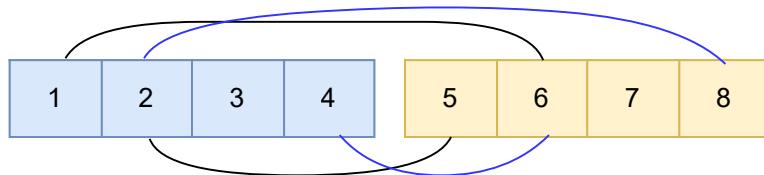


我们用 1-4 维护朋友关系，就这道题而言，是指关在同一个监狱的狱友，用 5-8 维护敌人关系，这道题里是指关在不同监狱的仇人。现在假如我们得到信息：1 和 2 是敌人，应该怎么办？

我们`merge(1, 2 + n), merge(1 + n, 2);`。这里`n`就等于 4，但我写成`n`这样更清晰：对于 1 个编号为`i`的元素，`i+n`是它的敌人。所以这里的意思就是：1 是 2 的敌人，2 是 1 的敌人。



现在我们令`merge(2, 4 + n), merge(2 + n, 4);`，也就是我们知道 2 和 4 是敌人：



也就是所谓敌人的敌人就是朋友，2 和 4 是敌人，2 和 1 也是敌人，所以这里，1 和 4 通过 $2+n$ 这个元素间接地连接起来了。这就是种类并查集工作的原理。

我们回到 7.8.3，AC 代码如下：

```
#include <cstdio>
#include <cctype>
#include <algorithm>

int read() { //快速读入，可忽略
    int ans = 0;
    char c = getchar();
    while (!isdigit(c))
        c = getchar();
    while (isdigit(c)) {
        ans = (ans << 3) + (ans << 1) + c - '0';
        c = getchar();
    }
}
```

```
    }

    return ans;
}

struct data { //以结构体方式保存便于排序
    int a, b, w;
} C[100005];

int cmp(data &a, data &b) {
    return a.w > b.w;
}

int fa[40005], rank[40005]; //以下为并查集
int find(int a) {
    return (fa[a] == a) ? a : (fa[a] = find(fa[a]));
}

int query(int a, int b) {
    return find(a) == find(b);
}

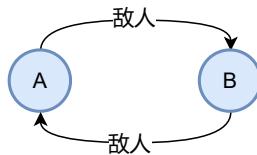
void merge(int a, int b) {
    int x = find(a), y = find(b);
    if (rank[x] >= rank[y])
        fa[y] = x;
    else
        fa[x] = y;
    if (rank[x] == rank[y] && x != y)
        rank[x]++;
}

void init(int n) {
    for (int i = 1; i <= n; ++i) {
```

```
    rank[i] = 1;
    fa[i] = i;
}
}

int main() {
    int n = read(), m = read();
    init(n * 2); //对于罪犯 i, i+n 为他的敌人
    for (int i = 0; i < m; ++i) {
        C[i].a = read();
        C[i].b = read();
        C[i].w = read();
    }
    std::sort(C, C + m, cmp);
    for (int i = 0; i < m; ++i) {
        if (query(C[i].a, C[i].b)) { //试图把两个已经被标记
            为“朋友”的人标记为“敌人”
            printf("%d\n", C[i].w); //此时的怒气值就是最大怒
            气值的最小值
            break;
        }
        merge(C[i].a, C[i].b + n);
        merge(C[i].b, C[i].a + n);
        if (i == m - 1) //如果循环结束仍无冲突，输出0
            puts("0");
    }
    return 0;
}
```

种类并查集可以维护敌人的敌人是朋友这样的关系，用更为具体的话来说，种类并查集（包括普通并查集）维护的是一种**循环对称**的关系。



所以如果是三个及以上的集合，只要每个集合都是等价的，且集合间的每个关系都是等价的，就能够用种类并查集进行维护。例如下面这道题：

例 7.8.4 (POJ P1182-食物链). 题目描述：动物王国中有三类动物 A, B, C, 这三类动物的食物链构成了有趣的环形。A 吃 B, B 吃 C, C 吃 A。

现有 N 个动物，以 1-N 编号。每个动物都是 A, B, C 中的一种，但是我们并不知道它到底是哪一种。

有人用两种说法对这 N 个动物所构成的食物链关系进行描述：

第一种说法是“1 X Y”，表示 X 和 Y 是同类。

第二种说法是“2 X Y”，表示 X 吃 Y。

此人对 N 个动物，用上述两种说法，一句接一句地说出 K 句话，这 K 句话有的是真的，有的是假的。当一句话满足下列三条之一时，这句话就是假话，否则就是真话。

- 当前的话与前面的某些真的话冲突，就是假话；
- 当前的话中 X 或 Y 比 N 大，就是假话；
- 当前的话表示 X 吃 X，就是假话。

你的任务是根据给定的 $N(1 \leq N \leq 50,000)$ 和 K 句话 ($0 \leq K \leq 100,000$)，输出假话的总数。

输入格式

第一行是两个整数 N 和 K，以一个空格分隔。

以下 K 行每行是三个正整数 D, X, Y，两数之间用一个空格隔开，其中 D 表示说法的种类。

若 $D=1$ ，则表示 X 和 Y 是同类；若 $D=2$ ，则表示 X 吃 Y。

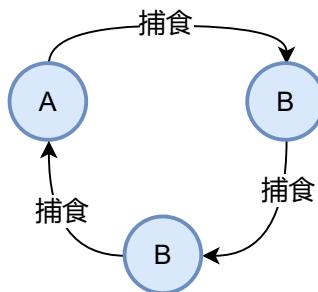
```
100 7  
1 101 1  
2 1 2  
2 2 3  
2 3 3  
1 1 3  
2 3 1  
1 5 5
```

输出格式

只有一个整数，表示假话的数目。

```
3
```

读过例题7.8.4，不难发现 A、B、C 三个种群天然地符合用种类并查集维护的要求：



于是我们可以用一个三倍大小的并查集进行维护，用 $i+n$ 表示 i 的捕食对象，而 $i+2n$ 表示 i 的天敌，AC 代码如下：

```
#include <cstdio>  
#include <cctype>  
  
int read() {  
    int ans = 0;  
    char c = getchar();
```

```
while (!isdigit(c))
    c = getchar();
while (isdigit(c)) {
    ans = (ans << 3) + (ans << 1) + c - '0';
    c = getchar();
}
return ans;
}

int fa[150005], rank[150005];

int find(int a) {
    return (fa[a] == a) ? a : (fa[a] = find(fa[a]));
}

int query(int a, int b) {
    return find(a) == find(b);
}

void merge(int a, int b) {
    int x = find(a), y = find(b);
    if (rank[x] >= rank[y])
        fa[y] = x;
    else
        fa[x] = y;
    if (rank[x] == rank[y] && x != y)
        rank[x]++;
}

void init(int n) {
    for (int i = 1; i <= n; ++i) {
        rank[i] = 1;
        fa[i] = i;
    }
}
```

```
}

}

int main() {
    int n = read(), m = read(), ans = 0;
    init(n * 3); //i吃i+n, 被i+2n吃
    for (int i = 0; i < m; ++i) {
        int opr, x, y;
        scanf("%d%d%d", &opr, &x, &y);
        if (x > n || y > n) //特判x或y不在食物链中的情况
            ans++;
        else if (opr == 1) {
            if (query(x, y + n) || query(x, y + 2 * n)) //如果已知x吃y, 或者x被y吃, 说明这是假话
                ans++;
            else {
                merge(x, y); //这是真话, 则x和y是一族
                merge(x + n, y + n); //x的猎物和y的猎物是一族
                merge(x + 2 * n, y + 2 * n); //x的天敌和y的天敌是一族
            }
        } else if (opr == 2) {
            if (query(x, y) || query(x, y + 2 * n)) //如果已知x与y是一族, 或者x被y吃, 说明这是假话
                ans++;
            else {
                merge(x, y + n); //这是真话, 则x吃y
                merge(x + n, y + 2 * n); //x的猎物吃y的猎物
                merge(x + 2 * n, y); //x的天敌吃y的天敌, 或者说y吃x的天敌
            }
        }
    }
}
```

```

    }
}

printf("%d\n", ans);
return 0;
}

```

7.9 树状数组

7.9.1 树状数组的引入

树状数组，即二进制下标树 (Binary Index Tree, BIT)，也是简洁优美的数据结构之一。最简单的树状数组需要支持两种操作：

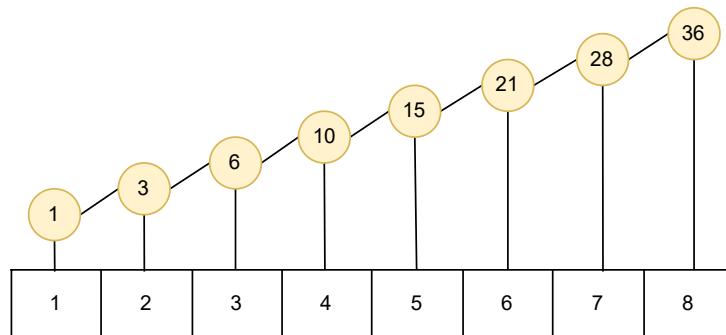
- **单点修改**：更改数组中一个元素的值；
- **区间查询**：查询一个区间内所有元素的和。

且两者的时间复杂度均为 $O(\log n)$ 。

我们先来看普通数组：

1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---

普通数组的**单点修改**的时间复杂度是 $O(1)$ 但**区间求和**的时间复杂度是 $O(n)$ ；同样我们也可以用**前缀和**的方式维护这个数组，具体如下：



这样的区间求和时间复杂度是 $O(1)$ ，但是单点修改会影响后面所有的元素，时间复杂度是 $O(n)$ 。

显然，程序最后跑多长时间，是由最慢的一环决定的，因此现在我们希望找到这样一种折中的方法：无论单点修改还是区间查询，它都能不那么慢地完成。

我们来看前缀和，如果我们要对区间 $[3, 5]$ 进行区间查询，我们只需要查询 $[1, 3)$ 和 $[1, 5]$ 区间随后相减即可，所以，当我们想要对区间 $[a, b]$ 进行区间查询时，我们只需要查询 $[1, a)$ 和 $[1, b]$ 再相减即可，因而区间查询问题也就转化为了求前 n 项和的问题。

现在我们试图寻找一种结构，一方面，单点修改时需要更新的区间不会太多；另一方面，区间查询时需要用来组合的区间也不会太多。

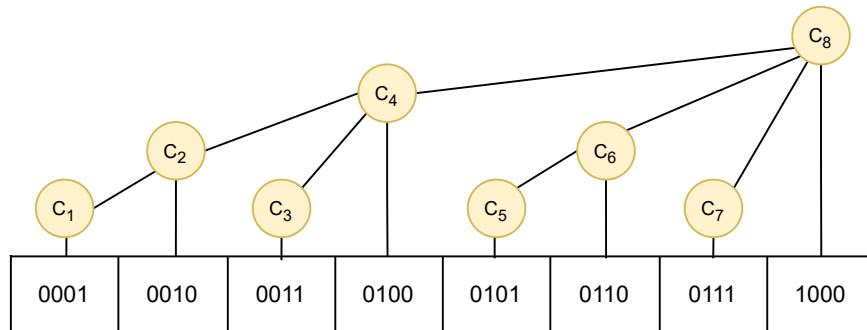
树状数组就是这样一种利用了二进制的结构，例如我们来看 11，它的二进制为 $(1011)_2$ ，如果我们要求前 11 项的和，可以分别查询 $((0000)_2, (1000)_2]$ 即 $(0, 8]$ 、 $((1000)_2, (1010)_2]$ 即 $(8, 10]$ ，以及 $((1010)_2, (1011)_2]$ 即 $(10, 11]$ 的区间和再相加。这个区间是如何得到的？我们每次都是去掉二进制数最右边的 1，具体如下：

$$(1010, \underline{1011}] \rightarrow (10, 11]$$

$$(1000, \underline{1010}] \rightarrow (8, 10]$$

$$(0000, \underline{1000}] \rightarrow (0, 8]$$

我们定义二进制数最右边的一个 1，连带着它之后的 0 为 $\text{lowbit}(x)$ 。那么我们用 C_i 维护区间 $(A_i - \text{lowbit}(A_i), A_i]$ ，这样显然查询前 n 项和时需要合并的区间数是少于 $\log_2 n$ 的。树状数组的结构大概像下面这样：



树状数组的更新就是一个“爬树”的过程，从改变的位置一路往上更新，直到 MAXN (也就是树状数组的容量)。

我们举个例子来看看这树是怎么爬的。现有 38 的二进制数 $(100110)_2$ ，包含它的最小区间是 $((100100)_2, (100110)_2]$ ，接下来，我们不难发现这个区间必然位于 $((1000000)_2, (101000)_2]$ 中，随后是 $((1000000)_2, (110000)_2]$ ，再然后就是 $(0, (1000000)_2]$ 。这个过程如下所示：

$$100\ 100 < 100\ 110 \leq 100\ \underline{1}10 \rightarrow (36, 38]$$

$$100\ 000 < 100\ 110 \leq 10\underline{1}\ 000 \rightarrow (32, 40]$$

$$100\ 000 < 100\ 110 \leq \underline{1}10\ 000 \rightarrow (32, 48]$$

$$000\ 000 < 100\ 110 \leq 1\ 000\ 000 \rightarrow (0, 64]$$

如上图，每一步都把从右边起一系列连续的 1 变为 0，再把这一系列 1 的前一位 0 变为 1。这样一来每一次加的值正是一个 $\text{lowbit}(x)$ ，所以，我们更新的区间数不会超过 $\log_2 \text{MAXN}$ ，我们也就实现一个单点修改和区间查询时间复杂度为 $O(\log n)$ 的数据结构。

7.9.2 树状数组的实现

在实现树状数组之前，我们需要解决 $\text{lowbit}(x)$ 的计算，如果我们选择转换为二进制再按位验证，不免太过复杂。我们可以用另外一种公式进行

计算：

$$\text{lowbit}(x) = (x) \& (-x)$$

我们知道计算机里有符号数一般是以**补码**的形式存储的， $-x$ 相当于 x 按位取反再加 1，会把结尾处原来 $1000\cdots$ 的形式，变成 $0111\cdots$ ，再变成 $1000\cdots$ ，而前面每一位都与原来相反。下面是两个例子：

(1)	(2)
0100 1010	0010 1000
反 1011 0101	反 1101 0111
补 1011 0110	补 1101 1000

这里根据图中的例子，我们再来复习一下计算机中的**原码**、**反码**和**补码**，一个数在计算机中是以**二进制**的形式表示的，称之为**机器数**。机器数是带**符号的**，在计算机用一个数的最高位存放**符号**，**正数为 0**，**负数为 1**。例如：十进制中的数 +7，计算机字长为 8 位，转换成二进制就是 0000 0111；如果是-7，就是 1000 0111。

- **原码：**数的二进制表示，若有符号，则最高位表示符号位，0 代表正，1 代表负。
 - 无符号位 $\in [0, 2^n - 1]$ ，例如 8 位的情况下，取值为 $[0, 255]$ ，即 $[0000\ 0000, 1111\ 1111]$ 。
 - 有符号位 $\in [-2^{n-1} - 1, 2^{n-1} - 1]$ ，例如 8 位的情况下，取值为 $[-127, 127]$ ，即 $[1111\ 1111, 0111\ 1111]$ 。
- **反码：**正数的反码就是原码本身，负数的反码则是**符号位不变，其他各位取反**。

- 补码：正数的补码还是原码本身，负数的补码依旧是符号位不变，其余各位取反，然后最低为加 1。

另一方面，逻辑运算中的与、或和异或运算也是二进制运算：

- 逻辑与：只有两位同时为 1 时才为 1，其余为 0，类似于乘法；
- 逻辑或：只有两位同时为 0 时才为 0，其余为 1，类似于加法；
- 逻辑异或：两位相同则为 0，两位不同则为 1。

我们再把 $(-x)$ 和 x 按位与，得到的结果便是 `lowbit(x)`，例如上述的 38，其二进制为 $(0010\ 0110)_2$ ，那么 -38 的二进制为 $(1101\ 1010)_2$ ，其运算过程大致如下：

0010 0110	0010 0110
1101 1001	1101 1010
1101 1010	0000 0010

也就是 `lowbit(38) == 2`，然后继续执行这个过程：

0010 1000	0010 1000
1101 0111	1101 1000
1101 1000	0000 1000

也就是 `lowbit(40) == 8`，接下来就是：

0011 0000	0011 0000
1100 1111	1101 0000
1101 0000	0001 0000

即 $\text{lowbit}(48) == 16$ 。现在我们有了理论基础，可以在此之上进行树状数组的设计：

单点修改

```
int tree[MAXN];
inline void update(int i, int x) {
    for (int pos = i; pos < MAXN; pos += lowbit(pos)) {
        tree[pos] += x;
    }
}
```

求前 n 项和

```
inline int query(int n) {
    int ans = 0;
    for (int pos = n; pos; pos -= lowbit(pos)) {
        ans += tree[pos];
    }
    return ans;
}
```

区间查询

```
inline int query(int a, int b) {
    return query(b) - query(a - 1);
}
```

照例，我们来看一道例题：

例 7.9.1 ((HDU P1166)-敌兵布阵). *Problem Description:* C 国的死对头 A 国这段时间正在进行军事演习，所以 C 国间谍头子 Derek 和他手下

Tidy 又开始忙乎了。*A* 国在海岸线沿直线布置了 N 个工兵营地, *Derek* 和 *Tidy* 的任务就是要监视这些工兵营地的活动情况。由于采取了某种先进的监测手段, 所以每个工兵营地的人数 *C* 国都掌握的一清二楚, 每个工兵营地的人数都有可能发生变动, 可能增加或减少若干人手, 但这些都逃不过 *C* 国的监视。

中央情报局要研究敌人究竟演习什么战术, 所以 *Tidy* 要随时向 *Derek* 汇报某一段连续的工兵营地一共有多少人, 例如 *Derek* 问: “*Tidy*, 马上汇报第 3 个营地到第 10 个营地共有多少人!” *Tidy* 就要马上开始计算这一段的总人数并汇报。但敌兵营地的人数经常变动, 而 *Derek* 每次询问的段都不一样, 所以 *Tidy* 不得不每次都一个一个营地的去数, 很快就精疲力尽了, *Derek* 对 *Tidy* 的计算速度越来越不满: “你个死肥仔, 算得这么慢, 我炒你鱿鱼!” *Tidy* 想: “你自己来算算看, 这可真是一项累人的工作! 我恨不得你炒我鱿鱼呢!” 无奈之下, *Tidy* 只好打电话向计算机专家 *Windbreaker* 求救, *Windbreaker* 说: “死肥仔, 叫你平时做多点 *acm* 题和看多点算法书, 现在尝到苦果了吧!” *Tidy* 说: “我知错了……” 但 *Windbreaker* 已经挂掉电话了。*Tidy* 很苦恼, 这么算他真的会崩溃的, 聪明的读者, 你能写个程序帮他完成这项工作吗? 不过如果你的程序效率不够高的话, *Tidy* 还是会受到 *Derek* 的责骂的。

输入格式:

第一行一个整数 T , 表示有 T 组数据。

每组数据第一行一个正整数 N ($N \leq 50000$), 表示敌人有 N 个工兵营地, 接下来有 N 个正整数, 第 i 个正整数 a_i 代表第 i 个工兵营地里开始时有 a_i 个人 ($1 \leq a_i \leq 50$)。

接下来每行有一条命令, 命令有 4 种形式:

- **Add i j** , i 和 j 为正整数, 表示第 i 个营地增加 j 个人 (j 不超过 30);
- **Sub i j** , i 和 j 为正整数, 表示第 i 个营地减少 j 个人 (j 不超过 30);

- Query i j, i 和 j 为正整数, $i \leq j$, 表示询问第 i 到第 j 个营地的总人数;
- End 表示结束, 这条命令在每组数据最后出现;

每组数据最多有 40000 条命令。

输出格式

对第 i 组数据, 首先输出“Case i:”和回车,

对于每个 Query 询问, 输出一个整数并回车, 表示询问的段中的总人数, 这个数保持在 int 以内。

```
#include <cstdio>
#include <cstring>

#define MAXN 50005
#define lowbit(x) ((x) & (-x))
int tree[MAXN];

inline void update(int i, int x) {
    for (int pos = i; pos < MAXN; pos += lowbit(pos))
        tree[pos] += x;
}

inline int query(int n) {
    int ans = 0;
    for (int pos = n; pos; pos -= lowbit(pos))
        ans += tree[pos];
    return ans;
}

inline int query(int a, int b) {
    return query(b) - query(a - 1);
}
```

```
int main() {
    int cases;
    scanf("%d", &cases);
    for (int I = 1; I <= cases; ++I) {
        memset(tree, 0, sizeof(tree));
        int n, x, a, b;
        char opr[10];
        printf("Case %d:\n", I);
        scanf("%d", &n);
        for (int i = 1; i <= n; ++i) {
            scanf("%d", &x);
            update(i, x);
        }
        while (scanf("%s", opr), opr[0] != 'E') {
            switch (opr[0]) {
                case 'A':
                    scanf("%d%d", &a, &b);
                    update(a, b);
                    break;
                case 'S':
                    scanf("%d%d", &a, &b);
                    update(a, -b);
                    break;
                case 'Q':
                    scanf("%d%d", &a, &b);
                    printf("%d\n", query(a, b));
            }
        }
    }
    return 0;
}
```

7.10 ST 表

ST 表 (Sparse Table, 稀疏表) 是一种简单数据结构，主要用于解决**RMQ**(Range Maximum/Minimum Query, 区间最大值/最小值查询) 问题¹，利用**倍增思想**，可以实现 $O(n \log n)$ 预处理和 $O(1)$ 查询。

ST 表通过使用二维数组 $f[] []$ ，对范围内的所有 $f[a][b]$ ，先算出并存储 $\max_{i \in [a, a+2^b]} A_i$ ²，这个过程称为**预处理**，查询时再利用这些子区间算出待求区间的最大值。

我们为了减少预处理的时间复杂度，这里选择使用**动态规划**的思想设计程序：

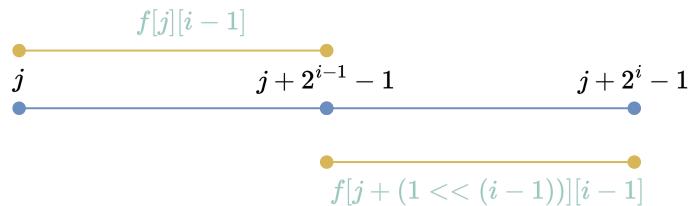
```

int f[MAXN][21];      // 第二维的大小根据数据范围决定，不小于
                      log(MAXN)
for(int i = 1; i <= n; ++i)
    f[i][0] = read();
for(int i = 1; i <= 20; ++i)
    for(int j = 1; j + (i << 1) - 1 <= n; ++j)
        f[j][i] = max(f[j][i - 1], f[j + (1 << (i - 1))][i - 1]);

```

原理如下：

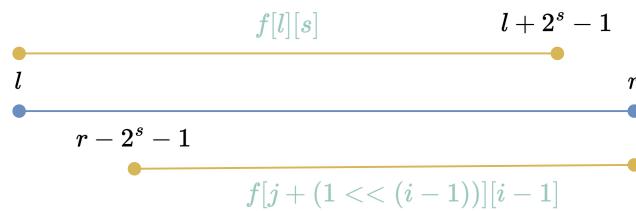
将区间平均分为两部分，前半段按定义是 $[j, j + 2^{i-1} - 1]$ 中的最大值，后半段则是 $[j + 2^{i-1} - 1, j + 2^i - 1]$ 中的最大值。



¹以最大值为例，给定一个数列 A 和区间 $[l, r]$ ，求 $\max_{i \in [l, r]} A_i$ 。

²在这里举出的例子都只包含整数，因此区间有时候也写成 $[a, a + 2^b - 1]$ 。

查询时，我们需要找到两个 $[l, r]$ 的子区间，它们的并集是 $[l, r]$ ，具体来说就是我们要找到一个整数 s ，使得两个子区间分别为 $[l, l + 2^s - 1]$ 和 $[r - 2^s + 1, r]$ 。原理如下：



我们希望前一个子区间的右端点尽可能接近 r ，即当 $l + 2^s - 1 = r$ 时，有 $s = \log_2(r - l + 1)$ ，此时 $r - 2^s - 1 = l$ ，且由于 $s \in Z$ ，所以我们取 $s = \lfloor \log_2(r - l + 1) \rfloor$ ，这两个子区间可以覆盖 $[l, r]$ 。

这样需要我们每次计算对数，于是我们可以对 \log 也进行一次递推的预处理：

```
for(int i = 2; i <= n; i++)
    Log2[i] = Log2[i / 2] + 1;
```

查询的代码如下：

```
for(int i = 0; i < m; i++) {
    int l = read(), r = read();
    int s = Log2[r - l + 1];
    printf("%d\n", max(f[l][s], f[r - (1 << s) + 1][s]));
}
```

ST 表除常用的最大值最小值处理外，凡是满足**结合律且可重复贡献**³的信息查询都可以使用 ST 表高效进行。显然最大值、最小值、最大公因数、最小公倍数、按位或、按位与都符合这个条件。可重复贡献的意义在于，可以对两个交集不为空的区间进行信息合并。

最后来看一道例题：

³假设二元运算 $f(x, y)$ 满足 $f(a, a) = a$ ，则称 f 是可重复贡献的。

例 7.10.1 (洛谷 P2880 [USACO07JAN] 平衡的阵容 Balanced Lineup). 每天, 农夫 John 的 $N(1 \leq N \leq 50,000)$ 头牛总是按同一序列排队。有一天, John 决定让一些牛们玩一场飞盘比赛。他准备找一群在对列中为置连续的牛来进行比赛。但是为了避免水平悬殊, 牛的身高不应该相差太大。

John 准备了 $Q(1 \leq Q \leq 200,000)$ 个可能的牛的选择和所有牛的身高 ($1 \leq \text{身高} \leq 1,000,000$)。他想知道每一组里面最高和最低的牛的身高差别。

输入格式:

第 1 行两个数 n, q

接下来 n 行, 每行一个数 h_i 代表每头牛的身高

再接下来 q 行, 每行两个整数 a 和 b , 表示询问第 a 头牛到第 b 头牛里的最高和最低的牛的身高差 ($1 \leq A \leq B \leq N$)

```
6 3  
1  
7  
3  
4  
2  
5  
1 5  
4 6  
2 2
```

输出格式:

输出每行一个数, 为最大数与最小数的差

```
6  
3  
0
```

这题属于 ST 表的模板题:

```
#include <bits/stdc++.h>
#define MAXN 50005
using namespace std;
int read() {
    int ans = 0;
    char c = getchar();
    while (!isdigit(c))
        c = getchar();
    while (isdigit(c)) {
        ans = ans * 10 + c - '0';
        c = getchar();
    }
    return ans;
}
int Log2[MAXN], Min[MAXN][17], Max[MAXN][17];
int main() {
    int n = read(), m = read();
    for (int i = 1; i <= n; ++i) {
        int x = read();
        Min[i][0] = x;
        Max[i][0] = x;
    }
    for (int i = 2; i <= n; ++i)
        Log2[i] = Log2[i / 2] + 1;
    for (int i = 1; i <= 16; ++i)
        for (int j = 1; j + (1 << i) - 1 <= n; ++j) {
            Min[j][i] = min(Min[j][i - 1], Min[j + (1 << (i - 1))][i - 1]);
            Max[j][i] = max(Max[j][i - 1], Max[j + (1 << (i - 1))][i - 1]);
        }
    for (int i = 0; i < m; ++i)
```

```

int l = read(), r = read();
int s = Log2[r - l + 1];
int ma = max(Max[l][s], Max[r - (1 << s) + 1][s]);
int mi = min(Min[l][s], Min[r - (1 << s) + 1][s]);
printf("%d\n", ma - mi);
}
return 0;
}

```

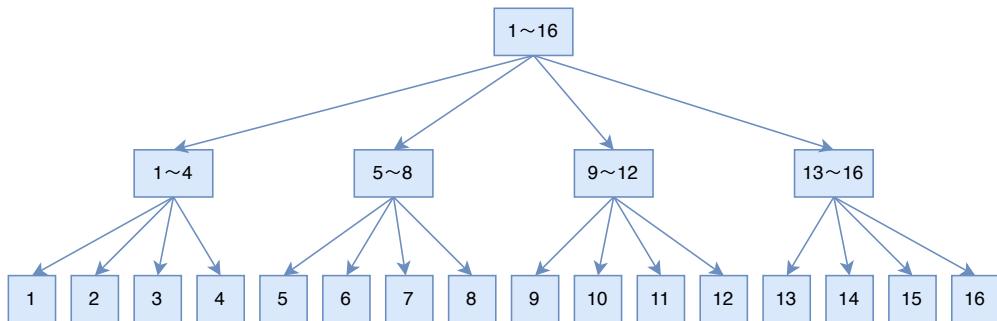
7.11 分块

分块是一种思想，其将一个整体划分为若干小块，对整块整体处理，对零散块单独处理。这里我们重点关注**块状数组**，块状数组是利用分块思想处理区间问题的一种数据结构。

块状数组将一个长度为 n 的数组划分为 a 块，则每块的长度为 $\frac{n}{a}$ 。对于一次区间操作，对区间内部的整块进行整体的操作，对区间边缘的零散块单独暴力处理。

显然分块的块数不能太多也不能太少，如果太少，区间中整块的数量会很少，我们要花费大量时间处理零散块；如果太多，又会让块的长度太短，失去整体处理的意义。通常来说，我们取块数为 \sqrt{n} ，这样在最坏情况下，要处理的块数接近 \sqrt{n} 个整块，还要对长度为 $\frac{2n}{\sqrt{n}} = 2\sqrt{n}$ 的零散块单独处理，总的时间复杂度为 $O(\sqrt{n})$ 。

分块的时间复杂度比不上诸如线段树和树状数组等对数级算法。但由此换来的是更高的灵活性。与线段树不同，块状数组并不要求所维护信息满足结合律，也不需要一层层地传递标记。但它们又有相似之处，线段树是一棵高度约为 $\log_2 n$ 的树，而块状数组则可以被看作一棵高度为 3 的树，不同的是，块状数组最顶层的信息不用维护：



预处理

为具体地使用块状数组，我们要先划定出每个块所占据的范围：

```

int sq = sqrt(n);
for (int i = 1; i <= sq; ++i) {
    st[i] = n / sq * (i - 1) + 1; // st[i] 表示 i 号块的第一个
                                    // 元素的下标
    ed[i] = n / sq * i; // ed[i] 表示 i 号块的最后一个元素的下
                        // 标
}
  
```

但是，数组的长度并不一定是一个完全平方数，所以这样下来很可能会漏掉一小块，我们把它们纳入最后一块中：

```
ed[sq] = n;
```

然后，我们为每个元素确定它所归属的块：

```

for (int i = 1; i <= sq; ++i)
    for (int j = st[i]; j <= ed[i]; ++j)
        bel[j] = i; // 表示 j 号元素归属于 i 块
  
```

在完成准备工作后，事情就很简单了，分块的代码量也许不比线段树小多少，但看起来要好理解很多，我们先来搞线段树模板题：

例 7.11.1 (洛谷-P3372 【模板】线段树 1). 如题，已知一个数列，你需要进行下面两种操作：

1. 将某区间每一个数加上 k 。
2. 求出某区间每一个数的和。

输入格式:

第一行包含两个整数 n, m , 分别表示该数列数字的个数和操作的总个数。

第二行包含 n 个用空格分隔的整数, 其中第 i 个数字表示数列第 i 项的初始值。

接下来 m 行每行包含 3 或 4 个整数, 表示一个操作, 具体如下:

1. 1 x y k : 将区间 $[x, y]$ 内每个数加上 k 。2. 2 x y : 输出区间 $[x, y]$ 内每个数的和。

```
5 5
1 5 4 2 3
2 2 4
1 2 3 2
2 3 4
1 1 5 1
2 1 4
```

输出格式:

输出包含若干行整数, 即为所有操作 2 的结果。

```
11
8
20
```

这个题数据范围只有 10^5 , 可以考虑用分块, 我们用一个 `sum` 数组来记录每一块的和, `mark` 数组来做标记。

读入和预处理数据

```
for (int i = 1; i <= n; ++i)
```

```

A[i] = read();
for (int i = 1; i <= sq; ++i)
    for (int j = st[i]; j <= ed[i]; ++j)
        sum[i] += A[j];

```

`sum[i]`保存第 i 个块的和。

区间修改

首先是区间修改,当 x 与 y 在同一块内时,直接暴力修改原数组和`sum`数组:

```

if (bel[x] == bel[y])
    for (int i = x; i <= y; ++i) {
        A[i] += k;
        sum[bel[i]] += k;
    }

```

否则,先暴力修改左右两边的零散区间:

```

for (int i = x; i <= ed[bel[x]]; ++i) {
    A[i] += k;
    sum[bel[i]] += k;
}
for (int i = st[bel[y]]; i <= y; ++i) {
    A[i] += k;
    sum[bel[i]] += k;
}

```

然后对中间的整块打上标记:

```

for (int i = bel[x] + 1; i < bel[y]; ++i)
    mark[i] += k;

```

区间查询

同样地，如果左右两边在同一块，直接暴力计算区间和。

```
if (bel[x] == bel[y])
    for (int i = x; i <= y; ++i)
        s += A[i] + mark[bel[i]]; // 注意要加上标记
```

否则，暴力计算零碎块：

```
for (int i = x; i <= ed[bel[x]]; ++i)
    s += A[i] + mark[bel[i]];
for (int i = st[bel[y]]; i <= y; ++i)
    s += A[i] + mark[bel[i]];
```

再处理整块：

```
for (int i = bel[x] + 1; i < bel[y]; ++i)
    s += sum[i] + mark[i] * size[i]; // 注意标记要乘上块长
```

于是我们用分块 A 掉了线段树的模板题，洛谷上跑得还挺快，比较数据范围很小。

但是，通常来说分块的题目通常很难也很灵活，我们以相对简单的一道题来结束分块的讲解，更多的技巧还需要勤加练习：

例 7.11.2 (洛谷-P2801 教主的魔法). 教主最近学会了一种神奇的魔法，能够使人长高。于是他准备演示给 $XMYZ$ 信息组每个英雄看。于是 N 个英雄们又一次聚集在了一起，这次他们排成了一列，被编号为 $1, 2, \dots, N$ 。

每个人的身高一开始都是不超过 1000 的正整数。教主的魔法每次可以把闭区间 $[L, R]$ ($1 \leq L \leq R \leq N$) 内的英雄的身高全部加上一个整数 W 。(虽然 $L = R$ 时并不符合区间的书写规范，但我们可以认为是单独增加第 $L(R)$ 个英雄的身高)

CYZ 、光哥和 ZJQ 等人不信教主的邪，于是他们有时候会问 WD 闭区间 $[L, R]$ 内有多少英雄身高大于等于 C ，以验证教主的魔法是否真的有效。

WD 巨懒，于是他把这个回答的任务交给了你。

输入格式：

第 1 行为两个整数 N, Q 。 Q 为问题数与教主的施法数总和。

第 2 行有 N 个正整数，第 i 个数代表第 i 个英雄的身高。

第 3 到第 $Q + 2$ 行每行有一个操作：

1. 若第一个字母为M，则紧接着有三个数字 L, R, W 。表示对闭区间 $[L, R]$ 内所有英雄的身高加上 W 。
2. 若第一个字母为A，则紧接着有三个数字 L, R, C 。询问闭区间 $[L, R]$ 内有多少英雄的身高大于等于 C 。

```
5 3
```

```
1 2 3 4 5
```

```
A 1 5 4
```

```
M 3 5 1
```

```
A 1 5 4
```

输出格式：

对每个A询问输出一行，仅含一个整数，表示闭区间 $[L, R]$ 内身高大于等于 C 的英雄数。

```
2
```

```
3
```

输入输出样例说明：原先 5 个英雄身高为 1, 2, 3, 4, 5，此时 $[1, 5]$ 间有 2 个英雄的身高大于等于 4。教主施法后变为 1, 2, 4, 5, 6，此时 $[1, 5]$ 间有 3 个英雄的身高大于等于 4。

区间加，询问区间大于等于 C 的元素个数，这个用线段树显然不方便，

因为对每个 C 开一棵线段树代价较大，我们发现虽然 N 最大为 10^6 ，但 Q 较小，分块可行。

零散块暴力处理起来显然很容易，问题在于如何对整块整体进行处理。实际上我们可以维护整块排序后的数组，然后对整块进行询问时直接**二分查找**。

注意，我们每次对零散块单独修改时，都需要更新排序后的数组，这听起来很暴力，但由于每个块相对较少，也可以接受。总的时间复杂度为 $O(q\sqrt{n} \log \sqrt{n})$ 。

```
/* ... */
const int MAXN = 1000005, SQ = 1005;
int st[SQ], ed[SQ], size[SQ], bel[MAXN];
void init_block(int n) { // 初始化
    int sq = sqrt(n);
    for (int i = 1; i <= sq; ++i) {
        st[i] = n / sq * (i - 1) + 1;
        ed[i] = n / sq * i;
    }
    ed[sq] = n;
    for (int i = 1; i <= sq; ++i)
        for (int j = st[i]; j <= ed[i]; ++j)
            bel[j] = i;
    for (int i = 1; i <= sq; ++i)
        size[i] = ed[i] - st[i] + 1;
}
int A[MAXN], mark[SQ];
vector<int> v[SQ]; // 这里用 vector 存排序后的数组
void update(int b) { // 更新排序后的数组
    for (int i = 0; i <= size[b]; ++i)
        v[b][i] = A[st[b] + i];
    sort(v[b].begin(), v[b].end());
}
```

```
int main() {
    int n = read(), m = read();
    int sq = sqrt(n);
    init_block(n);
    for (int i = 1; i <= n; ++i)
        A[i] = read();
    for (int i = 1; i <= sq; ++i)
        for (int j = st[i]; j <= ed[i]; ++j)
            v[i].push_back(A[j]);
    for (int i = 1; i <= sq; ++i)
        sort(v[i].begin(), v[i].end());
    while (m--) {
        char o;
        scanf(" %c", &o);
        int l = read(), r = read(), k = read();
        if (o == 'M') {
            if (bel[l] == bel[r]) { // 如果同属一块直接暴力
                for (int i = l; i <= r; ++i)
                    A[i] += k;
                update(bel[l]);
                continue;
            }
            for (int i = l; i <= ed[bel[l]]; ++i) // 对零散
                块暴力处理
                A[i] += k;
            for (int i = st[bel[r]]; i <= r; ++i)
                A[i] += k;
            update(bel[l]);
            update(bel[r]);
            for (int i = bel[l] + 1; i < bel[r]; ++i) // 打
                上标记
                mark[i] += k;
        }
    }
}
```

```
else {
    int tot = 0;
    if (bel[l] == bel[r]) {
        for (int i = l; i <= r; ++i)
            if (A[i] + mark[bel[l]] >= k)
                tot++;
        printf("%d\n", tot);
        continue;
    }
    for (int i = l; i <= ed[bel[l]]; ++i)
        if (A[i] + mark[bel[l]] >= k)
            tot++;
    for (int i = st[bel[r]]; i <= r; ++i)
        if (A[i] + mark[bel[r]] >= k)
            tot++;
    // 二分查找  $k - mark[i]$  的位置，因为整块都加上了  $mark[i]$ 
    // 其实就相当于  $k$  减去  $mark[i]$ 
    for (int i = bel[l] + 1; i < bel[r]; ++i)
        tot += v[i].end() - lower_bound(v[i].begin()
                                         , v[i].end(), k - mark[i]);
    printf("%d\n", tot);
}
return 0;
}
```

第八章 字符串

8.1 KMP 算法

KMP 算法 (全称 Knuth-Morris-Pratt 字符串查找算法, 由三位发明者的姓氏命名) 是可以在**文本串s**中快速查找**模式串p**的一种算法。

我们首先来看使用暴力匹配时的流程：所谓暴力匹配，即逐字符进行匹配 (比较 $s[i]$ 和 $p[j]$)，若当前字符匹配成功 ($s[i] == p[j]$)，就匹配下一个字符 ($++i, ++j$)，若失败， i 回溯， j 置为 0 ($i = i - j + 1, j = 0$)。具体如下所示：

```
// 暴力匹配
int i = 0, j = 0;
while (i < s.length()) {
    if (s[i] == p[j])
        ++i, ++j;
    else
        i = i - j + 1, j = 0;
    if (j == p.length()) { // 匹配成功
        cout << i - j << endl;
        i = i - j + 1;
        j = 0;
    }
}
```

我们根据代码来看一个例子，例如我们的文本串 $s = "abababcabaa"$ ，模式

串 $p = "ababcabaa"$:

a	b	a	b	c	a	b	a	a
a	b	a	b	c	a	b	a	a

从头开始匹配，第 1 个字符匹配成功，于是继续匹配，第 2-4 个字符也成功匹配：

a	b	a	b	a	b	c	a	b	a	a
a	b	a	b	c	a	b	a	a		

但我们发现下一位匹配失败，' $'a'$!= ' $'c'$ '，于是我们回溯：

a	b	a	b	c	a	b	a	a
a	b	a	b	c	a	b	a	a

匹配依然失败，继续回溯：

a	b	a	b	c	a	b	a	a
a	b	a	b	c	a	b	a	a

发现匹配成功，继续匹配：

a	b	a	b	c	a	b	a	a
a	b	a	b	c	a	b	a	a

就这样我们直到最后一位完成匹配。

设两个字符串的长度分别为 n 和 m ，则暴力匹配的最坏时间复杂度是 $O(nm)$ 。其根本原因在于 i 需要进行回溯，因此浪费了时间。然而，如果 i 不回溯，同时又把 j 置为 0，很可能会出现缺漏，如下图：

a	b	a	b	a	b	c	a	b	a	a

为了让j被赋予一个合适的值，我们引入了 **PMT**(Partial Match Table, 部分匹配表)。

对于j应该被赋值为多少，是只与模式串自身有关的。每个模式串，都对应着一张 PMT，例如"ababcabaa"对应的 PMT 如下：

	0	1	2	3	4	5	6	7	8
p	a	b	a	b	c	a	b	a	a
pmt	0	0	1	2	0	1	2	3	1

我们来看这张表，其中 $\text{pmt}[i]$ 代表的是从 $p[0]$ 开始到 $p[i]$ 为止子串的 **最长公共前后缀** 的长度。值得一提的是子串本身**不属于**前缀和后缀。

那么，我们就可以用 PMT 来确定j指针的位置，例如我们回到暴力算法第一次失败的情形：

a	b	<u>a</u>	<u>b</u>	a	b	c	a	b	a	a
a	<u>b</u>	a	b	c	a	b	a	a		

因为文本串s中的'a'字符和模式串p中的'c'字符没能正确匹配，我们需要在保证指针i不变的情况下，把指针j左移，但我们发现"abab"其实已经匹配成功的，这个子串拥有一个前缀"ab"和一个后缀"ab"，所以我们可以利用这个公共的前后缀"ab"，这个过程中我们令 $j = \text{pmt}[j - 1]$ ，具体如下所示：

a	b	<u>a</u>	<u>b</u>	a	b	c	a	b	a	a
a	<u>b</u>	a	b	c	a	b	a	a		

我们再来看一个例子，在这里我们令文本串 $s = "abacababacabad"$ ，令模式串 $p = "abacabad"$ ，具体如下所示：

a	b	a	c	a	<u>b</u>	a	b	a	c	a	b	a	d
<u>a</u>	<u>b</u>	a	c	a	b	a	d						

此时因 s 中的'b'与 p 中的'd'匹配失败，则 $j = pmt[j - 1]$ ，即符合条件的最长前缀所紧接着的下一位：

a	b	a	c	a	<u>b</u>	a	b	a	c	a	b	a	d
<u>a</u>	<u>b</u>	a	c	a	b	a	d						

我们发现仍不匹配，我们继续：

a	b	a	c	a	b	<u>a</u>	b	a	c	a	b	a	d
<u>a</u>	<u>b</u>	a	c	a	b	a	d						

a	b	a	c	a	b	<u>a</u>	b	a	c	a	b	a	d
<u>a</u>	b	a	c	a	b	a	d						

这次取得了成功。当然，我们并不总是能成功，有可能 j 指针一路减到了0，但 $s[i]$ 仍然不等于 $p[j]$ ，这时我们不再移动 j 指针。

上述代码实现如下：

```

for (int i = 0, j = 0; i < s.length(); ++i) {
    while (j && s[i] != p[j])
        j = pmt[j - 1]; // 不断前移j指针，直到成功匹配或移到头为止
    if (s[i] == p[j]) j++; // 当前位匹配成功，j指针右移
    if (j == p.length()) {
        // 对s[i - j + 1 .. i]进行一些操作
    }
}

```

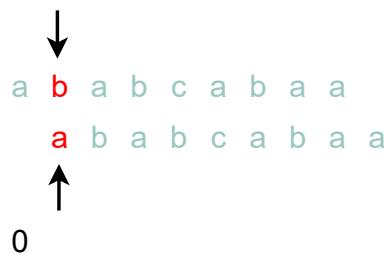
```

    j = pmt[j - 1];
}
}

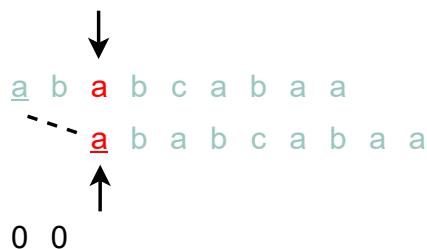
```

于是我们的问题变成了如何求 PMT 表，如果暴力求解，时间复杂度为 $O(m^2)$ ，这里提供一种做法：在错开一位后，让 **p** 自己去匹配自己，我们知道 $pmt[0] = 0$ ，而之后的每一位则可以通过在匹配过程中记录 **j** 值得到。

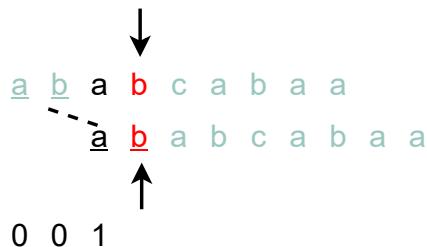
我们仍以刚刚的模式串 **p** = "ababcabaa" 为例：



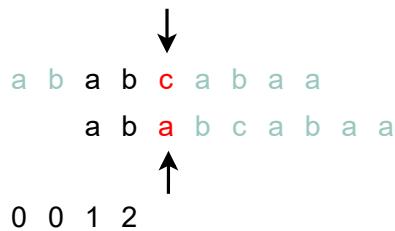
匹配失败，则令 $pmt[1] = -1 + 1$ ，**i** 指针后移：



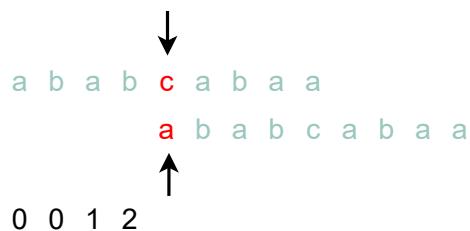
发现匹配成功，**j** 指针右移， $pmt[2] = 1$ ，然后两个指针都右移。



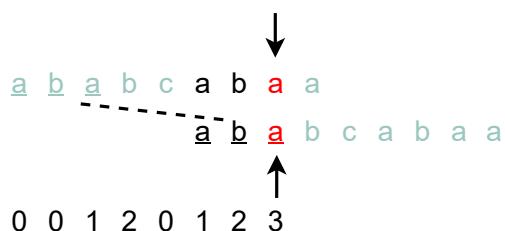
继续匹配成功， j 指针右移， $pmt[3] = 2$ 。



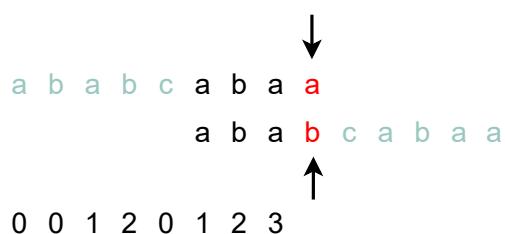
发现下一位匹配失败，但因为前面已经计算出来，我们仍可以像匹配文本串时那样地使用， $pmt[2 - 1]$ 即[1] = 0，所以退回到开头，此时j指针已经到了开头但仍未匹配成功，所以不再移动， $pmt[4] = j = 0$ 。



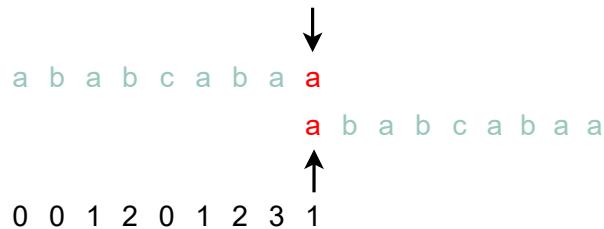
接下来仍按这种方法继续执行：



最后一位匹配失败，但这次我们先令 $j = pmt[j - 1] = 1$:



再次匹配，匹配成功，j指针右移一位，pmt[i] = j = 1。

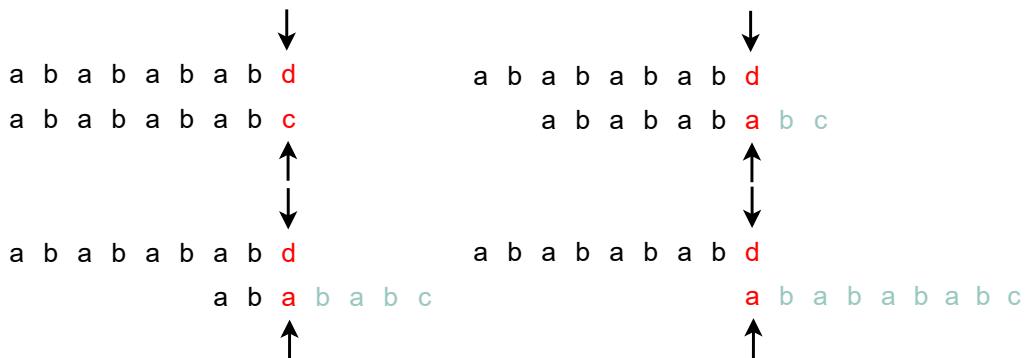


自此，我们通过一趟自我匹配，求出了 PMT，代码如下：

```
// pmt[0] = 0;
for (int i = 1, j = 0; i < plen; ++i) {
    while (j && p[i] != p[j]) j = pmt[j - 1];
    if (p[i] == p[j]) j++;
    pmt[i] = j;
}
```

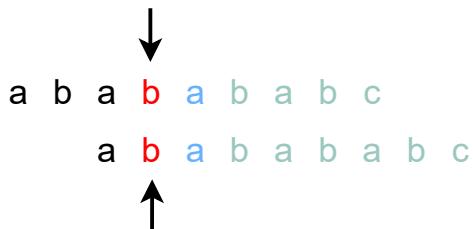
绝大多数情况下，上面的算法都够用了，所以很多人就管它叫 KMP 算法，但实际上，它只能称作 MP 算法，因为真正的 KMP 算法还有一个 Knuth 提出的常数优化。

例如 $p = "abababc"$ 这个模式串，若我们用它来匹配 $p = "abababd"$ ，在最后处要跳转 3 次才能发现匹配失败：



其实中间这几次跳转毫无意义，因为我们都知道'd'和'a'是不能匹配

的，却做了很多无用功，所以我们可以计算 PMT 时做一些小改动来避免这种情况。



例如上图，按道理匹配到这一步我们应该令 $pmt[i] = ++j (=2)$ 。但是我们发现， $p[i + 1]$ 与 $p[j + 1]$ 是相等的 'a'。也就是说稍后在匹配时，假如 j 指针为 4 时失配（说明 "ababa" 无法匹配），那在 j 指针为 2 时肯定也会失配（因为 "aba" 也无法匹配）。所以我们不如把路径压缩一下，也就是直接让 $pmt[i] = pmt[j] (=pmt[2-1])$ 而不是 $++j (=2)$ ，从而跳过 j 指针为 2 的情况。

```

void get_pmt(const string& s) {
    for (int i = 1, j = 0; i < s.length(); ++i) {
        while (j && s[i] != s[j]) j = pi[j - 1];
        if (s[i] == s[j]) {
            if (s[i + 1] == s[j + 1])
                pmt[i] = pmt[j++];
            else
                pmt[i] = ++j;
        }
        else
            pmt[i] = j;
    }
}

```

无论是 MP 算法还是 KMP 算法，其总时间复杂度都是 $O(m + n)$ ，这是因为 $++i$ 和 $++j$ 都只进行了 $m + n$ 次，虽然 j 在过程中有减小，但 j 在任

何时刻不可能小于 -1 ，所以 j 减小的次数也不可能超过 $n + m + 1$ 。

8.1.1 拓展 KMP 算法

Z 算法 (Z-Algorithm) 也被称为**拓展 KMP 算法**。

KMP 算法的核心就是那张**部分匹配表**，其实它也被称为**前缀函数**，记作 π 。对一个字符串而言，我们定义既是它前缀又是它后缀的字符串是它的 **border**，那 $\pi(i)$ 就表示 $s[0..i]$ 的最长 border 的长度。或者说， $\pi(i)$ 是满足 $s[0..x-1] = s[i-x+1..i]$ 的最大的 x ，特别地，令 $(0) = 0$ 。

而 Z 算法的核心是 **Z 函数**，它的定义与 π 非常相似：我们将 $Z(i)$ 定义为 s 与 $s[i..n-1]$ 的**最长公共前缀 (LCP)**。或者说， $Z(i)$ 是满足 $s[0..x-1] = s[i..i+x-1]$ 的最大的 x （特别地，令 $Z(0) = 0$ ）。

我们来看一个例子：设现有字符串 `aabcaabcaaaab`，那么它的 Z 函数值如下表所示：

下标	0	1	2	3	4	5	6	7	8	9	10	11	12
字符串	a	a	b	c	a	a	b	c	a	a	a	a	b
Z函数值	0	1	0	0	6	1	0	0	2	2	3	1	0

于是我们的问题变成了如何快速求出这张表。

设 $Z(i) \neq 0$ ，那么我们定义区间 $[i, i + Z[i] - 1]$ 为一个 **Z-Box**，显然，Z-Box 对应的子串一定也是整个字符串的一个前缀。例如，上表中 $[4, 9]$ 就是一个 Z-Box，它对应的子串是 `aabcaa`，也是整个字符串的一个前缀。

我们从左往右枚举下标 i ，同时维护 l 和 r ，用 $[l, r]$ 表示满足 $l \leq i$ 且 r 最大的 Z-Box。那么就可能出现三种情况：

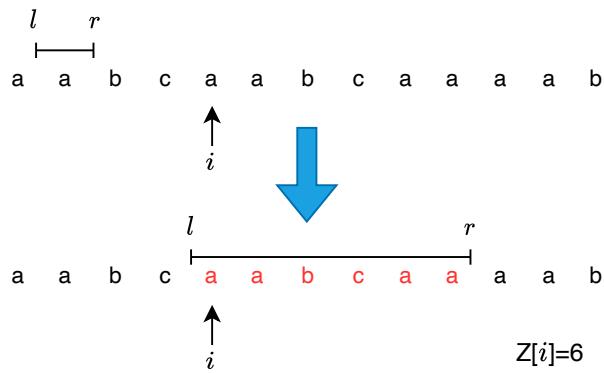
(1) 如果 $i > r$ ，说明我们已经完全越过了上一个 Z-Box，任何一个新的 Z-Box 的 r 都会更大，这时我们直接逐个比较，暴力计算 $Z[i]$ 的值。计算完成后，如果 $Z[i] \neq 0$ ，则更新 l 和 r 。

```
if (i > r) {
```

```

while (i + z[i] < n && s[z[i]] == s[i + z[i]])
    z[i]++;
l = i, r = i + z[i] - 1;
}

```

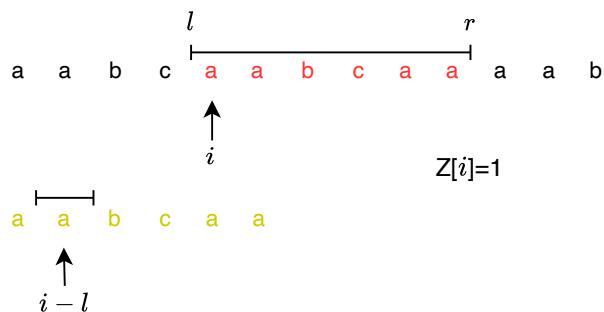


(2) 如果 $i \leq r$, 我们已经知道了 $s[l..r]$ 与 $s[0..r-l]$ 是相等的, 因此 $s[i..r]$ 与 $s[i-l..r-l]$ 也是相等的。所以我们考虑 $Z[i-l]$, 如果 $Z[i-l] < r-i+1$, 说明从 i 开始匹配和从 $i-l$ 开始匹配是一样的——匹配一定会在离开 Z-Box 前停止。即, $Z[i] = Z[i-l]$ 。

```

else if (z[i - 1] < r - i + 1)
    z[i] = z[i - 1];

```



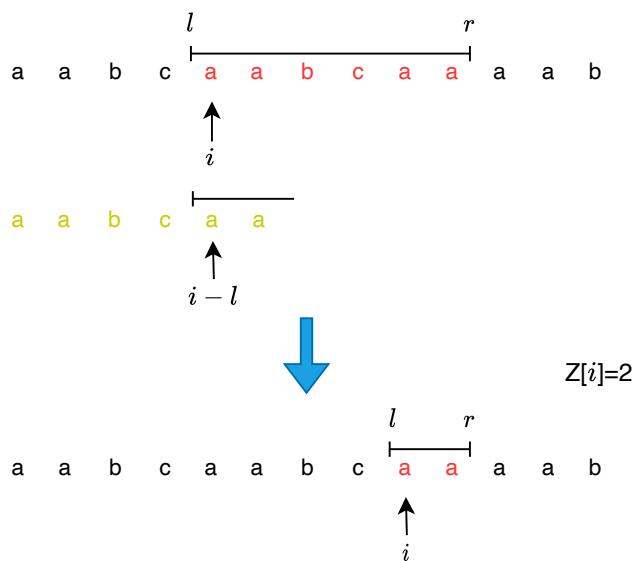
(3) 而如果 $Z[i-l] \geq r-i+1$, 说明整个 Z-Box 的剩余部分都可以匹配, 但后面的情况我们不清楚, 所以不能说 $Z[i] = Z[i-l]$ 。但我们知道 $Z[i]$

至少有 $r - i + 1$, 接下来我们暴力计算 Z-Box 还能向后延长多少即可。

```

else {
    z[i] = r - i + 1;
    while (i + z[i] < n && s[z[i]] == s[i + z[i]])
        z[i]++;
    l = i, r = i + z[i] - 1;
}

```



我们发现第一种和第三种情况的逻辑可以合并，最终得到以下代码：

```

for (int i = 1, l = 0, r = 0; i < n; ++i) {
    if (z[i - 1] < r - i + 1)
        z[i] = z[i - 1];
    else {
        z[i] = max(r - i + 1, 0);
        while (i + z[i] < n && s[z[i]] == s[i + z[i]])
            z[i]++;
        l = i, r = i + z[i] - 1;
    }
}

```

这个算法的复杂度是 $O(n)$ 的，内层循环每执行一次，都会使 r 加 1，执行次数不会超过 n 次。

应用

知道了 Z 函数的求法后，我们来看它的几个简单应用：

例 8.1.1. 给出字符串 a, b ，求 a 的每个后缀与 b 的 LCP。

设 $\$$ 为字符集外字符，求 $b + \$ + a$ 的 Z 函数，则 a 的后缀 $a[i..]$ 与 b 的 LCP 为 $Z(|b| + 1 + i)$ 。

例 8.1.2. 给出文本串 s 和模式串 p ，求 p 在 s 中的所有出现位置。

这是 KMP 和字符串哈希的经典题目，但也可以用 Z 算法。设 $\$$ 为字符集外字符，求 $p + \$ + s$ 的 Z 函数，则每一个 $Z(i) = |p|$ 都对应 p 在 s 中的一次出现。

例 8.1.3. 求 s 的所有 border。

虽然这个是 KMP 裸题，但也可以用 Z 算法。求 s 的 Z 函数。对于每一个 i ，如果 $i + Z(i) = |s|$ ，说明这个 Z-Box 对应一个 border。(注：与 KMP 不同，这里只是求所有 border，不是求所有前缀的 border)

例 8.1.4. 求 s 的每个前缀的出现次数。

求 s 的 Z 函数。对于每一个 i ，如果 $Z(i)$ 不等于 0，说明长度为 $Z(i), Z(i) - 1, \dots, 1$ 的前缀在此处各出现了一次，所以求一个后缀和即可。在这个问题中一般令 $Z(0) = |s|$ 。

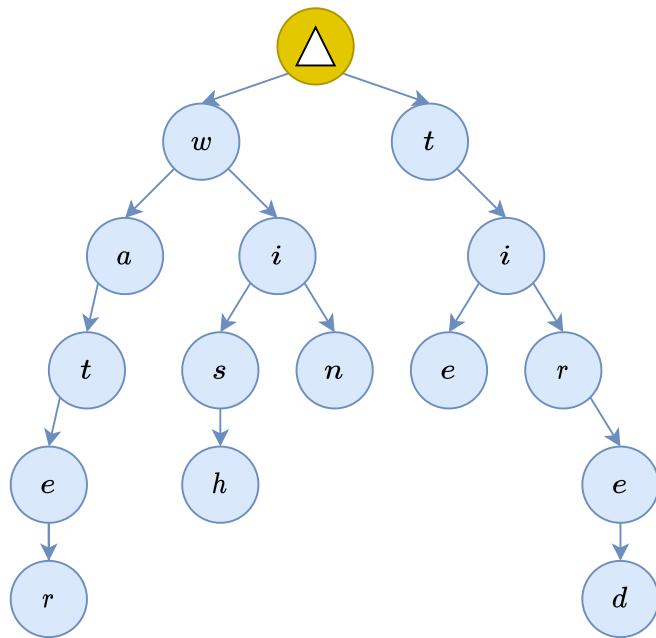
```

for (int i = n + 1; i < 2 * n + 1; ++i)
    S[z[i]]++;
for (int i = n; i >= 1; --i)
    S[i] += S[i + 1];

```

8.2 字典树

字典树 (Trie)，也称前缀树，是一个比较简单的数据结构，用来存储和查询字符串。例如，water, wish, win, tie, tired这几个单词可以用以下方式存储：



其中每个字符占据一个节点，拥有相同前缀的字符串可以共用部分节点。起始点是特殊点，我们将之设为节点 1，不存储字符。

建树的代码如下：

```

const int MAXN = 500005;
int next[MAXN][26], cnt; // 用类似链式前向星的方式存图，next[i][c]表示i号点所连、存储字符为c+'a'的点的编号
void init() // 初始化 {
    memset(next, 0, sizeof(next)); // 全部重置为0，表示当前点没有存储字符
    cnt = 1;
}
void insert(const string &s) { // 插入字符串
  
```

```

int cur = 1;
for (auto c : s) {
    // 尽可能重用之前的路径，如果做不到则新建节点
    if (!next[cur][c - 'a'])
        next[cur][c - 'a'] = ++cnt;
    cur = next[cur][c - 'a']; // 继续向下
}
}

```

字典树可以方便地查询某个前缀是否存在：

```

bool find_prefix(const string &s) { // 查找某个前缀是否出现过
    int cur = 1;
    for (auto c : s) {
        // 沿着前缀所决定的路径往下走，如果中途发现某个节点不存在，说明前缀不存在
        if (!next[cur][c - 'a'])
            return false;
        cur = next[cur][c - 'a'];
    }
    return true;
}

```

如果是查询某个字符串是否存在，可以另开一个exist数组，在插入完成时，把exist[叶子节点]设置为true，然后先按查询前缀的方法查询，在结尾处再判断一下exist的值。这是一种常见的套路，即用叶子节点代表整个字符串，保存某些信息。

字典树是一种空间换时间的数据结构，我们牺牲了字符串个数 \times 字符串平均字符数 \times 字符集大小的空间，但可以用 $O(n)$ 的时间查询，其中 n 为查询的前缀或字符串的长度。

例 8.2.1 (P2580 于是他错误的点名开始了). XS 中学化学竞赛组教练是一个酷爱炉石的人。

他会一边搓炉石一边点名以至于有一天他连续点到了某个同学两次，然后正好被路过的校长发现了然后就是一顿欧拉欧拉欧拉（详情请见已结束比赛 CON900）。

这之后校长任命你为特派探员，每天记录他的点名。校长会提供化学竞赛学生的人数和名单，而你需要告诉校长他有没有点错名。（为什么不直接不让他玩炉石。）

提示：

对于 40% 的数据， $n \leq 1000$, $m \leq 2000$ 。

对于 70% 的数据， $n \leq 10^4$, $m \leq 2 \times 10^4$ 。

对于 100% 的数据， $n \leq 10^4$, $m \leq 10^5$ 。

输入格式：

第一行一个整数 n ，表示班上人数。

接下来 n 行，每行一个字符串表示其名字（互不相同，且只含小写字母，长度不超过 50）。

第 $n + 2$ 行一个整数 m ，表示教练报的名字个数。

接下来 m 行，每行一个字符串表示教练报的名字（只含小写字母，且长度不超过 50）。

```
5
a
b
c
ad
acd
3
a
a
e
```

输出格式：

对于每个教练报的名字，输出一行。

如果该名字正确且是第一次出现，输出 OK，如果该名字错误，输出 WRONG，如果该名字正确但不是第一次出现，输出 REPEAT。

```
OK  
REPEAT  
WRONG
```

当然这个题其实用哈希表就可以轻松解决了，但也可以使用字典树。至于如何判断是否重复，只需要另开一个数组vis，每次查询后把叶子节点的vis设为true即可。代码如下：

```
#include <bits/stdc++.h>  
using namespace std;  
const int MAXN = 500005;  
namespace trie {  
    int next[MAXN][26], cnt;  
    bool vis[MAXN], exist[MAXN];  
    void init() {  
        memset(next, 0, sizeof(next));  
        cnt = 1;  
    }  
    void insert(const string &s) {  
        int cur = 1;  
        for (auto c : s) {  
            if (!next[cur][c - 'a'])  
                next[cur][c - 'a'] = ++cnt;  
            cur = next[cur][c - 'a'];  
        }  
        exist[cur] = true;  
    }  
    int find(const string &s) {  
        int cur = 1, ans;  
        for (auto c : s) {
```

```
    if (!next[cur][c - 'a'])
        return 0;
    cur = next[cur][c - 'a'];
}
if (!exist[cur])
    ans = 0;
else if (!vis[cur])
    ans = 1;
else
    ans = 2;
vis[cur] = true;
return ans;
}
} // namespace trie
int main() {
    int n;
    cin >> n;
    trie::init();
    while (n--) {
        string s;
        cin >> s;
        trie::insert(s);
    }
    int q;
    cin >> q;
    while (q--) {
        string s;
        cin >> s;
        switch (trie::find(s)) {
        case 0:
            cout << "WRONG" << endl;
            break;
        case 1:
```

```
    cout << "OK" << endl;
    break;
  case 2:
    cout << "REPEAT" << endl;
  }
}
return 0;
}
```

以上是使用命名空间的写法，原因在于避免与内置函数起冲突，接下来给出常规写法：

```
#include "bits/stdc++.h"

using namespace std;
const int MAXN = 500007;

// 字典树问题，二维数组构建
int tree[MAXN][26], cnt; // cnt 计数器
// 是否访问
bool vis[MAXN], exist[MAXN];

// 插入 插入的是字符串
void insert(string &s) {
  int curr = 1; // 字符串的第一个位置
  for (auto c: s) {
    // 有没有这个前缀
    if (!tree[curr][c - 'a']) {
      tree[curr][c - 'a'] = ++cnt;
    }
    curr = tree[curr][c - 'a'];
  }
  exist[curr] = true;
}
```

```
// 查找
int findd(string &s) { // 返回一个状态数
    int curr = 1, ans; // 存在，重复，不存在

    for (auto c: s) {
        if (!tree[curr][c - 'a']) {
            return 0; // 不存在
        }
        curr = tree[curr][c - 'a'];
    }

    if (!exist[curr]) {
        ans = 0;
    } else if (!vis[curr]) { // 未访问过
        ans = 1;
    } else {
        ans = 2;
    }

    vis[curr] = true;
    return ans;
}

int main() {
    memset(tree, 0, sizeof tree);
    cnt = 1;

    int n;
    cin >> n;

    while (n--) {
        string s;
        cin >> s;
        insert(s); // 插入字典树中
    }

    int q;
```

```

    cin >> q;
    while (q--) {
        string s;
        cin >> s;
        switch (findd(s)) {
            case 0:
                cout << "WRONG" << endl;
                break;
            case 1:
                cout << "OK" << endl;
                break;
            case 2:
                cout << "REPEAT" << endl;
                break;
        }
    }
    return 0;
}

```

实现方法大致相同，声明`tree[MAXN][26]`的数组，每个节点都至多有 26 个子节点，在建树过程中，如果已有前缀则直接下推，否则需要创建一个新的节点：

```

if (! tree[curr][c - 'a']) {
    tree[curr][c - 'a'] = ++ cnt;
}
curr = tree[curr][c - 'a'];

```

同时，还有一类字典树常见的问题：

例 8.2.2 (HDU-1251 统计难题). *Ignatius* 最近遇到一个难题，老师交给他很多单词（只有小写字母组成，不会有重复的单词出现），现在老师要他统计出以某个字符串为前缀的单词数量（单词本身也是自己的前缀）。

输入格式:

输入数据的第一部分是一张单词表，每行一个单词，单词的长度不超过 10，它们代表的是老师交给 Ignatius 统计的单词，一个空行代表单词表的结束。

第二部分是一连串的提问，每行一个提问，每个提问都是一个字符串。

```
banana
band
bee
absolute
acm

ba
b
band
abc
```

输出格式:

对于每个提问，给出以该字符串为前缀的单词的数量。

```
2
3
1
0
```

例题8.2.2统计的是字符串的前缀，因此可以用字典树来解决。即涉及到前缀问题，均可以考虑到字典树。

这里我们可以先把所有单词插入字典树，然后对于每个提问，我们只需要查询字典树即可。

```
#include "bits/stdc++.h"

using namespace std;
const int MAXN = 500007;
```

```
// 字典树问题，二维数组构建
int tree[MAXN][26], cnt; // cnt 计数器
int summ[MAXN];

// 插入
void insert(string &s) {
    int curr = 0; // 字符串的第一个位置
    for (auto c: s) {
        // 有没有这个前缀
        if (!tree[curr][c - 'a']) {
            tree[curr][c - 'a'] = ++cnt;
        }
        // 在遍历字符串的过程中，记录前缀数量
        summ[tree[curr][c - 'a']]++;
        curr = tree[curr][c - 'a'];
    }
}

// 查找
int findd(string &s) {
    int curr = 0;

    for (auto c: s) {
        if (!tree[curr][c - 'a']) {
            return 0; // 不存在
        }
        curr = tree[curr][c - 'a'];
    }
    return summ[curr];
}

int main() {
```

```
string s;
while (getline(cin, s)) { // 读取整行，回车结束
    if (s != "") {
        insert(s);
    } else {
        break;
    }
}
while (getline(cin, s)) {
    cout << findd(s) << endl;
}
return 0;
}
```

8.2.1 01 字典树

01 字典树 (01-trie) 是一种特殊的字典树，它的字符集只有 $\{0, 1\}$ ，主要用来解决一些异或问题。我们先看一道例题：

例 8.2.3 (HDU-4825 Xor Sum). *Zeus* 和 *Prometheus* 做了一个游戏。

Prometheus 给 *Zeus* 一个集合，集合中包含了 N 个正整数，随后向 *Zeus* 发起 M 次询问，每次询问中包含一个正整数 S ，之后 *Zeus* 需要在集合当中找出一个正整数 K ，使得 K 与 S 的异或结果最大。*Prometheus* 为了让 *Zeus* 看到人类的伟大，随即同意 *Zeus* 可以向人类求助。

你能证明人类的智慧么？

输入格式：

输入包含若干组测试数据，每组测试数据包含若干行。

输入的第一行是一个整数 $T(T < 10)$ ，表示共有 T 组数据。

每组数据的第一行输入两个正整数 $N, M(1 \leq N, M \leq 100000)$ ，接下来一行，包含 N 个正整数，代表 *Zeus* 的获得的集合，之后 M 行，每行一

个正整数 S , 代表 Prometheus 询问的正整数。所有正整数均不超过 2^{32} 。

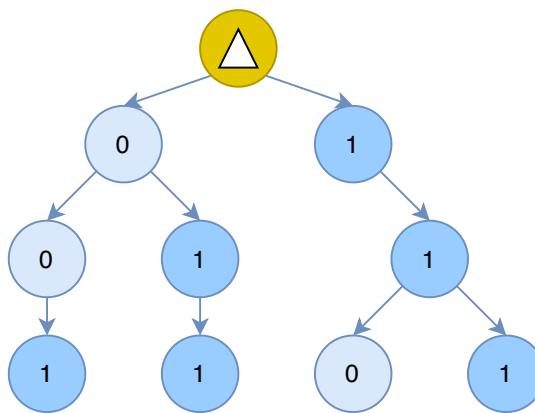
```
2
3 2
3 4 5
1
5
4 1
4 6 5 6
3
```

输出格式:

对于每组数据, 首先需要输出单独一行Case #?:, 其中间号处应填入当前的数据组数, 组数从 1 开始计算。对于每个询问, 输出一个正整数 K , 使得 K 与 S 异或值最大。

```
Case #1:
4
3
Case #2:
4
```

01 字典树的原理很简单, 把数字化为二进制后, 当作 01 串, 从高位到低位像普通字典树那样存储。例如, 如果有数 $(1)_2, (11)_2, (110)_2, (111)_2$, 存入 01 字典树就像这样, 但需要把各个数的位数补成一样多:



现在对于例题8.2.3，我们贪心地解决即可。如果我们要找与给定数异或**最大的**数，就尽可能走与该数当前位**不同的**路径，反之则尽可能走与当前位**相同的**路径。

```
#include "bits/stdc++.h"
#define ll long long
using namespace std;

const int MAXN = 1000007;

ll tree[32 * MAXN][2], cnt; // n个无符号32位整数，子节点个数 2
ll val[32 * MAXN], n, m;

// 建树
void insert(ll a) {
    ll curr = 0;
    // 从高位向低位
    for (ll i = 31; i >= 0; i--) {
        int c = (a >> i) & 1;
        if (!tree[curr][c]) {
            // 创建新节点
            tree[curr][c] = cnt++;
        }
        curr = tree[curr][c];
    }
    val[curr] = a;
}

// 查找
int findd(ll a) {
    int curr = 0;
    for (ll i = 31; i >= 0; i--) {
        int c = (a >> i) & 1;
```

```
// 走相反的路径
if (tree[curr][c ^ 1]) {
    curr = tree[curr][c ^ 1]; // 找到了相反的路
} else {
    curr = tree[curr][c];
}
}

return val[curr];
}

int main() {
    ll T;
    scanf("%lld", &T);
    ll q = 1; // 询问编号
    while (T--) {
        memset(tree, 0, sizeof tree);
        memset(val, 0, sizeof val);
        cnt = 1;
        scanf("%lld%lld", &n, &m);
        ll x; // 给出的数
        for (int i = 1; i <= n; i++) {
            scanf("%lld", &x);
            insert(x); // 插入字典树
        }
        printf("Case # %lld:\n", q++);
        while (m--) {
            scanf("%lld", &x);
            printf("%lld\n", findd(x));
        }
    }
    return 0;
}
```

另外还有一道题：

例 8.2.4 (洛谷 P4551 最长异或路径). 给定一棵 n 个点的带权树，结点下标从 1 开始到 n 。寻找树中找两个结点，求最长的异或路径。

异或路径指的是指两个结点之间唯一路径上的所有边权的异或。

输入格式：

第一行一个整数 n ，表示点数。

接下来 $n - 1$ 行，给出 u, v, w ，分别表示树上的 u 点和 v 点有连边，边的权值是 w 。

```
4
1 2 3
2 3 4
2 4 6
```

输出格式：

一行，一个整数表示答案。

```
7
```

提示：最长异或序列是 1, 2, 3，答案是 $7 = 3 \oplus 4$ 。

数据范围： $1 \leq n \leq 100000; 0 < u, v \leq n; 0 \leq w < 2^{31}$ 。

例题8.2.4非常巧妙，我们知道一个数异或同一个数两次相当于没有异或。因为公共段被抵消掉了，所以 $dis(i, j)$ 其实就是 $dis(1, i) \oplus dis(1, j)$ 。于是我们一趟 dfs 把每个点到节点 1 的异或路径求出来，问题就转换成了“给出一组数，从中选两个数异或，求最大值”。也就是对每一个数分别解决“从一组数中选一个数与给定数异或最大”的子问题。

```
#include<bits/stdc++.h>

using namespace std;

const int MAXN = 1000007;
```

```
// 1. 存图
int h[MAXN << 2], to[MAXN << 2], ne[MAXN << 2], w[MAXN << 2],
    idx;
void adde(int a, int b, int c) {
    ne[++idx] = h[a];
    h[a] = idx;
    to[idx] = b;
    w[idx] = c;
    // 反向加
    ne[++idx] = h[b];
    h[b] = idx;
    to[idx] = a;
    w[idx] = c;
}

// 2. 处理树 dfs
int XOR[MAXN]; // 从 当前点开始 到 1 的异或值
void dfs(int x, int fa, int cur) {
    XOR[x] = cur;
    for (int i = h[x]; i; i = ne[i]) {
        if (to[i] == fa) {
            continue;
        }
        dfs(to[i], x, cur ^ w[i]);
    }
}

// 3. 字典树部分
int tree[MAXN][2], cnt;

// 建树
void insert(int a) {
```

```
int curr = 0;
// 从高位向低位
for (int i = 31; i >= 0; i--) {
    // 获取当前位的值
    int c = (a >> i) & 1;
    if (!tree[curr][c]) {
        // 创建新节点
        tree[curr][c] = cnt++;
    }
    curr = tree[curr][c];
}

// 查找
int findd(int a) {
    int curr = 0;
    int ans = 0;
    for (int i = 31; i >= 0; i--) {
        // 获取当前位的值
        int c = (a >> i) & 1;
        // 走相反的路径
        if (tree[curr][c ^ 1]) {
            curr = tree[curr][c ^ 1]; // 就找到了相反的路
            ans += 1 << i; // 当前位置的异或值更新
        } else {
            curr = tree[curr][c];
        }
    }
    return ans;
}

int main() {
    int n;
```

```

    cin >> n;
    for (int i = 1; i < n; i++) {
        int a, b, c;
        cin >> a >> b >> c;
        adde(a, b, c);
    }
    dfs(1, 0, 0);
    for (int i = 1; i <= n; i++) {
        insert(XOR[i]); // 每一步都是存的当前点 到 根结点 的异或值
    }
    int ans = 0;
    for (int i = 0; i <= n; i++) {
        ans = max(ans, findd(XOR[i]));
    }
    cout << ans << endl;
    return 0;
}

```

如程序中所示，我们需要使用 DFS 来递归处理整棵树，且存储存储每个点到根的异或值，在处理树的过程中，就用 `XOR[i]` 来存储节点 i 到根结点 1 的异或值，并将这样每一步的异或值存入到字典树中。

最后只需要查询字典树，就可以求出任意两个点的最长异或路径。

总的来说，一组数中取任意个数求最大/最小异或和可以用**线性基**解决，取两个数求最大/最小异或和则可以用**01 字典树**解决。

8.3 Manacher 算法

Manacher 算法，俗称**马拉车**，是一种与**回文子串**有关的算法。

回文串根据其长度地奇偶性可以分为**奇回文串**和**偶回文串**两种，其中奇回文串通常比偶回文串好处理，因为它有一个唯一的中心，而且可以从中心向两边一步一步拓展得到。

不过，我们对字符串进行一些预处理后，就可以用处理奇回文串的方法处理偶回文串了。方法是：在每对相邻字符和字符串的两端加入一个字符集外字符，这里设为\$，例如，把abbab变成\$a\$b\$b\$a\$b\$。这样处理后，原来的bab变成了\$b\$a\$b\$，原来的abba变成了\$a\$b\$b\$a\$。可以发现，原来的奇回文子串会变成以普通字符为中心的奇回文子串；原来的偶回文子串会变成以\$为中心的奇回文子串。

对于一个字符串 s ，设 $d[i]$ 表示以 $s[i]$ 为中心的奇回文子串的数量。很明显，这 $d[i]$ 个回文子串的长度分别为 $1, 3, \dots, 2d[i] - 1$ 。如果 s 是由 t 预处理得到的，那么这些回文子串对应 t 中对应位置的 $\lfloor \frac{d[i]}{2} \rfloor$ 个回文子串，且最长的一个长度为 $d[i] - 1$ 。

Manacher 算法的作用，就是 $O(n)$ 地求出这个 d 数组。

我们在讲动态规划的时候提到过最长回文子串，题5.2.6当时用的是动态规划思想来解决，我们回顾一下动态规划解法：

对于一个子串而言，如果它是回文串，并且长度大于 2，那么将它首尾的两个字母去除之后，它仍然是个回文串。例如对于字符串ababa，如果我们已经知道bab是回文串，那么ababa一定是回文串，这是因为它的首尾两个字母都是a。

我们用 $P(i, j)$ 表示字符串 s 的第 i 到 j 个字母组成的串（下文表示成 $s[i, j]$ ）是否为回文串：

$$P(i, j) = \begin{cases} \text{true}, & \text{子串 } s_i \cdots s_j \text{ 是回文子串} \\ \text{false}, & \text{其他情况} \end{cases}$$

这里的其他情况包含两种情况：

- $s[i, j]$ 本身不是回文子串；
- $i > j$ ，此时 $s[i, j]$ 不合法。

那么我们就可以写出动态规划的状态转移方程：

$$P(i, j) = P(i + 1, j - 1) \wedge (S_i == S_j)$$

也就是说，只有 $s[i + 1, j - 1]$ 是回文串，并且 s 的第 i 和 j 个字母相同时， $s[i, j]$ 才会是回文串。

以上的所有讨论是建立在子串长度大于 2 的前提之上的，我们还需要考虑动态规划中的边界条件，即子串的长度为 1 或 2。对于长度为 1 的子串，它显然是个回文串；对于长度为 2 的子串，只要它的两个字母相同，它就是一个回文串。因此我们就可以写出动态规划的边界条件：

$$\begin{cases} P(i, i) &= \text{true} \\ P(i, i + 1) &= (S_i == S_{i+1}) \end{cases}$$

```
string longestPalindrome(string s) {
    int n = s.size();
    if (n < 2) {
        return s;
    }

    int maxLen = 1;
    int begin = 0;
    // dp[i][j] 表示 s[i..j] 是否是回文串
    vector<vector<int>> dp(n, vector<int>(n));
    // 初始化：所有长度为 1 的子串都是回文串
    for (int i = 0; i < n; i++) {
        dp[i][i] = true;
    }
    // 递推开始
    // 先枚举子串长度
    for (int L = 2; L <= n; L++) {
```

```
// 枚举左边界，左边界上限设置可以宽松一些
for (int i = 0; i < n; i++) {
    // 由 L 和 i 可以确定右边界，即 j - i + 1 = L 得
    int j = L + i - 1;
    // 如果右边界越界，就可以退出当前循环
    if (j >= n) {
        break;
    }

    if (s[i] != s[j]) {
        dp[i][j] = false;
    } else {
        if (j - i < 3) {
            dp[i][j] = true;
        } else {
            dp[i][j] = dp[i + 1][j - 1];
        }
    }
}

// 只要 dp[i][L] == true 成立，就表示子串 s[i..L] 是
// 回文，此时记录回文长度和起始位置
if (dp[i][j] && j - i + 1 > maxLen) {
    maxLen = j - i + 1;
    begin = i;
}
}

return s.substr(begin, maxLen);
}
```

这里仍要强调，在状态转移方程中，我们是从长度较短的字符串向长度较长的字符串进行转移的，因此一定要注意动态规划的循环顺序。动态规划解决的时间复杂度和空间复杂度都是 $O(n^2)$ 。

现在我们再来看看动态规划的状态转移方程：

$$\begin{cases} P(i, i) &= \text{true} \\ P(i, i+1) &= (S_i == S_j) \\ P(i, j) &= P(i+1, j-1) \wedge (S_i == S_j) \end{cases}$$

找到其中的状态转移链：

$$P(i, j) \leftarrow P(i+1, j-1) \leftarrow P(i+2, j-2) \leftarrow \cdots \leftarrow \text{某一边界情况}$$

可以发现，所有的状态在转移的时候的可能性都是唯一的。也就是说，我们可以从每一种边界情况开始**扩展**，也可以得出所有的状态对应的答案，也就是所谓的**中心扩展算法**。

边界情况仍是子串长度为 1 或 2 的情况。我们枚举每一种边界情况，并从对应的子串开始不断地向两边扩展，接下来也就两种情况：

- 如果两边的字母相同，我们就可以继续扩展，例如从 $P(i+1, j-1)$ 扩展到 $P(i, j)$ ；
- 如果两边的字母不同，我们就可以停止扩展，因为在这之后的子串都不能是回文串了。

其实，所谓**边界情况**对应的子串实际上就是我们**扩展出的回文串的回文中心**。

中心扩展算法的本质即为：枚举所有的“回文中心”并尝试“扩展”，直到无法扩展为止，此时的回文串长度即为此“回文中心”下的最长回文串长度。我们对所有的长度求出最大值，即可得到最终的答案。

```
pair<int, int> expandAroundCenter(const string& s, int left,
    int right) {
```

```
while (left >= 0 && right < s.size() && s[left] == s[right]) {
    --left;
    ++right;
}
return {left + 1, right - 1};
}

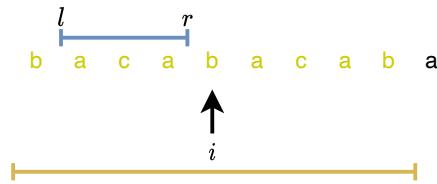
string longestPalindrome(string s) {
    int start = 0, end = 0;
    for (int i = 0; i < s.size(); ++i) {
        auto [left1, right1] = expandAroundCenter(s, i, i);
        auto [left2, right2] = expandAroundCenter(s, i, i + 1);
        if (right1 - left1 > end - start) {
            start = left1;
            end = right1;
        }
        if (right2 - left2 > end - start) {
            start = left2;
            end = right2;
        }
    }
    return s.substr(start, end - start + 1);
}
```

时间复杂度: $O(n^2)$, 其中 n 是字符串的长度, 因为长度为 1 和 2 的回文中心分别有 n 和 $n-1$ 个, 每个回文中心最多会向外扩展 $O(n)$ 次; 空间复杂度为 $O(1)$ 。

现在我们回到 Manacher 算法, Manacher 算法其实和 Z 算法的流程非常类似。我们以 bacabacaba 为例, 从左到右枚举 i , 并维护 l 和 r , 使得 $s[l..r]$ 是满足 $l \leq i$ 且 r 最靠右的奇回文串。当我们枚举到一个新的 i 时,

我们进行分类讨论：

- (1) 如果 $i > r$, 则直接暴力计算 $d[i]$, 同时更新 l 和 r 。

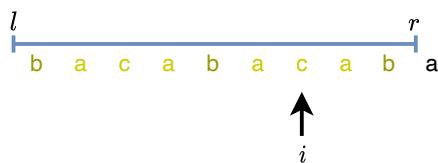


```

if (i > r) {
    while (i - d[i] >= 0 && i + d[i] < n && s[i - d[i]] == s[i + d[i]])
        d[i]++;
    l = i - d[i] + 1, r = i + d[i] - 1;
}

```

- (2) 如果 $i \leq r$, 找到 i 关于 $s[l..r]$ 中心的对称点 j , 这时如果以 j 为中心的最长奇回文串的左边界大于 l , 则直接令 $d[i] = d[j]$ 。这是因为, 在 $s[l..r]$ 的范围内, 关于中心点对称的子串一定是相等的。

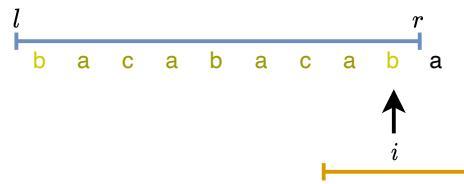


```

// int j = l + r - i;
else if (j - d[j] >= 1) { // 即 j - d[j] + 1 > l
    d[i] = d[j];
}

```

- (3) 如果以 j 为中心的最长奇回文串的左边界小于等于 l , 那么我们知道 $d[i]$ 至少有 $j - l + 1$ (或 $r - i + 1$), 但会不会更长还不清楚, 所以我们再尽可能向左右进行拓展, 同时更新 l 和 r 。



```

else {
    d[i] = j - l + 1;
    while (i - d[i] >= 0 && i + d[i] < n && s[i - d[i]] == s[i + d[i]])
        d[i]++;
    l = i - d[i] + 1, r = i + d[i] - 1;
}

```

和 Z 算法一样，我们重新组织一下逻辑，得到以下的代码：

```

for (int i = 0, l = 0, r = -1; i < n; ++i) {
    int j = l + r - i;
    d[i] = max(min(d[j], j - l + 1), 0);
    if (j - d[j] < 1) {
        while (i - d[i] >= 0 && i + d[i] < n && s[i - d[i]] == s[i + d[i]])
            d[i]++;
        l = i - d[i] + 1, r = i + d[i] - 1;
    }
}

```

不难发现，在整个过程中， r 都是单调不减的，而每次执行内层循环都相当于使 r 的值加 1，所以内层循环最多执行 n 次。整个算法的复杂度也就是 $O(n)$ 。

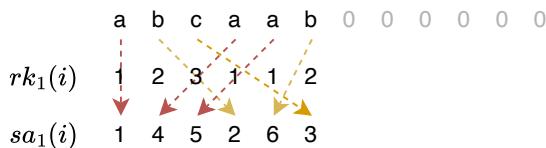
8.4 后缀数组

后缀数组 (Suffix Array, SA) 是解决很多与字符串相关的问题的有力工具。它实际上就是把字符串的所有后缀按字典序排序后得到的数组。

首先，为了方便，我们把字符串的长度扩充一倍，保证往后填充的字符比所有已有字符都小（记为 0），例如我们现在有个字符串 abcaab，我们用 0 进行扩充后得到 abcaab000000。注意，这样填充后并不会影响原先两个后缀的大小相对关系。

设 $rk_w(i)$ 表示从第 i 位开始、长度为 w 的子串在所有长度为 w 的子串中的排名（全 0 串的排名记为 0）， $sa_w(i)$ 表示长度为 w 的子串中字典序第 i 小的一个的开始位置。

我们可以轻松地求出 $rk_1(i)$ 和 $sa_1(i)$ ：



如果已知 rk_w ，那么我们就可以求出 sa_{2w} ，只需要以 $rk_w(sa_w(i))$ 为第一关键词、 $rk_w(sa_w(i) + w)$ 为第二关键词对 sa_w 进行一次排序就好。而如果我们得到了 sa_{2w} ，自然地也就可以得到 rk_{2w} 。



这个道理是很简单：对于两个等长的字符串 s 和 t ，如果 s 的字典序小于 t ，那么要么 s 的前半段的字典序小于 t 的前半段；要么它们前半段相

等，且 s 的后半段的字典序小于 t 的后半段。

所以我们可以从 rk_1/sa_1 出发，一步一步地得到 rk_2/sa_2 , $rk_4/sa_4 \dots$ 。当 $w \geq n$ 时， rk_w 也可以表示每个**后缀**的排名， sa_w 也可以表示每个**后缀**排序后的结果，也就是我们要的后缀数组。

在实际实现时，我们其实不需要求出 rk_1/sa_1 。因为除了最后一步外，我们只需要知道 rk_w 的相对大小关系，所以 $rk_1(i)$ 可以用字符本身代替。而除了最后一步外， sa_w 的唯一作用是用来求 rk_w ，但 sa_1 没有被用来求 rk_1 ，只是一个待排序数组，所以把待排序的下标按任意顺序放进去即可。不过注意，当 $n = 1$ 时，这样做会得到错误的 rk ，可以特判一下。

```
char s[MAXN];
int rk[MAXN << 1], sa[MAXN], tmp[MAXN << 1];
void init_sa(int n) { // 1-index
    if (n == 1) return void(rk[1] = sa[1] = 1); // 特判

    for (int i = 1; i <= n; ++i)
        sa[i] = i, rk[i] = s[i];
    for (int w = 1; w < n; w <= 1) {
        auto rp = [&](int x) { return make_pair(rk[x], rk[x + w]); };
        sort(sa + 1, sa + n + 1, [&](int x, int y) { return rp(x) < rp(y); });
        for (int i = 1, p = 0; i <= n; ++i)
            tmp[sa[i]] = rp(sa[i - 1]) == rp(sa[i]) ? p : ++p; // 注意相同的子串需要有相同的排名
        copy(tmp + 1, tmp + n + 1, rk + 1);
    }
}
```

这样实现的时间复杂度是 $O(n \log^2 n)$ 。

初步完成实现之后，我们要降低一下时间复杂度，注意到上面的算法的复杂度瓶颈在**排序**，所以我们可以直接在排序上做文章：注意到我们排的

是整数对，所以可以用**基数排序**代替，这样时间复杂度就降低到 $O(n \log n)$ 。

```
char s[MAXN];
int rk[MAXN << 1], sa[MAXN], tmp[MAXN << 1], cnt[MAXN];
void init_sa(int n) { // 1-index
    if (n == 1) return void(rk[1] = sa[1] = 1);

    for (int i = 1; i <= n; ++i)
        sa[i] = i, rk[i] = s[i];
    for (int w = 1, m = max(128, n); w < n; w <= 1) {
        fill(cnt + 1, cnt + m + 1, 0);
        for (int i = 1; i <= n; ++i)
            cnt[rk[sa[i] + w]]++;
        for (int i = 1; i <= m; ++i)
            cnt[i] += cnt[i - 1];
        for (int i = n; i >= 1; --i)
            tmp[cnt[rk[sa[i] + w]]--] = sa[i];

        fill(cnt + 1, cnt + m + 1, 0);
        for (int i = 1; i <= n; ++i)
            cnt[rk[tmp[i]]]++;
        for (int i = 1; i <= m; ++i)
            cnt[i] += cnt[i - 1];
        for (int i = n; i >= 1; --i)
            sa[cnt[rk[tmp[i]]]--] = tmp[i];

        auto rp = [&](int x) { return make_pair(rk[x], rk[x + w]); };
        for (int i = 1, p = 0; i <= n; ++i)
            tmp[sa[i]] = rp(sa[i - 1]) == rp(sa[i]) ? p : ++p;
        copy(tmp + 1, tmp + n + 1, rk + 1);
    }
}
```

然而，上面这份代码交上去后，我们会发现它跑得还不一定比 $O(n \log^2 n)$ 的版本好呢，这很正常，我们这份代码每次排序要遍历八次数组，这常数怎么想都很大，所以我们需要继续优化。

基数排序是先排第二关键词，再排第一关键词。我们尝试对第一趟排序进行优化，先来看看原本的过程：

	a	b	c	a	a	b	0	0	0	0	0	0
	$rk_1(i)$	1	2	3	1	1	2					
	$sa_1(i)$	1	4	5	2	6	3					
		ab	aa	ab	bc	b0	ca					
第一趟排序	$sa_2(i)$	6	4	3	1	5	2					

我们的目的是为了得到一个 $rk(sa_w(i) + w)$ 单调不减的数组。实际上，我们可以把第一趟排序后得到的数组分为两部分：第一部分，那些满足 $sa_w(i) + w > n$ 的 $sa_w(i)$ ，因为 $rk_w(sa_w(i) + w)$ 必然等于 0，所以它们将位于数组的开头部分。

然后我们观察发现，在 sa_w 中，如果 $sa_w(i) > w$ ，说明 $rk_w(sa_w(i))$ 在排序中一定会作为 $sa_w(i) - w$ 的第二关键词。而对 sa_w 而言， $rk_w(sa_w(i))$ 是单调不减的，所以，对于符合条件的 $sa_w(i)$ ，把 $t = sa_w(i) - w$ 依次放进数组，则 $rk(t + w)$ 就是单调不减的。这就是我们想要的数组第二部分，这两部分刚好覆盖了 1 到 n 的所有值。

	a	b	c	a	a	b	0	0	0	0	0	0
	$rk_1(i)$	1	2	3	1	1	2					
	$sa_1(i)$	1	4	5	2	6	3					
		a	a	a	b	b	c					
第一趟排序	$sa_2(i)$	6	3	4	1	5	2					

这跟我们原本直接计数排序得到的数组是不一样的，因为这是一种不

稳定的排序方法。但是没有关系，因为基数排序的第一趟排序可以不稳定。

注意：现在我们就需要注意初始化 sa_1 ，因为 sa 不再只在求 rk 中用到了。

此外，在上面的算法中，我们每次需要遍历 $\max(n, 128)$ 个数，实际上是很浪费的。注意到，我们每次都计算出了一个 p ，它其实就是 cnt 的值域。实际上，除了第一次循环后值域可能减小， cnt 的值域都是逐渐增长到 n 的。而且当它增长到 n 时，说明每个子串的 rk 互不相同，数组已经不会再有变化了，这时可以提前退出。

```
char s[MAXN];
int rk[MAXN << 1], sa[MAXN << 1], tmp[MAXN << 1], cnt[MAXN],
     rkt[MAXN];
void init_sa(int n) { // 1-index
    if (n == 1) return void(rk[1] = sa[1] = 1);

    int m = 128;
    for (int i = 1; i <= n; ++i)
        ++cnt[rk[i] = s[i]];
    for (int i = 1; i <= m; ++i)
        cnt[i] += cnt[i - 1];
    for (int i = n; i >= 1; --i)
        sa[cnt[rk[i]]--] = i;

    for (int w = 1;; w <= 1) {
        for (int i = n; i > n - w; --i)
            tmp[n - i + 1] = i;
        for (int i = 1, p = w; i <= n; ++i)
            if (sa[i] > w) tmp[++p] = sa[i] - w;
        fill(cnt + 1, cnt + m + 1, 0);
        for (int i = 1; i <= n; ++i)
            cnt[rkt[i] = rk[tmp[i]]]++;
        for (int i = 1; i <= m; ++i)
            cnt[i] += cnt[i - 1];
```

```

for (int i = n; i >= 1; --i)
    sa[cnt[rkt[i]]--] = tmp[i];
m = 0;
auto rp = [&](int x) { return make_pair(rk[x], rk[x + w]);
};
for (int i = 1; i <= n; ++i)
    tmp[sa[i]] = rp(sa[i - 1]) == rp(sa[i]) ? m : ++m;
copy(tmp + 1, tmp + n + 1, rk + 1);
if (n == m) break;
}
}

```

一点题外话

如果我们把 sa 所代表的字符串按顺序写下来，会发现一些有趣的性质：

	a	b	c	a	a	b
0	a	a	b			
1	a	b				
2	a	b	c	a	a	b
0	b					
1	b	c	a	a	b	
0	c	a	a	b		

可以看到，相邻的后缀之间可能有一些共同前缀。

我们令 $height[i] = lcp(sa[i], sa[i - 1])$ ，其中 lcp 表示最长公共前缀。利用这个 $height$ 数组，我们可以做很多事。

例如，我们可以按字典序遍历所有本质不同的子串。只需要遍历 $sa[i]$ 的前缀，但是跳过前 $height[i]$ 个，因为这 $height[i]$ 个前缀是与 $sa[i - 1]$ 共有的，之前一定已经被遍历过了。按此道理，也可以知道**字符串本质不同的子串数量为 $\frac{n(n-1)}{2} - \sum height[i]$** 。

还可以求两个子串的 LCP。显然，设两个子串分别为 $s_1 = s[l_1..r_1]$ $s_2 = s[l_2..r_2]$ ，则其最长公共前缀为 $\min(|s_1|, |s_2|, lcp(sa[l_1], sa[l_2]))$ 。所以我们只需要求两个后缀的 LCP，而很容易发现这正是 $\min_{l_1 < k \leq l_2} height[k](l_1 \leq l_2)$ 。这是一个 RMQ 问题，用线段树或 ST 表可以解决。

此时，求 $height$ 就需要证明一个结论：

推论 8.4.1. 设 $h[i] = height[rk[i]]$ ，则 $h[i] \geq h[i - 1] - 1$ 恒成立。

其实很容易理解，比如某个后缀是 cXZ ，它（字典序意义下的）上一个后缀是 cXY ，LCP 就是 cX （长度记为 $h[i - 1]$ ）。现在我们考察它（编号意义下的）下一个后缀 XZ ，我们已经知道字符串存在一个后缀 XY ，字典序比 XZ 小，而且它与 XZ 的 LCP 为 X （长度为 $h[i - 1] - 1$ ）。那么根据上面提到的性质，必然有 $h[i] \leq lcp(XZ, XY) = h[i - 1] - 1$ 。

利用这个结论，我们可以 $O(n)$ 出 $height$ 数组。

```
void init_ht(int n) {
    for (int i = 1, k = 0; i <= n; ++i) {
        if (k > 0) --k;
        while (s[i + k] == s[sa[rk[i] - 1] + k])
            ++k;
        ht[rk[i]] = k;
    }
}
```

8.5 确定有限状态自动机

通常在 OI 中提到的“自动机”往往指的是“确定有限状态自动机”，我们来看自动机的定义：

定义 8.5.1. 自动机是一个对信号序列进行判定的数学模型。

定义 8.5.1 中包含了许多名词：

- “**信号序列**”是指一连串有顺序的信号，例如字符串从前到后的每一个字符、数组从 1 到 n 的每一个数、数从高到低的每一位等。
- “**判定**”是指针对某一个命题给出**或真或假**的回答。

有时我们需要对一个信号序列进行判定，例如：判定一个二进制数是奇数还是偶数，判定一个字符串是否回文等等。

例如，你的选择序列是「家门 → 右拐 → 萍水西街 → 尚园街 → 古墩路 → 地铁站 → 下宁桥」，那你按顺序经过的路口可能是「家 → 家门口 → 萍水西街竞舟北路口 → 萍水西街尚圆街路口 → 尚园街古墩路口 → 古墩路中 → 三坝地铁站 → 下宁桥地铁站」。可以发现，上学的选择序列不止这一个。同样要去地铁站，你还可以从竞舟北路绕道，或者横穿文鼎苑抄近路。

而我们如果找到一个选择序列，就可以在地图上比划出这个选择序列能不能去学校。比如，如果一个选择序列是「家门 → 右拐 → 萍水西街 → 尚园街 → 古墩路 → 地铁站 → 钱江路 → 四号线站台 → 联庄」，那么它就不会带你去同一个学校，但是仍旧可能是一个可被接受的序列，因为目标地点可能不止一个。也就是说，我们通过这个地图和一组目的地，将信号序列分成了三类：

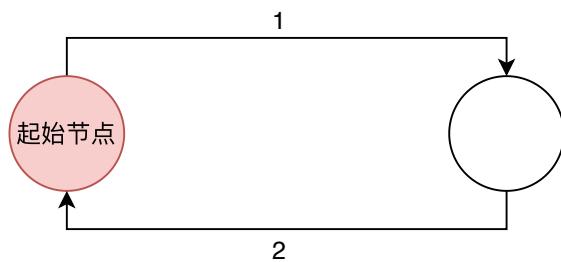
- 无法识别的信号序列，例如「家门 → ???」；
- 能去学校的信号序列；
- 不能去学校的信号序列。

我们将所有合法的信号序列分成了两类，完成了一个判定问题。

既然**自动机**是一个**数学模型**，那么显然不可能是一张地图。在我们对地图进行抽象之后，可以简化为一个有向图。因此，自动机的结构就是一张有向图。

而自动机的工作方式和流程图类似，不同的是：自动机的每一个结点都是一个判定结点；自动机的结点只是一个单纯的状态而非任务；自动机的边可以接受多种字符（不局限于 T 或 F）。

例如，完成「判断一个二进制数是不是偶数」的自动机如下：



从起始结点开始，从高到低接受这个数的二进制序列，然后看最终停在哪里，**如果最终停在红圈结点，则是偶数，否则不是**。

如果需要判定一个有限的信号序列和另外一个信号序列的关系（例如另一个信号序列是不是某个信号序列的子序列），那么常用的方法是针对那个有限的信号序列构建一个自动机。

需要注意的是，自动机只是一个**数学模型**，而不是**算法**，也不是**数据结构**。实现同一个自动机的方法有很多种，可能会有不一样的时空复杂度。

8.5.1 形式化定义

一个确定有限状态自动机 (DFA) 由五部分组成：

1. **字符集 (Σ)**，该自动机只能输入这些字符。
2. **状态集合 (Q)**。如果把一个 DFA 看成一张有向图，那么 DFA 中的状态就相当于图上的顶点。
3. **起始状态 ($start$)**, $start \in Q$, 一种特殊的状态，通常用 s 表示。
4. **接受状态集合 (F)**, $F \subseteq Q$, 是一组特殊的状态。

5. **转移函数 (δ)**, δ 是一个接受两个参数返回一个值的函数, 其中第一个参数和返回值都是一个状态, 第二个参数是字符集中的一个字符。如果把一个 DFA 看成一张有向图, 那么 DFA 中的转移函数就相当于顶点间的边, 而每条边上都有一个字符。

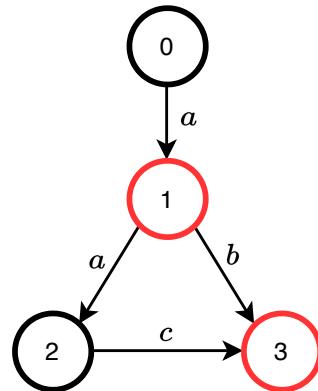
DFA 的作用就是**识别字符串**, 一个自动机 A , 若它能识别 (接受) 字符串 S , 那么 $A(S) = \text{True}$, 否则 $A(S) = \text{False}$ 。

当一个 DFA 读入一个字符串时, 从初始状态起按照转移函数一个一个字符地转移。如果读入完一个字符串的所有字符后位于一个接受状态, 那么我们称这个 DFA **接受**这个字符串, 反之我们称这个 DFA **不接受**这个字符串。

如果一个状态 v 没有字符 c 的转移, 那么我们令 $\delta(v, c) = \text{null}$, 而 null 只能转移到 null , 且 null **不属于**接受状态集合。无法转移到任何一个接受状态的状态都可以视作 null , 或者说, null 代指所有无法转移到任何一个接受状态的状态。

现在, 我们扩展定义转移函数 δ , 令其第二个参数可以接收一个字符串: $\delta(v, s) = \delta(\delta(v, s[1]), s[2..|s|])$, 扩展后的转移函数就可以表示从一个状态起接收一个字符串后转移到的状态。那么, $A(s) = [\delta(\text{start}, s) \in F]$ 。

例如, 一个接受且仅接受字符串 "a", "ab", "aac" 的 DFA:



我们再来看一个例子, 把字符串中的字符依次输入给 DFA, DFA 从初

始状态开始，不断在状态间转移。下图中的边权表示这个转移接受哪些字符，例如在 2 号状态输入 b，则会转移到 3 号状态；输入 c，则会转移到 1 号状态；输入其他字符则会进入 null 状态（一般不画出）。如果输入结束后，DFA 处于终止状态，就称这个字符串被接受了。例如，对于下图而言，a、acacab都可以被接受，而ac、abc不能被接受。

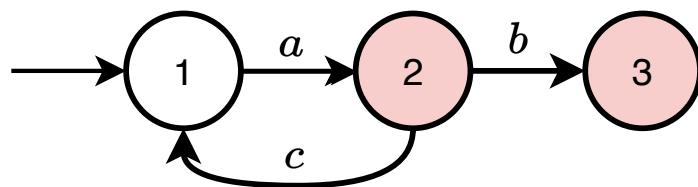


图 8.1: 无源箭头指向初始状态，绿色表示终止状态

能够被有限状态自动机描述的语言叫正则语言，上图表示的 DFA 就等价于正则表达式 $a(c a)^* b?$ 。

以上内容摘自 [oi-wiki](#)，如果看不懂没关系，我们来看一些例子：

例 8.5.2 (字符串转换整数 (atoi)). 请你来实现一个 `myAtoi(string s)` 函数，使其能将字符串转换成一个 32 位有符号整数 (类似 C/C++ 中的 `atoi` 函数)。

函数 `myAtoi(string s)` 的算法如下：

1. 读入字符串并丢弃无用的前导空格；
2. 检查下一个字符（假设还未到字符末尾）为正还是负号，读取该字符（如果有）。确定最终结果是负数还是正数。如果两者都不存在，则假定结果为正；
3. 读入下一个字符，直到到达下一个非数字字符或到达输入的结尾。字符串的其余部分将被忽略。
4. 将前面步骤读入的这些数字转换为整数（即， "123" → 123, "0032" → 32）。如果没有读入数字，则整数为 0。必要时更改符号（从步骤 2 开始）。

5. 如果整数数超过 32 位有符号整数范围 $[-2^{31}, 2^{31} - 1]$ ，需要截断这个整数，使其保持在这个范围内。具体来说，小于 -2^{31} 的整数应该被固定为 -2^{31} ，大于 $2^{31} - 1$ 的整数应该被固定为 $2^{31} - 1$ 。

返回整数作为最终结果。

输入格式

一行，包含一个字符串。

```
42
```

输出格式

一行，包含一个整数，代表最终结果。

```
42
```

样例输入 2

```
-42
```

样例输出 2

```
-42
```

样例输入 3

```
4193 with words
```

样例输出 3

```
4139
```

样例输入 4

```
+ -42
```

样例输出 4

```
0
```

样例输入 5

```
Word is 987
```

样例输出 5

```
0
```

例题8.5.2

8.6 后缀自动机

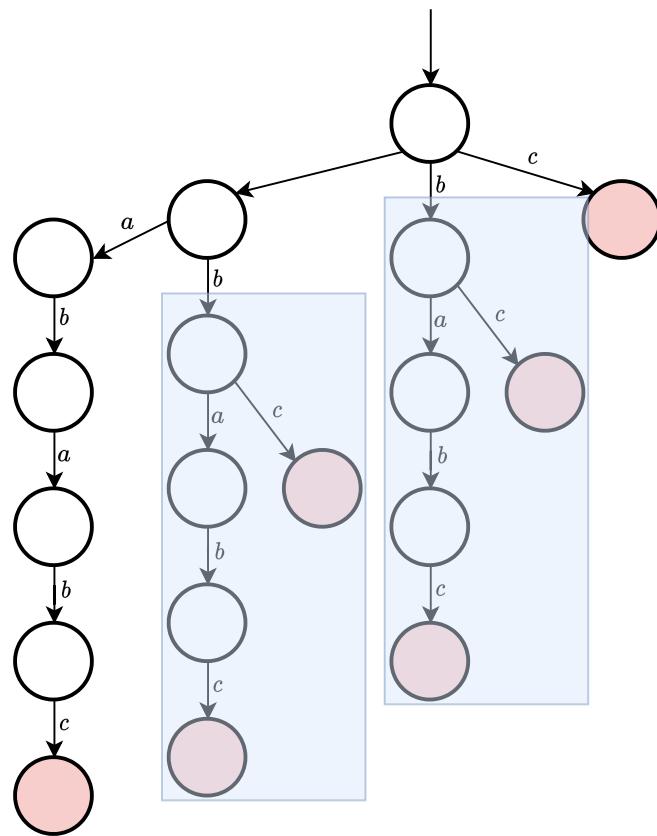
字符串 s 的后缀自动机 (SAM) 是一种特殊的 DFA，它符合以下性质：

1. 转移图是有向无环图，且每个转移接受且只接受一个字符；
2. 接受且只接受 s 的所有后缀；
3. 在符合以上性质的基础上，节点数最少。

直观上，字符串的 SAM 可以理解为给定字符串的所有子串的压缩形式。值得注意的事实是，SAM 将所有的这些信息以高度压缩的形式储存。对于一个长度为 n 的字符串，它的空间复杂度仅为 $O(n)$ 。此外，构造 SAM 的时间复杂度仅为 $O(n)$ 。

准确地说，一个 SAM 最多有 $2n - 1$ 个节点和 $3n - 4$ 条转移边。

如果只考虑上述提到的性质的前两条，其实只要把 s 的所有后缀插入字典树即可。例如，对于 $aababc$ ，在我们将自动机的示意图画出来后，我们不难发现其中存在着许多相似的结构：



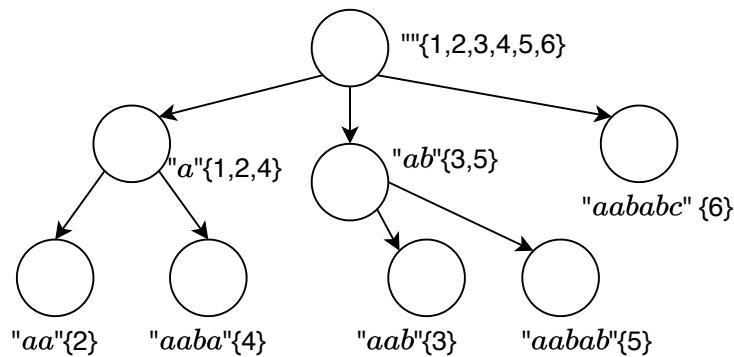
我们可以利用这种相似性，合并一些节点。

准确地说，我们定义 $\text{endpos}(p)$ 为模式串 p 在 s 中所有出现的结束位置的集合，那么 endpos 相同的模式串就可以共用一个节点。例如在 aababc 中， b 和 ab 的 endpos 都是 $3, 5$ ，说明它们总是一起出现，所以可以共用一个节点。我们把 endpos 相同的模式串组成的集合称为一个 endpos 等价类。

显然，对于同一个等价类内的两个模式串 p_1 和 p_2 ，如果 $p_1 < p_2$ ，那么 p_1 一定是 p_2 的后缀。因此，在一个等价类中，必然存在且只存在一个最长的串 p 。对于 p 的所有出现，根据位于 p 前面的那一个字符是什么，可以把原等价类划分为若干个等价类。例如在 aababc 中，我们已经知道 ab 出现了两次，但首次出现 ($\text{endpos} = 3$) 时前面那一个字符是 a ，再次出现 ($\text{endpos} = 5$) 时前面那一个字符是 b ， aab 和 bab 属于两个不同的等价类，这就把 $3, 5$ 划分成了 3 和 5 。需要注意的是，因为 p 前面可能没有字符（此时 p 是 s 的前缀），

所以划分后有可能丢失一个元素。

受此启发，可以发现一个重要的性质：*endpos* 等价类的数量是 $O(n)$ 级别的。不妨设空串所属的 *endpos* 等价类为 $1, 2, \dots, n$ ，那么所有等价类都可以认为是由此一步步划分而来的，形成一个树的结构，总数不会超过 $2n - 1$ （极限情况是满二叉树）。



这棵树称作 *parent tree*，它的节点与后缀自动机的状态一一对应。

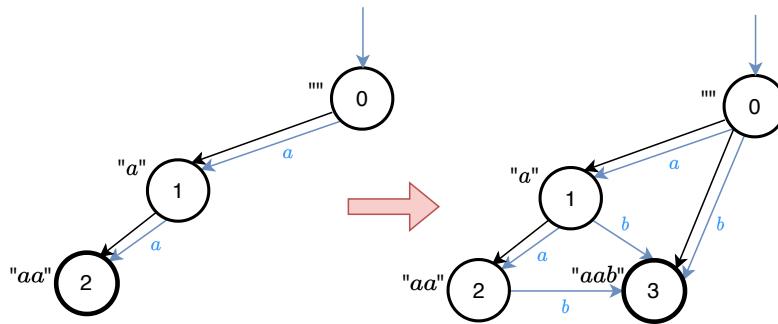
现在我们来一边构造 *parent tree* 一边构造后缀自动机。我们采取**动态构造**的方法，即一个一个地把字符加进去。那么问题转化为“向一个已知的字符串的后面添加一个字符，它的 SAM 会如何变化”。

设 $st.\text{nxt}[\text{ch}]$ 表示状态 st 接受字符 ch 后转移到的状态， $st.\text{fa}$ 为该状态在 *parent tree* 上的父节点， $st.\text{len}$ 为该状态对应的 *endpos* 等价类中最长串的长度， last 为加入新字符前整个字符串所在的等价类对应的状态。

一个基本的观察是：从 last 开始在 *parent* 树上往上爬，一定能遍历加入新字符前所有**后缀**的对应节点。加入一个新的字符，其实就是给之前部分后缀新增了转移。所以当新增字符 ch 时，我们创建一个新状态 cur （其中令 $\text{cur}.\text{len} = \text{last}.\text{len} + 1$ ），然后从 last 开始往上爬，对于遇到的每个状态 p ，如果 p 还不能通过 ch 转移，那我们就新增一个转移 $(p.\text{nxt}[\text{ch}] = \text{cur})$ ，然后继续往上爬，直到某个 p 可以通过 ch 转移到状态 q ，或者处理完根节点为止。这分为三种情况：

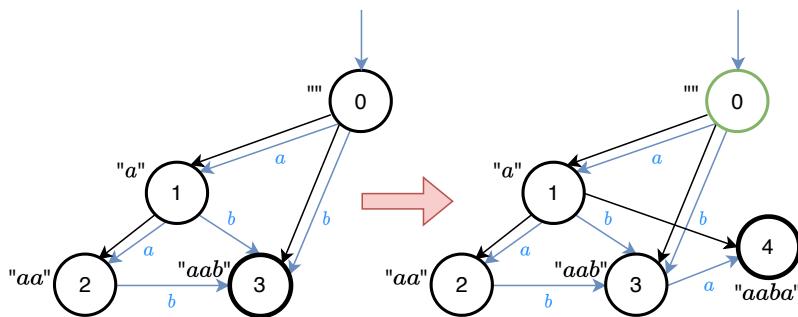
- 没有找到要找的 q 。这可能出现在加入了从未加入过的字符的时候，

此时直接令 `cur.fa` 为根节点然后退出即可。(退出后还需要把 `last` 设为 `cur`，这对三种情况都是一样的。)



黑色边为 **parent tree** 的边，蓝色边为SAM的转移边，圈内标的是最长串长度，旁边标注了最长串，加粗边框的为 `last`。

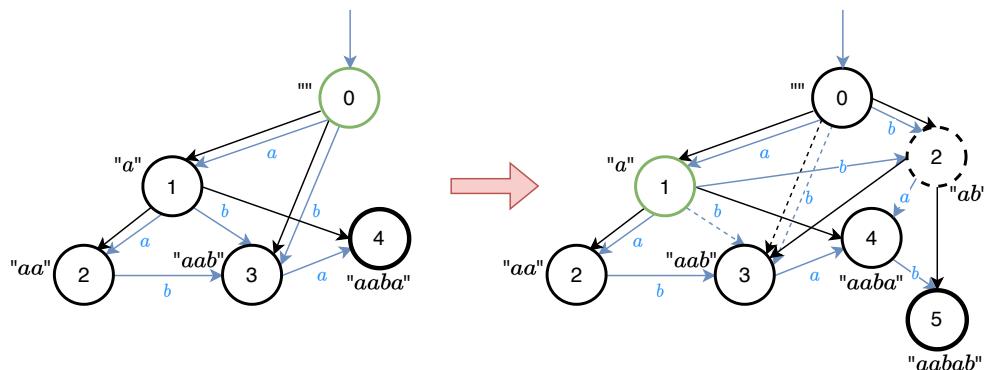
- $p.\text{len} + 1 == q.\text{len}$ 。这时我们说 $p \xrightarrow{ch} q$ 是一个连续的转移。这种情况下，我们直接令 `cur.fa = q` 然后退出即可。这是因为 `p` 所对应集合中每个字符串在后面加上一个 `ch` 都能构成一个 `q` 对应集合的字符串，而 `p` 对应集合都是原字符串的后缀，所以 `q` 对应集合都是新字符串的后缀，应作为 `cur` 的父亲节点。



橙色表示 `p`，紫色表示 `q`，爬到 `p` 的时候发现可以通过字符 `a` 转移到 `q`，且 $\text{len}[p] + 1 == \text{len}[q]$ ，属于第二种情况。

- $p.\text{len} + 1 != q.\text{len}$ 。这时我们说 $p \xrightarrow{ch} q$ 是一个不连续的转移。我们新建一个 `r` 节点，它拥有 `q` 节点的所有出边，且 `fa` 也与 `q` 节点相同，但

是 $r.\text{len} = p.\text{len} + 1$ 。我们从 p 节点继续往上爬，把所有接受 ch 而到达 q 的转移的目标改为 r (注意只要有一个节点不能接受 ch 那它的祖先都不能接受 ch ，要及时退出循环)。最后令 $\text{cur.fa} = \text{q.fa} = r$ 。这里不能用2中的方法是因为 q 不仅仅包含新字符串的后缀，比如下图中3号点除了"ab"还包含了"aab"，我们不得不将它拆分开。



这种情况相当于把原来的一个节点拆分成两个节点，新节点相当于是原节点的一个后缀。

在构建完SAM后，我们可以标记所有的终止节点——从代表整个字符串的节点(`last`)沿 `parent tree` 往上爬，途径的所有节点就是终止节点，它们代表了字符串的所有后缀。

```
namespace SAM {
    const int MAXM = 1e6 + 5;
    struct State {
        int fa, len, next[26];
    } sam[MAXM];
    int cnt = 1, last = 1;
    void insert(int ch) { // 插入时要 - 'a' (或其他)
        int cur = ++cnt, p;
        sam[cur].len = sam[last].len + 1;
        for (p = last; p && !sam[p].next[ch]; p = sam[p].fa)
```

```

    sam[p].next[ch] = cur;
    int q = sam[p].next[ch];
    if (q == 0) {
        sam[cur].fa = 1;
    } else if (sam[p].len + 1 == sam[q].len) {
        sam[cur].fa = q;
    } else {
        int r = ++cnt;
        sam[r] = sam[q];
        sam[r].len = sam[p].len + 1;
        for (; p && sam[p].next[ch] == q; p = sam[p].fa)
            sam[p].next[ch] = r;
        sam[cur].fa = sam[q].fa = r;
    }
    last = cur;
}
} // namespace SAM

```

一些规模方面的结论：

1. SAM的总状态数不超过 $2n - 1$, 当字符串形如abbb...时取到上界;
2. SAM的总转移数不超过 $3n - 4$, 当字符串形如abbb...bc时取到上界;
3. 构建SAM的复杂度为 $O(|\Sigma|n)$, Σ 为字符集。(瓶颈是复制节点, 所以如果字符集过大, 可以考虑用map代替数组。)

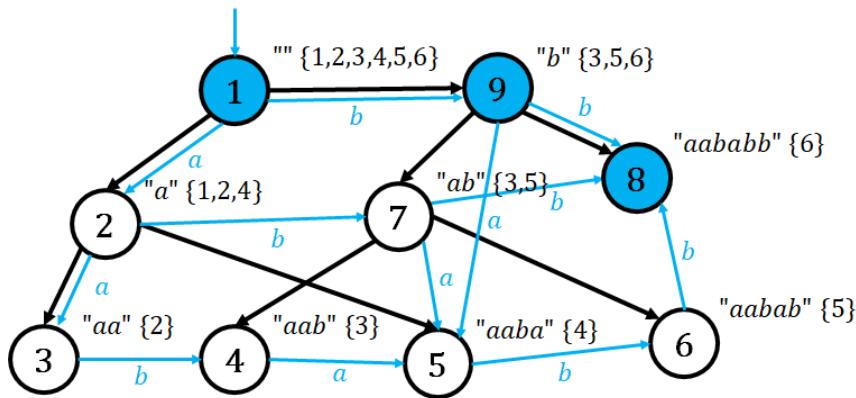
SAM的应用极其广泛, 这里举几个例子:

1. 求模式串p是否在字符串s中出现过:

根据p的字符在SAM上进行转移, 如果在某一步无法转移则代表没有出现过。

例如下图中, 当p为"aba"时, 走 $1 \rightarrow 2 \rightarrow 6 \rightarrow 5$ 可以处理完整个模

式串，所以该模式串出现过； p 为"baa"时，走 $1 \rightarrow 3 \rightarrow 5$ 后便无法再转移，所以该模式串未出现过。



此外，不难发现这个方法还可以求出出现过的最长前缀。

2. 求模式串 p 在字符串 s 中出现的次数：

其实就是 p 对应节点的 endpos 集合的大小，这个直接在 **parent** 树上 **dp** 就可以了。注意那些原字符串的前缀对应的节点，划分后 **endpos** 集合会丢失一个元素，所以这些节点的 **dp** 值要比子节点 **dp** 值之和多 1。这个直接在建 **SAM** 的时候预处理就行，直接令 **dp[cur] = 1**。

3. 求字符串 s 的最小表示法（循环同构字符串中字典序最小的一个）：

把 s 重复一遍，建出其 **SAM**，然后从开始节点往下走 $|s|$ 步，每次都走最小字符对应的边，得到的显然便是最小表示法。

4. 求 s 的本质不同子串个数：

从 **parent tree** 的角度，每个节点代表了 $p.\text{len} - \text{fa}(p).\text{len}$ 个子串，且每个节点代表的子串各不相同，所以只要对每个节点计算这个值然后相加即可。这个可以在建 **SAM** 途中动态维护。

5. 最长公共子串：

为 s_1 建 **SAM**，对 s_2 每一个前缀求它在 s_1 中出现过的最长后缀。这

个可以用类似 KMP 的方法，先尽量转移，转移不动则跳到 **parent tree** 上的父亲节点继续尝试转移，直到跳到根节点。整个过程中要维护当前最长后缀的长度。

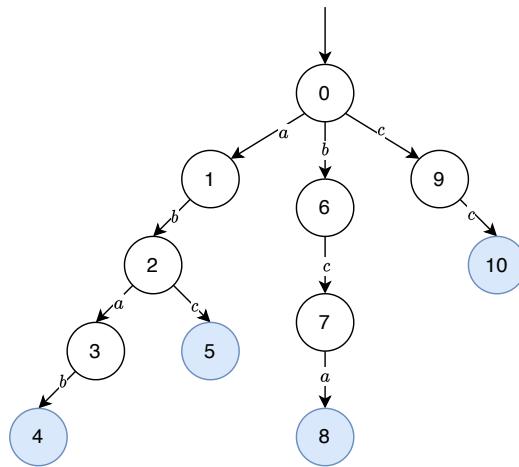
```
int lcs(const string &s) {
    int l = 0, p = 1, ans = 0;
    for (auto c : s) {
        c -= 'a';
        while (p != 1 && sam[p].next[c] == 0)
            p = sam[p].fa, l = sam[p].len;
        if (sam[p].next[c]) p = sam[p].next[c], l++;
        if (l > ans) ans = l;
    }
    return ans;
}
```

8.7 AC 自动机

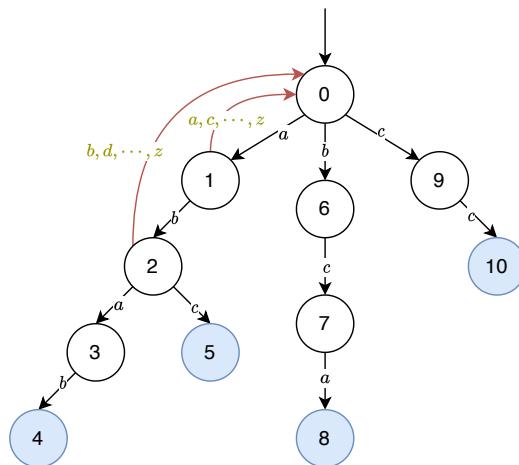
AC 自动机是以 **Trie** 的结构为基础，结合 **KMP** 的思想建立的自动机，用于解决多模式匹配等任务。

在开始讨论 AC 自动机之前，我们先来思考一个问题：给出若干个模式串，如何构建一个 DFA(即确定有限状态自动机)，接受所有以任一模式串结尾的文本串？

或者我们思考一个更简单的问题：如何构建接受所有模式串的 DFA？很明显，字典树就可以看做符合要求的自动机。例如，有模式串"abab"、"abc"、"bca"、"cc"，我们把它们插入**字典树**，可以得到：



为了使它不仅接受模式串，还接受以模式串结尾的文本串，一个看起来挺正确的改动是：使每个状态接受所有原先不能接受的字符，转移到初始状态（即根节点）。



图中仅画出了两个状态到根的转移，实际上每个状态都有到根的转移。

现在我们检查一下，我们发现"abbca"、"acbacc"可以接受。但是如果我们将尝试"abca"，我们会发现我们的自动机并不能接受它。稍加观察发现，我们在状态 5 接受 a 应该跳到状态 8 才对，而不是初始状态。某种意义上来说，状态 7 是状态 5 退而求其次的选择，因为状态 7 在 trie 上对应的字符串"bc"是状态 5 对应的字符串"abc"的后缀。既然状态 5 原本不能接受"abca"，

我们完全可以退而求其次看看状态 7 是否可以接受。这看起来很像 KMP 算法，因此，AC 自动机常常被人称作 **trie 上的 KMP**。

所以我们给每个状态分配一条`fail`边，它连向的是该状态对应字符串在 trie 上存在的最长真后缀所对应的状态。我们令所有状态`p`接受原来不能接受的字符`c`，转移到 `next(fail(p), c)`，特别地，根节点转移到自己。

为什么不需要像 KMP 算法一样，用一个循环不断进行退而求其次的选择呢？因为如果我们用 BFS 的方式进行上面的重构，我们可以保证 `fail(p)` 在`p`重构前已经重构完成了，类似于动态规划。

接下来的问题是建`fail`边。其实很简单，假设状态`p`(不为初始状态) 原本就可以接受字符`c`，那么有 `fail(next(p, c)) = next(fail(p), c)`，因为`p`对应字符串加上一个`c`后，其最长真后缀就是`p`对应字符串本身的最长真后缀加上一个`c`。特别地，如果`p`是初始状态则 `fail(next(p, c)) = p`。

于是，我们可以在一趟 BFS 中，一边建`fail`边一边重构自动机。

```

namespace acm {
    const int M = 1e6 + 5;
    int cnt;
    struct node {
        int next[26], fail, end;
    } G[M];
    void insert(const string &s) {
        int p = 0;
        for (auto c : s) {
            int &q = G[p].next[c - 'a'];
            if (!q) q = ++cnt;
            p = q;
        }
        G[p].end++;
    }
    void bfs() {
        queue<int> Q;

```

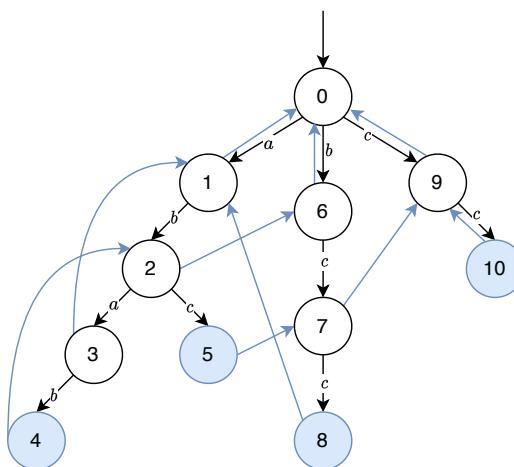
```

for (int i = 0; i < 26; ++i)
    if (G[0].next[i]) Q.push(G[0].next[i]);
while (Q.size()) {
    int u = Q.front();
    Q.pop();
    for (int i = 0; i < 26; ++i) {
        int &v = G[u].next[i], w = G[G[u].fail].next[i];
        if (v)
            G[v].fail = w, Q.push(v);
        else
            v = w;
    }
}
}

} // namespace acm

```

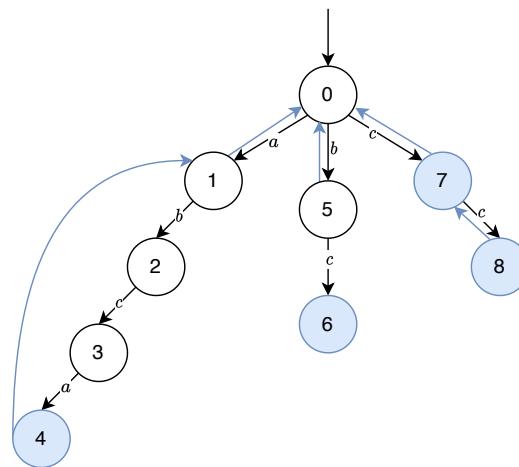
这样建fail边和重构完成后得到的自动机称为 **AC 自动机 (Aho-Corasick Automation)**。很明显建fail边和重构的总时间复杂度是 $O(n|\Sigma|)$ 。



蓝色边标出了所有 fail 边，重构的时候加入的边未画出。

我们发现fail边也形成一棵树，所以 AC 自动机包含两棵树：trie树和fail树。一个重要的性质是：如果当前状态 s 在某个终止状态 t 的fail树

的子树上，那么当前文本串就与 t 所对应模式串匹配。上面的例图可能不方便看出这个性质，我们来看模式串"abca"、"bc"、"c"和"cc"对应的 AC 自动机：



可以看出以"abc"、"bc"、"cc"和"c"结尾的文本串都与模式串"c"匹配，而以"abc"和"bc"结尾的文本串都与模式串"bc"匹配。我们可以利用这个性质进行多模式匹配。

接下来举一些 AC 自动机的应用：

- 给出若干个模式串 p_i 和一个文本串 s ，求有多少个模式串在文本串中出现过

将文本串输入自动机，每经过一个状态就向上爬fail树看在哪些终止状态的子树里。因为我们只需要知道是否出现过，不需要知道出现过几次，所以每次爬fail树可以把途经的点标记一下，下次就不爬了。这样的话查询的总复杂度不会超过状态数。

- 给出若干个模式串 p_i 和一个文本串 s ，求每个模式串在文本串中出现的次数

首先需要加强一下 insert 函数，记录一下每个终止状态对应哪些模式串。

```
int cnt, cid, C[N];
struct node {
    int next[26], fail;
    vector<int> ids; // ids不为空则为终止状态
} G[M];
void insert(const string &s) {
    int p = 0;
    for (auto c : s) {
        int &q = G[p].next[c - 'a'];
        if (!q) q = ++cnt;
        p = q;
    }
    G[p].ids.push_back(++cid);
}
```

查询的时候标记访问过的每个状态，最后做一趟树形 dp 计算每个状态的子树总共被经过多少次。

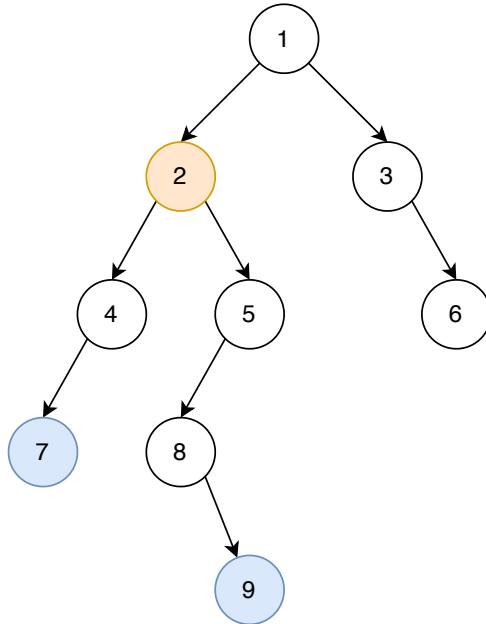
3. 给出若干模式串 p_i ，是否存在无限长度的文本串不匹配任一模式串
标记终止状态及其在 fail 树上的所有子节点为不可访问点，把剩下的
点建个图，跑一趟 DFS 看是否存在环。

第九章 树上的问题

9.1 最近公共祖先问题

在我们处理树上点与点关系的问题时,(例如,最简单的,树上两点的距离),常常需要获知树上两点的**最近公共祖先 (Lowest Common Ancestor, LCA)**。

例如我们来看一道**最近公共祖先**的问题:



我们要找到节点 7 和节点 9 最近的那个公共祖先节点,一个论朴素的做法是,首先进行一趟 `dfs`,求出每个点的深度:

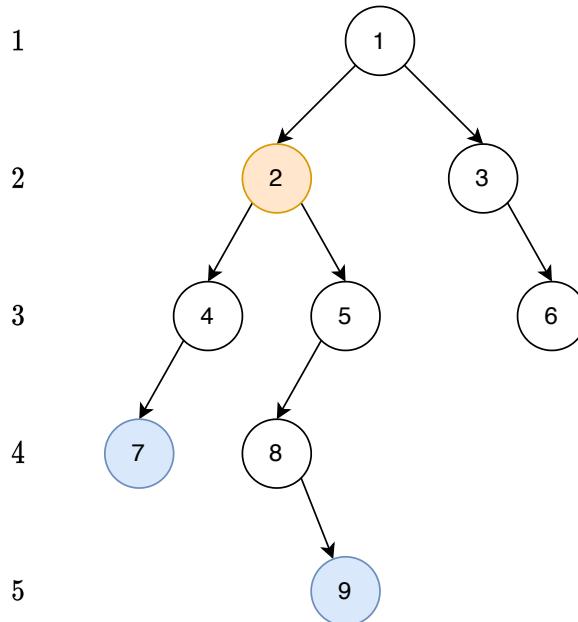
```

int dep[MAXN];
bool vis[MAXN];
void dfs(int cur, int fath = 0) {
    if (vis[cur]) {
        return;
    }
    vis[cur] = true;
    dep[cur] = dep[fath] + 1; // 每个点的深度等于父节点的深度+1
    for (int eg = head[cur]; eg != 0; eg = edges[eg].next) {
        dfs(edges[eg].to, cur);
    }
}

```

现在节点 9 比节点 7 的深度深，所以我们先让节点 9 往上“爬”，爬到与节点 7 深度相等为止。然后节点 7 和节点 9 再一起往上爬，直到两点相遇，那一点即为 LCA，这样下来，每次查询 LCA 的最坏时间复杂度是 $O(n)$ ：

Depth



有时候，我们需要进行很多次查询，这时朴素的 $O(n)$ 复杂度就不够用了，我们考虑空间换时间的**倍增算法**。我们用一个数组 $fa[i][k]$ 存储节点 i 的 2^k 级祖先，父节点为 1 级祖先，祖父节点为 2 级祖先 … 以此类推。

这本质上是一种**二进制拼凑**的思想，我们知道所有的十进制数都有其对应的二进制形式，其可以表示为 $D = 2^0 + 2^1 + \dots + 2^{k-1}$ 。现在我们需要求出两个点的深度差，即 $dep[x] - dep[y]$ ，但在实际操作中我们并不需要真的求出这个值，我们往往只需要 $dep[fa[x, k]] \geq dep[y]$ 是否成立即可，如果成立，则代表跳完后的点深度仍然大于 y ，继续往上跳，直到跳到与 y 同深度为止。这个过程可以在 dfs 途中动态规划得出：

```
// 在 dfs 中 ...
fa[cur][0] = fath;
for (int i = 1; i <= Log2[dep[cur]]; ++i)
    fa[cur][i] = fa[fa[cur][i - 1]][i - 1];
```

这样，往上爬的次数可以被大大缩短，首先还是先让两点深度相等：

```
if (dep[a] > dep[b]) // 不妨设 a 的深度小于等于 b
    swap(a, b);
while (dep[a] != dep[b]) // 跳到深度相等为止
    b = fa[b][Log2[dep[b] - dep[a]]]; // b 不断往上跳
```

例如，节点 a 和节点 b 本来相差 22 的深度，现在 b 不用往上爬 22 次，只需要依次跳 16、4、2 个单位，3 次便能达到与 a 相同的距离。

两者深度相等后，如果两个点已经相遇，那么问题就得以解决。如果尚未相遇，我们再让它们一起往上，同时我们也要意识到，我们并不需要让两个点跳到最近公共祖先，而是只需要让两个跳到最近公共祖先的子节点即可。

问题在于，如何确定每次要跳多少？正面解决也许不太容易，我们转而思考另一个问题：**如何在 a 、 b 不相遇的情况下跳到尽可能高的位置？** 如果找到了这个位置，它的父亲就是 LCA 了。

我们从可能跳的最大步数 $\log_2(\text{dep}[a])$ 开始，这样至多跳到 0 号点，不会越界，并不断减半步数：

```
for (int k = Log2[dep[a]]; k >= 0; k--)
    if (fa[a][k] != fa[b][k])
        a = fa[a][k], b = fa[b][k];
```

我们回过头看刚刚棵树，先尝试 $\log_2 4 = 2$ ，A、B 点的 2^2 级祖先都是 0，所以不跳。然后尝试 1，A、B 的 2^1 祖先都是 2，也不跳。最后尝试 0，A、B 的 1 级祖先分别是 4 和 5，跳。从而得到了答案，再往上一格所得到的 2 号点就是所求的最近公共祖先。

```
int Log2[MAXN], fa[MAXN][20], dep[MAXN]; // fa 的第二维大小不应
小于 log2(MAXN)
bool vis[MAXN];

void dfs(int cur, int fath = 0) {
    if (vis[cur])
        return;
    vis[cur] = true;
    dep[cur] = dep[fath] + 1;
    fa[cur][0] = fath;
    for (int i = 1; i <= Log2[dep[cur]]; ++i)
        fa[cur][i] = fa[fa[cur][i - 1]][i - 1];
    for (int eg = head[cur]; eg != 0; eg = edges[eg].next)
        dfs(edges[eg].to, cur);
}

int lca(int a, int b) {
    if (dep[a] > dep[b])
        swap(a, b);
    while (dep[a] != dep[b])
        b = fa[b][Log2[dep[b] - dep[a]]];
    if (a == b)
```

```

    return a;

for (int k = Log2[dep[a]]; k >= 0; k--)
    if (fa[a][k] != fa[b][k])
        a = fa[a][k], b = fa[b][k];
return fa[a][0];
}

int main() {
// ...
for (int i = 2; i <= n; ++i)
    Log2[i] = Log2[i / 2] + 1;
// ...
dfs(s); // 无根树可以随意选一点为根
// ...
return 0;
}

```

至于树上两点 u, v 的距离，有公式 $\text{dis}_{u,v} = \text{dep}_u + \text{dep}_v - 2\text{dep}_{\text{LCA}(u,v)}$ 。
 $O(n \log n)$ 预处理， $O(\log n)$ 查询，空间复杂度为 $O(n \log n)$ 。

需要提一下，以上都是针对**无权树**，如果树有权值，可以额外记录一下每个点到根的距离，然后用几乎相同的公式求出。

我们还是来看具体的题目：

例 9.1.1 (luogu-p3379 【模板】最近公共祖先 (LCA)). 如题，给定一棵有根多叉树，请求出指定两个点直接最近的公共祖先。

输入格式：

第一行包含三个正整数 N, M, S ，分别表示树的结点个数、询问的个数和树根结点的序号。

接下来 $N - 1$ 行每行包含两个正整数 x, y ，表示 x 结点和 y 结点之间有一条直接连接的边（数据保证可以构成树）。

接下来 M 行每行包含两个正整数 a, b , 表示询问 a 结点和 b 结点的最近公共祖先。

```
5 5 4
3 1
2 4
5 1
1 4
2 4
3 2
3 5
1 2
4 5
```

输出格式:

输出包含 M 行, 每行包含一个正整数, 依次为每一个询问的结果。

```
4
4
1
4
4
```

例题9.1.1在建树时可以采用存图的方式进行构建, 此外, 我们还需要设置一个哨兵节点, 当我们在执行倍增跳时, 可能会导致超过根节点, 即如果从节点 i 开始跳 2^j 步, 如果超过了根节点, 则设 $\text{fa}[i][j] = 0$, 并规定 $\text{depth}[0] = 0$ 。我们将先前的模板补全即可得到本题的解决方案:

```
#include <bits/stdc++.h>

using namespace std;

const int MAXN = 500010;
int n, m, s, x, y;
int Log2[MAXN], fa[MAXN][20], dep[MAXN], head[MAXN];
```

```
bool vis[MAXN];  
struct Edge {  
    int to, next;  
} edges[MAXN * 2];  
int idx = 0;  
  
void addEdge(int from, int to) {  
    edges[++idx] = (Edge) {to, head[from]};  
    head[from] = idx;  
}  
  
void dfs(int cur, int fath = 0) {  
    if (vis[cur]) {  
        return;  
    }  
    vis[cur] = true;  
    dep[cur] = dep[fath] + 1;  
    fa[cur][0] = fath;  
    for (int i = 1; i <= Log2[dep[cur]]; ++i) {  
        fa[cur][i] = fa[fa[cur][i - 1]][i - 1];  
    }  
    for (int eg = head[cur]; eg != 0; eg = edges[eg].next) {  
        dfs(edges[eg].to, cur);  
    }  
}  
  
int lca(int a, int b) {  
    if (dep[a] > dep[b]) {  
        swap(a, b);  
    }  
    for (int i = Log2[dep[b] - dep[a]]; i >= 0; --i) {  
        if (dep[fa[b][i]] >= dep[a]) {  
            b = fa[b][i];  
        }  
    }  
    while (a != b) {  
        a = fa[a][0];  
        b = fa[b][0];  
    }  
    return a;  
}
```

```
    }

}

if (a == b) {
    return a;
}

for (int k = Log2[dep[a]]; k >= 0; k--) {
    if (fa[a][k] != fa[b][k]) {
        a = fa[a][k], b = fa[b][k];
    }
}

return fa[a][0];
}

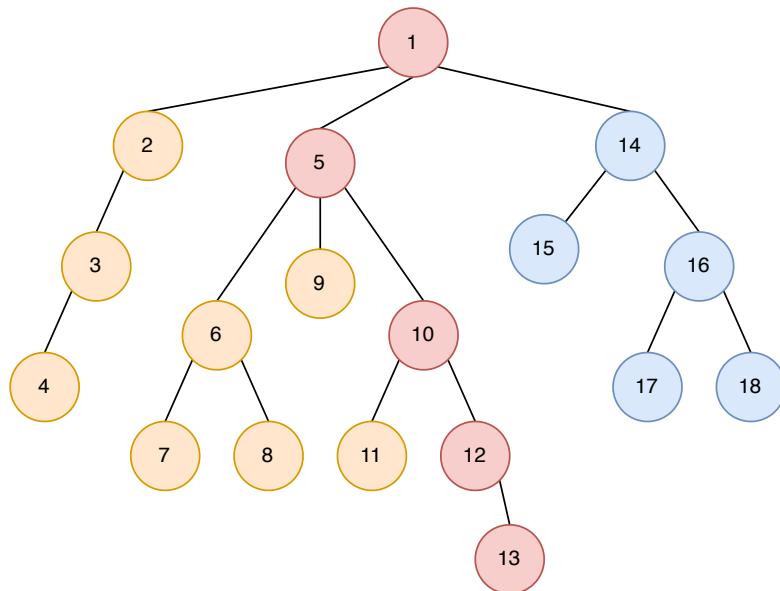
int main() {
    scanf("%d%d%d", &n, &m, &s);
    for (int i = 2; i <= n; ++i) {
        Log2[i] = Log2[i / 2] + 1;
    }
    fill(head, head + n + 1, 0);
    for (int i = 1; i < n; i++) {
        scanf("%d%d", &x, &y);
        addEdge(x, y);
        addEdge(y, x);
    }
    dfs(s);
    while (m--) {
        scanf("%d%d", &x, &y);
        printf("%d\n", lca(x, y));
    }
    return 0;
}
```

9.1.1 Tarjan 算法

为了求解 LCA 问题，我们最朴素的想法就是，从一个点出发向上回溯并标记每个经历过的点，直到回溯到根节点，再从另一个点出发，碰到的第一个被标记的点就是两点的最近公共祖先问题，这个算法被称为向上标记法。

Tarjan 算法是对向上标记法的一种改进，在我们进行深度优先遍历时，将点分为三大类：

- 已经搜索过，且经过回溯的点；
- 正在搜索的分支；
- 还未搜索到的点。



参照上图，例如我们正在搜索 13 号节点，我们去找所有与这个点相关的询问，从图中不难发现发现一些特点：

- 如果我们要求节点 4 和节点 13 的最近公共祖先，这个点会是 1；

- 如果我们要求节点 7 和节点 13 的最近公共祖先，这个点会是 5；
- 如果我们要求节点 11 和节点 13 的最近公共祖先，这个点会是 10。

这也就意味着，我们可以将已经搜索过的这一类点进行一次合并，例如图中，我们将节点 2、3、4 都合并到节点 1 中，只要访问其中的点，我们给出 1 号节点作为答案即可。同样的，这将很容易想到用并查集来实现，合并的时机是我们回溯完这个节点之后。

我们仍然根据例题9.1.1来实现，这里直接给出代码：

```
#include <bits/stdc++.h>

using namespace std;

const int N = 500010;
int n, m, s, idx;
int h[N], e[N * 2], ne[N * 2], p[N], res[N], st[N];
vector<pair<int, int>> query[N];

void add(int a, int b) {
    e[idx] = b, ne[idx] = h[a], h[a] = idx++;
}

int find(int x) {
    if (p[x] != x) {
        p[x] = find(p[x]);
    }
    return p[x];
}

void tarjan(int u) {
    st[u] = 1;
    for (int i = h[u]; ~i; i = ne[i]) {
        int j = e[i];
```

```
// 如果点未遍历
if (!st[j]) {
    tarjan(j);
    // 合并两个集合
    p[j] = u;
}
}

// 遍历所有与 u 相关的查询
for (auto& [y, id] : query[u]) {
    if (st[y] == 2) {
        res[id] = find(y); // 直接存储最近公共祖先的编号
    }
}

// 当前节点已遍历
st[u] = 2;
}

int main() {
    scanf("%d%d%d", &n, &m, &s);
    memset(h, -1, sizeof h);
    for (int i = 0; i < n - 1; i++) {
        int a, b;
        scanf("%d%d", &a, &b);
        add(a, b), add(b, a);
    }

    int query_count = 0; // 记录查询的编号
    for (int i = 0; i < m; i++) {
        int a, b;
        scanf("%d%d", &a, &b);
        if (a == b) {
            res[query_count] = a; // 如果a和b相同，最近公共祖先就是它们自己
        }
    }
}
```

```
    } else {
        // 和 a 相关的另一个节点是 b, 编号是 query_count
        query[a].push_back({b, query_count});
        // 和 b 相关的另一个节点是 a, 编号是 query_count
        query[b].push_back({a, query_count});
    }
    query_count++;
}

// 初始化并查集
for (int i = 1; i <= n; i++) {
    p[i] = i, st[i] = 0;
}
// 从根节点开始
tarjan(s);

for (int i = 0; i < query_count; i++) {
    printf("%d\n", res[i]);
}

return 0;
}
```

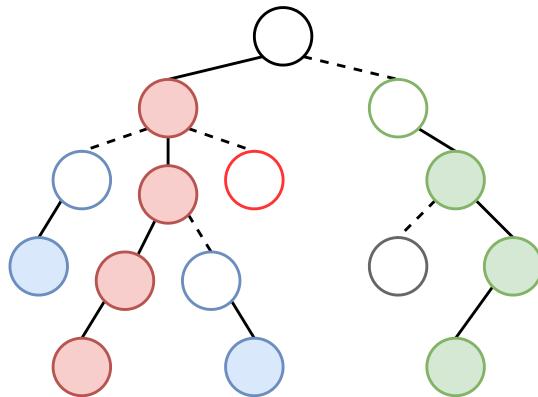
程序中，我们将未被访问到的点标记为 0，已访问的点标记为 2，并通过 `st[u]` 数组来进行存储。

9.2 重链剖分

树链剖分是把一棵树分割成若干条链，以便于维护信息的一种方法，其中最常用的是**重链剖分 (Heavy Path Decomposition, 重路径分解)**，所以一般提到树链剖分或树剖都是指**重链剖分**。

我们定义树上一个节点的子节点中子树最大的一个为它的**重子节点**，

其余的为**轻子节点**。一个节点连向其重子节点的边称为**重边**，连向轻子节点的边则为**轻边**。



如果把根节点看作轻的，那么从每个轻节点出发，不断向下走重边，都对应了一条链，于是我们把树剖分成了 l 条链，其中 l 是轻节点的数量。

重链剖分有一个重要的性质：对于节点数为 n 的树，从任意节点向上走到根节点，经过的轻边数量不超过 $\log n$ 。这是因为，如果一个节点连向父节点的边是轻边，就必然存在子树不小于它的兄弟节点，那么父节点对应子树的大小一定超过该节点的两倍。每经过一条轻边，子树大小就翻倍，所以最多只能经过 $\log n$ 条。

我们通过两次dfs来进行重链剖分：

第一趟 dfs，先得到每个节点的fa(父节点)、sz(子树大小)、dep(深度)、hson(重子节点)：

```
void dfs1(int p, int d = 1) {
    int size = 1, maxTree = 0;
    dep[p] = d;
    for (auto q: edges[p])
        if (!dep[q]) {
            dfs1(q, d + 1);
            fa[q] = p;
            size += sz[q];
            if (sz[q] > maxTree)
```

```
    hson[p] = q, maxTree = sz[q];  
}  
sz[p] = size;  
}
```

第二趟dfs，得到每个节点的top(链头，即所在的重链中深度最小的那个节点)：

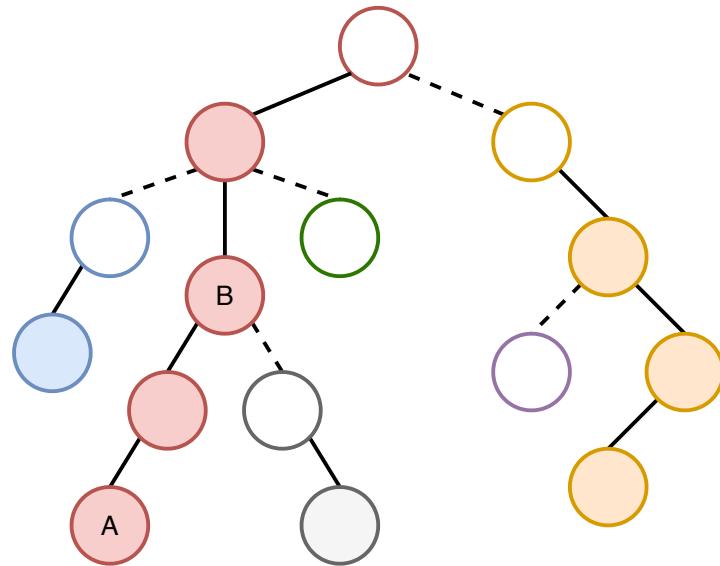
```
// 需要先把根节点的 top 初始化为自身  
void dfs2(int p) {  
    for (auto q: edges[p]) {  
        if (!top[q]) {  
            if (q == hson[p])  
                top[q] = top[p];  
            else  
                top[q] = q;  
            dfs2(q);  
        }  
    }  
}
```

这样便完成了剖分。

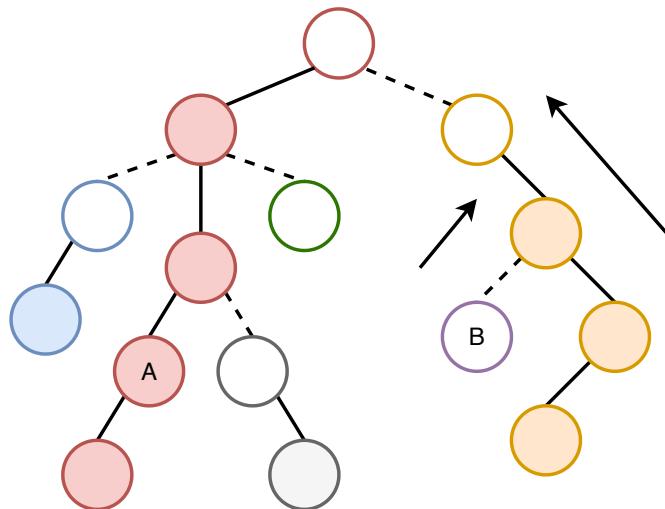
接下来我们来看一些应用：

9.2.1 LCA 问题

树剖可以单次 $O(\log n)$ 地求 LCA，且常数较小。假如我们要求两个节点的 LCA，如果它们在同一条链上，那直接输出深度较小的那个节点就可以了。



否则，LCA 要么在链头深度较小的那条链上，要么就是两个链头的父节点的 LCA，但绝不可能在链头深度较大的那条链上。所以我们可以直接把链头深度较大的节点用其链头的父节点代替，然后继续求它与另一者的 LCA。



由于在链上我们可以 $O(1)$ 地跳转，每条链间由轻边连接，而经过轻边的次数又不超过 $\log n$ ，所以我们实现了 $O(\log n)$ 的 LCA 查询。

```
int lca(int a, int b) {
    while (top[a] != top[b]) {
        if (dep[top[a]] > dep[top[b]])
            a = fa[top[a]];
        else
            b = fa[top[b]];
    }
    return (dep[a] > dep[b] ? b : a);
}
```

综上所述，这里给出一个可行的程序：

```
#include<bits/stdc++.h>

using namespace std;

const int MAXN = 500005;

vector<int> edges[MAXN];

// 节点深度，当前节点的父节点，链头节点，子树大小，重子节点
int dep[MAXN], fa[MAXN], top[MAXN], sz[MAXN], hson[MAXN];

// DFS1：计算每个节点的深度、子树大小、父节点和重儿子
void dfs1(int p, int d = 1) {
    int size = 1, maxTree = 0;
    dep[p] = d; // 设置当前节点的深度
    for (auto q: edges[p])
        if (!dep[q]) {
            fa[q] = p; // 设置 q 的父节点为 p
            dfs1(q, d + 1);
            size += sz[q]; // 累加子树大小
            if (sz[q] > maxTree) // 找到最大的子树
                hson[p] = q, maxTree = sz[q]; // 设置重子节点
        }
}
```

```
    }

    sz[p] = size; // 设置当前节点的子树大小
}

// DFS2: 设置每个节点的链头节点
void dfs2(int p) {
    for (auto q: edges[p]) {
        if (!top[q]) { // 如果 q 节点的链顶尚未设置
            if (q == hson[p])
                top[q] = top[p]; // 重子节点继承当前节点的链头
            else
                top[q] = q; // 其他子节点设置自身为链头
            dfs2(q); // 递归访问 q 节点
        }
    }
}

// LCA: 计算两个节点的最近公共祖先
int lca(int a, int b) {
    while (top[a] != top[b]) { // 当 a 和 b 不在同一条链上
        if (dep[top[a]] > dep[top[b]]) // 比较链顶的深度
            a = fa[top[a]]; // a 跳到上一个链的顶端的父节点
        else
            b = fa[top[b]]; // b 跳到上一个链的顶端的父节点
    }
    return (dep[a] > dep[b] ? b : a); // 返回深度较小的节点
}

int main() {
    int n, m, s;
    cin >> n >> m >> s;
    for (int i = 1; i < n; ++i) {
        int x, y;
```

```

    cin >> x >> y;
    edges[x].push_back(y);
    edges[y].push_back(x);
}

dfs1(s); // 以 s 为根进行第一次 DFS
top[s] = s; // 根节点的链顶是自身
dfs2(s); // 以 s 为根进行第二次 DFS

while (m--) {
    int a, b;
    cin >> a >> b;
    cout << lca(a, b) << endl;
}

return 0;
}

```

在程序中，我们按照常规流程，通过两次 DFS 操作求出重子节点并获取到每条链的链头节点，然后再进行 LCA 查询。

9.2.2 结合数据结构

在进行了树链剖分后，我们便可以配合线段树等数据结构维护树上的信息，这需要我们改一下第二次dfs的代码，我们用dfs数组记录每个点的dfs序，用rdfs数组记录每棵子树的最大dfs序：

```

// 需要先把根节点的 top 初始化为自身
int cnt;

void dfs2(int p) {
    dfsn[p] = ++cnt;
    if (hson[p] != 0) {

```

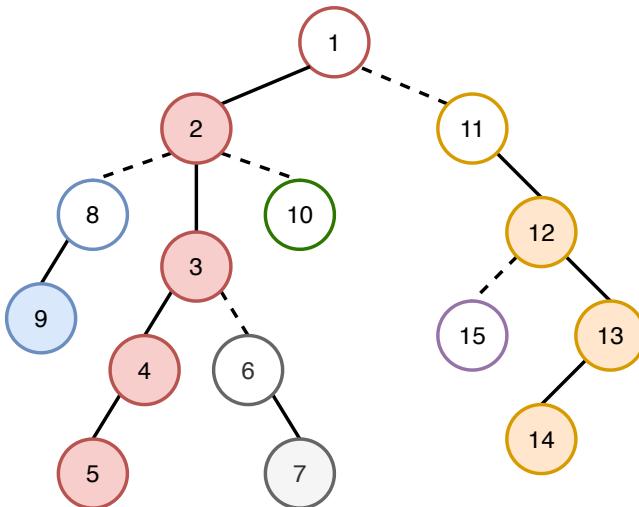
```

    top[hson[p]] = top[p];
    dfs2(hson[p]);
}

for (auto q: edges[p])
    if (!top[q]) {
        top[q] = q;
        dfs2(q);
    }
    rdfs[p] = cnt;
}

```

注意到，每棵子树的dfs序都是连续的，且根节点dfs序最小；而且，如果我们优先遍历重子节点，那么同一条链上的节点的dfs序也是连续的，且链头节点dfs序最小。



所以就可以用线段树等数据结构维护区间信息（以点权的和为例），例如路径修改：

```

void update_path(int x, int y, int z) {
    while (top[x] != top[y]) {
        if (dep[top[x]] > dep[top[y]]) {
            update(dfsn[top[x]], dfsn[x], z);

```

```

    x = fa[top[x]];
} else {
    update(dfsn[top[y]], dfsn[y], z);
    y = fa[top[y]];
}
}

if (dep[x] > dep[y])
    update(dfsn[y], dfsn[x], z);
else
    update(dfsn[x], dfsn[y], z);
}

```

路径查询：

```

int query_path(int x, int y) {
    int ans = 0;
    while (top[x] != top[y]) {
        if (dep[top[x]] > dep[top[y]]) {
            ans += query(dfsn[top[x]], dfsn[x]);
            x = fa[top[x]];
        } else {
            ans += query(dfsn[top[y]], dfsn[y]);
            y = fa[top[y]];
        }
    }
    if (dep[x] > dep[y])
        ans += query(dfsn[y], dfsn[x]);
    else
        ans += query(dfsn[x], dfsn[y]);
    return ans;
}

```

子树修改：

```
void update_subtree(int x, int z) {
```

```
update(dfsn[x], rdfs[x], z);
}
```

子树查询：

```
int query_subtree(int x) {
    return query(dfsn[x], rdfs[x]);
}
```

这里需要指出：建线段树的时候不是按节点编号建，而是按dfs序建，类似这样：

```
for (int i = 1; i <= n; ++i)
    B[i] = read();
// ...
for (int i = 1; i <= n; ++i)
    A[dfsn[i]] = B[i];
build();
```

9.3 树的重心

计算以无根树每个点为根节点时的最大子树大小，这个值最小的点称为**无根树的重心**。在这里我们将首先证明树的重心的一些数学性质，并给出寻找重心的方法。

在接下来的部分，我们将用 `mss(maximum subtree size)` 表示最大子树大小。用 $\text{size}_u(v)$ 表示以 u 为根节点时包含 v 的子树的大小，用 $u \leftrightarrow v$ 和 $u \longleftrightarrow v$ 这样的记法表示边和简单路径。此外，我们设整棵树大小为 n 。

我们首先证明一些简单的引理：

引理 9.3.1. 设 u 、 v 相邻，则 $\text{size}_u(v) + \text{size}_v(u) = n$ 。

因为树上任意节点 w 要么在以 u 为根 v 所在的子树上，此时 $w = v$ 或

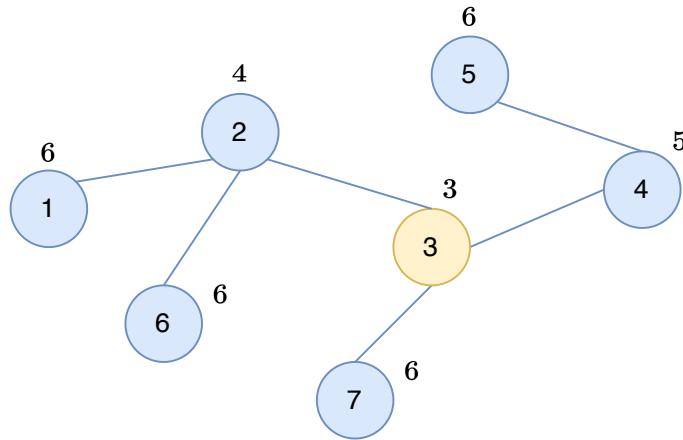
存在 $w \longleftrightarrow v \leftrightarrow u$; 要么在以 v 为根 u 所在的子树上, 此时 $w = u$ 或存在 $w \longleftrightarrow u \leftrightarrow v$ 。

引理 9.3.2. 设存在 $u \leftrightarrow v \leftrightarrow w$, 则 $\text{size}_u(v) > \text{size}_v(w)$ 。

因为 $\text{size}_v(u) + \text{size}_v(w) < n$, 故 $\text{size}_v(w) < n - \text{size}_v(u) = \text{size}_u(v)$ 。实际上, 设 $u \longleftrightarrow v \longleftrightarrow w$, 根据不等式的传递性, $\text{size}_u(v) > \text{size}_v(w)$ 也成立。

无根树的重心主要有以下性质:

1. 某个点是树的**重心**等价于它最大子树大小**不大于**整棵树大小的一半。

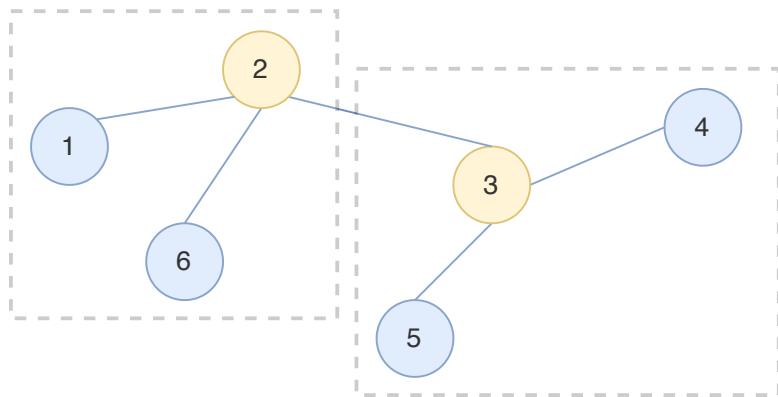


充分性: 假设 u 是树的重心, v 与它相邻且 $\text{size}_u(v) > \frac{n}{2}$ 。那么以 v 为根节点时, $\text{size}_v(u) = n - \text{size}_u(v) < \frac{n}{2} < \text{size}_u(v) = \text{mss}(u)$, 同时对于 $w \neq u$ 又有 $\text{size}_v(w) < \text{size}_u(w) = \text{mss}(u)$ 。所以, $\text{mss}(v) < \text{mss}(u)$, 与 u 是重心矛盾。因此原命题成立。

必要性: 假设 u 是树的重心, 且任意相邻的节点 v 都满足 $\text{size}_u(v) \leq \frac{n}{2}$, 那么 $\text{mss}(u) \leq \frac{n}{2}$, 故 $\text{size}_v(u) = n - \text{size}_u(v) \geq \frac{n}{2} \geq \text{mss}(u)$, 则 $\text{mss}(v) \geq \text{mss}(u)$ 。对于任意不与 u 相邻的节点 w , 都存在简单路径 $w \longleftrightarrow v \leftrightarrow u$, 那么 $\text{size}_w(v) > \text{size}_v(u) \geq \text{mss}(u)$, 因此 $\text{mss}(w) >$

$\text{mss}(u)$ 。综上， u 是所有节点中 mss 最小的（或之一），也就是树的重心。

2. 树至多有两个重心。



如果树有两个重心，那么它们相邻，此时树一定有偶数个节点，且可以被划分为两个大小相等的分支，每个分支各自包含一个重心。

假设 u 和 v 是树的两个重心，且它们之间的路径（不包含端点）有 k 个节点。我们知道 $\text{mss}(u) \geq \text{mss}(v)$ 且 $\text{mss}(u) \leq \text{mss}(v)$ ，所以 $\text{mss}(u) = \text{mss}(v)$ 。

那么 u 的最大子树必然包含 $u \longleftrightarrow v$ ，否则设 w 是 u 最大子树上任意一点，有 $\text{mss}(u) = \text{size}_u(w) < \text{size}_v(u) \leq \text{mss}(v)$ ，与前面的结论矛盾。同理， v 的最大子树也一定包含 $u \longleftrightarrow v$ 。

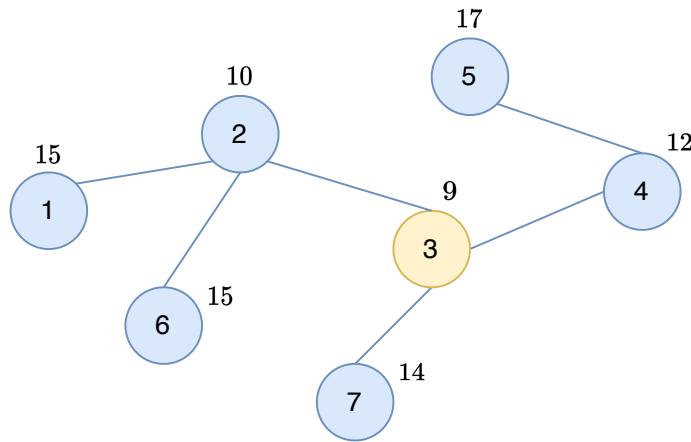
设 $u \longleftrightarrow v$ 除端点外包含 k 个顶点，那么 $\text{mss}(u) = \text{size}_u(v) = k + \text{mss}(v)$ ，可知 $k = 0$ 。因此，重心必然相邻。

假设树至少存在三个重心，那么它们两两相邻，但这会形成环，是不可能的。所以树最多有两个重心。

由上述条件，我们知道 $\text{size}_u(v) = \text{size}_v(u)$ 。而我们又有 $\text{size}_u(v) + \text{size}_v(u) = n$ ，于是 $\text{size}_u(v) = \text{size}_v(u) = \frac{n}{2}$ 。所以，在有两个重心时，

树的大小一定是偶数，且可以被划分为两个大小相等的分支，每个分支各自包含一个重心。

3. 树中所有点到某个点的距离和中，到重心的距离和是最小的；如果有两个重心，那么到它们的距离和一样。反过来，距离和最小的点一定是重心。



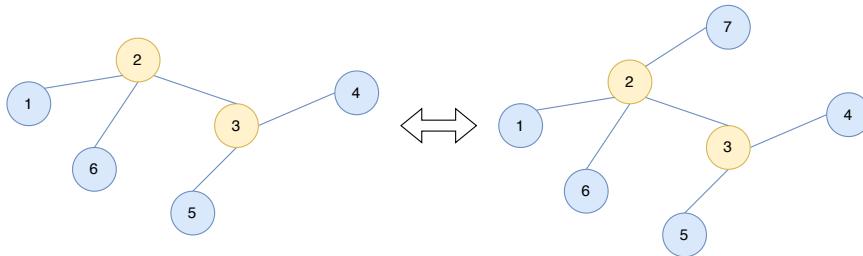
设所有点到某个点 p 的距离和为 $\text{dis}(p)$ 。那么对于树上任意一个点 u ，如果存在相邻的点 v 使得 $\text{size}_u(v) > \frac{n}{2}$ ，则 $\text{dis}(v) < \text{dis}(u)$ ，因为这种移动使得 u 所在的子树上的点对 dis 的贡献减 1，而其他点对 dis 的贡献加 1，所以 $\text{dis}(v)$ 比 $\text{dis}(u)$ 少了 $\text{size}_u(v) - (n - \text{size}_u(v)) = 2\text{size}_u(v) - n > 0$ 。

除了重心以外的点都至少存在一个相邻点使得 dis 更小，根据偏序关系的性质，重心应当比其他点的 dis 都小。

假设有两个重心 u 和 v 。当我们从 u 移动到 v 时， u 所在分支对 dis 的贡献加了 1， v 所在分支对 dis 的贡献减了 1。由于 u 所在分支的大小为 $\text{size}_v(u)$ ， v 所在分支的大小为 $\text{size}_u(v)$ ，而我们已经知道 $\text{size}_u(v) = \text{size}_v(u)$ ，所以这两个贡献变化刚好互相抵消了。这就导出了 $\text{dis}(u) = \text{dis}(v)$ 。

上面已经说明了如果一个点不是重心，它相邻的某个点的 dis 一定比它小，所以不是距离和最小的点。因此，距离和最小的点就是重心。

4. 往树上增加或减少一个叶子，如果原节点数是奇数，那么重心可能增加一个，原重心仍是重心；如果原节点数是偶数，重心可能减少一个，另一个重心仍是重心。



重心的每棵子树的节点数都是小于等于 $\frac{n}{2}$ 的，很显然，节点数等于 $\frac{n}{2}$ 的子树最多只有一棵。这时 n 显然是偶数，而且这棵子树的根节点也必然是重心，它也拥有一棵节点数等于 $\frac{n}{2}$ 的子树。

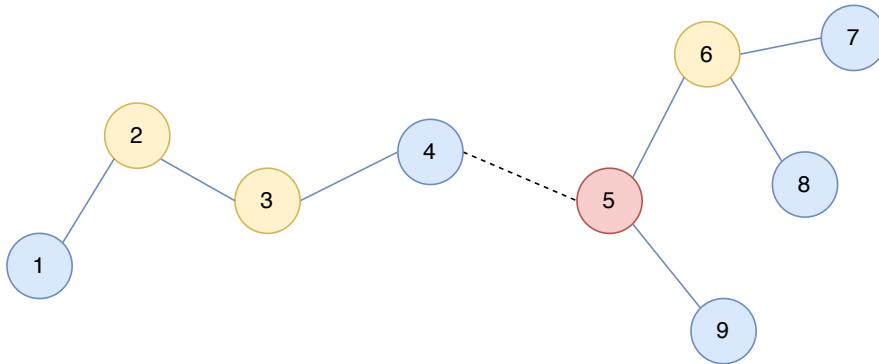
增加一个叶子，节点数变成 $n+1$ 。如果此时原来的某个重心不再是重心，那么添加叶子的那棵子树原来的节点数一定大于 $\frac{n+1}{2} - 1 = \frac{n-1}{2}$ ，只有节点数等于 $\frac{n}{2}$ 的子树才符合要求，这时 n 只能是偶数。往这棵子树上添加一个节点后，这棵子树的根节点（即“另一个重心”）仍然是重心，它拥有一个节点数为 $\frac{n}{2}$ 的子树和另外一系列节点数总和为 $\frac{n}{2}$ 的子树，均小于 $\frac{n+1}{2}$ 。

如果增加叶子后出现了新的重心，那么添加叶子的那颗子树原来的节点数大于 $\frac{n}{2}$ ，却又小于等于 $\frac{n+1}{2} - 1 = \frac{n-1}{2}$ ，那就只能是 $\frac{n-1}{2}$ ，这时 n 一定是奇数。这时原来的重心拥有一棵大小为 $\frac{n+1}{2}$ 的子树和一系列节点数总和为 $\frac{n-1}{2}$ 的子树，仍然是重心。

同理可证，删除的情形。

5. 把两棵树通过一条边相连得到一棵新的树，则新的重心在较大的一棵

树一侧的连接点与原重心之间的简单路径上。如果两棵树大小一样，则重心就是两个连接点。



两棵树大小一样的情形显然，我们重点证明两棵树大小不一样的情况。

设两棵树的节点数分别为 n 和 m ，不妨设 $n > m$ ，我们先建出较大的树，然后一个点一个点地把较小的树添加进来。按照性质 4，这个过程中可能会出现重心的增加和减少，我们把一组增加和减少称为移动，则重心一定往连接点方向移动。

现在只需要证明重心不可能逾越自己一侧的连接点。事实上，当重心达到自己一侧的连接点时，设这时已经添加了 k 个节点，那么想要通过再添加一个节点让另一侧的连接点也成为重心，需要满足 $k = \frac{k+n-1}{2}$ ，即 $k = n - 1$ 。然而 $k < m \leq n - 1$ （注意 k 不能等于 m ，因为还要再添加一个节点），所以这是不可能的。

为了找出一棵树的重心，我们利用性质 1，一趟dfs即可找到：

```

// sz: 存储每个节点子树大小, mss: 存储每个节点最大子树大小
int n, sz[MAXN], mss[MAXN];
// 重心列表
vector<int> ctr;

// 寻找树的重心
void dfs(int p, int fa = 0) {
  ...
}
  
```

```
// 当前节点子树大小为 1
sz[p] = 1;
// 当前节点最大子树大小为 0
mss[p] = 0;

// 遍历当前节点的边
for (auto [to, w]: edges[p]) {
    // 不是当前节点的父节点
    if (to != fa) {
        dfs(to, p);
        // 更新当前节点的最大子树大小
        mss[p] = max(mss[p], sz[to]);
        // 更新当前节点的子树大小，将子节点的子树大小累加到当前
        // 节点
        sz[p] += sz[to];
    }
}

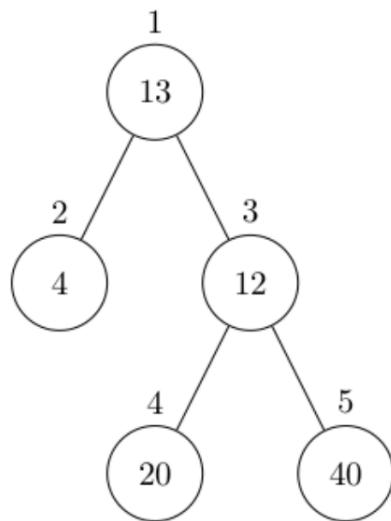
// 更新当前节点的最大子树大小
mss[p] = max(mss[p], n - sz[p]);

// 如果当前节点的最大子树大小小于等于总节点数的一半，将其加
// 入到重心列表中
if (mss[p] <= n / 2) {
    ctr.push_back(p);
}
}
```

我们来看一道例题：

例 9.3.3 (luogu-P1364 医院设置). 设某地区的排布类似一棵二叉树，其中，圈中的数字表示结点中居民的人口。圈边上数字表示结点编号，现在要求在某个结点上建立一个医院，使所有居民所走的路程之和为最小，同时约

定，相邻接点之间的距离为 1。如下图所示：



图中，若医院建在 1 处，则距离和 $= 4 + 12 + 2 \times 20 + 2 \times 40 = 136$ ；
 若医院建在 3 处，则距离和 $= 4 \times 2 + 13 + 20 + 40 = 81$ 。

输入格式：

第一行一个整数 n ，表示树的结点数。

接下来的 n 行每行描述了一个结点的状况，包含三个整数 w, u, v ，其中 w 为居民人口数， u 为左链接（为 0 表示无链接）， v 为右链接（为 0 表示无链接）。

```

5
13 2 3
4 0 0
12 4 5
20 0 0
40 0 0
  
```

输出格式：

一个整数，表示最小距离和。

81

对于类似9.3.3的问题，即题干中包含有医院、快递点、信号塔等到其他所有点距离最小的问题时，当图是无环图时，我们都可以考虑使用重心来解决。经过之前的分析，这里直接给出程序：

```
#include <bits/stdc++.h>

using namespace std;

const int MAXN = 1001;

int n, cnt, tot, ans, sum;
// 前向星
int h[MAXN], ne[MAXN], to[MAXN], val[MAXN];
// 重心
int sz[MAXN], dis[MAXN], fa[MAXN];

void add(int x, int y) {
    to[cnt] = y;
    ne[cnt] = h[x];
    h[x] = cnt++;
}

void dfs(int u) {
    int i, v;
    sz[u] += val[u];
    for (i = h[u]; i != -1; i = ne[i]) {
        v = to[i];
        if (v != fa[u]) {
            fa[v] = u;
            dfs(v);
            sz[u] += sz[v]; // 累加子树大小
        }
    }
}
```

```
    }

}

if (2 * sz[u] >= tot && !ans) {
    ans = u; // 判断是否是重心
}

}

void dfs1(int u) {
    int i, v;
    sum += (dis[u] - 1) * val[u];
    for (i = h[u]; i != -1; i = ne[i]) {
        v = to[i];
        if (!dis[v]) {
            dis[v] = dis[u] + 1;
            dfs1(v);
        }
    }
}

int main() {
    int i, x, y;
    scanf("%d", &n);
    memset(h, -1, sizeof(h));
    for (i = 1; i <= n; i++) {
        scanf("%d %d %d", &val[i], &x, &y);
        if (x) add(i, x), add(x, i);
        if (y) add(i, y), add(y, i);
        tot += val[i];
    }
    dfs(1);
    dis[ans] = 1;
    dfs1(ans);
    printf("%d", sum);
```

```

    return 0;
}

```

9.4 点分治

点分治或重心剖分 (Centroid Decomposition) 是树分治的一种，主要处理一些树上路径问题。

假如我们要遍历树上所有的路径，最朴素的想法是枚举端点，对每个点来一趟dfs，复杂度 $O(n^2)$ 。为了效率更高地解决问题，我们引入**分治思想**。对于每个点，我们分别考虑包含这个点的路径和不包含这个点的路径。对于前者，我们做一趟dfs；对于后者，我们删除该点后，对所有子树递归地处理即可。

但是如果直接做，复杂度是不稳定的，例如当树是一条链时，有可能会退化成 $O(n^2)$ 。然而，如果每次都**选择子树的重心作为子树的根**，那么复杂度就可以保证为 $O(n \log n)$ 。因为按照这种选法，每个递归问题中子树的大小都不超过整棵树的一半，所以每次递归问题规模可以下降至一半或更低。

同样的，我们来看一道例题：

例 9.4.1 (CF161D Distance in Tree). 一棵树是一种不包含任何环的连通图。

树中两个顶点之间的距离是这两个顶点之间最短路径的长度（以边为单位）。给定一棵有 n 个顶点的树和一个正整数 k 。

请找出树中有多少个顶点对的距离恰好为 k 。注意，顶点对 (v, u) 和 (u, v) 被认为是相同的顶点对。

输入格式：

第一行包含两个整数 n 和 k $1 \leq n \leq 50000, 1 \leq k \leq 500$ ，即顶点的数量和顶点之间所需的距离。

接下来的 $n-1$ 行描述边，格式为 " a_i b_i "(不带引号)($1 \leq ai, bi \leq n, ai \neq bi$)，其中 a_i 和 b_i 是由第 i 条边连接的顶点。

所有给定的边都是不同的。

输出格式：

输出一个整数，即树中顶点对之间距离恰好为 k 的顶点对的数量。

题目9.4.1的目的是求无根树中长度为 k 的路径数目。

对于当前处理的节点的每一棵子树，我们用一个数组temp存该子树中得到的路径长，哈希表lens存其他子树已得到过的路径长。每次搜索完一个子树，就把temp里的信息放到lens里面去。

```
#include <bits/stdc++.h>
#define ll long long

using namespace std;
const int MAXN = 50005, MAXK = 505;
vector<int> edges[MAXN]; // 存储树的邻接表表示

namespace PointPartition { // 使用命名空间来包含重心分解相关函数和变量
    /**
     * @param ctr 当前子树的重心
     * @param n 当前子树的节点个数
     * @param sz 存储每个节点子树大小的数组
     */
    int ctr, n, sz[MAXN];
    bool del[MAXN]; // 标记点是否已被删除（在重心分解过程中）

    /**
     * 寻找一个子树的重心
     * @param p 表示当前节点
     * @param fa 表示当前节点的父节点（默认为 0，即根节点）
     */
    void dfs(int p, int fa = 0) {
```

```
sz[p] = 1; // 初始化当前节点子树大小为 1 (包含当前节点)
int mss = 0; // 初始化当前节点最大子树大小为 0

// 遍历当前节点的边
for (auto to: edges[p]) {
    // 如果遍历到的节点没有被删除且不是当前节点的父节点，则
    // 继续递归搜索
    if (!del[to] && to != fa) {
        dfs(to, p); // 递归搜索子节点
        if (ctr != -1) return; // 找到重心后及时退出
        mss = max(mss, sz[to]); // 更新当前节点的最大子树大小
        sz[p] += sz[to]; // 更新当前节点的子树大小，将子节点的
        // 子树大小累加到当前节点
    }
}

// 更新当前节点的最大子树大小，考虑将整棵树分为两部分后，
// 当前节点的最大子树大小
mss = max(mss, n - sz[p]);

// 如果当前节点的最大子树大小小于等于总节点数的一半，将其
// 设为重心
if (mss <= n / 2) {
    ctr = p;
    sz[fa] = n - sz[p]; // 将 sz[fa] 修正为以 ctr 为根节点时
    // 的值
}
}

/***
 * k 需要的距离
 * temp 临时存储路径长度的数组
 * cntt 临时数组的长度
*/
```

```
* cnt 结果计数器
* lens 存储不同长度路径的计数数组
*/
int k, temp[MAXN], cntt, cnt, lens[MAXK];

/**
* 遍历一个节点的某个子树，计算结果
* @param p 表示当前节点
* @param fa 表示当前节点的父节点
* @param len 表示当前路径长度
*/
void dfs2(int p, int fa, int len) {
    if (len > k) return; // 剪枝
    cnt += lens[k - len] + (len == k); // 更新最终答案
    temp[cntt++] = len; // 存入新的路径长

    // 历遍当前节点的边
    for (auto to: edges[p]) {
        // 如果遍历到的节点没有被删除且不是当前节点的父节点，则
        // 继续递归搜索
        if (!del[to] && to != fa)
            dfs2(to, p, len + 1); // 递归搜索子节点，并将路径长度
            // 加一
    }
}

void run(int p) {
    // 先统计经过p点的答案
    for (auto to: edges[p])
        if (!del[to]) {
            dfs2(to, p, 1); // 遍历子节点
            for (int i = 0; i < cntt; ++i) // 将临时数组里的数据腾
                // 出来
        }
}
```

```
    lens[temp[i]]++;
    cntt = 0; // 清空临时数组
}

fill(lens, lens + MAXK, 0); // 重置 lens 数组
// 再统计不经过 p 点的答案
del[p] = 1; // 标记当前节点已被删除
for (auto to: edges[p])
    if (!del[to]) {
        n = sz[to];
        ctr = -1;
        dfs(to); // 找到重心
        run(ctr); // 递归处理
    }
}

int count(int n0, int k0) {
    n = n0, k = k0, ctr = -1;
    dfs(1); // 寻找重心
    run(ctr); // 运行重心分解
    return cnt; // 返回结果
}

} // namespace PointPartition

int main() {
    int n, k, u, v;
    cin >> n >> k;
    for (int i = 0; i < n - 1; ++i) {
        cin >> u >> v;
        edges[u].push_back(v);
        edges[v].push_back(u);
    }
    cout << PointPartition::count(n, k) << endl; // 输出计算结果
}
```

```

    return 0;
}

```

重新确定重心每棵子树的大小的原理是：在以`to`为根节点`dfs`时，相对于以`ctr`为根节点`dfs`，重心周围的点只有一个会得到错误的`sz`，而且它会变得比重心还大。它正确的大小是`n - sz[ctr]`。如果不需要卡常的话，直接以`ctr`为根节点重新`dfs`一次也可以。

例 9.4.2 (CF1101D GCD Counting). 您将得到一个由 n 个顶点组成的树。每个顶点上都写有一个数字，顶点 i 上的数字等于 a_i 。

我们用函数 $g(x, y)$ 表示从顶点 x 到顶点 y 的简单路径上顶点所具有的数字的最大公约数（包括这两个顶点）。同时，我们用 $dist(x, y)$ 表示顶点 x 和 y 之间的简单路径上的顶点数量，包括端点。对于每个顶点 x ， $dist(x, x) = 1$ 。

您的任务是计算在 $g(x, y) > 1$ 的顶点对中， $dist(x, y)$ 的最大值。

输入格式：

第一行包含一个整数 n ，代表顶点的数量 ($1 \leq n \leq 2 \times 10^5$)。

第二行包含 n 个整数 a_1, a_2, \dots, a_n ($1 \leq a_i \leq 2 \times 10^5$)，代表写在顶点上的数字。

然后有 $n - 1$ 条线，每条线包含两个整数 x 和 y ($1 \leq x, y \leq n, x \neq y$)，表示连接顶点 x 和顶点 y 的边。

可以保证这些边形成一棵树。

输出格式：

如果没有一对顶点 x, y 使 $g(x, y) > 1$ ，打印 0。否则打印此类对中 $dist(x, y)$ 的最大值。

例题9.4.2的题意是求最长的 gcd 大于 1 的路径。

在处理经过 p 的路径时，不能简单地把每条子树最长路径相加。因为可能存在这样的结构：

```

6
/
4   9

```

从 6 出发，向左或向右都可以得到长度为 1 的路径，但是整个路径却并不满足 gcd 大于 1。

一种可行的做法是，枚举 p 的质因子，对于每个质因子找最长路径。注意质因子可以 $O(n \log n)$ 预处理，而一个 2×10^5 以内的整数最多只有 6 个质因子，因为 $2 \times 3 \times 5 \times 7 \times 11 \times 13 \times 17 = 510510 > 2 \times 10^5$ 。

```

#include <bits/stdc++.h>
#define ll long long

using namespace std;

const int MAXN = 200005;
vector<int> edges[MAXN], factors[MAXN];
int val[MAXN];

namespace PointPartition {
    int ctr, n, sz[MAXN];
    bool del[MAXN]; // 点是否已被删除
    void dfs(int p, int fa = 0) { // 找一个重心
        sz[p] = 1;
        int mss = 0;
        for (auto to: edges[p]) {
            if (!del[to] && to != fa) {
                dfs(to, p);
                if (ctr != -1) return;
                mss = max(mss, sz[to]);
                sz[p] += sz[to];
            }
        }
    }
}

```

```
mss = max(mss, n - sz[p]);
if (mss <= n / 2) {
    ctr = p;
    sz[fa] = n - sz[p];
}
}

int temp, maxlen, olen, fac;

void dfs2(int p, int fa, int len) { // 遍历一个节点的某个子
    tree
    maxlen = max(maxlen, len + olen + 1); // 更新最终答案
    temp = max(len, temp); // 存入新的路径长
    for (auto to: edges[p]) {
        if (!del[to] && to != fa && val[to] % fac == 0) {
            dfs2(to, p, len + 1);
        }
    }
}

void run(int p) {
    // 先统计经过p点的答案
    maxlen = max(maxlen, int(val[p] != 1));
    for (auto f: factors[val[p]]) {
        fac = f;
        for (auto to: edges[p]) {
            if (!del[to] && val[to] % f == 0) {
                dfs2(to, p, 1);
                olen = max(olen, temp);
                temp = 0;
            }
        }
    }
    olen = 0;
```

```
}

// 再统计不经过p点的答案
del[p] = 1;
for (auto to: edges[p]) {
    if (!del[to]) {
        n = sz[to];
        ctr = -1;
        dfs(to); // 找到重心
        run(ctr); // 递归处理
    }
}
}

int max(int n0) {
    n = n0, ctr = -1;
    // fill(del + 1, del + n + 1, 0);
    dfs(1);
    run(ctr);
    return maxlen;
}

} // namespace PointPartition
bool isnp[MAXN];
vector<int> primes; // 质数表
void init(int n) {
    isnp[0] = isnp[1] = 1;
    for (int i = 2; i <= n; i++) {
        if (!isnp[i]) {
            primes.push_back(i);
        }
        for (int p: primes) {
            if (p * i > n) { break; }
            isnp[p * i] = 1;
            if (i % p == 0) { break; }
        }
    }
}
```

```
    }

}

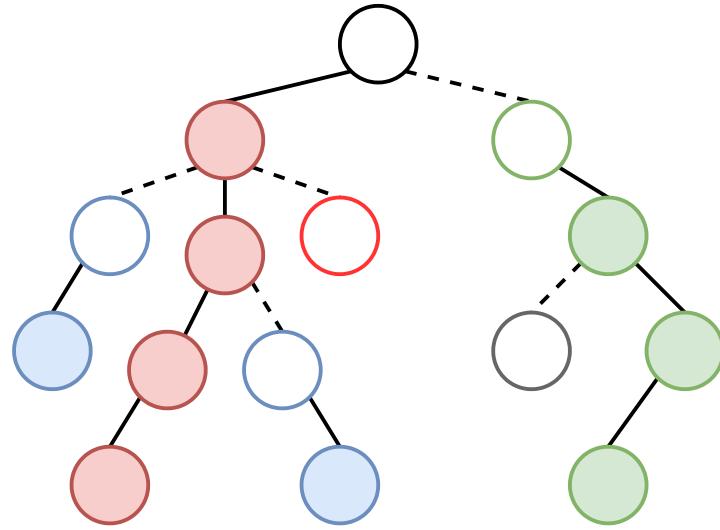
}

int main() {
    init(MAXN - 1);
    for (auto p: primes) {
        for (int i = p; i < MAXN; i += p) {
            factors[i].push_back(p);
        }
    }
    int n, u, v;
    cin >> n;
    for (int i = 1; i <= n; ++i)
        cin >> val[i];
    for (int i = 0; i < n - 1; ++i) {
        cin >> u >> v;
        edges[u].push_back(v);
        edges[v].push_back(u);
    }
    cout << PointPartition::max(n) << endl;
    return 0;
}
```

9.5 长链剖分

在经过对重链剖分的探讨后，**长链剖分**就非常容易理解了。**重链剖分**定义子树大小最大的为重子节点，而**长链剖分**定义子树最深深度最深的为重子节点。

我们来看一个图，这个图在重链剖分曾经提到过，但巧合的是它也符合长链剖分的性质，代码思路也比较清晰：



```

// 深度，部分链长，父节点，重儿子，链头，dfs序，子树最大dfs
// 序，dfs序对应节点

int dep[N], len[N], fa[N], hson[N], top[N], lenc[N], dfsn[N],
    mdfsn[N], node[N], cnt;

void dfs1(int p, int d = 1) {
    len[p] = 1, dep[p] = d;
    for (auto q: edges[p])
        if (!dep[q]) {
            dfs1(q, d + 1);
            fa[q] = p;
            if (len[q] + 1 > len[p])
                hson[p] = q, len[p] = len[q] + 1;
        }
    }

void dfs2(int p, int tp) {
    dfsn[p] = ++cnt;
    top[p] = tp;
    node[cnt] = p;
    if (hson[p]) dfs2(hson[p], tp);
}

```

```

for (auto q: edges[p])
    if (!top[q])
        dfs2(q, q);
    mdfsn[p] = cnt;
}

void cut(int r = 1) {
    dfs1(r);
    dfs2(r, r);
}

```

这里与需要提一下：并非所有题目都需要用到上述的所有参数。

长链剖分有两个重要的性质：

1. 任意节点 p 的 k 级祖先 q 所在的链的长度一定大于 k 。

这是因为， $q \rightarrow p$ 的长度为 $k+1$ ，如果 p 和 q 在同一条链上，那这条链的长度自然大于 k ；如果 p 和 q 不在同一条链上，设 q 所在链的链头和链尾分别是 top 和 $tail$ ，那 $top \rightarrow q \rightarrow tail$ 必须比 $top \rightarrow q \rightarrow p$ 长，所以链的长度也大于 k 。

2. 任意节点 p 到根节点最多经过 \sqrt{n} 级别的轻边。

因为每经过一条轻边到一条新的链，新的链的长度一定严格大于上一条链（可以利用性质 1 简单证明），由于树上所有链的长度和为 n ，所以经过的链的长度形成一个和小于等于 n 的递增序列，当每次长度加 1 时序列最长，长度为 \sqrt{n} 级别。

相比于重链剖分的 $\log n$ ，这里的 \sqrt{n} 显得不够看，所以长链剖分并不常用于解决重链剖分擅长的那类问题。我们常常利用的是其他性质。

例 9.5.1 (P5903 【模板】树上 k 级祖先). 给定一棵 n 个点的有根树。

有 q 次询问，第 i 次询问给定 x_i, k_i ，要求点 x_i 的 k_i 级祖先，答案为 ans_i 。特别地， $ans_0 = 0$ 。

本题中的询问将在程序内生成。给定一个随机种子 s 和一个随机函数 $\text{get}(x)$ 。你需要按顺序依次生成询问。

设 d_i 为点 i 的深度，其中根的深度为 1。

对于第 i 次询问：

- $x_i = ((\text{get}(s) \text{ xor } ans_{i-1}) \bmod n) + 1;$
- $k_i = (\text{get}(s) \text{ xor } ans_{i-1}) \bmod d_{x_i}.$

输入格式：

第一行三个整数 n, q, s 。

第二行 n 个整数 $f_{1\dots n}$ ，其中 f_i 表示 i 的父亲。特别地，若 $f_i = 0$ ，则 i 为根。

输出格式

一行一个整数，表示 $\text{xor}_{i=1}^q i \times ans_i$ 。

样例输入：

```
6 3 7
5 5 2 2 0 3
```

样例输出：

```
1
```

样例说明：

$x_1 = 4, k_1 = 1, ans_1 = 2;$

$x_2 = 6, k_2 = 3, ans_2 = 5;$

$x_3 = 3, k_3 = 0, ans_3 = 3;$

故输出 1。

此外，随机函数 `get()` 如下：

```
#define ui unsigned int
ui s;
```

```

inline ui get(ui x) {
    x ^= x << 13;
    x ^= x >> 17;
    x ^= x << 5;
    return s = x;
}

```

对于这个问题，我们可以用树上倍增 $O(n \log n) - O(\log n)$ 地解决，方法显然；也可以用重链剖分 $O(n) - O(\log n)$ 地解决，方法是在重链上跳，直到跳到 k 级祖先所在的链上，然后利用dfs序定位 k 级祖先。当然同样的套路利用长链剖分可以 $O(n) - O(\sqrt{n})$ 解决，但那太没意义了，这里要提供的是 $O(n \log n) - O(1)$ 的方法。

首先，我们同样进行一个树上倍增的预处理， $O(n \log n)$ 地预处理出每个节点的 2^k 级祖先。然后我们再对每一个链头节点 p 预处理出它的 $1, 2, \dots, len[p] - 1$ ($len[p]$ 为链长) 级祖先节点和链上的 $1, 2, \dots, len[p] - 1$ 级子孙节点。显然因为所有链的链长加起来是 n ，所以这部分是 $O(n)$ 的。

对于每个询问，设 $2^i \leq k \leq 2^{i+1}$ ($k = 0$ 时特判)，考虑 p 的 2^i 级祖先 q ，它所在的链的链长一定大于 2^i 。由于我们要找的是 q 的 $k - 2^i$ 级祖先，而 $k - 2^i < 2^{i+1} - 2^i = 2^i < len[top[q]]$ ，所以它一定已经被预处理过了（要么在链上，要么是链头的小于链长级祖先），算一下差值，查询就好。

```

vector<int> anc[N], des[N];

void init(int r, int n) {
    cut(r);
    for (int i = 1; i <= __lg(n); ++i)
        for (int j = 1; j <= n; ++j)
            fa[i][j] = fa[i - 1][fa[i - 1][j]];
    for (int i = 1; i <= n; ++i) {
        if (top[i] == i) {

```

```

        for (int j = 0, p = i; j < len[i]; ++j, p = fa[0][p])
            anc[i].push_back(p);
        for (int j = 0; j < len[i]; ++j)
            des[i].push_back(node[dfsn[i] + j]);
    }
}

}

int query(int p, int k) {
    if (k == 0) return p; // 特判
    int i = __lg(k), q = fa[i][p];
    int tp = top[q];
    int d = k - (1 << i) + dep[tp] - dep[q];
    if (d > 0)
        return anc[tp][d];
    else
        return des[tp][-d];
}

```

例 9.5.2 (CF1009F Dominant Indices). 你将获得一棵由 n 个节点组成的根节点为 1 的无向树。

我们将节点 x 的深度数组表示为一个无限序列 $[d_{x,0}, d_{x,1}, d_{x,2}, \dots]$, 其中 $d_{x,i}$ 是满足以下两个条件的节点 y 的数量:

1. x 是 y 的祖先;
2. 从 x 到 y 的简单路径恰好经过 i 条边。

节点 x 的深度数组的主要指数 (或简称节点 x 的主要指数) 是指满足以下条件的索引 j :

1. 对于每个 $k < j$, $d_{x,k} < d_{x,j}$;
2. 对于每个 $k > j$, $d_{x,k} \leq d_{x,j}$ 。

对于树中的每个节点，计算其主要指数。

对于例题9.5.2，我们设 $dp[p][x]$ 表示节点 p 的子树中到 p 距离为 x 的节点数，那么显然：

$$\begin{cases} \sum_{p \rightarrow q} dp[q][x-1] & x > 0 \\ 1 & x = 0 \end{cases}$$

其中 k 可以在转移过程中计算。

那么为了节约时间，我们直接复用重子树的dp数组，在其前面添加一个1，再暴力转移轻子树。由于每个轻子树的dp数组只会被用于暴力转移一次，而每个轻子树对应一条链，轻子树的dp数组长度正好为链长，所有链长和为 n ，所以总复杂度是 $O(n)$ 。

当然，我们可以使用倒序的vector来实现，但有一个更优雅的实现，是使用指针。假设有一个共享的buf，我们用dp[p]指向下标从dfs[n][p]开始、长度为len[p]的一片区域。这样dp[p]刚好就是在dp[hson[p]]的前面多了一位。以下是实现的部分：

```
int buf[N], *dp[N], k[N];

void dfs2(int p, int tp) {
    dfsn[p] = ++cnt;
    top[p] = tp;
    node[cnt] = p;
    dp[p] = buf + cnt; // dp[p] 指向下标从 dfsn[p] 开始、长度为 len[p]
                       // 的一片区域
    if (hson[p]) dfs2(hson[p], tp);
    for (auto q: edges[p])
        if (!top[q])
            dfs2(q, q);
    mdfsn[p] = cnt;
}
```

```
void dfs3(int p, int fa = 0) {
    if (hson[p]) {
        dfs3(hson[p], p);
        k[p] = k[hson[p]] + 1;
    }
    dp[p][0] = 1;
    if (dp[p][k[p]] <= 1) k[p] = 0;
    for (auto q: edges[p]) {
        if (q != fa && q != hson[p]) {
            dfs3(q, p);
            for (int i = 1; i <= len[q]; ++i) {
                dp[p][i] += dp[q][i - 1];
                if (dp[p][i] > dp[p][k[p]] || dp[p][i] == dp[p][k[p]]
                    && i < k[p])
                    k[p] = i;
            }
        }
    }
}
```

第十章 数论

10.1 欧几里得算法

在讨论拓展欧几里得算法之前，我们先来了解一下欧几里得算法（即辗转相除法），欧几里得算法可以用来计算两个数的最大公约数，例如我们现在要来求 1112 和 695 的最大公因数，我们可以如下操作：

$$\begin{aligned} 1112 & \quad 695 \quad 695 + 417 \times 1 = 1112 \\ 695 & \quad 417 \quad 417 + 278 \times 1 = 695 \\ 417 & \quad 278 \quad 278 + 139 \times 1 = 417 \\ 278 & \quad 139 \quad 0 + 139 \times 2 = 278 \\ & \qquad \longrightarrow 139 \quad 0 \end{aligned}$$

当余数变为 0 的时候，最后一个操作的除数 139 即是 1112 和 695 的最大公约数。

欧几里得算法的代码用递归实现起来非常简洁：

```
int gcd(int a, int b){  
    if (b != 0)  
        return gcd(b, a % b);  
    else  
        return a;  
}
```

接下来我们尝试证明欧几里得算法：

证明：不妨设 $a > b$, 则 a 可以表示成 $a = kb + r (a, b, k, r \in Z^+, \text{且 } r < b)$, 则 $r = a \bmod b$ 。

假设 d 是 a, b 的一个公约数, 记作 $d|a, d|b$, 即 a 和 b 都可以被 d 整除。因为 $r = a - kb$, 两边同时除以 d 得 $\frac{r}{d} = \frac{a}{d} - k\frac{b}{d} = m$, 由于 a, b 均是 d 的倍数可知 m 为整数, 因此 $d|r$, 即 r 可被 d 整除, 因此 d 也是 b 和 $a \bmod b$ 的公约数。

假设 d 是 $b, a \bmod b$ 的公约数, 则 $d|b, d|(a - kb) (k \in Z)$, 进而 $d|a$ 。因此 d 也是 a, b 的公约数。

因此 (a, b) 和 $(b, a \bmod b)$ 的公约数是一样的, 其最大公约数也必然相等, 得证。

10.2 拓展欧几里得算法

欧几里得算法可以为我们计算出两个数的最大公约数, 拓展欧几里得算法则可以在辗转相除的执行过程中求出不定方程 $ax + by = c$ 的一组解, 其中 c 是 $\gcd(a, b)$ ¹ 的整数倍。

定理 10.2.1 (裴蜀定理). 设 $a, b \in N$ 且 a, b 不全为 0, 则存在整数 x, y 使得 $ax + by = \gcd(a, b)$ 。

扩展欧几里得算法的证明也与欧几里得算法的证明相似, 我们尝试从裴蜀定理开始:

证明：不妨设 $a \geq b$, 当 $b = 0$ 时, $\gcd(a, b) = a$, 由于 c 是 a, b 最大公因数的倍数, 显然成立。

¹gcd 即 Greatest Common Divisor, 最大公因数, $\gcd(a, b)$ 即 a, b 的最大公因数。

另一方面，当 $b \neq 0$ 时，在 $ax + by = \gcd(a, b)$ 两边同时除 $\gcd(a, b)$ 得到 $\frac{a}{\gcd(a, b)}x + \frac{b}{\gcd(a, b)}y = 1$ ，令 $\frac{a}{\gcd(a, b)} = a_1, \frac{b}{\gcd(a, b)} = b_1$ ，则 a_1 与 b_1 互质，转证 $a_1x + b_1y = 1$ 。

回顾欧几里得算法 $\gcd(a_1, b_1) = \gcd(b_1, a_1 \bmod b_1) = \dots = 1$ ，现在我们将每一步取模运算的值记为 r_i ，那么上式变成：

$$\gcd(a_1, b_1) = \gcd(b_1, r_1) = \gcd(r_1, r_2) = \dots = \gcd(r_{n-1}, r_n) = 1$$

现在，将辗转相除的过程体现为带余除法：

$$a_1 = k_1 b_1 + r_1$$

$$b_1 = k_2 r_1 + r_2$$

...

$$r_{n-3} = k_{n-1} r_{n-2} + r_{n-1}$$

$$r_{n-2} = k_n r_{n-1} + r_n \quad (k_i \in \mathbb{Z})$$

不妨假设当辗转相除法到互质时 $r_n = 1$ ，则有 $r_{n-2} = k_n r_{n-1} + 1$ ，也就是

$$1 = r_{n-2} - k_n r_{n-1} \tag{10.1}$$

根据式子 $r_{n-3} = k_{n-1} r_{n-2} + r_{n-1}$ 有 $r_{n-1} = r_{n-3} - k_{n-1} r_{n-2}$ ，将之带入到式子 10.1 中得到

$$1 = (1 + k_n k_{n-1})r_{n-2} - k_n r_{n-3}$$

迭代这个过程可以逐步消去 r_{n-2}, \dots, r_1 ，最终得到 $a_1x + b_1y = 1$ ，其中 x, y 只与 k_i 有关，而 $k_i \in \mathbb{Z}$ ，即 $a_1x + b_1y = 1$ 有整数解，也就是 $ax + by = \gcd(a, b)$ 有整数解，得证。

现在我们假设在 $ax + by = c$ 有整数解，我们在 $ax + by = c$ 两边同时

除 $\gcd(a, b)$ 得到 $\frac{a}{\gcd(a, b)}x + \frac{b}{\gcd(a, b)}y = \frac{c}{\gcd(a, b)}$, 根据定理10.2.1, 等式左边是整数, 那么意味着等式右边 $\frac{c}{\gcd(a, b)}$ 也必是整数, 因此我们也就得到了一个推论:

推论 10.2.2. 设 $a, b \in Z$, 当且仅当 c 是 $\gcd(a, b)$ 的整数倍时, 存在 $x, y \in Z$ 使得 $ax + by = c$ 。

同样的, 我们来求 $1112x + 695y = 139$ 的一组解, 我们回顾欧几里得算法的求解流程:

$$\begin{aligned} 1112 & \quad 695 \quad 695 + 417 \times 1 = 1112 \\ 695 & \quad 417 \quad 417 + 278 \times 1 = 695 \\ 417 & \quad 278 \quad 278 + 139 \times 1 = 417 \\ 278 & \quad 139 \quad 0 + 139 \times 2 = 278 \end{aligned}$$

扩展欧几里得算法的流程则如下:

$$\begin{aligned} (139 \quad 0) \quad 139 &= 139 \times 1 + 0 \times 0 \\ (278 \quad 139) \quad 139 &= 278 \times 0 + 139 \times 1 \\ (417 \quad 278) \quad 139 &= 417 \times 1 + 278 \times (-1) \\ (695 \quad 417) \quad 139 &= 695 \times (-1) + 417 \times 2 \\ (1112 \quad 695) \quad 139 &= 1112 \times 2 + 695 \times (-4) \end{aligned}$$

我们可以得到 $1112x + 695y = 139$ 的一组解是 $x = 2, y = -4$ 。

在简单完成完成了推导与证明之后, 接下来我们需要用代码来实现:

```
/*
 * 扩展欧几里得算法。
 * 不仅求得两整数m和n的最大公约数, 还能找到整数x和y, 使得 m*x +
 * n*y = gcd(m, n)。
 */
```

```
/*
int exgcd(int m, int n, int &x, int &y) {
    // 临时变量，用于存储中间的系数
    int x1, y1, x0, y0;

    x0 = 1; y0 = 0;
    x1 = 0; y1 = 1;

    x = 0;
    y = 1;

    int r = m % n; // 初始余数
    int q = (m - r) / n; // 初始商

    while (r) {
        // 更新系数
        x = x0 - q * x1;
        y = y0 - q * y1;

        x0 = x1; y0 = y1;
        x1 = x; y1 = y;
        // 更新m, n, 余数和商以进行下一次迭代
        m = n;
        n = r;
        r = m % n;
        q = (m - r) / n;
    }
    // 返回最大公约数
    return n;
}
```

以上是对拓展欧几里得算法的介绍，根据本节所讲算法，完成下题。

例 10.2.3 (洛谷 P1082-同余方程). 关于 x 的同余方程 $ax \equiv 1 \pmod{b}$ 的

最小正整数解。

输入格式

一行，包含两个正整数 a, b ，空格隔开。

输出格式

一个正整数 x_0 ，即最小正整数解。输入数据保证一定有解。

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

int exgcd(int a, int b, int &x, int &y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }
    int d = exgcd(b, a % b, y, x);
    y -= (a / b) * x;
    return d;
}

int main() {
    int a, b, x, y;
    cin >> a >> b;
    exgcd(a, b, x, y);
    cout << ((x % b) + b) % b;
    return 0;
}
```

10.3 逆元

首先，我们来看两个概念：

定义 10.3.1 (同余). 设 m 为正整数，如果有两个整数 a, b 满足 $m|(a - b)$ ，则称整数 a, b 对模 m 同余，记作 $a \equiv b \pmod{m}$ 。

定义 10.3.2 (逆元). 如果 $ab \equiv 1 \pmod{p}$, 则代表 ab 在模 p 的意义下互为逆元, 记作 $a = \text{inv}(b)$ 。

取模运算对于加法、减法和乘法都是封闭的, 所以在进行大数计算时可以步骤间取模来避免溢出, 即:

$$\begin{aligned} (a + b) \% p &== (a \% p + b \% p) \% p \\ (a - b) \% p &== (a \% p - b \% p) \% p \\ (a * b) \% p &== (a \% p * b \% p) \% p \end{aligned}$$

但取模运算对除法来说并不封闭, 所以为了解决取模意义下除法的问题而引入逆元。 $\text{inv}(b)$ 可以视作模 p 意义下的 $\frac{1}{b}$, 那么对于模 p 意义下的 $\frac{a}{b}$ 则可以视作 $a \cdot \text{inv}(b) \pmod{p}$ 。即:

$$(a / b) \% p = (a * \text{inv}(b)) \% p = (a \% p * \text{inv}(b) \% p) \% p$$

求解逆元有 2 种方法: 费马小定理、拓展欧几里得。

10.3.1 拓展欧几里得法求解

拓展欧几里得算法 前文已经介绍过, 事实上这也是最常用的求逆元方式。如果 $\gcd(a, p) = 1$, 则有 $ax + py = 1$, 在等式两边同时对 p 取余就得到了 $ax \equiv 1 \pmod{p}$, 所以只需解出该情况下的 x 就得到了 a 的逆元。拓展欧几里得算法的程序如下:

```
typedef long long LL;

void exgcd(LL a, LL b, LL &x, LL &y, LL &d) {
    if (b == 0) d = a, x = 1, y = 0;
    else exgcd(b, a % b, x, y, d), t = x, x = y, y = t - a / b * y;
}

LL inv(LL t, LL p) { //无解情况
    LL d, x, y;
    ex_gcd(t, p, x, y, d);
    return d == 1 ? (x % p + p) % p : -1;
}
```

}

10.3.2 费马小定理求解

定理 10.3.3 (费马小定理). 若 p 是质数, 且 $\gcd(a, p) = 1$, 则有 $a^{p-1} \equiv 1 \pmod{p}$ 。

费马小定理规定了 p 一定为一个质数, 所以 a 和 p 一定互质, 那么双方在模 p 的意义下同时除 a 可得 $a^{p-2} \equiv \frac{1}{a} \pmod{p}$ 。也就是 $\text{inv}(a) = a^{p-2} \pmod{p}$, 这需要用快速幂来计算 a^{p-2} 。

那么, 我们根据快速幂的思路, 我们来求解 a^{p-2} :

```
ll quickPow(ll a, ll n, ll p) { // 快速幂
    ll ans = 1;
    while (n) {
        if (n % 2 == 1)
            ans = ans % p * a % p;
        a = a % p * a % p;
        n >>= 1;
    }
    return ans;
}

ll inv(ll a, ll p) {
    return quickPow(a, p - 2, p);
}
```

注: p 必须要是质数。

10.3.3 线性递推法

前文所述是通常求逆元的两种方法, 但是洛谷的一道题却无法使用, 下面我们来看一道例题:

例 10.3.4. 给定 n, p , 求 $1 - n$ 中所有整数在模 p 意义下的乘法逆元, $1 \leq n \leq 3 \times 10^6$ 。

输入格式:

一行两个正整数 n, p

输出格式:

输出 n 行, 第 i 行表示 i 在模 p 下的乘法逆元。

本题采用线性递推的解法, 思路如下: 设 $p = aq + r$, 即 $q = \lfloor \frac{p}{a} \rfloor, r = p \bmod a$, 在模 p 意义下, 有 $aq + r \equiv 0 \pmod{p}$, 移项得到 $a = -r \cdot \text{inv}(q) \pmod{p}$, 那么有 $\text{inv}(a) = -q \cdot \text{inv}(r) \pmod{p}$ 。即 $\text{inv}(a) = -\lfloor \frac{p}{a} \rfloor \cdot \text{inv}(p \bmod a) \pmod{p}$ 。

```
ll Inv[MAXN] = {0, 1};

inline ll mod(ll a, ll p) {
    return (a % p + p) % p;
}

ll inv(ll a, ll p){
    if (Inv[a])
        return Inv[a];
    Inv[a] = mod(-p / a * inv(p % a, p), p);
    return Inv[a];
}
```

10.4 中国剩余定理

中国剩余定理, 也称孙子定理, 来源于孙子算经的一个问题:

例 10.4.1 (孙子算经). 有物不知其数, 三三数之剩二, 五五数之剩三, 七七数之剩二。问物几何?

这实际上是一组同余方程，同余方程的形式如下：

$$\begin{cases} x \equiv b_1 \pmod{a_1} \\ x \equiv b_2 \pmod{a_2} \\ \dots \\ x \equiv b_n \pmod{a_n} \end{cases}$$

数学家在研究中发现，这一方程组有解的一个充分条件是 a_1, a_2, \dots, a_n 两两互质，并用构造法给出了这种情况下方程的通解。

我们将孙子算经的形式翻译一下，题意是要我们找到一个 x 使得下面的式子成立：

$$\begin{cases} x \equiv 2 \pmod{3} \\ x \equiv 3 \pmod{5} \\ x \equiv 2 \pmod{7} \end{cases}$$

我们要找一个 x 使得三个式子同时成立有点困难，但如果要找到 n_1, n_2 和 n_3 分别使得 $n_1 \equiv 2 \pmod{3}, n_2 \equiv 2 \pmod{5}, n_3 \equiv 2 \pmod{7}$ 成立就相对简单一点。

现在假设 n_1, n_2 和 n_3 已知，但这三个数并不是我们的目标，我们的目标是求出 x 使得三个式子同时成立，那么我们是否可以简单的令 $x = n_1 + n_2 + n_3$ ？我们需要验证一下猜想，已知 $n_1 \equiv 2 \pmod{3}$ ，什么情况下才会有 $n_1 + n_2 \equiv 2 \pmod{3}$ ？显然只有当 n_2 是 3 的倍数的情况下才会成立。那么在此基础上， $n_1 + n_2 + n_3 \equiv 2 \pmod{3}$ 这个式子成立的条件必然是 n_2, n_3 都是 3 的倍数。

顺着刚刚的思路推下去，符合方程组的条件是： n_1 是 35 的倍数， n_2 是 21 的倍数， n_3 是 15 的倍数。也就是说，现在我们只需要解出下面三个同

余方程即可得到答案：

$$\begin{cases} 35m_1 \equiv 2 \pmod{3} \\ 21m_2 \equiv 3 \pmod{5} \\ 15m_3 \equiv 2 \pmod{7} \end{cases}$$

我们注意到模数两两互质，即有 $\gcd(35, 3) = \gcd(21, 5) = \gcd(15, 7) = 1$ ，因而我们可以用拓展欧几里得算法求解：

$$\begin{cases} 35 \frac{m_1}{2} \equiv \frac{2}{2} \pmod{3} \\ 21 \frac{m_2}{3} \equiv \frac{3}{3} \pmod{5} \\ 15 \frac{m_3}{2} \equiv \frac{2}{2} \pmod{7} \end{cases} = \begin{cases} 35\omega_1 \equiv 1 \pmod{3} \\ 21\omega_2 \equiv 1 \pmod{5} \\ 15\omega_3 \equiv 1 \pmod{7} \end{cases}$$

其中， $m_1 = 2\omega_1, m_2 = 3\omega_2, m_3 = 2\omega_3$ ，解得 $\omega_1 = 2, \omega_2 = 1, \omega_3 = 1$ ，进而得到 $m_1 = 2\omega_1 = 4, m_2 = 3\omega_2 = 3, m_3 = 2\omega_3 = 2$ ，最后带入得到：

$$\begin{cases} n_1 = 35m_1 = 140 \\ n_2 = 21m_2 = 63 \\ n_3 = 15m_3 = 30 \end{cases}$$

此时 $x = n_1 + n_2 + n_3 = 233$ ，是同余方程的一个特解，特解的含义为所有解中的一个，因为对于原方程来说，所有与 233 在模 105 意义下同余的数都是这个方程组的解，如果要求最小正数解只需对 105 取模即可，这里得出来是 23。

现在我们不失一般化的总结刚刚的过程，设 $p = \prod_{i=1}^n a_i$ （即所有模数的乘积），并设 $r_i = \frac{p}{a_i}$ （在“物不知数”中即为 35、21 和 15）。于是 $w_i = \text{inv}(r_i)|_{a_i}$ （表示 r_i 在模 a_i 意义下的逆元）， $m_i = b_i w_i$ ，而 $n_i = r_i m_i$ ，所有 n_i 相加即得 x 。

综合一下可以得到一个公式：

$$x \equiv \sum_{i=1}^n b_i r_i [r_i]^{-1} \Big|_{a_i} \pmod{p}$$

以及对应的代码实现：

```
// a是模数数组，b是余数数组，n是数组长度
long long CRT(long long a[], long long b[], long long n) {
    ll p = 1, x = 0;
    for (int i = 0; i < n; ++i) {
        p *= a[i];
    }
    for (int i = 0; i < n; ++i) {
        ll r = p / a[i];
        // inv() 求逆元
        x += (b[i] * r * inv(r, a[i])) % p;
    }
    return x % p; // 返回最小正整数解
}
```

接下来是一个例题：

例 10.4.2 (luogu-1495 曹冲养猪). 自从曹冲搞定了大象以后，曹操就开始捉摸让儿子干些事业，于是派他到中原养猪场养猪，可是曹冲满不高兴，于是在工作中马马虎虎，有一次曹操想知道母猪的数量，于是曹冲想狠狠要曹操一把。

举个例子，假如有 16 头母猪，如果建了 3 个猪圈，剩下 1 头猪就没有地方安家了，如果建造了 5 个猪圈，但是仍然有 1 头猪没有地方去，然后如果建造了 7 个猪圈，还有 2 头没有地方去。

你作为曹总的私人秘书理所当然要将准确的猪数报给曹总，你该怎么办？

输入格式

第一行包含一个整数 $n(n \leq 10)$ 代表建立猪圈的次数；

解下来 n 行，每行两个整数 $a_i, b_i(b_i \leq a_i \leq 1000)$ ，表示建立了 a_i 个猪圈，有 b_i 头猪没有去处。你可以假定 $a_i \sim a_j$ 互质。

```
3
3 1
5 1
7 2
```

输出格式

输出包含一个正整数，即为曹冲至少养母猪的数目。

```
16
```

将例题10.4.2写成同余方程的形式可以得到：

$$\begin{cases} x \equiv 1 \pmod{3} \\ x \equiv 1 \pmod{5} \\ x \equiv 2 \pmod{7} \end{cases}$$

解决方法与之前相同，下面给出 AC 程序：

```
#include <bits/stdc++.h>
#define ll long long

using namespace std;

// 拓展欧几里得算法 ax + by
ll exgcd(ll a, ll b, ll &x, ll &y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }
}
```

```
ll d = exgcd(b, a % b, y, x);
y -= (a / b) * x;
return d;
}

// 逆元
ll inv(ll a, ll p) {
    ll x, y;
    exgcd(a, p, x, y);
    return (x % p + p) % p;
}

// 解同余方程
ll CRT(ll a[], ll b[], ll n) {
    ll p = 1, x = 0;
    for (int i = 0; i < n; ++i)
        p *= a[i];
    for (int i = 0; i < n; ++i) {
        ll r = p / a[i];
        x += (b[i] * r * inv(r, a[i])) % p;
    }
    return x % p;
}

int main() {
    ll n, a[10], b[10];
    cin >> n;
    for (int i = 0; i < n; ++i){
        cin >> a[i] >> b[i];
    }
    cout << CRT(a, b, n) << endl;
    return 0;
}
```

10.5 素数筛

素数筛法，是一种快速“筛”出 $2 \sim n$ 之间所有素数的方法。朴素的筛法叫埃氏筛 (the Sieve of Eratosthenes, 埃拉托色尼筛)，它的过程是这样的：我们把 $2 \sim n$ 的数按顺序写出来：

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

从前往后看，找到第一个未被划掉的数 2，这说明它是质数。然后把 2 的倍数 (不包括 2) 划掉：

2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ 9 ~~10~~ 11 ~~12~~ 13 ~~14~~ 15 ~~16~~

下一个未被划掉的数是 3，它也是质数，把 3 的倍数划掉：

2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11 ~~12~~ 13 ~~14~~ ~~15~~ ~~16~~

接下来应该是 5，但是 5 已经超过 $\sqrt{16}$ 了，所以遍历结束，剩下未被划掉的都是素数：

2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11 ~~12~~ 13 ~~14~~ ~~15~~ ~~16~~

代码如下：

```
bool is_prime[MAXN];  
  
void primeSieve(int n) {  
    for (int i = 2; i * i <= n; i++)  
        if (!is_prime[i])
```

```

    for (int j = i * i; j <= n; j += i)
        is_prime[j] = true;
}

```

上述代码十分简洁，时间复杂度是 $O(n \log \log n)$ 。但是我们可能会发现，在筛的过程中我们会重复筛到同一个数，例如 12 同时被 2 和 3 筛到，30 同时被 2、3 和 5 筛到。所以我们引入欧拉筛，也叫线性筛，可以在 $O(n)$ 时间内完成对 $2 \sim n$ 的筛选。它的核心思想是：**让每一个合数被其最小质因数筛到。**

我们这次除了把 $2 \sim n$ 列出来，还维护一个质数表：

```

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
primes:()

```

还是从头到尾遍历，第一个数是 2，未被划掉，把它放进质数表：

```

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
primes:(2)

```

然后我们用 2 去乘质数表里的每个数，划掉它们：

```

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
primes:(2)

```

下一个数是 3 加入质数表，同时需要划掉 6、9：

```

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
primes:(2, 3)

```

下一个数是 4，需要注意的是：这里划掉的数也要遍历，只是**不加入质数**

表，先划掉 8，但我们不划掉 12，因为 $12 = 2 \times 6 = 3 \times 4$ ，应该由它的最小质因数 2 筛掉，而不是 3。

实际上，对于 x ，我们遍历到质数表中的 p ，且发现 $p|x$ 时，就应当停止遍历质数表。因为：设 $x = pr(r \geq p)$ (p 本身是 x 的最小质因数)，那么对于任意 $p' > p$ ，有 $p'x = pp'r = p(p'r)$ ，说明 $p'x$ 的最小质因数不是 p' ，我们不应该在此划掉它。

举个例子，如这里的 2 能整除 4，则 $4 = 2 \times 2$ ，那么对于任何大于 2 的质数 p ，都有 $4p = 2 \times 2p$ ，其中 2 是一个比 p 更小的质数，所以 $4p$ 应该被 2 筛掉而不是被 p 筛掉。

下一个 5 加入质数表，划掉 10, 15：

```
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
primes:(2, 3, 5)
```

下一个 6，划掉 12，6 被 2 整除，跳过……按这样的步骤进行下去，可以筛掉所有的合数，并得到一张质数表。

```
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
primes:(2, 3, 5)
```

我们可以保证每个合数都被筛过，设任意合数 $x = pr(r \geq p)$ ，其中 p 是 x 的最小质因数，又设 $r = pr'(r' \geq p')$ ， p' 是 r 的最小质因数。在处理 r 时，要遍历质数表，直到遇到 p' 时才结束，所以任意小于等于 p' 的质数与 r 的乘积，都会在此时被筛掉。

而由于一定有 $p \leq p'$ (因为 x 的最小质因数是 p ，而不是 p')，所以在处理到 r 时， rp 一定会被筛到。具体实现如下：

```
bool is_prime[MAXN];
vector<int> primes; // 质数表
void eulerSieve(int n) {
```

```

for (int i = 2; i <= n; i++) {
    if (!is_prime[i]) {
        primes.push_back(i);
    }
    for (int p : primes) {
        if (p * i > n) {
            break;
        }
        is_prime[p * i] = true;
        if (i % p == 0) {
            break;
        }
    }
}

```

有一点需要注意：判断越界那里最好使用乘法而不是除法，一般不会溢出，计算机算除法比乘法要慢得多。

完成洛谷P3383。

10.6 欧拉函数

数论中的欧拉函数 $\varphi(x)$ 是一个非常重要的函数，它定义为小于（或不大于） x 但与 x 互质的正整数的数量，例如 $\varphi(12) = 4$ ，有 1、5、7、11 与之互质。特别地，规定 $\varphi(1) = 1$ 。对于欧拉函数，主要有如下性质：

1. 若 p 是质数，则 $\varphi(p^n) = p^{n-1}(p - 1)$
2. 若 $a|x$ ，则 $\varphi(ax) = a\varphi(x)$
3. 若 a, b 互质，则 $\varphi(a)\varphi(b) = \varphi(ab)$

对于上述的三个性质，我们先看第三点，欧拉函数的第三点性质称为**积性函数**，这里给出积性函数的定义，但需要注意的是**欧拉函数是积性函数，不是完全积性函数**。

定义 10.6.1 (积性函数). 对于任意互质的整数 a, b 有性质 $f(ab) = f(a)f(b)$ 的数论函数。

定义 10.6.2 (完全积性函数). 对于任意整数 a, b 有性质 $f(ab) = f(a)f(b)$ 的数论函数。

首先是第一个性质，在不大于 p^n 的正整数中，不与之互质的只有 p 的倍数： $p, 2 \cdot p, 3 \cdot p, \dots, p^{n-1} \cdot p$ ，共 p^{n-1} 个，所以 $\varphi(p^n) = p^n - p^{n-1} = p^{n-1}(p-1)$ 。

紧接着，对于第二个性质，设那 $\varphi(x)$ 个整数分别为 $d_1, d_2, \dots, d_{\varphi(x)}$ ，由于 $a|x$ ，这些数也都与 a 互质，故小于 ax 且与之互质。所以如果 $a > 1$ ，则 $d_1 + x, d_2 + x, \dots, d_{\varphi(x)} + x$ 也小于 ax 且与之互质，如果 $a > 2$ ，则 $d_1 + 2x, d_2 + 2x, \dots, d_{\varphi(x)} + 2x$ 也小于 ax 且与之互质……一共有 a 组，所以一共有 $a\varphi(x)$ 个小于 ax 且与之互质的正整数。

求解欧拉函数，我们首先把正整数质因数分解： $x = p_1^{k_1} p_2^{k_2} \cdots p_n^{k_n}$ ，其中所有 $p_i^{k_i}$ 两两互质，那么根据欧拉函数的性质有： $\varphi(x) = p_1^{k_1-1}(p_1 - 1) \cdot p_2^{k_2-1}(p_2 - 1) \cdots p_n^{k_n-1}(p_n - 1)$ ，即： $\varphi(x) = x \cdot \frac{p_1-1}{p_1} \cdot \frac{p_2-1}{p_2} \cdots \frac{p_n-1}{p_n}$ 。

这里给出一个解决的方案：

```
int phi(int n) {
    int res = n;
    for (int i = 2; i * i <= n; i++) {
        if (n % i == 0) {
            // 先除再乘防止溢出
            res = res / i * (i - 1);
        }
        // 每个质因数只处理一次，可以把已经找到的质因数除干净
        while (n % i == 0) {
            n /= i;
        }
    }
    return res;
}
```

```
    }
}

if (n > 1) {
    // 最后剩下的部分也是原来的n的质因数
    res = res / n * (n - 1);
}

return res;
}
```

这个方案的时间复杂度为 $O(\sqrt{n})$ 。

第十一章 图论

11.1 最短路问题

在本节，我们要解决的目标是图论中最为基础的一个问题：最短路径问题，最短路问题可以分为两类：**单源最短路**和**多源最短路**。单源最短路只需要求一个**固定的起点**到各个顶点的最短路径，多源最短路则要求得出**任意两个顶点**之间的最短路径。

11.1.1 Floyd 算法

我们通常用Floyd算法解决**多源最短路**问题，Floyd算法的核心如下：

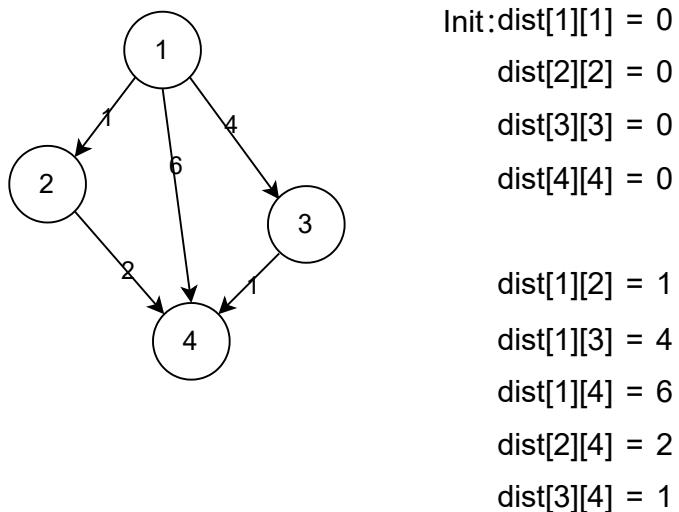
```
int dist[400][400];
void Floyd(int n) {
    for (int k = 1; k <= n; ++k)
        for (int i = 1; i <= n; ++i)
            for (int j = 1; j <= n; ++j)
                dist[i][j] = min(dist[i][j], dist[i][k] +
                    dist[k][j]);
}
```

Floyd算法的实现本身非常简洁，同时，我们还可以从代码中发现这是一种**动态规划**思想，即每一次循环更新经过前k个节点，*i*到*j*的最短路径。

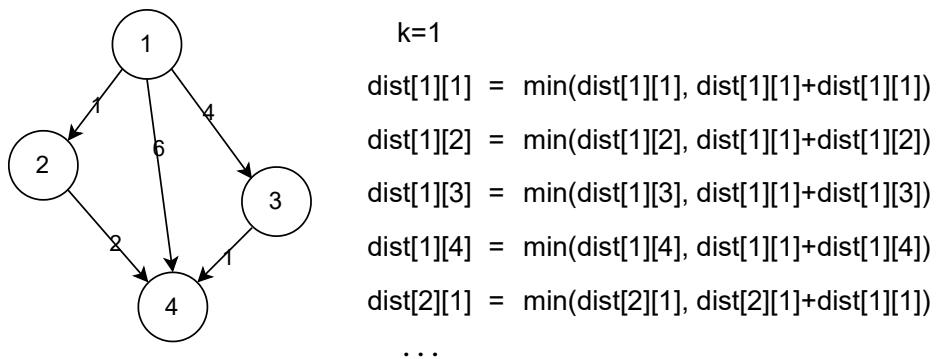
初始化的时候，我们把每个点到自己的距离设为0，每新增一条边，就把

从这条边的起点到终点的距离设为此边的边权。其他距离初始化为INF(一个超过边权数据范围的大整数，注意防止溢出)。

我们同样通过一个例子来体会Floyd算法，对于一个有向图进行初始化操作：

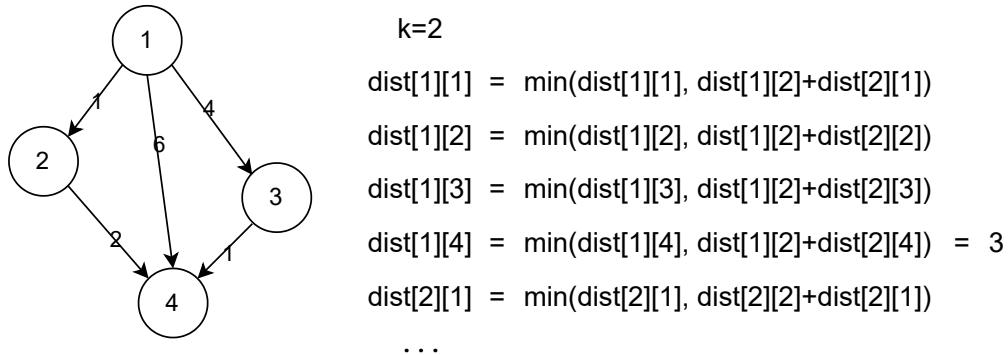


第一趟， $k = 1$ ：



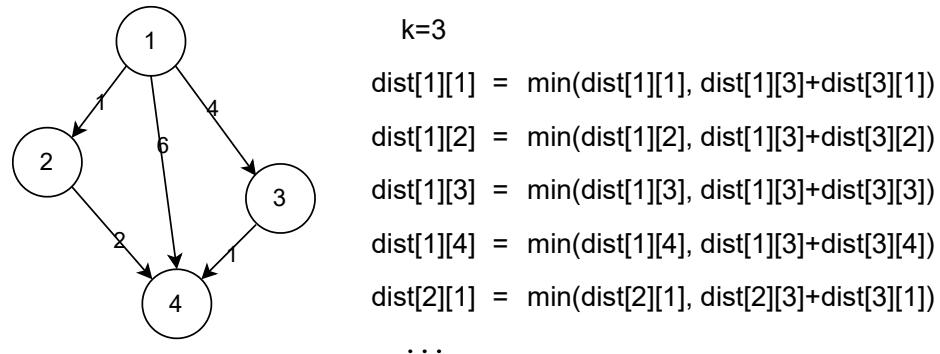
这里的含义是，图中任意两点之间的距离和这两点经由节点 1 后的距离取最小值，但由于除节点 1 以外所有的节点都没有到达节点 1 的路径，因而没有一个距离能经由节点 1 缩短，例如，节点 2 到节点 4 的距离是2，而 $\text{dist}[2][1] + \text{dist}[1][4] = \text{INF} + 6$ ，未能缩短路径。

第二趟, $k = 2$:



原本节点 1 到节点 4 的直接距离是 6, 而经由节点 2, $\text{dist}[1][2] = 1$, $\text{dist}[2][3] = 2$, 因此 $\min(6, 1 + 2) = 3$, 也就是节点 1 到节点 4 的路径经由节点 2 缩短了。

第三趟, $k = 3$:



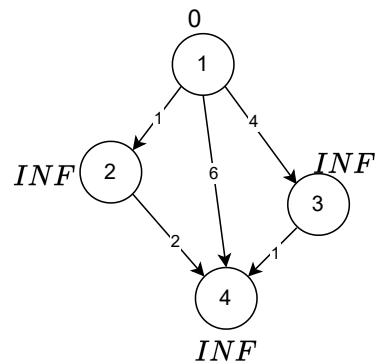
注意, 虽然 $1 \rightarrow 3 \rightarrow 4$ 的路径比 $1 \rightarrow 4$ 短, 但由于 $\text{dist}[1][4]$ 已经被更新为 3 了, 所以这一趟并不能更新距离。接下来 $k = 4$ 显然也更新不了任何点。

回顾流程我们可以发现, 每一趟循环, 实际上都是在考察**能不能经由 k 点, 把 i 到 j 的距离缩短**。此外, Floyd 算法的时间复杂度是 $O(n^3)$, 且空间复杂度为 $O(n^2)$, 所以只适用于数据规模较小的情形。

11.1.2 Bellman-Ford 算法

我们通常更为关注**单源最短路**问题，即固定起点的最短路径问题，因为起点固定，我们只需要一个数一维组`dist[]`即可存储每个点到起点的距离。

同样我们来看一个例子，如图所示一个有向有权图，现在设节点 1 是起点，则我们初始化`dist[1] = 0`，其他距离初始化为`INF`。



假设没有负权环，我们要找两点之间的最短路径，设起点是 S ，终点是 E ，那么这条路上的最短路径一定以 $S \rightarrow P_1 \rightarrow P_2 \rightarrow \dots \rightarrow P_N \rightarrow E$ 的形式呈现出来，且点的总个数一定**不大于** n 。

现在我们进行松弛操作，所谓松弛操作，即起点 S 和终点 E 之间，能否经由节点 x 使得距离变短：

```

dist[E] = min(dist[E], dist[x] + e[x][E]);
//这里的e[x][E]表示x、E之间的距离，具体形式可能根据存图方法
//不同而改变
  
```

于是，我们为了要找到最短路径，需要以下步骤：

1. 先松弛 S, P_1 ，此时`dist[P1]`必然等于`e[S][P1]`；
2. 再松弛 P_1, P_2 ，因为 $S \rightarrow P_1 \rightarrow P_2$ 是最短路的一部分，最短路的子路也是最短路，所以`dist[P2]`不可能小于`dist[P1] + e[P1][P2]`，因

此它会被更新为 $\text{dist}[P_1] + e[P_1][P_2]$ ，即 $e[S][P_1] + e[P_1][P_2]$ 。

3. 再松弛 P_2, P_3, \dots, P_n ，以此类推，最终我们可以发现 $\text{dist}[E]$ 必然等价于 $e[S][P_1] + e[P_1][P_2] + \dots$ ，这恰好就是最短路径。

问题在于，我并不知道这些 P_1, P_2 代表什么，我们只要找到它们，就可以得到最短路径，而 Bellman-Ford 算法的做法就是：**把所有边松弛一遍**。

同样的，因为我们要求的是最小值，且边权都是正的，所以多余的松弛操作不会使某个 dist 比最小值还小。那么把所有边的端点松弛完一遍后，我们可以保证 S, P_1 已经被松弛过了，现在我们要松弛 P_1, P_2 ，我们就直接的**再把所有边松弛一遍**。

现在我们松弛了 P_1, P_2, \dots, P_n ，继续这么松弛下去，但循环总要有个尽头，所以我们还需要找到一个结束的标志，之前我们发现：最短路上的点的总个数一定不大于 n ，尽管一般而言最短路上的顶点数比 n 少得多，但因为多余的松弛操作不会影响结果，我们不妨**把所有边松弛 $n - 1$ 遍**！

这就是 Bellman-Ford 算法，是一种很暴力的算法，它的时间复杂度是 $O(nm)$ 。代码如下：

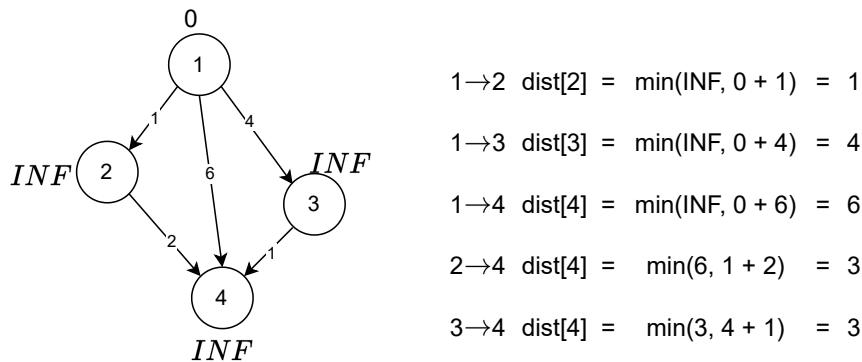
```
void Bellman_Ford(int n, int m) {
    for (int j = 0; j < n - 1; ++j)
        for (int i = 1; i <= m; ++i)
            dist[edges[i].to] = min(dist[edges[i].to], dist[edges[i].from] + edges[i].w);
}
```

这里用的是**链式前向星存图**，但是建议存的时候多存一个 from，方便遍历所有边。当然其实并没什么必要，这里直接暴力存边集就可以了，因为这个算法并不关心每个点能连上哪些边。

虽然通常情况下我们不考虑**负权环**，但 Bellman-Ford 算法是可以很简单地处理负权环的，只需要**再多对每条边松弛一遍**，如果这次还有点被更新，就说明存在负权环。因为没有负权环时，最短路上的顶点数一定小于 n ，

而存在负权环时，可以无数次地环绕这个环，最短路上的顶点数是无限的。

我们根据上例来看 Bellman-Ford 算法，具体下图所示：



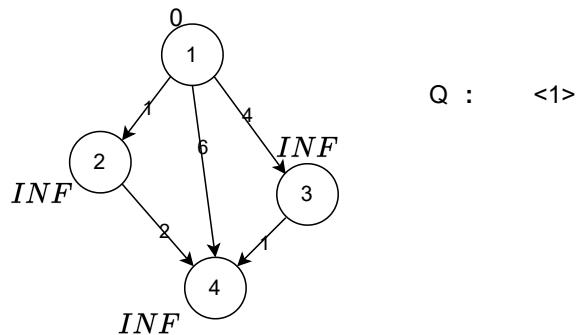
虽然本例只遍历了一遍所有边就把所有最短路求出来了，但为了保证求出正解，通常还需要遍历两次。

11.1.3 SPFA 算法

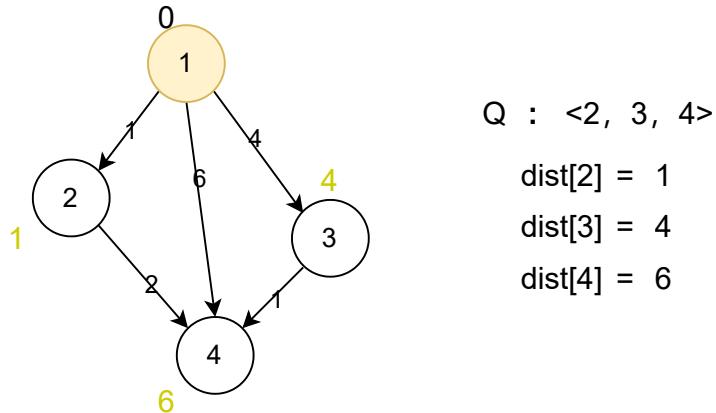
Bellman-Ford 算法 $O(mn)$ 级的时间复杂度还是太高，我们可以继续优化，显然我们不需要每次松弛所有的点，只松弛那些可能更新的点。

观察发现，第一次松弛 S, P_1 时，可能更新的点只可能是 S 能直接到达的点。然后下一次可能被更新的则是 S 能直接到达的点能直接到达的点。SPFA 算法正是利用了这种思想，利用队列优化 Bellman-Ford 算法。

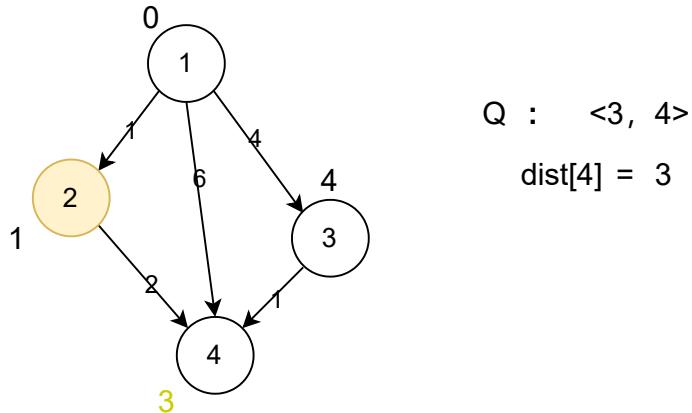
我们来看这个例子，一开始，把起点放进队列：



随后考察节点 1，它可以到达点 2、3、4，于是节点 1 出队，节点 2、3、4 依次入队，入队时松弛相应的边：



现在队首是节点 2，节点 2 出队，由于节点 2 可以到达节点 4，我们松弛 $2 \rightarrow 4$ ，但是节点 4 已经在队列里了，所以 4 号点就不入队了：

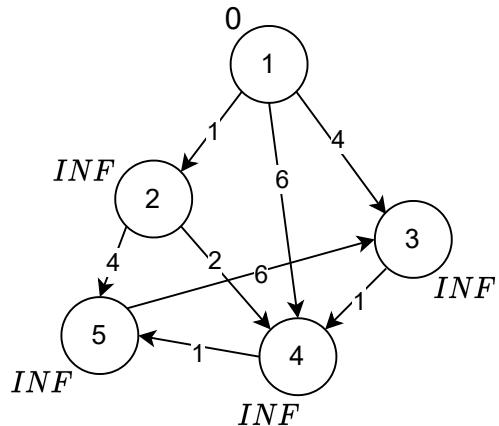


因为这张图非常简单，之后的流程无非是节点 3 出队，松弛 $3 \rightarrow 4$ ，然后节点 4 出队。当队列为空时，流程结束。

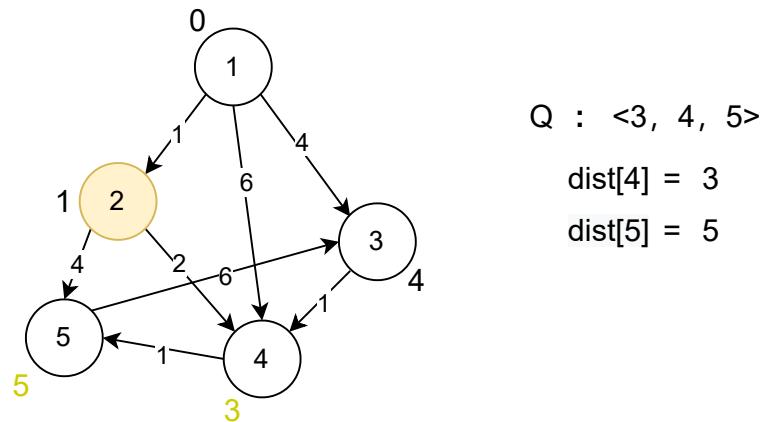
这里需要提一下，如果某个节点已经在队列中，我们就不再将之入队。

为了表明 SPFA 的优越性，我们再来看一个稍微复杂一点的图，现有

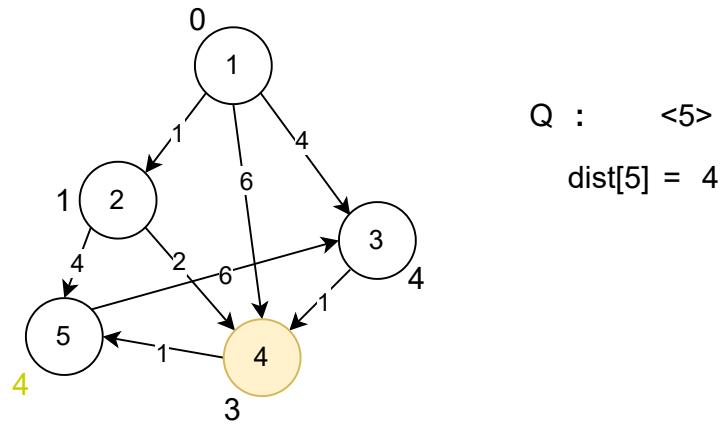
一张五个节点的有向带权图，这张图如果按 Bellman-Ford 算法需要松弛 $8 \times 4 = 32$ 次，现在我们的目的是用 SPFA 算法解决这个问题：



显然前几步跟上次是一致的，我们松弛了 $1, 2, 1, 3, 1, 4$ 。然后队首元素是节点 2，我们让 2 出队，并松弛 $2, 4, 2, 5$ 。由于节点 5 未在队列中，节点 5 入队。



现在队首节点是节点 3，我们根据先前的经验，节点 3 并不能产生最短路径的更新，所以这里就不展示节点 3 相关的过程。随后，节点 4 成为了队首节点，节点 4 出队，松弛 $4, 5$ ，然后节点 5 已经在队列中，所以节点 5 不入队：



最后节点 5 出队, $\text{dist}[3]$ 未被更新, 所以 3 号点通往的点不会跟着被更新, 因此 3 号点不入队, 循环结束。

这个过程中, 我们只进行了 6 次松弛, 远小于 B-F 算法的 32 次, 虽然进行了入队和出队, 但在 n, m 很大时, SPFA 通常还是显著快于 B-F 算法的, 随机数据下期望时间复杂度是 $O(m + n \log n)$ 。

我们可以总结出 SPFA 算法有以下几个特点:

1. 只让当前点能到达的点入队;
2. 如果一个点已经在队列里, 便不重复入队;
3. 如果一条边未被更新, 那么它的终点不入队。

SPFA 算法的原理是: 我们的目标是松弛完 $S \rightarrow P_1 \rightarrow P_2 \rightarrow \dots \rightarrow E$, 所以我们先把 S 能到达的所有点加入队列, 则 P_1 一定在队列中。然后对于队列中每个点, 我们都把它能到达的所有点加入队列, 但不重复入队, 这时我们又可以保证 P_2 一定在队列中。另外, 值得注意的是, 如果 $P_i \rightarrow P_{i+1}$ 是目标最短路上的一段, 那么在松弛这条边时它一定会被更新, 所以如果一条边未被更新, 它的终点就不入队。

我们在这里用一个 `inqueue[]` 数组来记录一个点是否在队列里, 于是 SPFA 的代码如下:

```

void SPFA(int s) {
    queue<int> Q;
    Q.push(s);
    while (!Q.empty()) {
        int p = Q.front();
        Q.pop();
        inqueue[p] = 0;
        for (int e = head[p]; e != 0; e = edges[e].next) {
            int to = edges[e].to;
            if (dist[to] > dist[p] + edges[e].w) {
                dist[to] = dist[p] + edges[e].w;
                if (!inqueue[to]) {
                    inqueue[to] = 1;
                    Q.push(to);
                }
            }
        }
    }
}

```

SPFA 算法也可以判负权环，我们可以用一个数组记录每个顶点进队的次数，当一个顶点进队超过 n 次时，就说明存在负权环。

同时，这里我们给出 SPFA 的vector实现：

```

struct Edge {
    int to, w;
};

vector<Edge> edges[MAXN];
vector<int> dist(MAXN, INT_MAX);
vector<bool> inqueue(MAXN, false);

void SPFA(int s) {
    queue<int> Q;

```

```
Q.push(s);
dist[s] = 0;
inqueue[s] = true;

while (!Q.empty()) {
    int u = Q.front();
    Q.pop();
    inqueue[u] = false;

    for (int i = 0; i < edges[u].size(); i++) {
        int to = edges[u][i].to;
        int w = edges[u][i].w;
        if (dist[to] > dist[u] + w) {
            dist[to] = dist[u] + w;
            if (!inqueue[to]) {
                inqueue[to] = true;
                Q.push(to);
            }
        }
    }
}

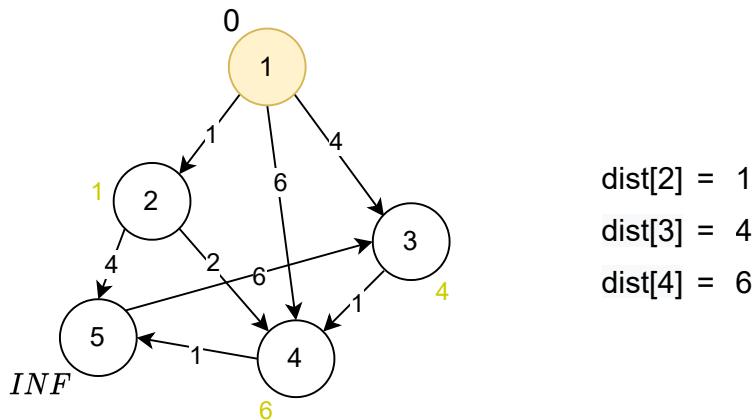
}
```

11.1.4 Dijkstra 算法

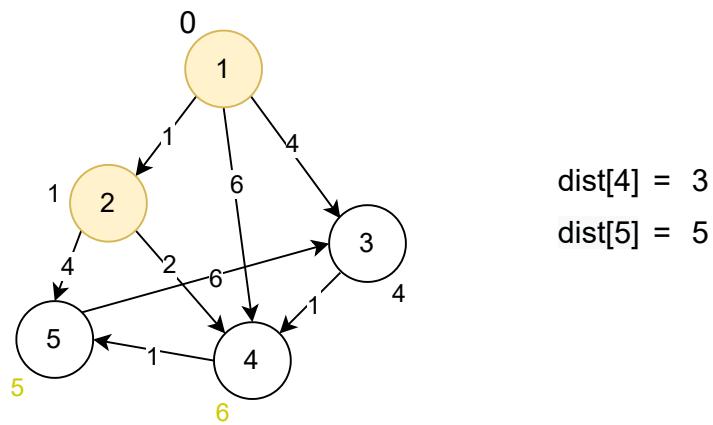
Dijkstra 算法基于一种**贪心**的思想，其复杂度稳定，但 Dijkstra 算法不能处理存在负权环的问题。

假设现有一张**没有负边**的图，首先进行的就是初始化操作，起点到自身的距离自然为 0，现在我们对起点和它能直接到达的所有点进行松弛。同时，因为没有负边，离起点最近的那个顶点的**dist**一定已经是最终结果，即目前的最短路径，因为在没有负边的情况下，不可能经由其他点，使得从起

点到该点的距离变得更短。



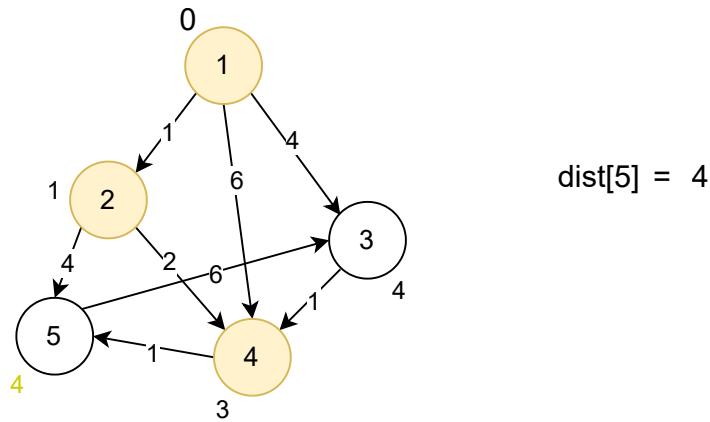
现在我们考察离起点最近的点，即节点 2：



我们对节点 2 和它能直接到达的点进行松弛，这时 dist 保存的是起点直接到达或经由节点 2 到达每个点的最短距离。

这时候，我们取出未访问过的 dist 最小的点，即节点 4，考虑到其他路径都至少要从起点直接到达、或者经由节点 2 到达另一个点 P ，再从 P 到达节点 4，因此这个点的 dist 也不可能变得更短了。

我们继续这个流程，接下来需要松弛的点就是节点 4 能够直接到达的节点：



然后分别考察节点 3 和节点 5，直到所有点都被访问过。

总的来说，Dijkstra 算法的流程就是，不断取出离顶点最近且没有被访问过的点，松弛它和它能到达的所有点。

于是我们的问题变成了如何取出离顶点最近的节点。如果暴力寻找，那就是朴素的Dijkstra算法，时间复杂度是 $O(n^2)$ ，我们可以考虑采取堆结构进行优化。具体来说就是我们可以用一个优先队列来维护所有节点，这样可以实现在 $O(m \log m)$ 的时间内跑完最短路。

首先写一个结构体：

```
struct Polar{
    int dist, id;
    Polar(int dist, int id) : dist(dist), id(id){}
};
```

然后写一个仿函数，再构建优先队列：

```
struct cmp{
    bool operator ()(Polar a, Polar b){ // 重载()运算符，使其成为一个仿函数
        return a.dist > b.dist; // 这里是大于，使得距离短的先出队
}
```

```
};

priority_queue<Polar, vector<Polar>, cmp> Q;
```

最后就是Dijkstra算法的实现：

```
void Dij(int s) {
    dist[s] = 0;
    Q.push(Polar(0, s));
    while (!Q.empty()) {
        int p = Q.top().id;
        Q.pop();
        if (vis[p])
            continue;
        vis[p] = 1;
        for (int e = head[p]; e != 0; e = edges[e].next) {
            int to = edges[e].to;
            dist[to] = min(dist[to], dist[p] + edges[e].w);
            if (!vis[to])
                Q.push(Polar(dist[to], to));
        }
    }
}
```

此外，也可以利用 STL 里的pair实现优先队列：

```
typedef pair<int, int> Pair;
priority_queue<Pair, vector<Pair>, greater<Pair>> Q;
```

这样一来，Dijkstra算法的实现就会略有不同：

```
void Dij(int s) {
    dist[s] = 0;
    Q.push(make_pair(0, s));
    while (!Q.empty()) {
        int p = Q.top().second;
        Q.pop();
```

```

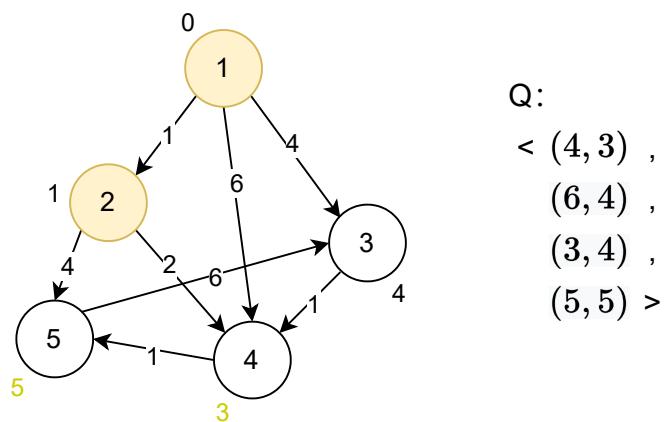
    if (vis[p])
        continue;
    vis[p] = 1;
    for (int e = head[p]; e != 0; e = edges[e].next) {
        int to = edges[e].to;
        dist[to] = min(dist[to], dist[p] + edges[e].w);
        if (!vis[to])
            Q.push(make_pair(dist[to], to));
    }
}
}

```

这样可以省去写结构体和仿函数的步骤，因为pair已经内建了比较函数。

也许你会想，每个步骤不是应该取当前离源点最近、且未被访问过的元素吗，但我们现在每次让一个pair进入优先队列，这时pair里面存储的dist是那时该点到源点的距离，我怎么能保证每次取出来的点恰是离源点最近的点呢？

其实是这样做的，在一个点被访问前，优先队列里会存储这个点被更新的整个历史。比如下面这个状态，队列里既有(6, 4)又有(3, 4)，但是(3, 4)会比(6, 4)先出队，等到(6, 4)出队的时候，4这个点已经被访问了，所以不会有影响。



堆优化的Dijkstra复杂度稳定且较低，但仍要强调：Dijkstra算法不能处理负边，原因其实很明显，如果有负边，就不能保证离源点最近的那个点的dist不会被更新了。

这里我们给出 Dijkstra 的vector实现，其中用一个结构体Polar来修改优先队列的比较函数：

```
// vector 实现
struct Edge {
    int to, w;
};

struct Polar {
    int dist, id;
    bool operator>(const Polar& o) const {
        return dist > o.dist;
    }
};

vector<Edge> edges[MAXN];
vector<int> dist(MAXN, INT_MAX);
vector<bool> vis(MAXN, false);
priority_queue<Polar, vector<Polar>, greater<Polar>> Q;

void Dij(int s) {
    dist[s] = 0;
    Q.push({0, s});

    while (!Q.empty()) {
        Polar top = Q.top();
        Q.pop();
        int u = top.id;

        if (vis[u]) continue;

        for (const Edge& e : edges[u]) {
            int v = e.to;
            int w = e.w;
            if (dist[v] > dist[u] + w) {
                dist[v] = dist[u] + w;
                Q.push({dist[v], v});
            }
        }
    }
}
```

```

    vis[u] = true;

    for (int i = 0; i < edges[u].size(); i++) {
        int to = edges[u][i].to, w = edges[u][i].w;
        if (dist[to] > dist[u] + w) {
            dist[to] = dist[u] + w;
            if (!vis[to])
                Q.push({dist[to], to});
        }
    }
}
}

```

11.1.5 打印路径

之前的所有操作都只是为了求出最短路径的长度，如果我们要具体路径输出呢，其实也很简单，我们只需要用一个`pre[]`数组存储每个点的父节点即可。

单源最短路的起点是固定的，所以每条路有且仅有一个祖先节点，一步步溯源上去的路径是唯一的。相反，这里不能存子节点，因为从源点下去，有很多条最短路径。

每当更新一个点的`dist`时，顺便更新一下它的`pre`。这种方法对 SPFA 和 Dijkstra 算法都适用，以下是对 SPFA 的修改：

```

if (edges[e].w + dist[p] < dist[to]) {
    pre[to] = p;      // 在这里加一行
    dist[to] = edges[e].w + dist[p];
    if (!inqueue[to]) {
        Q.push(to);
        inqueue[to] = 1;
    }
}

```

以打印从 1 到 4 的最短路为例打印最短路径：

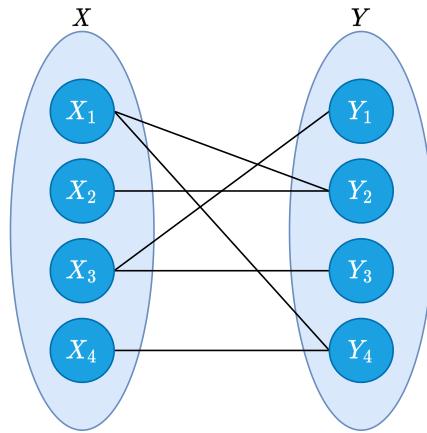
```
int a = 4;
while (a != 1) {
    printf("%d<-", a);
    a = pre[a];
}
printf("%d", a);
```

但这样打印出的结果是反向的箭头，如果想得到正向的箭头，可以先将结果压入数组再逆序打印。

11.2 匈牙利算法

匈牙利算法 (Hungarian algorithm)，主要用于解决一些与二分图匹配有关的问题，所以我们需要来了解一下二分图的定义。

定义 11.2.1 (二分图). 二分图 (*Bipartite graph*) 是一类特殊的图，它可以被划分为两个部分，每个部分内的点互不相连。下图就是一种二分图。

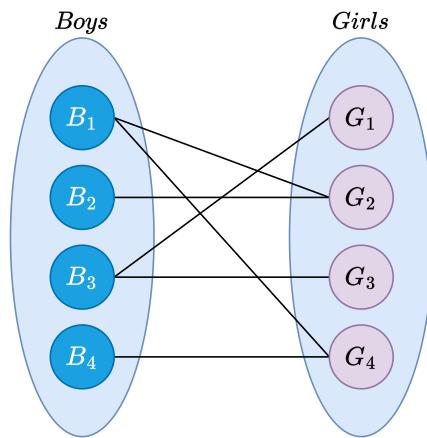


在上面的二分图中，每条边的端点都分别处于点集 X 和 Y 中。匈牙利算法主要用来解决两个问题：求二分图的**最大匹配数**和**最小点覆盖数**。

11.2.1 最大匹配问题

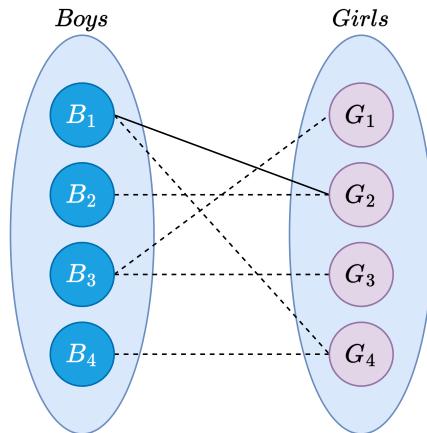
我们来看一个“红娘”的问题，现有两个集合 Boys 和 Girls，分别代表男学生和女学生，边表示他们之间存在暧昧关系。

最大匹配问题，即在二分图中最多能找到多少条没有公共端点的边，在这里就相当于：我们可以撮合任意有暧昧关系的一对男女同学，那么最多能够成全多少对。

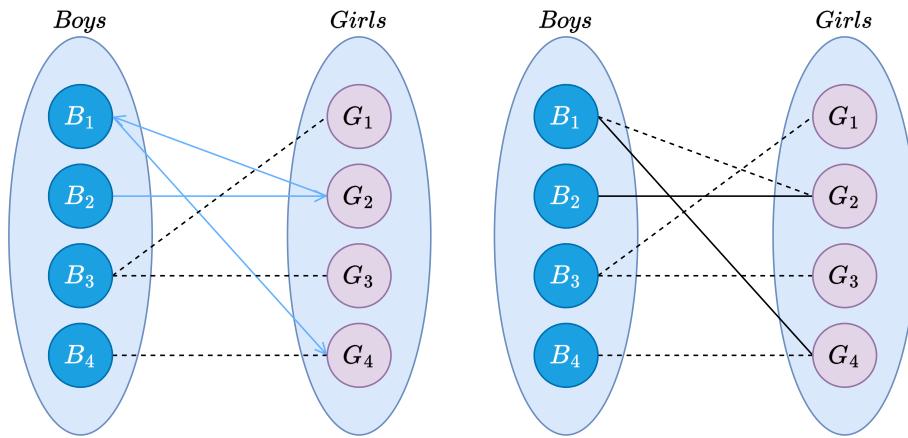


我们来看匈牙利算法的运作流程：

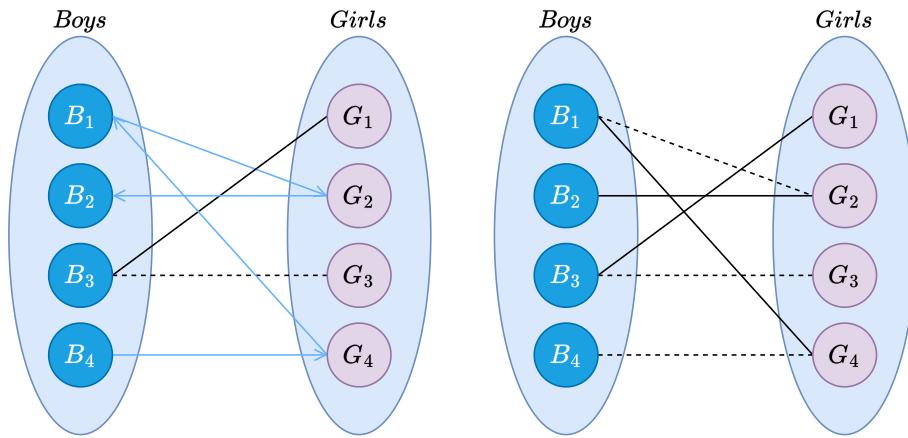
首先我们注意到 B_1 同学，与之直接相连的有 G_2 即二者之间存在暧昧关系，那么我们就暂时连接他们（但并未真正确定）：



然后我们看 B_2 同学, B_2 同学也喜欢 G_2 同学, 但是 G_2 同学此时已经有了对象, 于是我们倒回去看 B_1 同学, 我们发现 B_1 还有别的选择, 且 G_4 同学还没被安排, 于是我们给 B_1 安排上 G_4 , B_2 安排上 G_2 :



随后我们来看 B_3 同学, 发现 B_3 可以与 G_1 同学直接相连, 于是我们将之匹配:



最后来看 B_4 同学, 发现 B_4 同学只钟情 G_4 同学, 但 G_4 同学目前匹配的是 B_1 同学, 尽管 B_1 同学还可以匹配 G_2 同学, 但如果 B_1 选了 G_2 , 那么 B_2 就没有选择, 因此 B_4 没有可匹配的点。

以上就是匈牙利算法的大致流程, 我们来看具体实现:

```
int M, N; //M, N分别表示左、右侧集合的元素数量
int Map[MAXM][MAXN]; //邻接矩阵存图
int p[MAXN]; //记录当前右侧元素所对应的左侧元素
bool vis[MAXN]; //记录右侧元素是否已被访问过
bool match(int i) {
    for (int j = 1; j <= N; ++j)
        if (Map[i][j] && !vis[j]) { //有边且未访问
            vis[j] = true; //记录状态为访问过
            if (p[j] == 0 || match(p[j])) { //如果暂无匹配,
                或者原来匹配的左侧元素可以找到新的匹配
                p[j] = i; //当前左侧元素成为当前右侧元素
                的新匹配
                return true; //返回匹配成功
            }
        }
    return false; //循环结束, 仍未找到匹配, 返回匹配失败
}
int Hungarian() {
    int cnt = 0;
    for (int i = 1; i <= M; ++i) {
        memset(vis, 0, sizeof(vis)); //重置vis数组
        if (match(i))
            cnt++;
    }
    return cnt;
}
```

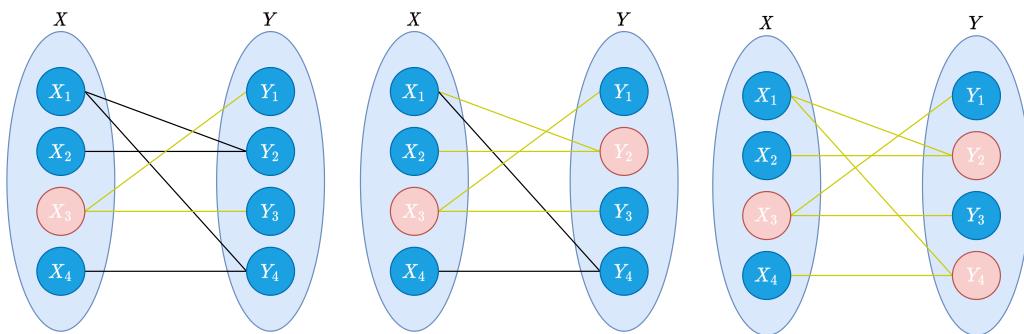
11.2.2 最小点覆盖问题

最小点覆盖问题，即找到最少的一些点，使二分图所有的边都至少有一个端点在这些点之中。换言之，即删除包含这些点的边，可以删掉所有

边。

定理 11.2.2 (König 定理). 一个二分图中的最大匹配数等于这个图中的最小点覆盖数。

根据定理，我们从左侧一个未匹配成功的点出发，走一趟匈牙利算法的流程，所有左侧未经过的点和右侧经过的点，即组成最小点覆盖：



下面是匈牙利算法的一个例题：

例 11.2.3 (P1129 矩阵游戏). 小 Q 是一个非常聪明的孩子，除了国际象棋，他还很喜欢玩一个电脑益智游戏——矩阵游戏。

矩阵游戏在一个 $n \times n$ 黑白方阵进行（如同国际象棋一般，只是颜色是随意的）。每次可以对该矩阵进行两种操作：

- 行交换操作：选择矩阵的任意两行，交换这两行（即交换对应格子的颜色）。
- 列交换操作：选择矩阵的任意两列，交换这两列（即交换对应格子的颜色）。

游戏的目标，即通过若干次操作，使得方阵的主对角线（左上角到右下角的连线）上的格子均为黑色。

对于某些关卡，小 Q 百思不得其解，以致他开始怀疑这些关卡是不是根本就是无解的！于是小 Q 决定写一个程序来判断这些关卡是否有解。

输入格式:

第一行包含一个整数 T , 表示数据的组数, 对于每组数据, 输入格式如下:

第一行为一个整数, 代表方阵的大小 n 。接下来 n 行, 每行 n 个非零即一的整数, 代表该方阵。其中 0 表示白色, 1 表示黑色。

```

2
2
0 0
0 1
3
0 0 1
0 1 0
1 0 0

```

输出格式:

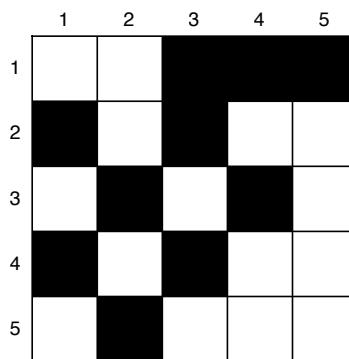
对于每组数据, 输出一行一个字符串, 若关卡有解则输出 Yes, 否则输出 No。

```

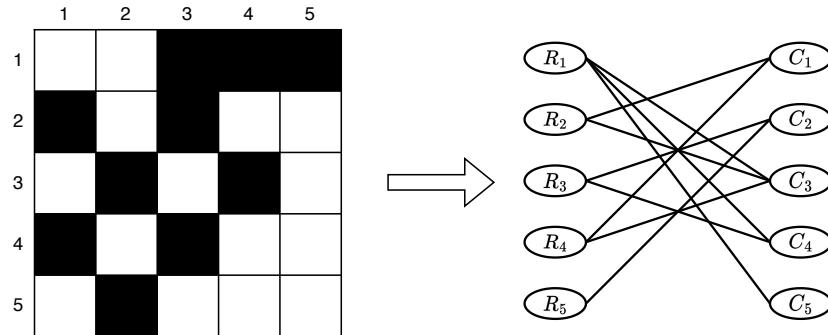
No
Yes

```

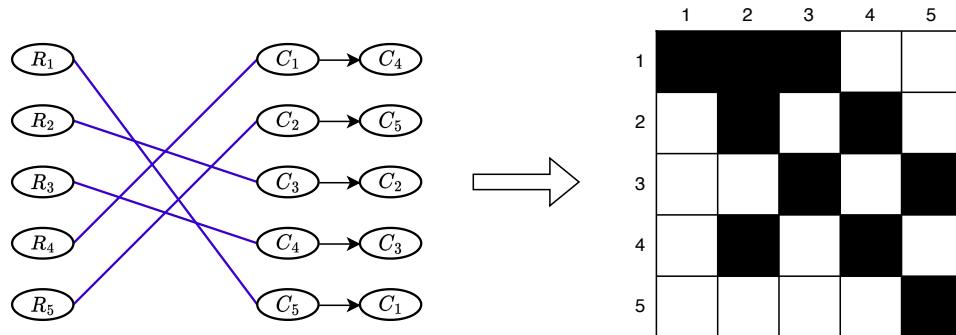
对于例题11.2.3, 我们可以通过匈牙利算法来解决, 首先需要将这个问题转化为二分图的形式: 我们将每行每列看成一个独立的集合, 行和列可以唯一确定一个格子。我们来看一个例子:



在这里我们将行和列分别看成一个集合，考虑到问题的特殊性，我们只需要关注主对角线上的格子，即对于第 i 行，第 i 个格子可以有同行哪些列交换过来（即格子为黑色），这些格子所在的列即是与行的连线，于是我们就可以将这个问题转化为二分图的最大匹配问题：



我们按照匈牙利算法进行最大匹配后，只需要将匹配的列重新编号（即交换）则可以完成题目要求：



综上所述，只需要进行最大匹配后最大匹配数等于 n ，则该问题有解，否则无解，如下面这个无解的例子：

0	1	0	0	0
1	0	1	0	0
0	0	1	0	1
1	0	1	0	0
1	1	0	0	0

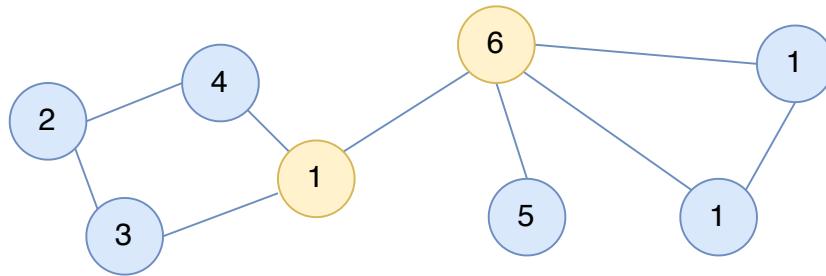
11.3 图的连通性

11.3.1 有向图的强连通分量

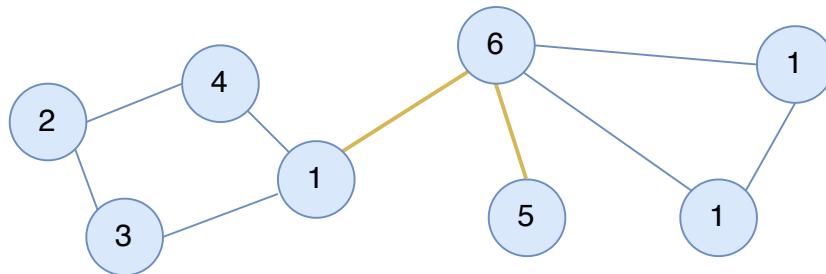
Tarjan 算法是基于深度优先搜索 (DFS) 的算法，用于求解图的连通性问题。Tarjan 算法可以在线性时间内求出无向图的割点与桥，进一步地可以求解无向图的双连通分量；同时，也可以求解有向图的强连通分量、必经点与必经边。

接下来是一些必要的定义：

定义 11.3.1 (割点). 若从图 G 中删除节点 x 以及所有与 x 关联的边之后，图将被分成两个或两个以上的不相连的子图，那么称 x 为图的割点。



定义 11.3.2 (桥). 若从图 G 中删除边 e 之后，图将分裂成两个不相连的子图，那么称 e 为图的桥或割边。



定义 11.3.3 (途径). 图 G 中一个非空的点和边交替出现的有限序列被称为途径，如 $W = \{v_{i_0} e_{i_0} v_{i_1} e_{i_1} v_{i_2} e_{i_2} \cdots v_{i_{k-1}} e_{i_k} v_{i_k}\}$ ，其中与边 e_{i_j} 关联的两个顶

点分别是 $v_{i_{j-1}}$ 与 v_{i_j} 。途径是图中最基本的一种路径，其中 v_{i_0} 称为途径的起点， v_{i_k} 称为途径的终点，其余各点称为内部顶点。

值得注意的是，途径允许经过重复的点和边，在途径的基础上引出迹和路的概念。

定义 11.3.4 (迹). 若某途径 W 中不包含重复的边，则称此途径为迹。

定义 11.3.5 (路). 若某途径 W 中不包含重复的顶点，我们称此途径为路。

对于路，可以使用顶点顺次排开表示，例如 $P = v_1v_2v_3 \dots$ ，路的长等于路中包含的边数。

在有向图当中，如果 u, v 存存在一条从 u 到 v 的路径且存在一条 v 到 u 的路径（双向可达），那么我们称 u, v 是强连通的。

若图 G 中的每对顶点 u, v 都是强连通的，那么我们称该图是一个强连通图。

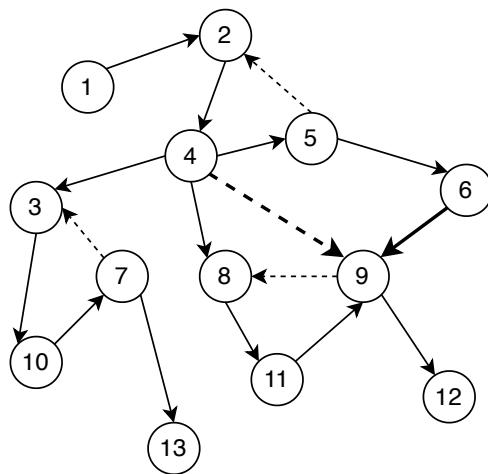
定义 11.3.6 (强连通分量). 且当一个有向图的子图是强连通子图且不是任何一个强连通子图的真子图，则称子图为极大强连通子图，也称其为该有向图的强连通分量 (*Strongly Connected Component, SCC*)。

定义 11.3.7 (DFS 生成树). 以图 G 某个节点出发进行深搜过程中搜索，每个节点只访问一次，所有被访问过的节点与边所形成的树称之为一棵树，亦称之为搜索树、生成树。

以下图为例，有向图的 DFS 生成树有 4 种边，但这 4 种边不一定全部出现：

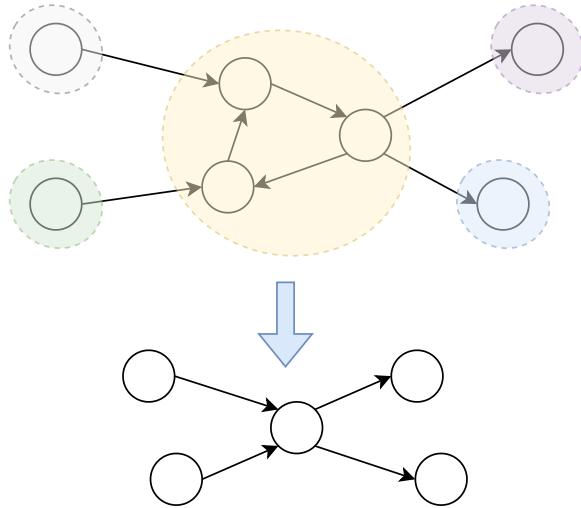
- 树枝边：搜索树中从某节点到其叶节点的路径，即 (x, y) ，其中节点 x 是 y 的父节点。
- 前向边：搜索树中从某节点到其子树中的节点的路径，即 (x, y) ，其中节点 x 是 y 的祖先节点。如图中 $(4, 9)$ 。

- 后向边（返祖边）：搜索树中从某节点到其祖先节点的路径，即 (x, y) ，其中节点 y 是 x 的祖先节点。如图中 $(5, 2), (3, 7), (9, 8)$ 。
- 横叉边：搜索时找到已访问的节点，但该节点并非当前节点的祖先节点。如图中 $(6, 9)$ 。



对于这里的概念只需要理解、认识即可。

介绍完强连通分量后，我们通常利用强连通分量将任意一个**有向图**通过**缩点**的方式转化为一个**有向无环图**。其中缩点所做的是将每个连通分量分别缩成一个点并重新连接。



如图所示，上图中原有五个强连通分量，经过缩点后，每个强连通分量缩成一个点得到新的图。一般来说，强连通分量的一个应用在于：求出一个图中所有的强连通分量，然后将所有的强连通分量缩成一个点，随后就可以采用如拓扑序等方法来处理问题。

现在我们思考一个问题：如何判断点 x 是否是强连通分量中的点？显然有两种情况可以判断：

- 存在后向边可以找到祖先节点；
- 存在横叉边，通过横叉边可以找到祖先节点。

为了求出强连通分量，我们需要首先引入时间戳的概念，即在搜索时给每个点一个编号。那么对于每个点需要定义两个时间戳：

- $\text{dfn}[u]$ 记录遍历到点 u 的时刻；
- $\text{low}[n]$ 记录从点 u 开始走所能遍历到的最长时间戳是什么。

换句话解释就是， $\text{low}[n]$ 记录的是点 u 包含其子树所有节点中能访问到的最早被访问到的节点的时间戳。

现在我们要做的是求出当前节点所在的强连通分量中的最高点（最早被访问到的节点），那么如果节点 u 是其所在的强连通分量的最高点，则会有： $\text{dfn}[u] == \text{low}[u]$ 。实现的结果如下：

```
void tarjan(int u) {
    dfn[u] = low[u] = ++time;
    // 将顶点 u 压入一个栈(stk)中并标记为在栈内
    stk[++top] = u; inStk[u] = true;
    // 循环遍历顶点 u 的所有邻接顶点。
    // h 数组是邻接表的头部数组,
    // next 是邻接表中的“下一个”数组
    for (int i = head[u]; ~i; i = next[u]) {
        // e 数组存储与顶点 u 相邻的顶点
```

```

int v = e[i];
// 节点 v 未被访问过
if (!dfn[v]) {
    tarjan(v);
    low[u] = min(low[u], low[v]);
    // 顶点 v 已经在栈中，意味着找到了一条后向边
} else if (inStk[v]) {
    low[u] = min(low[u], dfn[v]);
}
}

// 发现强连通分量的根节点
if (low[u] == dfn[u]) {
    int x;
    // 更新强连通分量编号
    ++sccCnt;
    do {
        x = stk[top--];
        inStk[x] = false;
        // 当前节点属于新的强连通分量
        id[x] = sccCnt;
    } while(x != u);
}
}

```

程序中，栈保存的元素可以简单理解为：栈中所有的点都不是其所在的强连通分量的最高点（当前尚未遍历完的强连通分量的所有点都被存储到栈中）。这个算法的时间复杂度为 $O(n + m)$ 。下面给出一个简单的缩点方法：

```

for (i = 1, i <= n; i++) {
    for (i 的所有邻点 j) {
        if (id[i] != id[j]) {
            // 加一条新边
            addEdge(id[i], id[j]);
        }
    }
}

```

```
    }  
}  
}
```

具体来看下面这道题：

例 11.3.8 (acwing-367 学校网络). 一些学校连接在一个计算机网络上，学校之间存在软件支援协议，每个学校都有它应支援的学校名单（学校 A 支援学校 B ，并不表示学校 B 一定要支援学校 A ）。

当某校获得一个新软件时，无论是直接获得还是通过网络获得，该校都应立即将这个软件通过网络传送给它应支援的学校。

因此，一个新软件若想让所有学校都能使用，只需将其提供给一些学校即可。

现在请问最少需要将一个新软件直接提供给多少个学校，才能使软件能够通过网络被传送到所有学校？

最少需要添加几条新的支援关系，使得将一个新软件提供给任何一个学校，其他所有学校就都可以通过网络获得该软件？

输入格式：

第 1 行包含整数 N ，表示学校数量。

第 $2 \cdots N+1$ 行，每行包含一个或多个整数，第 $i+1$ 行表示学校 i 应该支援的学校名单，每行最后都有一个 0 表示名单结束（只有一个 0 即表示该学校没有需要支援的学校）。

```
5  
2 4 3 0  
4 5 0  
0  
0  
1 0
```

输出格式：

输出两个问题的结果，每个结果占一行。

1

2

问题11.3.8是强连通分量的一个应用案例，在这个例子中，学校之间的网络可以被视为一个有向图，学校是顶点，支持关系是有向边。

题中有两个问题，第一个问题：要求最少需要直接提供给多少个学校新软件，以确保所有学校都能通过网络得到软件。这个问题实际上是在问我们需要在图中找到多少个顶点，使得从这些顶点出发，可以通过一系列有向边到达图中的所有其他顶点。这等价于**找到最小路径覆盖或者图的最小顶点集合**，这些顶点的出度为0(即没有其他学校它们需要支持的)。这些学校代表着强连通分量的“根”，如果我们在这些学校中提供软件，所有在该强连通分量内的学校都可以得到软件。

第二个问题：需要添加多少条新的支持关系，以确保无论向哪个学校提供新软件，其他所有学校都可以通过网络得到这个软件。这个问题等价于将原图转换成一个强连通图的最小边数。我们可以计算图中的强连通分量，然后建立一个缩点图，这是一个新的有向图，其中每个强连通分量是一个顶点。在缩点图中，我们需要找到一个方式将这些顶点连接起来，使得整个图成为强连通的。

为了解决这个问题，我们使用 Tarjan 算法来找到所有的强连通分量，然后分别解决这两个问题：

1. 计算如度为0的强连通分量的数量，这是第一个问题的答案。
2. 在缩点图中，找到入度为0和出度为0的顶点数量，我们需要至少添加较大者的边数，以便使图变得强连通，这是第二个问题的答案。

实现如下：

```
#include "bits/stdc++.h"
```

```
using namespace std;

const int N = 107, M = 10007;

int head[N], e[M], nextt[M], idx;
int dfn[N], low[N], timee;
int stk[N], top;
bool inStk[N];
int id[N], sccCnt;
// 统计出度入度
int dIn[N], dOut[N];

// 添加有向边
void add(int a, int b) {
    e[idx] = b, nextt[idx] = head[a], head[a] = idx++;
}

void tarjan(int u) {
    dfn[u] = low[u] = ++timee;
    stk[++top] = u, inStk[u] = true;

    for (int i = head[u]; ~i; i = nextt[i]) {
        int v = e[i];
        if (!dfn[v]) {
            tarjan(v);
            low[u] = min(low[u], low[v]);
        } else if (inStk[v]) {
            low[u] = min(low[u], dfn[v]);
        }
    }

    if (dfn[u] == low[u]) {
        ++sccCnt;
    }
}
```

```
int x;
do {
    x = stk[top--];
    inStk[x] = false;
    id[x] = sccCnt;
} while (x != u);
}

int main() {
    int n;
    cin >> n;
    memset(head, -1, sizeof head);
    for (int i = 1; i <= n; i++) {
        int t;
        while (cin >> t, t) {
            add(i, t);
        }
    }
    // tarjan 缩点
    for (int i = 1; i <= n; i++) {
        if (!dfn[i]) {
            tarjan(i);
        }
    }
    // 统计出度入度
    for (int i = 1; i <= n; i++)
        for (int j = head[i]; j != -1; j = nextt[j]) {
            int k = e[j];
            int a = id[i], b = id[k];
            // a和b不在一个强连通分量时
            if (a != b) {
                dOut[a]++;
            }
        }
}
```

```
        dIn[b]++;
    }

}

int a = 0, b = 0;
// 统计出度、入度为0的强连通分量
for (int i = 1; i <= sccCnt; i++) {
    if (!dIn[i]) a++;
    if (!dOut[i]) b++;
}

cout << a << "\n";
if (sccCnt == 1) {
    cout << "0" << "\n";
} else {
    cout << max(a, b) << "\n";
}

return 0;
}
```

其中add(int a, int b)函数实现了一个利用邻接表添加有向边的操作，我们具体来看：

```
void add(int a, int b) {
    // 在边数组e中记录边的目标顶点b
    e[idx] = b;
    // nextt数组记录这个边节点在邻接表中的下一个节点，即当前
    // 顶点a的邻接表头部的索引
    nextt[idx] = head[a];
    // head数组记录每个顶点的邻接表的起始边的索引，这里将当
    // 前边的索引idx设置为顶点a的邻接表头
    head[a] = idx++;
}
```

每次调用add函数时，都会执行以下步骤：

1. 将目标顶点 b 存储在边数组e的 idx 索引处；
2. 将当前顶点 a 的邻接表头部的索引 ($head[a]$) 存储在nextt数组的 idx 索引处；如果 $head[a]$ 是 -1 ，这意味着顶点 a 之前没有邻接顶点，那么 $nextt[idx]$ 也将是 -1 ；
3. 更新 $head[a]$ 为当前边的索引 idx ，这样 $head[a]$ 现在指向最新添加的边。
4. idx 自增，为下一次添加边做准备。

这种方法将所有边存储在一个数组中，并使用head数组和nextt数组来追踪每个顶点的邻接边。head数组的每个元素指向该顶点的邻接表中的第一条边的索引，而nextt数组则用于找到同一个顶点的邻接表中的下一条边。在遍历图的邻接表时，可以从 $head[u]$ 开始，然后连续访问 $nextt[i]$ ，直到它的值为 -1 ，即到达了邻接表的末尾。

此外，由于当下算法竞赛中允许使用vector来代替数组，通过使用可自增的vector实现邻接表可以大幅简化先前链式前向星，上述 Tarjan 模板的修改如下：

```
void tarjan(int u) {
    dfn[u] = low[u] = ++timee;
    stk[++top] = u, inStk[u] = true;
    for (int i = 0; i < edges[u].size(); i++) {
        int v = edges[u][i];
        if (!dfn[v]) {
            tarjan(v);
            low[u] = min(low[u], low[v]);
        } else if (inStk[v]) {
            low[u] = min(low[u], dfn[v]);
        }
    }
}
```

```
    }

    if (dfn[u] == low[u]) {
        ++sccCnt;
        int x;
        do {
            x = stk[top--];
            inStk[x] = false;
            id[x] = sccCnt;
        } while (x != u);
    }
}
```

例题11.3.8的程序修改如下：

```
#include "bits/stdc++.h"
using namespace std;

const int N = 107;
int dfn[N], low[N], timee;
int stk[N], top;
bool inStk[N];
int id[N], sccCnt;
int dIn[N], dOut[N];
vector<vector<int>> edges(N); // 使用 vector 存储图的边

// 添加有向边
void add(int a, int b) {
    edges[a].push_back(b);
}

void tarjan(int u) {
    dfn[u] = low[u] = ++timee;
    stk[++top] = u, inStk[u] = true;
```

```
for (int i = 0; i < edges[u].size(); i++) {
    int v = edges[u][i];
    if (!dfn[v]) {
        tarjan(v);
        low[u] = min(low[u], low[v]);
    } else if (inStk[v]) {
        low[u] = min(low[u], dfn[v]);
    }
}

if (dfn[u] == low[u]) {
    ++sccCnt;
    int x;
    do {
        x = stk[top--];
        inStk[x] = false;
        id[x] = sccCnt;
    } while (x != u);
}
}

int main() {
    int n;
    cin >> n;
    for (int i = 1; i <= n; i++) {
        int t;
        while (cin >> t, t) {
            add(i, t);
        }
    }
    // tarjan 缩点
    for (int i = 1; i <= n; i++) {
        if (!dfn[i]) {
```

```
        tarjan(i);
    }
}

// 统计出度入度
for (int i = 1; i <= n; i++) {
    for (int k : edges[i]) {
        int a = id[i], b = id[k];
        // a和b不在一个强连通分量时
        if (a != b) {
            dOut[a]++;
            dIn[b]++;
        }
    }
}

int a = 0, b = 0;
// 统计出度、入度为0的强连通分量
for (int i = 1; i <= sccCnt; i++) {
    if (!dIn[i]) a++;
    if (!dOut[i]) b++;
}

cout << a << "\n";
if (sccCnt == 1) {
    cout << "0" << "\n";
} else {
    cout << max(a, b) << "\n";
}

return 0;
}
```

11.3.2 无向图的双连通分量

双连通分量也被称为**重连通分量**，是一个针对无向图连通性的概念。不同于有向图只有一种强连通分量，无向图有**两种**双连通分量：

- **边双连通分量 (e-DCC)**: 极大的不含有桥的连通区域称为边连通分量；即任意两个点都有至少两条**不重复**的路径相连。
- **点双连通分量 (v-DCC)**: 极大的不含有割点的连通区域称为点连通分量；即任意两条边都在一个简单环中。

这里需要注意的是：在一个点双连通分量中，如果存在割点，那么这个割点一定属于两个连通分量。

边双连通分量

我们在用 Tarjan 算法求边双连通分量时同样需要连个参数：`dfn` 和 `low`，这里的 `dfn` 和 `low` 的含义与强连通分量中的含义相同。

为了找到双连通分量，即意味着我们需要找到存在的所有桥，那么如何判断一条边是否是桥呢？例如我们由点 x 经过一条边走到点 y ，如果我们不能通过其他路径由点 y 回到点 x ，那么这条边就是桥。即如果始终存在 $\text{low}[y] > \text{dfn}[x]$ ，则边 (x, y) 是桥。

我们的最终目的是找出所有的双连通分量，一种简单的方法是：找到所有的桥，然后将桥删除，剩下的就是双连通分量；此外，我们还可以用类似于有向图找强连通分量的方法使用用一个栈帮助记录双连通分量。

我们直接来看一道例题：

例 11.3.9 (acwing-395 冗余路径). 为了从 F 个草场中的一个走到另一个，奶牛们有时不得不路过一些她们讨厌的可怕的树。

奶牛们已经厌倦了被迫走某一条路，所以她们想建一些新路，使每一对草场之间都会至少有两条相互分离的路径，这样她们就有多一些选择。

每对草场之间已经有至少一条路径。

给出所有 R 条双向路的描述，每条路连接了两个不同的草场，请计算最少的新建道路的数量，路径由若干道路首尾相连而成。

两条路径相互分离，是指两条路径没有一条重合的道路。

但是，两条分离的路径上可以有一些相同的草场。

对于同一对草场之间，可能已经有两条不同的道路，你也可以在它们之间再建一条道路，作为另一条不同的道路。

输入格式：

第 1 行输入 F 和 R 。

接下来 R 行，每行输入两个整数，表示两个草场，它们之间有一条道路。

```
7 7  
1 2  
2 3  
3 4  
2 5  
4 5  
5 6  
5 7
```

输出格式：

输出一个整数，表示最少的需要新建的道路数。

```
2
```

例题11.3.9题目很长，但简单翻一下不难发现题目的要求是：给定一个无向图，求出最少需要添加多少条边，使得图中任意两点之间都至少有两条不重复的路径，即成为边双连通分量。这个问题与例题11.3.8类似，例题11.3.8要求能够将有向图改为强连通图，而这里要求将无向图改为边双连通图。

将有向图改为强连通图至少需要加 $\max\{p, q\}$ 条边，这里的 p 和 q 分别是出度为 0 和入度为 0 的强连通分量的个数；将无向图改为边双连通则至少需要加 $\lfloor \frac{cnt+1}{2} \rfloor$ 条边，这里的 cnt 是缩点之后度数为 1 的点的个数。

```
#include <bits/stdc++.h>

using namespace std;

const int N = 5070, M = 20010;

int h[N], e[M], ne[M], idx;
int dfn[N], low[N], timestamp;
int stk[N], top;
int id[N], dcc_cnt;
bool is_bridge[M];
int d[N]; // 存储每个连通分量的度

void add(int a, int b) {
    e[idx] = b, ne[idx] = h[a], h[a] = idx++;
}

void tarjan(int u, int from) {
    dfn[u] = low[u] = ++timestamp;
    stk[++top] = u;

    for (int i = h[u]; ~i; i = ne[i]) {
        int j = e[i];
        if (!dfn[j]) {
            tarjan(j, i);
            low[u] = min(low[u], low[j]);
            if (low[j] > dfn[u]) {
                // 如果子节点的 low 值大于 u 的 dfn 值，说明这条边是桥
                // is_bridge[i] = is_bridge[i ^ 1] = true;
            }
        }
    }
}
```

```
    is_bridge[i] = true;
    if (i % 2 == 0) {
        is_bridge[i + 1] = true; // 如果 i 是偶数，则 i
        +1 是与之配对的边
    } else {
        is_bridge[i - 1] = true; // 如果 i 是奇数，则 i
        -1 是与之配对的边
    }
}
} else if (i != (from ^ 1)) {
    low[u] = min(low[u], dfn[j]);
}
}
if (dfn[u] == low[u]) {
    ++dcc_cnt;
    int y;
    do {
        y = stk[top--];
        id[y] = dcc_cnt;
    } while (y != u);
}
}

int main() {
    int n, m;
    cin >> n >> m;
    memset(h, -1, sizeof(h));
    while (m--) {
        int a, b;
        cin >> a >> b;
        add(a, b), add(b, a);
    }
    tarjan(1, -1);
```

```

for (int i = 0; i < idx; i++) {
    if (is_bridge[i]) {
        d[id[e[i]]]++;
    }
}

int cnt = 0;
for (int i = 1; i <= dcc_cnt; i++) {
    if (d[i] == 1) {
        cnt++;
    }
}

cout << (cnt + 1) / 2 << endl;
return 0;
}

```

在无向图求解双连通分量时，最核心的目的是找出图中存在的所有桥，那么如果在使用邻接表时需要存储边的编号，以便直接将桥标记出来。因此我们需要用一个结构体数组来存储边的信息：

```

struct node {
    int to, idx;
};

vector<vector<node>> edges(N);

```

并在此基础上修改add函数：

```

void add(int a, int b) {
    edges[a].push_back({b, idx++});
}

```

在tarjan函数中，我们需要修改for循环中的遍历方式：

```
void tarjan(int u, int fromEdge) {
    dfn[u] = low[u] = ++timestamp;
    stk[++top] = u;

    for (auto edge : edges[u]) {
        int j = edge.to, edge_index = edge.idx;
        if (!dfn[j]) {
            tarjan(j, edge_index);
            low[u] = min(low[u], low[j]);
            if (low[j] > dfn[u]) {
                is_bridge[edge_index] = true;
            }
        } else if (edge_index != (fromEdge ^ 1)) {
            low[u] = min(low[u], dfn[j]);
        }
    }

    if (dfn[u] == low[u]) {
        ++dcc_cnt;
        int x;
        do {
            x = stk[top--];
            id[x] = dcc_cnt;
        } while (x != u);
    }
}
```

至此我们可以完成对 Tarjan 算法的修改，这里给出修改后的程序：

```
#include <bits/stdc++.h>

using namespace std;

const int N = 5070, M = 20010;
```

```
struct node {
    int to, idx;
};

int idx;
int dfn[N], low[N], timestamp;
int stk[N], top;
int id[N], dcc_cnt;
bool is_bridge[M];
int d[N]; // 存储每个连通分量的度
vector<vector<node>> edges(N);

void add(int a, int b) {
    edges[a].push_back({b, idx++});
}

void tarjan(int u, int fromEdge) {
    dfn[u] = low[u] = ++timestamp;
    stk[++top] = u;

    for (auto edge : edges[u]) {
        int j = edge.to, edge_index = edge.idx;
        if (!dfn[j]) {
            tarjan(j, edge_index);
            low[u] = min(low[u], low[j]);
            if (low[j] > dfn[u]) {
                is_bridge[edge_index] = true;
            }
        } else if (edge_index != (fromEdge ^ 1)) {
            low[u] = min(low[u], dfn[j]);
        }
    }
}
```

```
if (dfn[u] == low[u]) {
    ++dcc_cnt;
    int x;
    do {
        x = stk[top--];
        id[x] = dcc_cnt;
    } while (x != u);
}

int main() {
    int n, m;
    cin >> n >> m;
    while (m--) {
        int a, b;
        cin >> a >> b;
        add(a, b), add(b, a);
    }

    tarjan(1, -1);

    for (int i = 0; i < edges.size(); i++) {
        for (auto edge : edges[i]) {
            if (is_bridge[edge.idx]) {
                d[id[i]]++;
                d[id[edge.to]]++;
            }
        }
    }

    int cnt = 0;
    for (int i = 1; i <= dcc_cnt; i++) {
        if (d[i] == 1) {
```

```
    cnt++;
}

}

cout << (cnt + 1) / 2 << endl;

return 0;
}
```

值得注意的是，由于`vector`嵌套存储边时并没有边的编号，这将影响到我们在`tarjan`函数中判断桥的条件，以及在更新点`u`的`low[u]`值时需要判断这两条边是否是同一条边（考虑到无向图存储时的特殊性）。而链式前向星的存储包含边的序号，在求解边双连通分量的时候各有优势，如何使用请自行判断。

点双连通分量

点双连通分量同样需要我们保存`dfn`和`low`数组，此时我们的核心目的是判断某个点是否是割点。

我们考虑一个情况：有两个点 x 和 y ，且两点通过一条边相连，如果始终有 $\text{low}[y] \geq \text{dfn}[x]$ ，那么会有两种情况：

1. 若点 x 不是根节点，则点 x 是割点；
2. 若点 x 是根节点，且至少要有两个这样的 y 子节点，并有 $\text{low}[y_i] \geq \text{dfn}[x]$ ，则 x 是割点。

针对割点的问题，我们先看一个模板题：

例 11.3.10 (luogu-3388 【模板】割点). 给出一个 n 个点， m 条边的无向图，求图的割点。

输入格式：

第一行输入两个正整数 n, m 。

下面 m 行每行输入两个正整数 x, y 表示 x 到 y 有一条边。

```
6 7
1 2
1 3
1 4
2 5
3 5
4 5
5 6
```

输出格式：

第一行输出割点个数。

第二行按照节点编号从小到大输出节点，用空格隔开。

```
1
5
```

例题11.3.10明确要求我们求出无向图中存在哪些割点。为了解决这个问题，我们除开常规的变量定义外，我们需要一个**root**变量来记录每个点的根节点，同时由于我们不需要记录每个点属于哪个连通分量，所以这里并不需要**stack**。

```
int dnf[N], low[N], timestamp;
int root;
set<int> s; // 存储割点
```

同时修改**tarjan**函数，使其能够判断割点：

```
void tarjan(int u) {
    dnf[u] = low[u] = ++timestamp;
    int son = 0;
    for (int i = h[u]; ~i; i = ne[i]) {
        int j = e[i];
```

```
if (!dfn[j]) {
    tarjan(j);
    low[u] = min(low[u], low[j]);
    // 如果 u 不是根节点，且 low[j] >= dfn[u]，则 u 是割点
    if (low[j] >= dfn[u]) {
        son++; // 记录子树个数
        if (u != root || son > 1) {
            s.insert(u);
        }
    }
} else {
    low[u] = min(low[u], dfn[j]);
}
}
```

到这里，我们给出本题的解决方案：

```
#include <bits/stdc++.h>

using namespace std;

const int N = 5e5 + 10, M = 4e6 + 10;

int h[N], e[M], ne[M], idx;
int dfn[N], low[N], timestamp;
int root;
set<int> s; // 存储割点

void add(int a, int b) {
    e[idx] = b, ne[idx] = h[a], h[a] = idx++;
}
```

```
void tarjan(int u) {
    dfn[u] = low[u] = ++timestamp;
    int son = 0;
    for (int i = h[u]; ~i; i = ne[i]) {
        int j = e[i];
        if (!dfn[j]) {
            tarjan(j);
            low[u] = min(low[u], low[j]);
            // 如果u不是根节点，且low[j] >= dfn[u]，则u是割点
            // 如果u是根节点，且有两个以上的子树，则u是割点
            if (low[j] >= dfn[u]) {
                son++; // 记录子树个数
                if (u != root || son > 1) {
                    s.insert(u);
                }
            }
        } else {
            low[u] = min(low[u], dfn[j]);
        }
    }
}

int main() {
    int n, m;
    ios_base::sync_with_stdio(false);
    cin >> n >> m;
    // 链式前向星初始化
    memset(h, -1, sizeof h);
    while (m--) {
        int a, b;
        cin >> a >> b;
        add(a, b), add(b, a);
    }
}
```

```
}

for (root = 1; root <= n; root++) {
    if (!dfn[root]) {
        tarjan(root);
    }
}

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

for (auto i : s) {
    cout << i << " ";
}
return 0;
}
```

到此为止我们给出了一个求割点的方法，但我们最终的目的是求出所有的点双连通分量，同样类似于边双连通分量或是强连通分量，我们都需要用一个栈来辅助，不同点在于什么时候出栈。点双连通分量的弹出时机是：

```
if (x 是孤立点) {
    // x是点双连通分量
}

// 当我们从点x搜向点y
// ...
if (dfn(x) <= low(y)) {
    son++;
    if (x != root || son > 1) {
        // x是割点
        // 将栈中元素出栈，直到y
        // 且x也属于该点双连通分量
    }
}
```

我们还是来看一个实际的题目：

例 11.3.11 (acwing-396 矿场搭建). 煤矿工地可以看成是由隧道连接挖煤点组成的无向图。

为安全起见，希望在工地发生事故时所有挖煤点的工人都能有一条出路逃到救援出口处。

于是矿主决定在某些挖煤点设立救援出口，使得无论哪一个挖煤点坍塌之后，其他挖煤点的工人都有一条道路通向救援出口。

请写一个程序，用来计算至少需要设置几个救援出口，以及不同最少救援出口的设置方案总数。

输入格式：

输入文件有若干组数据，每组数据的第一行是一个正整数 N ，表示工地的隧道数。

接下来的 N 行每行是用空格隔开的两个整数 S 和 T ($S \neq T$)，表示挖煤点 S 与挖煤点 T 由隧道直接连接。

注意，每组数据的挖煤点的编号为 $1 \sim Max$ ，其中 Max 表示由隧道连接的挖煤点中，编号最大的挖煤点的编号，可能存在没有被隧道连接的挖煤点。

输入数据以 0 结尾。

```
9
1 3
4 1
3 5
1 2
2 6
1 5
6 3
1 6
3 2
```

```
6  
1 2  
1 3  
2 4  
2 5  
3 6  
3 7  
0
```

输出格式:

每组数据输出结果占一行。

其中第 i 行以 Case i: 开始（注意大小写，Case 与 i 之间有空格，i 与：之间无空格，：之后有空格）。

其后是用空格隔开的两个正整数，第一个正整数表示对于第 i 组输入数据至少需要设置几个救援出口，第二个正整数表示对于第 i 组输入数据不同最少救援出口的设置方案总数。

输入数据保证答案小于 2^{64} ，输出格式参照以下输入输出样例。

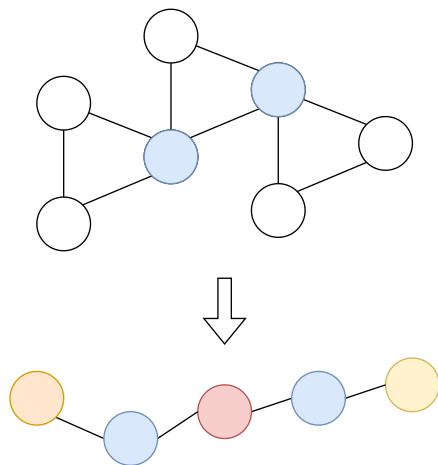
```
Case 1: 2 4  
Case 2: 4 1
```

例题11.3.11同样题目很长，我们将它翻译成人话即：给定一个无向图，问最少在几个点上设置出口，可以使得不管哪个点坍塌，其余所有点都可以与某个出口连通。

那么分析问题：首先我们需要设置出口数量至少为 2，此外，由于图之间未必连通，我们可以分别去看每个连通块：

1. 如果连通块内无割点，意味着删去某个点，图依然连通，则可以任选两点作为出口；
2. 如果连通块内有割点，则首先需要先进行缩点处理，缩点后将每个割点单独视为一个点，随后从每个 v-DCC 出发，向其包含的每个割点

连一条边。



接下来就需要关注每个点双连通分量的度，如果某个 $v - \text{DCC}$ 的度为 1，则需要在其内部（非割点）选择一个点作为出口；如果某个 $v - \text{DCC}$ 的度大于 1，则无需设置出口。

为了解决这个问题，我们需要定义一个栈，这里与之前的处理方案一样：

```
int stk[N], top;
```

同时，我们需要将 Tarjan 算法进行修改：

```
void tarjan(int u) {
    dfn[u] = low[u] = ++timestamp;
    stk[++top] = u;
    // 该点是孤立点
    if (u == root && h[u] == -1) {
        dcc_cnt++;
        dcc[dcc_cnt].push_back(u);
        return;
    }
    // 该点不是孤立点
    int son = 0;
```

```

for (int i = h[u]; ~i; i = ne[i]) {
    int j = e[i];
    if (!dfn[j]) {
        tarjan(j);
        low[u] = min(low[u], low[j]);
        if (low[j] >= dfn[u]) {
            son++;
            if (u != root || son > 1) {
                cut[u] = true;
            }
            dcc_cnt++;
        }
        int x;
        /**
         * 循环出栈，考虑到一个割点同时属于两个点双
         * 连通分量
         * 当前最高点 u 还要被用在更高的包含 u 的连
         * 通块中
         */
        do {
            x = stk[top--];
            dcc[dcc_cnt].push_back(x);
        } while (x != j);
        dcc[dcc_cnt].push_back(u);
    }
} else {
    low[u] = min(low[u], dfn[j]);
}
}
}

```

这里给出问题的解决方案：

```

#include "bits/stdc++.h"
#define ll long long

```

```
using namespace std;

const int N = 1007, M = 507;

int h[N], e[M], ne[M], idx;
int dfn[N], low[N], timestamp;
int stk[N], top;
int dcc_cnt;
vector<int> dcc[N];
bool cut[N];
int root;

void add(int a, int b) {
    e[idx] = b, ne[idx] = h[a], h[a] = idx++;
}

void tarjan(int u) {
    dfn[u] = low[u] = ++timestamp;
    stk[++top] = u;
    // 判断孤立点
    if (u == root && h[u] == -1) {
        dcc_cnt++;
        dcc[dcc_cnt].push_back(u);
        return;
    }

    int son = 0;
    for (int i = h[u]; ~i; i = ne[i]) {
        int j = e[i];
        if (!dfn[j]) {
            tarjan(j);
            low[u] = min(low[u], low[j]);
        }
    }
}
```

```
    if (low[j] >= dfn[u]) {
        son++;
        if (u != root || son > 1) {
            cut[u] = true;
        }
        dcc_cnt++;
        int x;
        do {
            x = stk[top--];
            dcc[dcc_cnt].push_back(x);
        } while (x != j);
        dcc[dcc_cnt].push_back(u);
    }
} else {
    low[u] = min(low[u], dfn[j]);
}
}

int main() {
    int T = 1;
    int n, m;
    while (cin >> m, m) {
        for (int i = 1; i <= dcc_cnt; i++) {
            dcc[i].clear();
        }
        idx = n = timestamp = top = dcc_cnt = 0;
        memset(h, -1, sizeof(h));
        memset(dfn, 0, sizeof(dfn));
        memset(cut, 0, sizeof(cut));

        while (m--) {
            int a, b;
```

```
    cin >> a >> b;
    n = max(n, b), n = max(n, a);
    add(a, b), add(b, a);
}

for (root = 1; root <= n; root++) {
    if (!dfn[root]) {
        tarjan(root);
    }
}

int res = 0;
ll num = 1;
for (int i = 1; i <= dcc_cnt; i++) {
    int cnt = 0;
    for (int j = 0; j < dcc[i].size(); j++) {
        if (cut[dcc[i][j]]) {
            cnt++;
        }
    }
    if (cnt == 0) {
        if (dcc[i].size() > 1) {
            res += 2;
            num *= dcc[i].size() * (dcc[i].size() - 1) /
                2;
        } else {
            res++;
        }
    } else if (cnt == 1) {
        res++;
        num *= dcc[i].size() - 1;
    }
}
```

```
    cout << "Case " << T++ << ":" " << res << " " << num <<
        endl;
}
return 0;
}
```

11.4 负环问题

先前我们强调过，Dijkstra 算法基于贪心思想，在求解最短路问题时不能处理存在负环的情况。那么如果出现负环，则要用Bellman-Ford或是SPFA 算法。判断负环的方法有两种：

1. 统计每个点的入队次数，如果有一个点入队 n 次，则说明存在负环，这也是 Bellman-Ford 算法的实现过程；
2. 统计当前每个点的最短路中所包含的边数，如果某点的最短路所包含的边数大于等于 n ，则也说明存在环，这是 SPFA 算法。

目前比较推荐使用 SPFA 算法来解决负环问题。首先通过一个题目来复习一下 SPFA 算法：

例 11.4.1 (acwing-904 虫洞). 农夫约翰在巡视他的众多农场时，发现了很多令人惊叹的虫洞。虫洞非常奇特，它可以看作是一条 **单向** 路径，通过它可以使你回到过去的某个时刻（相对于你进入虫洞之前）。

农夫约翰的每个农场中包含 N 片田地， M 条路径（双向）以及 W 个虫洞。

现在农夫约翰希望能够从农场中的某片田地出发，经过一些路径和虫洞回到过去，并在他的出发时刻之前赶到他的出发地。他希望能够看到出发之前的自己，请你判断一下约翰能否做到这一点。

下面我们将给你提供约翰拥有的农场所数量 F , 以及每个农场所的完整信息。

已知走过任何一条路径所花费的时间都不超过 10000 秒, 任何虫洞将他带回的时间都不会超过 10000 秒。

输入格式:

第一行包含整数 F , 表示约翰共有 F 个农场所。

对于每个农场所, 第一行包含三个整数 N, M, W 。

接下来 M 行, 每行包含三个整数 S, E, T , 表示田地 S 和 E 之间存在一条路径, 经过这条路径所花的时间为 T 。

再接下来 W 行, 每行包含三个整数 S, E, T , 表示存在一条从田地 S 走到田地 E 的虫洞, 走过这条虫洞, 可以回到 T 秒之前。

```
2
3 3 1
1 2 2
1 3 4
2 3 1
3 1 3
3 2 1
1 2 3
2 3 4
3 1 8
```

输出格式:

输出共 F 行, 每行输出一个结果。

如果约翰能够在出发时刻之前回到出发地, 则输出 YES, 否则输出 NO。

```
NO
YES
```

例题11.4.1实则是一个求负环的问题。我们可以用 SPFA 算法来解决这个问题:

```
#include "bits/stdc++.h"

using namespace std;

const int N = 510, M = 5210;

int h[N], e[M], w[M], ne[M], idx;
int dist[N];
int cnt[N];
bool st[N];
int n, m1, m2;

void add(int a, int b, int c) {
    e[idx] = b;
    w[idx] = c;
    ne[idx] = h[a];
    h[a] = idx++;
}

bool spfa() {
    memset(dist, 0, sizeof dist);
    memset(cnt, 0, sizeof cnt);
    memset(st, 0, sizeof st);

    queue<int> q;
    for (int i = 1; i <= n; i++) {
        q.push(i);
        st[i] = true;
    }

    while (q.size()) {
        int t = q.front();
```

```
q.pop();
st[t] = false;

for (int i = h[t]; ~i; i = ne[i]) {
    int j = e[i];
    if (dist[j] > dist[t] + w[i]) {
        dist[j] = dist[t] + w[i];
        cnt[j] = cnt[t] + 1;
        if (cnt[j] >= n) return true;
        if (!st[j]) {
            q.push(j);
            st[j] = true;
        }
    }
}
return false;
}

int main() {
    int T;
    scanf("%d", &T);

    while (T--) {
        cin >> n >> m1 >> m2;
        memset(h, -1, sizeof h);
        idx = 0;
        for (int i = 0; i < m1; i++) {
            int a, b, c;
            cin >> a >> b >> c;
            add(a, b, c), add(b, a, c);
        }
        for (int i = 0; i < m2; i++) {
```

```

        int a, b, c;
        cin >> a >> b >> c;
        add(a, b, -c);
    }

    if (spfa()) {
        cout << "YES" << endl;
    } else {
        cout << "NO" << endl;
    }
}

return 0;
}

```

程序中我们在第二次添加边的时候，由于是虫洞，添加时就可以视为添加一个负的边：

```

for (int i = 0; i < m2; i++) {
    // 虫洞 回到 t 秒前 时间 time-t 秒
    // 单向负边
    int a, b, c;
    cin >> a >> b >> c;
    add(a, b, -c);
}

```

那么分析问题，不难发现我们只要能够经过某个环使得耗时为负（即环的总长度为负），我们就能够看到出发之前的自己。所以我们来看核心的 SPFA 算法：

```

bool spfa() {
    // 初始化所有的距离，一开始都是 0
    memset(dist, 0, sizeof dist);
    // 最短路的边数

```

```
memset(cnt, 0, sizeof cnt);
// 判重数组
memset(st, 0, sizeof st);

queue<int> q;
// 所有点入队
for (int i = 1; i <= n; i++) {
    q.push(i);
    st[i] = true;
}

// 只要队列不空，每次取出队头元素
while (q.size()) {
    int t = q.front();
    q.pop();
    // 出队标记
    st[t] = false;

    for (int i = h[t]; ~i; i = ne[i]) {
        int j = e[i];
        // 松弛操作
        if (dist[j] > dist[t] + w[i]) {
            dist[j] = dist[t] + w[i];
            // 最短路的边数 + 1
            cnt[j] = cnt[t] + 1; // 从t到j多了一条边w[t][j]
        }
        // 如果边数大于n，即存在环，返回true
        if (cnt[j] >= n) {
            return true;
        }
        // 如果j没有入队，入队
        if (!st[j]) {
            q.push(j);
            st[j] = true;
        }
    }
}
```

```
        }
    }
}

// 整个更新完都没有发现负环，则返回 false
return false;
}
```

此外，负环问题常与 01 分数规划类型的问题联合，例如：

例 11.4.2 (acwing-361 观光奶牛). 给定一张 L 个点、 P 条边的有向图，每个点都有一个权值 $f[i]$ ，每条边都有一个权值 $t[i]$ 。

求图中的一个环，使“环上各点的权值之和”除以“环上各边的权值之和”最大。

输出这个最大值。

输入格式:

第一行包含两个整数 L 和 P 。

接下来 L 行每行一个整数，表示 $f[i]$ 。

再接下来 P 行，每行三个整数 $a\ b\ t[i]$ ，表示点 a 和 b 之间存在一条边，边的权值为 $t[i]$ 。

```
5 7
30
10
10
5
10
1 2 3
2 3 2
3 4 5
3 5 2
4 5 5
```

5 1 3
5 2 2

输出格式:

输出一个数表示结果，保留两位小数。

6.00

例题11.4.2不长，我们来简单翻译一下：让我们找到一个环，并计算这个环中所有的点权的和 F 和边权 T ，我们需要使得这个 $\frac{F}{T}$ 的值最大。那么所有形如这个类型的问题都被称为 **01 分数规划问题**。

这类问题有一个通用的做法，就是用二分法来解决。就这个问题来看，由于所有点的权重都是大于 0 的，那么我们可以合理推测最终的 $\frac{F}{T} > 0$ ，另一方面，如果要让这些值尽可能大，我们都会将点的权重设置为 1000(即题中给的上限)，边权设置为 1(即题中给的下限)，这样我们得到的最大值就是 $\frac{1000}{1} = 1000$ ，所以我们的答案必然在 $(1, 1000)$ 之间。

紧接着我们需要二分这个区间找到答案，令 $\text{mid} = (0 + 1000) / 2$:

- 若 $\frac{F}{T} \geq \text{mid}$ ，则说明答案在 $[\text{mid}, 1000]$ 之间；
- 若 $\frac{F}{T} < \text{mid}$ ，则说明答案在 $[0, \text{mid}]$ 之间。

此外，由于这个题目既有点权又有边权，我们可以把点权放到边上，由于我

们需要使得 $\frac{F=\sum_i f_i}{T=\sum_i t_i}$ 最大，又有 $F, T > 0$ ，所以我们可以进行转化，例如：

$$\begin{aligned}\frac{\sum_i f_i}{\sum_i t_i} &> \text{mid} \\ \sum_i f_i &> \text{mid} \cdot \sum_i t_i \\ \sum_i f_i - \text{mid} \cdot \sum_i t_i &> 0 \\ \sum_i (f_i - \text{mid} \cdot t_i) &> 0\end{aligned}$$

那么在记录每条边的边权时，我们就可以记录边权为 $f_i - \text{mid} \cdot t_i$ ，那么问题就变成了考察图中是否存在正环 ($\sum_i (f_i - \text{mid} \cdot t_i) > 0$)。

我们给出本题的解决方案：

```
#include <bits/stdc++.h>

using namespace std;

const int N = 1010, M = 5010;
int h[N], e[M], ne[M], w[M], idx;
int f[N];
double dis[N];
int cnt[N], q[N];
bool st[N];
int n, m;

void add(int x, int y, int d) {
    e[idx] = y, ne[idx] = h[x], w[idx] = d, h[x] = idx++;
}

bool check(double mid) {
    // 距离多少不影响最终结果，所以距离可以不初始化
    memset(dis, 0, sizeof(dis));
    memset(cnt, 0, sizeof(cnt));
    // 循环队列
```

```
int tt = 0, hh = 0;

for (int i = 1; i <= n; ++i) st[i] = true, q[tt++] = i;
while (hh != tt) {
    int x = q[hh++];
    if (hh == N) hh = 0;
    st[x] = false;
    for (int i = h[x]; i != -1; i = ne[i]) {
        int y = e[i];
        if (dis[y] < dis[x] + f[x] - mid * w[i]) {
            dis[y] = dis[x] + f[x] - mid * w[i];
            cnt[y] = cnt[x] + 1;
            // 边数大于等于n, 存在负环
            if (cnt[y] >= n) {
                return true;
            }
            if (!st[y]) {
                st[y] = true;
                q[tt++] = y;
                if (tt == N) {
                    tt = 0;
                }
            }
        }
    }
}
return false;
}

int main() {
    memset(h, -1, sizeof(h));
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= n; ++i) {
```

```
    scanf("%d", &f[i]);
}

int s, e, d;
for (int i = 0; i < m; ++i) {
    scanf("%d%d%d", &s, &e, &d);
    add(s, e, d);
}

double l = 0, r = 1000;
// 如果两位小数，保留到  $1e-4$ ，一般多两位小数
while (r - l > 1e-4) {
    double mid = (l + r) / 2;
    // 放大区间
    if (check(mid)) {
        l = mid;
    } else { // 否则缩小区间
        r = mid;
    }
}

printf("%.2lf\n", r);
return 0;
}
```

11.5 差分约束系统

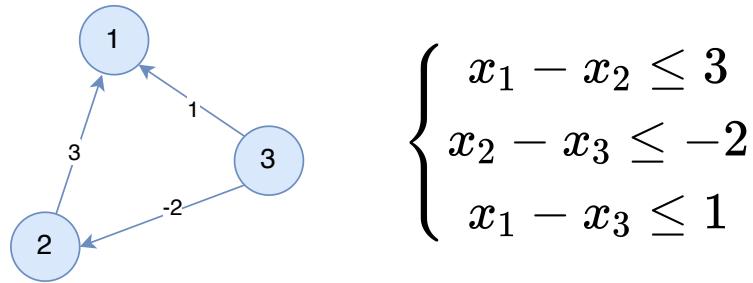
差分约束系统 (System of Differential Equations) 问题是对一组不等式求解的问题，通常在算法竞赛中，给出一系列的不等关系情况，就可以尝试把它们转化为差分约束系统来解决。

差分约束系统的定义具体如下所示，其中， y_i 是已知的常数，差分约束的每个不等式称为一个约束条件，都是两个未知量之差小于或等于某个

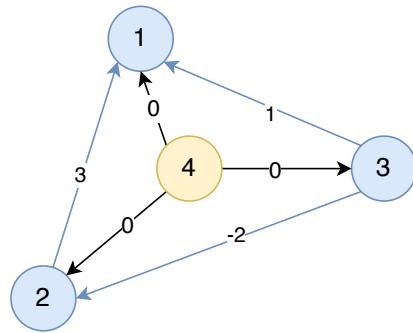
常数。

$$\begin{cases} x_{c_1} - x_{c'_1} \leq y_1 \\ x_{c_2} - x_{c'_2} \leq y_2 \\ \dots \\ x_{c_n} - x_{c'_n} \leq y_n \end{cases}$$

为了解决这个不等式，我们需要一些转换，将 $x - y \leq c$ 的形式移项得到 $x \leq y + c$ ，观察这个不等式，不难发现这个不等式类似于图论问题中的最短路 $\text{dist}[u] \leq \text{dist}[v] + \omega$ ，也就是说对于每对点 c_i 和 c'_i ，我们都可以建立一条边 $c'_i \rightarrow c_i$ ，边权为 y_i 。这样建出的有向图，它的每个顶点都对应差分约束系统中的一个未知量，源点到每个顶点的最短路对应这些未知量的值，而每条边对应一个约束条件。



最短路问题需要有一个源点，在差分约束系统中实际上取哪个点作为源点并无影响，只是有时候我们得到的图并不连通，这样会导致结果出现 INF，为了避免这种情况，通常的解决方法是人为添加一个人工的源点，通过这个人工源点解决图不连通的问题。



现在我们就以 $n + 1$ 号点为源点求各点的最短路。需要注意的是，添加了人工源点之后也添加了约束条件：

$$\begin{cases} x_1 - x_2 \leq 3 \\ x_2 - x_3 \leq -2 \\ x_1 - x_3 \leq 1 \\ x_1 - x_4 \leq 0 \\ x_2 - x_4 \leq 0 \\ x_3 - x_4 \leq 0 \end{cases}$$

原问题只需要求出人工源点到各个节点的最短路径即可，因为这求出的只是一组解，通过简单的数学计算可知，在符合差分约束系统的一组解上加上或减去同一个数，得到的解同样符合原系统。

具体如这道模板题：

例 11.5.1 (luogu-5960 【模板】差分约束). 给出一组包含 m 个不等式，有 n 个未知数的不等式组，求任意一组满足这个不等式组的解。形如：

$$\begin{cases} x_{c_1} - x_{c'_1} \leq y_1 \\ x_{c_2} - x_{c'_2} \leq y_2 \\ \dots \\ x_{c_m} - x_{c'_m} \leq y_m \end{cases}$$

输入格式:

第一行为两个正整数 n, m , 代表未知数的数量和不等式的数量。

接下来 m 行, 每行包含三个整数 c, c', y , 代表一个不等式 $x_c - x_{c'} \leq y$ 。

```
3 3
1 2 3
2 3 -2
1 3 1
```

输出格式:

一行, n 个数, 表示 $x_1, x_2 \dots x_n$ 的一组可行解, 如果有多组解, 请输出任意一组, 无解请输出NO。

```
0 -2 0
```

样例解释:

一种可行的方法是 $x_1 = 0, x_2 = -2, x_3 = 0$ 。

$$\begin{cases} 0 - (-2) = 2 \leq 3 \\ -2 - 0 = -2 \leq -2 \\ 0 - 0 = 0 \leq 1 \end{cases}$$

例题11.5.1即是上文给出的样例, 分析刚刚已经给出, 这里直接来看解决方案:

```
#include <bits/stdc++.h>

using namespace std;

const int MAXN = 5007, MAXM = 10007;
int cnt_edge, head[MAXN];
bool inQueue[MAXN];
int cnt[MAXN], dis[MAXN];
queue<int> Q;
```

```
struct {
    int to, next, w;
} edges[MAXM];

int read() {
    int ans = 0, sgn = 1;
    char c = getchar();
    while (!isdigit(c)) {
        if (c == '-') {
            sgn *= -1;
        }
        c = getchar();
    }
    while (isdigit(c)) {
        ans = ans * 10 + c - '0';
        c = getchar();
    }
    return ans * sgn;
}

void add_edge(int from, int to, int w) {
    edges[++cnt_edge].next = head[from];
    edges[cnt_edge].to = to;
    edges[cnt_edge].w = w;
    head[from] = cnt_edge;
}

bool SPFA(int s, int n) {
    memset(dis, 127, sizeof(dis));
    dis[s] = 0;
    Q.push(s);
    while (!Q.empty()) {
        int p = Q.front();
```

```
if (cnt[p] > n) {
    return false;
}
Q.pop();
inQueue[p] = false;
for (int eg = head[p]; eg != 0; eg = edges[eg].next) {
    int to = edges[eg].to;
    if (edges[eg].w + dis[p] < dis[to]) {
        dis[to] = edges[eg].w + dis[p];
        if (!inQueue[to]) {
            Q.push(to);
            inQueue[to] = true;
            cnt[to]++;
        }
    }
}
return true;
}

int main() {
    int n = read(), m = read();
    for (int i = 0; i < m; ++i) {
        int u = read(), v = read(), w = read();
        add_edge(v, u, w);
    }
    for (int i = 1; i <= n; ++i) {
        add_edge(n + 1, i, 0);
    }
    if (SPFA(n + 1, n)) {
        for (int i = 1; i <= n; ++i) {
            printf("%d ", dis[i]);
        }
    }
}
```

```

} else {
    puts("NO");
}
return 0;
}

```

考虑到 0 号点可能被题目使用，因此我们构建人工节点选用的是 $n + 1$ 号节点：

```

for (int i = 1; i <= n; ++i) {
    add_edge(n + 1, i, 0);
}

```

随后我们选用 SPFA 算法解决这个问题，选择 SPFA 算法的理由是，图中如果存在负环，则代表没有最短路，也就是所谓无解的情况：

```

if (SPFA(n + 1, n)) {
    for (int i = 1; i <= n; ++i) {
        printf("%d ", dis[i]);
    }
} else {
    puts("NO");
}

```

SPFA 算法的实现之前已经提到过，这里不再赘述。至此，我们总结一下差分约束系统的解决方案：

1. **构建差分约束图**：将差分约束系统中的每个不等式转化为一条边。
2. **确定起始点**：建立人工节点 $n + 1$ ，并连接到其他所有点，距离为 0。
3. 运行 SPFA 算法，只要能跑出来就一定有解，否则就代表图中存在负环，即无解。

此外，实际问题中的不等式可能会有一些特殊的情况，有些情况可以转化为差分约束需要的情况，例如：

- $x_1 - x_2 \geq y$ 可以同乘 -1 转化为 $x_2 - x_1 \leq -y$;
- $x_1 - x_2 = y$ 转化为 $x_1 - x_2 \leq y$ 和 $x_2 - x_1 \leq y$;
- $x_1 - x_2 < y$, 在整数的情况下可以转化为 $x_1 - x_2 \leq y - 1$;
- $x_1 - x_2 > y$, 在整数的情况下可以转化为 $x_2 - x_1 \leq -y - 1$ 。

同时,有一些题目中还可能隐含一些不等关系,例如下面这道题:

例 11.5.2 (luogu-1250 种树). 路边的地区被分割成块,并被编号成 $1, \dots, n$ 。每个部分为一个单位尺寸大小并最多可种一棵树。

每个居民都想在门前种些树,并指定了三个号码 b, e, t 。这三个数表示该居民想在地区 b 和 e 之间(包括 b 和 e)种至少 t 棵树。

居民们想种树的各自区域可以交叉。你的任务是求出能满足所有要求的最少的树的数量。

输入格式:

输入的第 1 行是一个整数,代表区域的个数 n 。

输入的第 2 行是一个整数,代表房子个数 h 。

第 3 到第 $h + 2$ 行,每行三个整数,第 $i + 2$ 行的整数依次为 b_i, e_i, t_i ,代表第 i 个居民想在 b_i 和 e_i 之间种至少 t_i 棵树。

```
9
4
1 4 2
4 6 2
8 9 2
3 5 2
```

输出格式:

输出一行一个整数,代表最少的树木个数。

```
5
```

对于例题11.5.2来说，如果以每一块地区上树的数量为变量问题将比较棘手，因此我们选择使用前缀和，令前缀和为 $S[i]$ ，那么题目给出的条件就可以被转化为 $S[E] - S[B - 1] \geq T$ ，此外还需要注意到题目要求每一块上只能种一棵树，所以我们有 $0 \leq S[i + 1] - S[i] \leq 1$ 。

随后我们建立人工源点 s ，将其他所有点连接到 s ，下面是一个可行的解决方案：

```
#include <bits/stdc++.h>

using namespace std;

const int MAXN = 30005, MAXM = 5005;
int n, m, s, cnt_edge, head[MAXN];
bool inQueue[MAXN];
int cnt[MAXN], dis[MAXN];
queue<int> Q;
struct {
    int to, next, w;
} edges[MAXN * 5];

int read() {
    int ans = 0, sgn = 1;
    char c = getchar();
    while (c < '0' || c > '9') {
        if (c == '-') sgn = -1;
        c = getchar();
    }
    while (c >= '0' && c <= '9') {
        ans = ans * 10 + c - '0';
        c = getchar();
    }
    return ans * sgn;
}
```

```
void add_edge(int from, int to, int w) {
    edges[+cnt_edge].next = head[from];
    edges[cnt_edge].to = to;
    edges[cnt_edge].w = w;
    head[from] = cnt_edge;
}

void SPFA(int s) {
    for(int i = 0; i <= n + 1; i++) dis[i] = 1e9;
    dis[s] = 0;
    Q.push(s);
    inQueue[s] = true;
    while (!Q.empty()) {
        int p = Q.front();
        Q.pop();
        inQueue[p] = false;
        for (int eg = head[p]; eg; eg = edges[eg].next) {
            int to = edges[eg].to;
            if (dis[to] > dis[p] + edges[eg].w) {
                dis[to] = dis[p] + edges[eg].w;
                if (!inQueue[to]) {
                    Q.push(to);
                    inQueue[to] = true;
                }
            }
        }
    }
}

int main() {
    n = read(), m = read();
    s = n + 1;
```

```

memset(head, -1, sizeof(head));

for(int i = 0; i <= n; i++) {
    add_edge(s, i, 0);
}

for(int i = 1; i <= m; i++) {
    int a = read(), b = read(), c = read();
    add_edge(b, a - 1, -c);
}

for(int i = 1; i <= n; i++) {
    add_edge(i - 1, i, 1);
    add_edge(i, i - 1, 0);
}

SPFA(s);

int ans = *min_element(dis, dis + n + 1);
printf("%d\n", dis[n] - ans);

return 0;
}

```

题目要求在指定的区间 [b, e] 内至少种植 t 棵，为了满足这个条件并计算最少的树木数量而使用了一种类似于前缀和的思想，尽管我们没有直接声明一个前缀和数组。

添加边以维护节点之间的相对关系和前缀和的逻辑

```

for(int i = 1; i <= n; i++) {
    // 从 i-1 到 i 添加权重为 1 的边，表示从前一个区域到当前
    // 区域最多增加 1 棵树
    // 这实现了树木数量的累加效果，即前缀和
    add(i - 1, i, 1);

    // 从 i 到 i-1 添加权重为 0 的边，表示可以不增加树木而回
    // 到前一个区域
    add(i, i - 1, 0);
}

```

在这个程序中，前缀和的逻辑体现在通过 SPFA 算法计算最短路径的过程

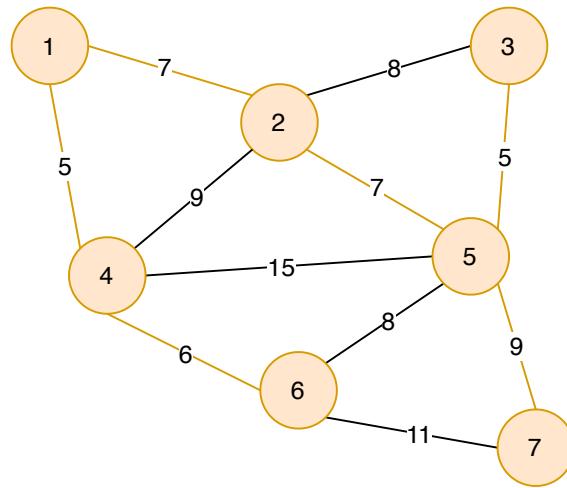
中。通过添加权重为 1 的边，我们保证了当我们从一个节点移动到下一个节点时，树的数量最多增加 1。这样，当我们从源点 s 到任意点 i 的最短路径被计算出来时， $\text{dis}[i]$ 实际上表示的是从起点到点 i 所需要种植的最少树木数量，这本质上是一个前缀和的过程。

最后，题目要求最少的树木个数，所以最终结果是 $\text{dis}[n] - \text{ans}$ 。

11.6 最小生成树

最小生成树这一概念是指在一个无向连通图中所有边的权重之和最小。需要注意的是，最小生成树并不一定是唯一的，有时会有多个最小生成树，只有当图中各边的边权不相等时最小生成树才唯一；此外，只有连通图才有生成树，而对于非连通图，只存在生成森林。

最小生成树有两种算法：1. Prim 算法；2. Kruskal 算法。两种算法都基于贪心思想。



让我们通过一个题目来体会两种做法：

例 11.6.1 (luogu-P3366 【模板】最小生成树). 如题，给出一个无向图，求出最小生成树，如果该图不连通，则输出 orz。

输入格式:

第一行包含两个整数 N, M , 表示该图共有 N 个结点和 M 条无向边。

接下来 M 行每行包含三个整数 X_i, Y_i, Z_i , 表示有一条长度为 Z_i 的无向边连接结点 X_i, Y_i 。

```
4 5
1 2 2
1 3 2
1 4 3
2 3 4
3 4 3
```

输出格式:

如果该图连通, 则输出一个整数表示最小生成树的各边的长度之和。如果该图不连通则输出orz。

```
7
```

11.6.1 Prim 算法

Prim 算法的基本思想是: 从一个节点开始, 每次选择距离最小的一个结点, 并用新的边更新其他结点的距离。这个算法与 Dijkstra 算法有些类似, 但 Prim 算法更适合于稠密图。

对于 Prim 算法, 我们给出一个可行的方案:

```
#include <bits/stdc++.h>

using namespace std;

const int MAXN = 5007, MAXM = 200007;
const int INF = 0x3f3f3f3f;

int n, m, ans;
```

```
int dis[MAXN], last[MAXN], idx;
bool vis[MAXN];
struct node {
    int to, next, cost;
} e[2 * MAXM];

// 前向星构图
void add(int u, int v, int w) {
    e[++idx].to = v;
    e[idx].next = last[u];
    last[u] = idx;
    e[idx].cost = w;
}

void prim() {
    for (int i = 1; i <= n; i++) {
        dis[i] = INF;
    }
    dis[1] = 0;
    for (int j = 1; j <= n; j++) {
        int u = -1, minn = INF;
        for (int i = 1; i <= n; i++) {
            if (dis[i] < minn && !vis[i]) {
                u = i;
                minn = dis[i];
            }
        }
        if (u == -1) {
            ans = -1;
            return;
        }
        vis[u] = 1;
        ans += dis[u];
```

```
    for (int i = last[u]; i; i = e[i].next) {
        int v = e[i].to;
        if (!vis[v] && dis[v] > e[i].cost) dis[v] = e[i].
            cost;
    }
}

int main() {
    cin >> n >> m;
    for (int i = 1; i <= m; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        add(u, v, w);
        add(v, u, w);
    }
    prim();
    if (ans == -1) {
        printf("orz\n");
    } else {
        printf("%d\n", ans);
    }
    return 0;
}
```

这里简单提一下稀疏图与稠密图的概念，由于图由点和边组成，稀疏图与稠密图是一个相对的概念，若图中边的数量远远小于点的数量，则称为稀疏图，否则称为稠密图。

11.6.2 Kruskal 算法

Kruskal 算法的基本思想是：每次选择一条权最小的边，并判断是否形成了回路，如果形成了回路则舍弃这条边，否则加入最小生成树。

对于用 Kruskal 算法解决例题11.6.1，我们需要实现一个并查集，并用它来判断是否形成了回路，这就需要一个结构体来辅助实现，我们来看具体的做法：

```
#include <bits/stdc++.h>

using namespace std;

const int MAXN = 5007;
const int MAXM = 200007;

int fa[MAXN], deep[MAXN];
int n, m, tot = 0, ans = 0;
struct node {
    int u, v, w;
} e[MAXM];

// 修改排序规则
bool cmp(node a, node b) {
    return a.w < b.w;
}

// 并查集 查找
int find(int x) {
    if (fa[x] == x) return x;
    else return fa[x] = find(fa[x]);
}

// 并查集 合并
```

```
void unite(int x, int y) {
    x = find(x), y = find(y);
    if (x == y) return;
    if (deep[x] < deep[y]) fa[x] = y;
    else {
        fa[y] = x;
        if (deep[x] == deep[y]) deep[x]++;
    }
}

// 并查集 判断是否在同一个集合
bool same(int x, int y) {
    return find(x) == find(y);
}

int main() {
    cin >> n >> m;
    for (int i = 1; i <= n; i++) {
        fa[i] = i, deep[i] = 0;
    }
    for (int i = 1; i <= m; i++) {
        cin >> e[i].u >> e[i].v >> e[i].w;
    }
    // 逆序排序
    sort(e + 1, e + m + 1, cmp);
    for (int i = 1; i <= m; i++) {
        if (same(e[i].u, e[i].v)) {
            continue;
        } else {
            unite(e[i].u, e[i].v);
            ans += e[i].w;
            tot++;
        }
    }
}
```

```

    }

    if (tot < n - 1) {
        printf("orz");
    } else {
        printf("%d", ans);
    }
    return 0;
}

```

11.7 拓扑排序

拓扑排序是一种针对有向图的排序算法，其基本思想是：对于一个 DAG(有向无环图)，对其所有顶点进行排序，使得若存在一条从顶点 u 到顶点 v 的有向边，则 u 排在 v 的前面。可以证明，一个有向无环图一定存在拓扑排序，因此如果一个图存在拓扑排序，则可以称这个图为拓扑图，或称之为有向无环图。

拓扑排序本质是一个 BFS 算法，执行拓扑排序需要与先统计所有点的入度，在开始时先将所有入度为零的点入队，随后执行 BFS，算法的大致流程如下：

```

queue<int> q;
所有入度为0的点入队;
while (!q.empty()) {
    取出队头节点 t;
    枚举 t 的所有出边 t -> j {
        删除边 t -> j;
        j 的入度 d[j]--;
        if (d[j] == 0) {
            入队 j;
        }
    }
}

```

}

有一点需要注意的是，BFS 需要队列来辅助实现，但在拓扑排序问题中通常不使用 STL 中的队列，转而是用数组来模拟一个队列，具体原因我们可以通过例题来说明。

例 11.7.1 (luogu-B3644 【模板】拓扑排序 / 家谱树). 有个人的家族很大，辈分关系很混乱，请你帮整理一下这种关系。

给出每个人的后代的信息。

输出一个序列，使得每个人的后辈都比那个人后列出。

输入格式：

第 1 行一个整数 N ($1 \leq N \leq 100$)，表示家族的人数。

接下来 N 行，第 i 行描述第 i 个人的后代编号 $a_{i,j}$ ，表示 $a_{i,j}$ 是 i 的后代。

每行最后是 0 表示描述完毕。

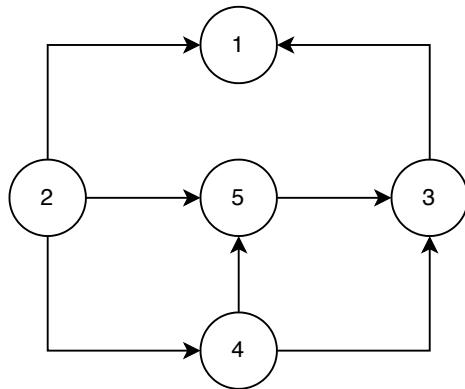
```
5
0
4 5 1 0
1 0
5 3 0
3 0
```

输出格式：

输出一个序列，使得每个人的后辈都比那个人后列出。如果有多种不同的序列，输出任意一种即可。

```
2 4 5 3 1
```

例题11.7.1的输入是一个有向图，其形状如下图所示，不难发现这是一个有向无环图，因此可以用拓扑排序算法来解决。



拓扑排序的大致流程已经通过伪代码给出，下面我们用程序来实现这个算法：

```

#include <bits/stdc++.h>

using namespace std;

const int N = 110, M = N * N / 2;

int n;
int h[N], e[M], ne[M], idx;
int q[N]; // 数组模拟队列
int d[N]; // 点的入度

void add(int a, int b) {
    e[idx] = b, ne[idx] = h[a], h[a] = idx++;
}

void topSort() {
    int head = 0, tail = -1; // head 队头指针, tail 队尾指针
    for (int i = 1; i <= n; i++) {
        if (!d[i]) {
            q[++tail] = i;
        }
    }
}
  
```

```
}

while (head <= tail) {
    int t = q[head++];
    for (int i = h[t]; ~i; i = ne[i]) {
        int j = e[i];
        if (--d[j] == 0)
            q[++tail] = j;
    }
}

int main() {
    cin >> n;
    memset(h, -1, sizeof h);

    for (int i = 1; i <= n; i++) {
        int son;
        while (cin >> son, son) {
            add(i, son);
            d[son]++;
        }
    }

    topSort();

    for (int i = 0; i < n; i++)
        printf("%d ", q[i]);
}

return 0;
}
```

刚刚我们提到会用一个数组来模拟队列，其原因在于每次出队时，我们只需要执行`int t = q[head++]`即可。这样做好处有两点：1. 如果需要输出序列，我们只需要输出`q[0] ... q[n-1]`即可；2. 如果我们无法确定是否存在拓扑序列，我们可以在 BFS 执行完成之后判断`tail == n - 1`是否成立，如果成立则说明存在拓扑序列。

同样我们再来看一道例题：

例 11.7.2 (CF510C Fox And Names). *Fox Ciel* 将要在 FOCS 上发表一篇论文。她听说了一个谣言：论文上的作者列表总是按字典顺序排序的。

在检查了一些例子后，她发现有时候这不是真的。在一些论文上，作者的名字并没有按照正常意义上的字典顺序排序。但总是存在这样的情况：在对字母表中字母的顺序进行某些修改，即作者的顺序变成了字典顺序！

她想知道，是否存在一种拉丁字母的顺序，使得她提交的论文上的名字按照字典顺序排列。如果存在，你应该找出这样的任意一种顺序。

字典顺序的定义如下：当我们比较 s 和 t 时，首先找到最左边的不同字符的位置： $s_i \neq t_i$ 。如果没有这样的位置（即 s 是 t 的前缀，反之亦然），则最短的字符串较小。否则，我们根据它们在字母表中的顺序比较字符 s_i 和 t_i 。

输入格式：

第一行包含一个整数 n ($1 \leq n \leq 100$) 代表字符串的数量。

以下 n 行中的每一行都包含一个字符串 $name_i$ ($1 \leq |name_i| \leq 100$)。每个字符串只包含小写字母，所有的字符串都是不同的。

```
3
rivest
shamir
adleman
```

输出格式：

如果存在这样的字母顺序，使得给定的字符串按字典顺序排序，输出这样的任意一个顺序作为字符 $a - z$ 的新排列（即首先输出修改后的字母表的第一个字母，然后是第二个，依此类推），否则输出 `Impossible`。

```
bcdefghijklmnopqrstuvwxyz
```

对于例题11.7.2，我们这里也直接给出做法：

```
#include <bits/stdc++.h>

using namespace std;

const int N = 210;
char s[N][N];
// 点数量较少，用邻接矩阵存图
int vis[26][26], len[N], d[26];
char ans[N];
int n;

void add(int a, int b) {
    if (!vis[a][b]) {
        vis[a][b] = 1; // 点a指向点b
        d[b]++;
    }
}

void topSort() {
    int idx = 0; // ans数组的索引
    for (int i = 0; i < 26; i++) {
        int j = 0;
        for (; j < 26; j++) {
            if (d[j] == 0) {
                break;
            }
        }
    }
}
```

```
}

if (j == 26) { // 特判，没有入度为0的点
    // 没有入度为0的点代表形成一个环
    printf("Impossible\n");
    return;
}

d[j] = -1; // 标记为已访问
ans[idx++] = j + 'a';
for (int k = 0; k < 26; k++) {
    if (vis[j][k]) {
        d[k]--;
    }
}
ans[idx] = '\0'; // 字符串结束符，标志字符串的结束
printf("%s\n", ans);
}

int main() {
    while (scanf("%d", &n) != EOF) { // 循环读取输入直到文件结束
        memset(d, 0, sizeof d);
        memset(vis, 0, sizeof vis);
        for (int i = 0; i < n; i++) {
            scanf("%s", s[i]);
            len[i] = strlen(s[i]); // 记录每个字符串的长度
        }
        bool flag = true;
        for (int i = 0; i < n - 1; i++) {
            int len1 = len[i], len2 = len[i + 1];
            bool found = false;
            // 比较相邻名字，找到第一个不同的字符
            for (int j = 0; j < min(len1, len2); j++) {
                if (s[i][j] != s[i + 1][j]) {
```

```
    add(s[i][j] - 'a', s[i + 1][j] - 'a'); // 添加字符顺序关系
    found = true;
    break;
}
}

// 如果没有找到不同的字符且前一个名字比后一个长，则无法排序
if (!found && len1 > len2) {
    flag = false;
    break;
}
}

if (!flag) {
    printf("Impossible\n");
} else {
    topSort();
}
}

return 0;
}
```

11.8 单源次短路问题

次短路问题即求解图中第二短的路径，通常次短路问题分为两类四种：

1. 边**不可重复经过**的次短路问题、边**可重复经过**的次短路问题；2. **严格**次短路（次短路长度必须**大于**最短路长度）、**非严格**次短路（次短路长度可以**大于或等于**最短路长度）。

11.8.1 边不可重复经过的次短路问题

例 11.8.1 (luogu-P1491 集合位置). 现在提出这样的一个问题：给出 n 个点的坐标，其中第一个为野猫的出发位置，最后一个为大家的集合位置，并给出哪些位置点是相连的。野猫从出发点到达集合点，总会挑一条最近的路走，如果野猫没找到最近的路，他就会走第二近的路。请帮野猫求一下这条第二最短路径长度。

特别地，选取的第二短路径不会重复经过同一条路，即使可能重复走过同一条路多次路程会更短。

输入格式：

第一行是两个整数 $n(1 \leq n \leq 200)$ 和 $m(1 \leq m \leq 10000)$ ，表示一共有 n 个点和 m 条路；

以下 n 行每行两个数 $x_i, y_i, (-500 \leq x_i, y_i \leq 500)$ ，代表第 i 个点的坐标；

再往下的 m 行每行两个整数 $p_j, q_j, (1 \leq p_j, q_j \leq n)$ ，表示两个点之间有一条路，数据没有重边，可能有自环。

```
3 3
0 0
1 1
0 2
1 2
1 3
2 3
```

输出格式：

只有一行包含一个数，为第二最短路线的距离（保留两位小数），如果存在多条第一短路径，则答案就是第一最短路径的长度；如果不存在第二最短路径，输出 -1。

```
2.83
```

例题11.8.1要求的是一张图的**非严格**次短路，且每条边**不可重复经过**。

为了解决这个问题，可以选择尝试「删边法」，即先对原图进行最短路计算，记录最短路的路径，之后每次删去**最短路的一条边**，重新跑最短路，在这几次跑出来的结果中，取最小值，就是最终答案。

这个方法是可行的，首次次短路和最短路的路径必然是不同的。因为最短路的路径肯定是整张图中最优的，如果每次去掉最短路上的一条路径，剩下的路径必然**不会比**最短路的路径更优。在删掉一条边后的剩下的路径中取得最短路径，肯定不会比原先图的最短路径更优，那么这一条路就必然是次短路径。

对于这道题而言，考虑到题中给出的是点的坐标，因此需要记录每个点的坐标位置，在连边时用两点间距离公式算出距。这里给出解决方案：

```
#include <bits/stdc++.h>
const int N = 2e4 + 5;
using namespace std;

int head[N], cnt = 0;
struct edge {
    int nextt, to;
    double w;
} e[N << 1]; // 无向图，双倍存边

void add(int x, int y, double z) {
    cnt++;
    e[cnt].nextt = head[x];
    head[x] = cnt;
    e[cnt].to = y;
    e[cnt].w = z;
}

int n, m, prevv[N];
```

```
int x[N], y[N];

// 两点之间距离
double cal(int p, int q) {
    return sqrt((x[p] - x[q]) * (x[p] - x[q]) + (y[p] - y[q]) *
        (y[p] - y[q]));
}

class Dijkstra {
public:
    struct point {
        int id;
        double dis;

        bool operator<(const point& other) const {
            return dis > other.dis;
        }
    };

    priority_queue<point> q;
    double dist[N];
    bool vis[N];

    void dijkstra(int dx, int dy) {
        for (int i = 1; i <= n; i++) {
            dist[i] = DBL_MAX;
        }
        memset(vis, 0, sizeof(vis));
        dist[1] = 0;
        q.push({1, 0});
        while (!q.empty()) {
            int tmp = q.top().id;
            q.pop();
            for (int i = 1; i <= n; i++) {
                if (dist[i] > dist[tmp] + cal(tmp, i)) {
                    dist[i] = dist[tmp] + cal(tmp, i);
                    q.push({i, dist[i]});
                }
            }
        }
    }
}
```

```
if (!vis[tmp]) {
    vis[tmp] = 1;
    for (int i = head[tmp]; i; i = e[i].nextt) {
        int v = e[i].to;
        double w = e[i].w;
        if (tmp == dx && v == dy || tmp == dy && v
            == dx) {
            continue;
        }
        if (dist[v] > dist[tmp] + w) {
            dist[v] = dist[tmp] + w;
            q.push({v, dist[tmp] + w});
            if (dx == -1 && dy == -1) {
                prevv[v] = tmp;
            }
        }
    }
}
};

int main() {
    cin >> n >> m;
    for (int i = 1; i <= n; i++) {
        cin >> x[i] >> y[i];
    }
    for (int i = 1; i <= m; i++) {
        int p, q;
        cin >> p >> q;
        double dis = cal(p, q);
        add(p, q, dis);
        add(q, p, dis);
```

```

    }

Dijkstra dij;
dij.dijkstra(-1, -1);
double ans = DBL_MAX;
for (int i = n; i != 1; i = prevv[i]) {
    int dx = i, dy = prevv[i];
    dij.dijkstra(dx, dy);
    ans = min(ans, dij.dist[n]);
}
if (ans == DBL_MAX) {
    printf("-1");
    return 0;
}
printf("%.2lf", ans);
return 0;
}

```

如果题中要求计算的是严格次短路，那么在第一次求解最短路时，将最短路记录下来。之后在遍历 `prevv` 数组求删边后的最短路时，若求出来和第一次求解最短路的路径长度相同，则忽略当前结果。

11.8.2 边可重复经过的次短路问题

例 11.8.2 (P2865 (USACO06NOV) Roadblocks G). 贝茜把家搬到了一个小农场，但她常常回到 *FJ* 的农场去拜访她的朋友。贝茜很喜欢路边的风景，不想那么快地结束她的旅途，于是她每次回农场，都会选择第二短的路径，而不象我们所习惯的那样，选择最短路。

贝茜所在的乡村有 $R(1 \leq R \leq 10^5)$ 条双向道路，每条路都联结了所有的 $N(1 \leq N \leq 5000)$ 个农场中的某两个。贝茜居住在农场 1，她的朋友们居住在农场 N (即贝茜每次旅行的目的地)。

贝茜选择的第二短的路径中，可以包含任何一条在最短路中出现的道路，并且，一条路可以重复走多次。当然咯，第二短路的长度必须严格大于最短路（可能有多条）的长度，但它的长度必须不大于所有除最短路外的路径的长度。

输入格式：

第一行两个整数 R 和 N ，表示有 R 条双向道路，联结了 N 个农场。

接下来 R 行，每行包含三个整数 A 、 B 、 D ，表示第 A 个农场和第 B 个农场之间有一条双向道路，其长度为 D 。

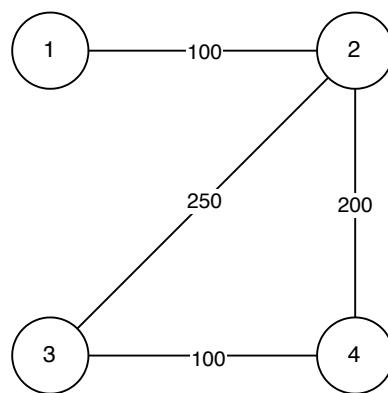
```
4 4
1 2 100
2 4 200
2 3 250
3 4 100
```

输出格式：

只有一行，包含一个整数，表示贝茜的第二短路的长度。

```
450
```

例题11.8.2要求的是一张图中每条边可重复经过的严格次短路。例题的图如下所示，最短路是 $1 \rightarrow 2 \rightarrow 4$ ，长度为 $100 + 200 = 300$ ，次短路则是 $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ ，长度为 $100 + 250 + 100 = 450$ 。



考虑到可以重复经过路径，为了解决这个问题，这里选择维护两个数组，分别记录**最短路**和**次短路**。

- 当**最短路**可以更新时，那么**次短路**就继承之前的**最短路**的长度。
- 当**次短路**可以更新，且**次短路**更新后不会超过**最短路**长度，那么**次短路**更新。

详细来说：

1. **更新次短路的处理**：当发现一条新的**次短路**时，我们需要将这个新的**次短路**信息（顶点编号和距离）也放入优先队列中。这是因为不能确定当前的**次短路**就是最终的最优解，它可能在之后的计算中被进一步优化，所以需要保留一个继续更新的机会。
2. **去掉访问标记（vis 数组）**：在这个改进的算法中，不再使用传统 Dijkstra 算法中的 **vis** 数组。原因是，即使一个顶点的**最短路**已经确定，它的**次短路**可能还需要更新。使用 **vis** 数组可能会过早地终止对某些路径的探索，导致错过潜在的**次短路**。
3. **优先队列元素的处理**：传统算法中队列顶部元素的距离总是对应于该顶点的当前**最短距离**。与传统 Dijkstra 算法不同，在该版本中需要同时取出优先队列顶部元素的顶点 **id** 和距离。这是因为队列中可能同时包含**最短路**和**次短路**的信息，因此需要根据实际的距离值来判断当前处理的是**最短路**还是**次短路**，从而决定使用哪个 **dist** 数组。

```
#include <bits/stdc++.h>
const int N = 1e5 + 5;
using namespace std;

int n, m, s;

struct edge {
    int nextt, to, w;
} e[N << 1];
```

```
int cnt = 0, head[N];

void edge_add(int x, int y, int z) {
    cnt++;
    e[cnt].nextt = head[x];
    head[x] = cnt;
    e[cnt].to = y;
    e[cnt].w = z;
}

class Dijkstra {
public:
    struct point {
        int id, dis;
        bool operator < (const point& b) const {
            return dis > b.dis;
        }
    };

    priority_queue<point> q;
    int dist[N][5];

    void dijkstra() {
        memset(dist, 0x3f, sizeof(dist));
        dist[1][1] = 0;
        q.push({1, 0});
        while (!q.empty()) {
            int tmp = q.top().id;
            int dis_now = q.top().dis;
            q.pop();
            for (int i = head[tmp]; i; i = e[i].nextt) {
                int v = e[i].to;
                int w = e[i].w;
```

```
        if (dist[v][1] > dis_now + w) {
            dist[v][2] = dist[v][1];
            dist[v][1] = dis_now + w;
            q.push({v, dist[v][1]});
            q.push({v, dist[v][2]});
        } else if (dist[v][2] > dis_now + w && dist[v][1] < dis_now + w) {
            dist[v][2] = dis_now + w;
            q.push({v, dist[v][2]});
        }
    }
}
};

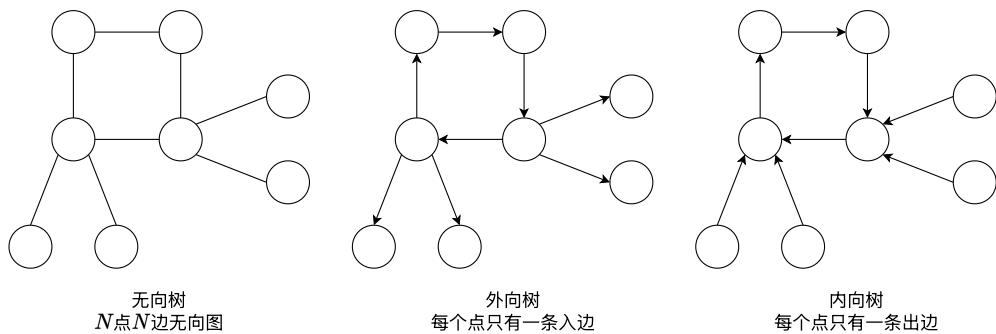
int main() {
    cin >> n >> m;
    for (int i = 1; i <= m; i++) {
        int x, y, z;
        cin >> x >> y >> z;
        edge_add(x, y, z);
        edge_add(y, x, z);
    }
    Dijkstra dij;
    dij.dijkstra();
    printf("%d", dij.dist[n][2]);
    return 0;
}
```

同样，若这道题要求非严格次短路，只需将当次短路可以更新时的后半部分条件稍作修改即可，即将 $\text{dist}[v][1] < \text{dis_now} + w$ 中的判断条件修改为 $\text{dist}[v][1] \leq \text{dis_now} + w$ 。

11.9 基环树

定义 11.9.1 (基环树). 基环树是一个节点数等于边数的连通图。

根据定义，基环树就是有 n 个点 n 条边的连通图，由于比树只出现了一个环，就称之为基环树。换言之，如果从图中移除环上的任意一条边，剩下的图就变成了一棵树。特别的，若有图中存在多个子图均为基环树，则该图成为基环树森林。



以上是三种特殊的基环树，这三种树有十分优秀的性质，即可以**直接将环作为根**，随后可以对每个环的子树进行单独处理，最后再处理环。

11.9.1 找环

处理无向图时，可以根据拓扑排序找出环上的所有点：

```
void topsort() {
    int l = 0, r = 0;
    // 将所有入度为 1 的节点加入队列
    for (int i = 1; i <= n; i++) {
        if (in[i] == 1) {
            q[++r] = i;
        }
    }
}
```

```

// 处理队列中的节点
while (l < r) {
    int now = q[++l]; // 取出队首节点
    for (int i = ls[now]; i; i = a[i].next) {
        int y = a[i].to;
        // 只处理入度大于1的节点(可能在环上)
        if (in[y] > 1) {
            in[y]--; // 将邻接节点的入度减1
            // 如果减1后入度变为1, 将其加入队列
            if (in[y] == 1) {
                q[++r] = y;
            }
        }
    }
}
}

```

其原理在于：对于无向图，每条边会给它连接的两个顶点各增加 1 的入度，而度为 1 的点一定是树枝的叶子节点，不可能在环上。因此，我们把所有入度为 1 的点加入队列，随后每次从队列中取出一个点，将其所有邻接点的入度减 1，如果减完后某个邻接点的入度变为 1，就再把该点加入队列。这个过程实际上是在“剥离”图中的树枝结构，所有不在环上的点，最终都会经历入度变为 1 的阶段，然后被处理，而环上的点，因为总是至少有来自环内的两个邻居，所以入度永远不会降到 1。最终，经过查找之后，入度 ≥ 2 的点就是环上的点。

如果是有向图，则使用 DFS 来处理：

```

void check(int x) {
    vis[x] = true;
    if (vis[d[x]]) {
        ring = x;
    } else {

```

```

    check(father[x]);
}
return;
}

```

其中，`ring` 是环上的一个点，`d[]` 表示每个节点的下一个节点。

接下来我们将探讨几种解决问题的方案：

11.9.2 断环法

例 11.9.2 (P5022 旅行). 小 Y 了解到， X 国的 n 个城市之间有 m 条双向道路。每条双向道路连接两个城市。不存在两条连接同一对城市的道路，也不存在一条连接一个城市和它本身的道路。并且，从任意一个城市出发，通过这些道路都可以到达任意一个其他城市。小 Y 只能通过这些道路从一个城市前往另一个城市。

小 Y 的旅行方案是这样的：任意选定一个城市作为起点，然后从起点开始，每次可以选择一条与当前城市相连的道路，走向一个没有去过的城市，或者沿着第一次访问该城市时经过的道路后退到上一个城市。当小 Y 回到起点时，她可以选择结束这次旅行或继续旅行。需要注意的是，小 Y 要求在旅行方案中，每个城市都被访问到。

为了让自己的旅行更有意义，小 Y 决定在每到达一个新的城市（包括起点）时，将它的编号记录下来。她知道这样会形成一个长度为 n 的序列。她希望这个序列的字典序最小，你能帮帮她吗？对于两个长度均为 n 的序列 A 和 B ，当且仅当存在一个正整数 x ，满足以下条件时，我们说序列 A 的字典序小于 B 。

- 对于任意正整数 $1 \leq i < x$ ，序列 A 的第 i 个元素 A_i 和序列 B 的第 i 个元素 B_i 相同。
- 序列 A 的第 x 个元素的值小于序列 B 的第 x 个元素的值。

输入格式：

第一行包含两个整数 $n, m (m \leq n)$, 中间用一个空格分隔。

接下来 m 行, 每行包含两个整数 $u, v (1 \leq u, v \leq n)$, 表示编号为 u 和 v 的城市之间有一条道路, 两个整数之间用一个空格分隔。

```
6 6  
1 3  
2 3  
2 5  
3 4  
4 5  
4 6
```

输出格式:

包含一行, n 个整数, 表示字典序最小的序列。相邻两个整数之间用一个空格分隔。

```
1 3 2 4 5 6
```

例题11.9.2简要来说就是: 给定一棵树(这棵树可能是基环树), 从节点 1 出发, 每到达一个新的点就记录下编号, 求一种走法使得记录下来的编号字典序最小。

本题的特殊点在于, 首先不确定树是否是基环树, 如果是一棵普通的树, 则每次按字典序小的点走, 简单排序一下即可得到结果; 那么如果是基环树, 题目中表明 100% 的数据和所有样例, $1 \leq n \leq 5000$, 在数据量不大、点不多的情况下, 环本身不会对题目造成最终影响, 因此可以暴力删边将基环树变为一棵普通的树, 然后计算答案。

```
#include <bits/stdc++.h>  
  
using namespace std;  
const int N = 5010;  
  
int cnt = 0, head[N], in[N];
```

```
struct edge {
    int to, nextt;
} e[N << 1];

void edge_add(int x, int y) {
    cnt++;
    e[cnt].to = y;
    e[cnt].nextt = head[x];
    head[x] = cnt;
    in[y]++;
}

// 存储一条边的两个端点
struct line {
    int x, y;
} l[N << 2];

int n, m, t, ans[N];

/***
 * state[N]: 用于存储当前遍历的节点顺序
 * w[N]: 用于存储环上的节点。
 * q[N]: 用于拓扑排序的队列。
 */
int state[N], w[N], q[N];

/***
 * k[N][N]: 标记某条边是否被删除
 * v[N]: 标记节点是否被访问过
 */
bool k[N][N], v[N];

// 拓扑求环
```

```
bool topsort() {
    int l = 0, r = 0;
    for (int i = 1; i <= n; i++) {
        if (in[i] == 1) {
            q[++r] = i;
        }
    }
    while (l < r) {
        int now = q[++l];
        for (int i = head[now]; i; i = e[i].nextt) {
            int y = e[i].to;
            if (in[y] > 1) {
                in[y]--;
                if (in[y] == 1) {
                    q[++r] = y;
                }
            }
        }
    }
    if (r == n) {
        return true;
    }
    return false;
}

bool cmp(line x, line y) {
    return x.y > y.y;
}

void dfs(int x) { // 走一遍
    state[++t] = x;
    v[x] = true;
    for (int i = head[x]; i; i = e[i].nextt) {
```

```
    int y = e[i].to;
    if (k[x][y] || v[y]) {
        continue;
    }
    dfs(y);
}

// 检查当前遍历顺序是否字典序更小。
void check() {
    int i;
    bool flag = false;
    for (i = 1; i <= n; i++) {
        if (state[i] < ans[i]) {
            flag = true;
            break;
        } else if (state[i] > ans[i]) {
            return;
        }
    }
    if (!flag) {
        return;
    }
    for (; i <= n; i++) {
        ans[i] = state[i];
    }
}

void getAns(int xs) { // 暴力删边
    int x = xs, b = 0, i;
    do {
        w[++b] = x;
        in[x] = 1;
```

```
for (i = head[x]; i; i = e[i].nextt) {
    int y = e[i].to;
    if (in[y] > 1) {
        x = y;
        break;
    }
}
} while (i); //记录环的每个点

w[++b] = xs;

for (int i = 1; i < b; i++) { //枚举删除的边
    k[w[i]][w[i + 1]] = k[w[i + 1]][w[i]] = true;
    memset(v, 0, sizeof(v));
    t = 0;
    dfs(1);
    check();
    k[w[i]][w[i + 1]] = k[w[i + 1]][w[i]] = false;
}
}

int main() {
    memset(ans, 127 / 3, sizeof(ans));
    cin >> n >> m;
    for (int i = 1; i <= m; i++) {
        int x, y;
        cin >> x >> y;
        l[i] = (line) {x, y};
        l[i + m] = (line) {y, x};
    }
    sort(l + 1, l + 1 + 2 * m, cmp); // 排序
    cnt = 1;
    for (int i = 1; i <= 2 * m; i++) {
```

```
    edge_add(l[i].x, l[i].y);
}

if (m == n - 1) { //普通树
    dfs(1);
    for (int i = 1; i <= n; i++) {
        printf("%d ", state[i]);
    }
    return 0;
}

topsort();
for (int i = 1; i <= n; i++) {
    if (in[i] > 1) {
        getAns(i);
        break;
    }
}
for (int i = 1; i <= n; i++) {
    printf("%d ", ans[i]);
}
}
```

11.9.3 二次 DP 法

例 11.9.3 (P2607 (ZJOI2008) 骑士). 战火绵延，人民生灵涂炭，组织起一个骑士军团加入战斗刻不容缓！国王交给了你一个艰巨的任务，从所有的骑士中选出一个骑士军团，使得军团内没有矛盾的两人（不存在一个骑士与他最痛恨的人一同被选入骑士军团的情况），并且，使得这支骑士军团最具有战斗力。

为了描述战斗力，我们将骑士按照 1 至 n 编号，给每名骑士一个战斗力的估计，一个军团的战斗力为所有骑士的战斗力总和。

输入格式:

第一行包含一个整数 n , 描述骑士团的人数。

接下来 n 行, 每行两个整数, 按顺序描述每一名骑士的战斗力和他最痛恨的骑士。

```
3
10 2
20 3
30 1
```

输出格式:

输出一行, 包含一个整数, 表示你所选出的骑士军团的战斗力。

```
30
```

对于例题11.9.3, 也即环形的问题, 可以像环形 dp 一样将一条边强行断开处理, 然后强行连上再进行处理, 对于该题, 类似于「没有上司的舞会」, 因此考虑利用二次树形 dp 的方法, 先找到环, 然后强行将环断开进行一次 dp, 然后强行连上进行一次 dp, 两次答案求最大值。具体实现如下:

```
#include <bits/stdc++.h>
#define ll long long

using namespace std;

const int N = 1000010;
ll cnt, head[N], root;
struct edge {
    ll to, next;
} a[N];

void edge_add(ll x, ll y) {
    a[++cnt].to = y;
    a[cnt].next = head[x];
```

```
    head[x] = cnt;
}

ll n, x, ans, d[N], ring;
/***
 * w[N]: 存储节点的价值
 * f[N]: 包含当前节点的最优解
 * g[N]: 不包含当前节点的最优解
 */
ll w[N], fa[N], f[N], g[N];
bool vis[N];

void check(ll x) {
    vis[x] = true;
    if (vis[d[x]]) ring = x;
    else check(d[x]);
    return;
}

void treeDP(ll x) {
    vis[x] = true;
    f[x] = w[x];
    g[x] = 0;
    for (ll i = head[x]; i; i = a[i].next) {
        ll y = a[i].to;
        /**
         * 如果 y 不是环中的节点 ( $y \neq ring$ ) , 正常处理子树
         * 如果 y 是环中的节点 ( $y == ring$ ) , 不处理, 在这里把环
         * 断开
        */
        if (y != ring) {
            treeDP(y);
            g[x] += max(f[y], g[y]);
        }
    }
}
```

```
        f[x] += g[y];
    } else {
        f[y] = -0x3f3f3f / 3;
    }
}

int main() {
    cin >> n;
    for (ll i = 1; i <= n; i++) {
        cin >> w[i] >> d[i];
        edge_add(d[i], i);
    }
    for (ll i = 1; i <= n; i++) {
        if (vis[i]) {
            continue;
        }
        check(i);
        treeDP(ring);
        ll maxs = max(f[ring], g[ring]);
        ring = d[ring];
        root = 0;
        treeDP(ring);
        ans += max(maxs, max(f[ring], g[ring]));
    }
    printf("%lld", ans);
}
```

11.10 欧拉路径与欧拉回路

欧拉路径和欧拉回路是图论中两个重要的概念，我们先来看定义：

定义 11.10.1 (欧拉路径). 欧拉路径是一条从某个节点为起点出发，每条边只经过一次且能够到达终点的路径。

定义 11.10.2 (欧拉回路). 欧拉回路是一条从某个节点出发，每条边只经过一次且回到该节点的路径。

欧拉路径与欧拉回路就是生活中所谓的“一笔画”问题，那么如果要做到“一笔画”需要哪些条件呢？在这里我们直接给出定理：

定理 11.10.3. 对于一个无向图，存在欧拉路径的充分必要条件是：该图中度数为奇数的节点只能有 0 个或 2 个。

对于一个有向图，存在欧拉路径的充分必要条件是：

1. 所有点的入度均等于出度；
2. 除起点与终点外，所有点的入度均等于出度，起点满足出度比入度多 1，终点满足入度比出度多 1。

定理 11.10.4. 对于一个无向图，存在欧拉回路的充分必要条件是：该图中度数为奇数的节点只能有 0 个。

对于一个有向图，存在欧拉回路的充分必要条件是：所有点的入度均等于出度。

欧拉路径与欧拉回路问题在用程序处理的时候，与一般的图论问题不同的点在于需要用边来判重，

第十二章 动态规划

12.1 动态规划基础

动态规划是每个 OI 选手首个需要跨过的难关，动态规划问题在前文已经简单探讨过，这里单独开一个章节讨论，其原因在于动态规划是一种思想而非具体的算法，因而不会像其他算法一样存在可以套用的模板。

通常情况下，动态规划问题常常应用于问题中出现求最大值、最小值、方案数，或是存在最优选择、先后手必胜等字眼，这是由于**动态规划的本质是记忆化搜索加上决策过程**。例如例题12.1.1就是一个求解最大值的问题，我们为了确保我们找到的值确实是所有值中的最大值，我们需要把所有的情况都列举出来并比较，这个过程是烦杂且费时的。

例 12.1.1 (打气球). 你在游乐场发现一个玩具枪射击气球的游戏，游戏规则如下：有 n 个气球，编号为 0 到 $\text{nums}[n - 1]$ ，每个气球上都标有一个数字，这些数字存在数组 nums 中。

打破第 i 个气球可以获得 $\text{nums}[i - 1] * \text{nums}[i] * \text{nums}[i + 1]$ 枚硬币，求所能获得硬币的最大数量。

这里的 $i - 1$ 和 $i + 1$ 代表和 i 相邻的两个气球的序号。如果 $i - 1$ 或 $i + 1$ 超出了数组的边界，那么就当它是一个数字为 1 的气球。

动态规划所做的就是将中间过程的计算结果保留下来，并在所有的中间过程中选出最大值进而确保求出的值是最终结果。

学习动态规划需要关注能够使用动态规划解决问题的三个条件：

- 最优子结构；
- 子问题无后效性；
- 子问题重叠。

我们将在后续学习过程中深入探讨这三个条件。

12.2 线性 DP

线性 DP 指的是在解决问题时存在一个明显的线性顺序，这类问题称之为线性 DP。我们来看一个经典的例题：

例 12.2.1 (acwing-898 数字三角形). 给定一个如下图所示的数字三角形，从顶部出发，在每一结点可以选择移动至其左下方的结点或移动至其右下方的结点，一直走到底层，要求找出一条路径，使路径上的数字的和最大。

7
3 8
8 1 0
2 7 4 4
4 5 2 6 5

输入格式

第一行包含整数 n ，表示数字三角形的层数。

接下来 n 行，每行包含若干整数，其中第 i 行表示数字三角形第 i 层包含的整数。

5
7
3 8

8	1	0		
2	7	4	4	
4	5	2	6	5

输出格式

输出一行，这一行只包含一个整数，表示最大的路径数字和。

30

我们分析例题12.2.1，解决这个问题需要我们从起点开始每次向下推进一行，直到走到最后一行，并在最后一行中查找最大值，这就是一个线性的顺序。为了解决这个问题，我们首先定义状态，我们定义一个二维数组 $dp[i][j]$ 代表从数字三角形的顶部到第*i*层第*j*个位置的最大路径和。

同时，下一行的点都只能由紧邻的上一行的一或两个点到达，例如我们要到达例题中第四行的点 7，我们可以从第三行的点 8 向右下走或是由第三行的点 1 向左下走。而 $dp[i][j]$ 代表的是到达当前点的最大路径和，因而我们不需要关注路径到底是怎么走的，只需要关注哪一条路到达目标点得到的值更大，所以我们得到状态转移方程：

$dp[i][j] = \max(dp[i - 1][j - 1], dp[i - 1][j]) + triangle[i][j];$

其中 $triangle[i][j]$ 是数字三角形第 *i* 层第 *j* 个位置的数值。

此外，我们需要特别处理每一层的第一个和最后一个元素，因为它们只有一个父节点。初始化条件是 $dp[1][1] = triangle[1][1]$ ，即数字三角形的顶部元素。

最后，我们只需要找到最后一层中 dp 值的最大值，即为所求的最大路径和。

#include<bits/stdc++.h>
using namespace std;
const int N = 510;

```
int main() {
    int n;
    cin >> n;
    int triangle[N][N];
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= i; j++) {
            cin >> triangle[i][j];
        }
    }
    int dp[N][N];
    // 初始化
    dp[1][1] = triangle[1][1];

    // 动态规划部分
    for (int i = 2; i <= n; i++) {
        for (int j = 1; j <= i; j++) {
            if (j == 1) {
                dp[i][j] = dp[i - 1][j] + triangle[i][j];
            } else if (j == i) {
                dp[i][j] = dp[i - 1][j - 1] + triangle[i][j];
            } else {
                dp[i][j] = max(dp[i - 1][j - 1], dp[i - 1][j]) +
                           triangle[i][j];
            }
        }
    }

    // 找到最后一层中的最大值
    int ans = dp[n][1];
    for (int i = 2; i <= n; i++) {
        ans = max(ans, dp[n][i]);
    }
}
```

```
cout << ans << endl;  
return 0;  
}
```

例 12.2.2 (acwing-895 最长上升子序列). 给定一个长度为 N 的数列, 求数值严格单调递增的子序列的长度最长是多少。

输入格式

第一行包含整数 N 。

第二行包含 N 个整数, 表示完整序列。

```
7  
3 1 2 1 8 5 6
```

输出格式

输出一个整数, 表示最大长度。

```
4
```

数据范围

- $1 \leq N \leq 10^3$
- $-10^9 \leq$ 序列中的数 $\leq 10^9$

例题12.2.2也是一道经典的动态规划问题, 称为**最长上升子序列 (LIS)**问题。同样的, 我们定义一个状态数组 dp , 其中 $\text{dp}[i]$ 表示以第 i 个数结尾的最长上升子序列的长度。

由于我们定义 $\text{dp}[i]$ 为以第 i 个数结尾的最长上升子序列的长度。这意味着我们考虑的子序列必须以第 i 个数结束。这样定义的好处在于, 我们可以根据前面的结果来确定当前的结果。

那么, 为了得到一个以第 i 个数结尾的上升子序列, 我们需要在前 $i-1$ 个数中找到一个数, 它既小于第 i 个数, 又有最长的上升子序列。这样, 我

们就可以将第 i 个数接在这个子序列的后面，得到一个更长的上升子序列。因此，我们可以得到以下状态转移方程：

```
dp[i] = max(dp[i], dp[j] + 1);
// 0 <= j < i 且 nums[j] < nums[i]
```

这个方程的意思是：对于每一个 j (j 小于 i)，如果 $\text{nums}[j] < \text{nums}[i]$ ，那么我们可以考虑将 $\text{nums}[i]$ 接在以 $\text{nums}[j]$ 结尾的上升子序列后面。这样，新的子序列的长度就是 $\text{dp}[j] + 1$ 。

初始化条件是 $\text{dp}[i] = 1$ ，因为每个数本身就是一个长度为 1 的上升子序列。最后，我们只需要找到 dp 数组中的最大值，即为所求的最长上升子序列的长度。

```
#include <bits/stdc++.h>

using namespace std;

const int N = 1010;

int main() {
    int n;
    cin >> n;
    int nums[N];
    for (int i = 1; i <= n; i++) {
        cin >> nums[i];
    }
    int dp[N];
    // 初始化
    for (int i = 1; i <= n; i++) {
        dp[i] = 1;
    }

    // 动态规划部分
```

```

for (int i = 2; i <= n; i++) {
    for (int j = 1; j < i; j++) {
        if (nums[j] < nums[i]) {
            dp[i] = max(dp[i], dp[j] + 1);
        }
    }
}

// 找到 dp 数组中的最大值
int ans = dp[1];
for (int i = 2; i <= n; i++) {
    ans = max(ans, dp[i]);
}
cout << ans << endl;
return 0;
}

```

我们回顾两道例题，动态规划的关键在于**状态定义和状态转移方程**，在例题12.2.1中我们定义 $dp[i][j]$ 代表从数字三角形的顶部到第*i*行第*j*列的最大路径和，在例题12.2.2中我们定义 $dp[i]$ 表示以第*i*个数结尾的最长上升子序列的长度，两道题的状态数组维度不同。在这里，我们不难发现，*x*维的状态数组所带来的状态转移方程的时间复杂度是 $O(n^x)$ ，因此，在解决动态规划问题时设计状态数组，我们应当从低维开始考虑。

现在，我们将例题12.2.2升级一下，将数据范围限定为：

- $1 \leq N \leq 10^5$
- $-10^9 \leq$ 序列中的数 $\leq 10^9$

当 N (即序列长度) 达到 10^5 时，之前时间复杂度为 $O(n^2)$ 的动态规划方法将会超时，因此我们需要考虑一个更为高效的解决方法，使用**贪心算法**和**二分查找**来解决这个问题：

维护一个数组tails，其中tails[i]表示长度为*i*+1的上升子序列的最小可能尾部元素，显然，这个数组是单调递增的。

对于序列中的每一个元素，我们使用**二分查找**来查找该元素在tails中插入的位置。如果这个位置超过了tails的当前长度，那么我们就在tails的末尾添加这个元素，表示我们找到了一个更长的上升子序列。否则，我们更新tails中相应位置的元素，表示我们找到了一个尾部元素更小的同样长度的上升子序列。

最后，tails的长度就是最长上升子序列的长度。

```
#include <bits/stdc++.h>

using namespace std;

const int N = 100010;

// 二分查找函数，返回x在tails中的位置，如果x不在tails中，则返回x
// 应该插入的位置。
int binary_search(const vector<int>& tails, int x) {
    int l = 0, r = tails.size() - 1;
    while (l <= r) {
        int mid = (l + r) / 2;
        if (tails[mid] < x) {
            l = mid + 1; // 如果x大于中间值，搜索右半部分
        } else {
            r = mid - 1;
        }
    }
    return l; // 返回x应该插入的位置
}

int main() {
    int n;
    cin >> n;
```

```
int nums[N];
for (int i = 0; i < n; i++) {
    cin >> nums[i];
}

// tails[i] 表示长度为 i + 1 的上升子序列的最小可能尾部元素
vector<int> tails;
for (int i = 0; i < n; i++) {
    // 查找 nums[i] 在 tails 中的位置或应该插入的位置
    int pos = binary_search(tails, nums[i]);
    if (pos == tails.size()) {
        // 如果 nums[i] 大于 tails 中的所有元素，表示我们找到了
        // 一个更长的上升子序列
        tails.push_back(nums[i]);
    } else {
        // 否则，更新 tails 中相应位置的元素，表示我们找到了一
        // 个尾部元素更小的同样长度的上升子序列
        tails[pos] = nums[i];
    }
}
cout << tails.size() << endl;
return 0;
}
```

12.3 背包 DP

背包问题是一种组合优化的 NP 完全 (NP-Complete, NPC) 问题, NPC 问题是没有多项式时间复杂度的解法的, 但是利用动态规划, 我们可以以伪多项式时间复杂度求解背包问题。背包问题通常包括三种:

- 0-1 背包问题;
- 完全背包问题;

- 多重背包问题。

12.3.1 0-1 背包问题

例 12.3.1 (背包问题). 有 n 个物品和一个容量为 W 的背包，每个物品有重量 w_i 和价值 v_i 两种属性，要求选若干物品放入背包使背包中物品的总价值最大且背包中物品的总重量不超过背包的容量。

例题12.3.1所描述的是一道经典的背包问题，从题干中我们可以得到：每个物体只有选与不选两种可能的状态，对应二进制中的 0 和 1，因而这类问题被称为 **0-1 背包问题**。

针对这个问题，显然我们可以暴力求解，对于每个物品都考虑其选与不选的情况：

```
// 递归函数，用于暴力搜索
int knapsackRecursive(vector<int>& values, vector<int>& weights,
    int capacity, int index) {
    // 递归的边界条件
    if (index < 0 || capacity <= 0) {
        return 0;
    }

    // 当前物品的重量
    int currentWeight = weights[index];
    // 当前物品的价值
    int currentValue = values[index];

    // 不选择当前物品的情况
    int withoutCurrent = knapsackRecursive(values, weights,
        capacity, index - 1);

    // 选择当前物品的情况
    int withCurrent = 0;
```

```

if (currentWeight <= capacity) {
    withCurrent = currentValue + knapsackRecursive(values,
        weights, capacity - currentWeight, index - 1);
}

// 返回不选择和选择当前物品中的较大值
return max(withoutCurrent, withCurrent);
}

int knapsack(vector<int>& values, vector<int>& weights, int
capacity) {
    // 物品的个数
    int n = values.size();

    // 调用递归函数求解
    return knapsackRecursive(values, weights, capacity, n - 1);
}

```

自然，时间复杂度也就到了 $O(2^n)$ ，这是我们所不能接受的。

我们来看动态规划的做法，首先设状态数组 $dp[i][j]$ 代表将前 i 件物品装进限重为 j 的背包可以获得的最大价值，其中 $0 \leq i \leq N, 0 \leq j \leq W$ 。

现在我们考虑状态转移方程，假设我们已经处理好了前 $i - 1$ 个物品，那么对于第 i 个物品，当其不放入背包时，背包的剩余容量不变，背包中物品的总价值也不变，因此目前的最大价值为 $dp[i - 1][j]$ ，当其放入背包时，背包的剩余容量会减小 w_i ，背包中物品的总价值会增大 v_i ，故这种情况的最大价值为 $dp[i - 1][j - w[i]] + v[i]$ 。

于是，对于 0-1 背包问题的状态转移方程为：

$$dp[i][j] = \max(dp[i - 1][j], dp[i - 1][j - w[i]] + v[i])$$

同时，我们需要时刻注意，只有当 $j \geq w_i$ 时才可以放入背包。

// 创建一个二维数组 dp ，用于保存子问题的解

```

vector<vector<int>> dp(n + 1, vector<int>(capacity + 1, 0));
// 填充 dp 数组
for (int i = 1; i <= n; ++i) {
    int currentWeight = weights[i - 1];
    int currentValue = values[i - 1];
    for (int j = 1; j <= capacity; ++j) {
        if (currentWeight <= j) {
            dp[i][j] = max(dp[i - 1][j], currentValue + dp[i - 1][j - currentWeight]);
        } else {
            dp[i][j] = dp[i - 1][j];
        }
    }
}
// 返回背包中物品的最大总价值
return dp[n][capacity];

```

0-1 背包问题的特征较为明显：存在一个确定的上限 W ，价值量 v 和对上限的消耗 w ，例如下面这道题：

例 12.3.2 (NOIP2005 普及组-采药). 辰辰是个天资聪颖的孩子，他的梦想是成为世界上最伟大的医师。为此，他想拜附近最有威望的医师为师。

医师为了判断他的资质，给他出了一个难题。

医师把他带到一个到处都是草药的山洞里对他说：“孩子，这个山洞里有一些不同的草药，采每一株都需要一些时间，每一株也有它自身的价值。我会给你一段时间，在这段时间里，你可以采到一些草药。如果你是一个聪明的孩子，你应该可以让采到的草药的总价值最大。”

如果你是辰辰，你能完成这个任务吗？

输入格式

第一行有 2 个整数 T ($1 \leq T \leq 1000$) 和 M ($1 \leq M \leq 100$)，用一个空格隔开， T 代表总共能够用来采药的时间， M 代表山洞里的草药的数目。

接下来的 M 行每行包括两个在 1 到 100 之间（包括 1 和 100）的整数，分别表示采摘某株草药的时间和这株草药的价值。

```
70 3  
71 100  
69 1  
1 2
```

输出格式

输出在规定的时间内可以采到的草药的最大总价值。

```
3
```

我们首先来分析例题12.3.2的题干，重要信息包括：山洞里有一些**不同的草药**，采不同的药需要**不同的时间**，**不同的药有自身的价值**，并且需要在**限定期限里得到最大价值**。

显然这是一个**0-1 背包问题**：限定的时间即为背包容量，采药消耗的时间为每个物品的体积，每种药有价值，于是我们可以得出做法：

```
#include <bits/stdc++.h>  
  
using namespace std;  
  
int main() {  
    int T, M;  
    cin >> T >> M;  
  
    vector<int> time(M);  
    vector<int> value(M);  
    for (int i = 0; i < M; i++) {  
        cin >> time[i] >> value[i];  
    }  
  
    vector<int> dp(T + 1, 0);
```

```
for (int i = 0; i < M; i++) {
    for (int j = T; j >= time[i]; j--) {
        dp[j] = max(dp[j], dp[j - time[i]] + value[i]);
    }
}
cout << dp[T] << endl;
return 0;
}
```

相信您也一定已经发现，我们在内存循环中用的都是**逆序循环**，即从大到小不断减少背包容量，这是因为我们需要用**逆序循环**的方法来避免重复选取。

使用逆序循环在背包问题中是一个常见的技巧，尤其是当我们处理 0-1 背包问题时，使用逆序循环的原因主要有两点：

1. **避免重复计算**：在我们的动态规划转移方程中，我们需要确保在考虑选择当前物品时**不会重复使用该物品**。

假设我们使用正序循环，那么在某一轮内部的计算中，可能会重复使用当前的物品，而使用逆序循环可以确保我们只使用一次当前物品。

2. **确保依赖关系**：动态规划的核心在于状态转移，我们的当前状态可能依赖于之前的状态。

这或许难以理解，我们不妨换一种说法，首先我们来看二维数组的状态转移方程：

```
dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - w[i]] + v[i])
```

根据二维的状态转移方程，我们不难发现，其实对于**二维数组的背包**来说，正序和逆序是无所谓的，因为你把状态都保存了下来，如果要想知道 $dp[i][j]$ ，你就需要从 $dp[i - 1][j]$ 和 $dp[i - 1][j - w[i]] + v[i]$ 两个状态转移而来。

而一维数组的背包是会覆盖之前的状态的，因而需要使用逆序循环。我们来看一维数组的状态转移方程：

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

其中， $dp[j]$ 表示在执行 i 次循环后（此时已经处理 i 个物品），前 i 个物体放到容量 j 的背包时的最大价值，即之前的 $dp[i][j]$ 。

与二维相比较，它删去了第一维，但是二者表达的含义是相同的，只不过 $dp[j]$ 一直在重复使用，这带来的影响就是第 i 次循环会覆盖第 $i-1$ 次循环的结果。

仅仅从状态转移方程的角度来说，这其中有许多对应相等的关系，比如 $dp[i-1][j]$ 和 $dp[j]$ 就是相等的，这是为什么呢？

我们来简单求一下这几个值：

- 前 $i-1$ 个物品放到容量 j 的背包中带来的收益

$$dp[i-1][j]$$

由于在执行第 i 次循环时， $dp[j]$ 存储的是前 i 个物体放到容量为 j 的背包时的最大价值，在求前 i 个物体放到容量 j 时的最大价值（即之前的 $dp[i][j]$ ）时，我们正在执行第 i 次循环， $dp[v]$ 的值还是在第 $i-1$ 次循环时存下的值，在此时取出的 $dp[j]$ 就是前 $i-1$ 个物体放到容量 j 的背包时的最大价值，即 $dp[i-1][j]$ 。

- 前 $i-1$ 件物品放到容量为 $j - w[i]$ 的背包中带来的收益

$$dp[i][j - w[i]] + v[i]$$

由于在执行第 i 次循环前， $dp[0 \dots V]$ 中保存的是第 $i-1$ 次循环的结果，也就是前 $i-1$ 个物体分别放到容量 $0 \dots V$ 时的最大价值，即 $dp[i-1][0 \dots V]$ ，则在执行第 i 次循环前， dp 数组中 $j - w[i]$ 的位置存储就是我们要找的前 $i-1$ 件物品放到容量为 $j - w[i]$ 的背包中带来的收益（即之前的 $dp[i][j - w[i]]$ ）。

由于在执行 j 时还没执行到 $j - w[i]$ ，因此 $dp[j - w[i]]$ 保存的还是第 $i - 1$ 次循环的结果，即在执行第 i 次循环且背包容量为 j 时，此时的 $dp[j]$ 存储的是 $dp[i-1][j]$ ， $dp[j - w[i]]$ 存储的是 $dp[i][j - w[i]]$ 。

相反，如果在执行第 i 次循环时，背包容量按照 $0 \cdots V$ 的顺序遍历一遍来检测第 i 件物品是否能放。此时在执行第 i 次循环且背包容量为 j 时，此时的 $dp[j]$ 存储的是 $dp[i - 1][j]$ ，但是，此时 $dp[j - w[i]]$ 存储的是 $dp[i][j - w[i]]$ 。因为 $j > j - w[i]$ ，所以第 i 次循环中，执行背包容量为 j 时，容量为 $j - w[i]$ 的背包已经计算过了，即 $dp[j - w[i]]$ 中存储的是 $dp[i][j - w[i]]$ ，它会从一开始就装入某个物品，只是为了价值最大，重复是肯定要存在的，这就意味着对于 01 背包问题，按照增序枚举背包容量是不对的。

12.3.2 完全背包问题

完全背包是 0-1 背包问题的一个变种，与 0-1 背包不同就是每种物品可以有无限多个。我们修改问题 12.3.1 的描述即可得到完全背包问题：

例 12.3.3 (完全背包问题). 有 n 种物品和一个容量为 W 的背包，每种物品有无限多个且包含重量 w_i 和价值 v_i 两种属性，要求选若干物品放入背包使背包中物品的总价值最大且背包中物品的总重量不超过背包的容量。

尽管问题有所改动，但我们的目标没变，所以我们仍可定义与 01 背包问题几乎完全相同的状态，用 $dp[i][j]$ 表示将前 i 种物品装进限重为 j 的背包可以获得的最大价值。

考虑初始状态，我们仍然可以将 $dp[0][0 \dots W]$ 初始化为 0，表示将没有物品装入书包的最大价值为 0。那么当 $i > 0$ 时 $dp[i][j]$ 也有两种情况：

- 如果不装入第 i 种物品，即 $dp[i - 1][j]$ ，这与 0-1 背包相同；
- 如果装入第 i 种物品，与 01 背包不同，因为每种物品有无限个（但注意背包容量有限），所以此时不应该转移到 $dp[i - 1][j - w[i]]$ 而应

该转移到 $dp[i][j - w[i]]$ ，即装入第*i*种商品后还可以再继续装入第*j*种商品。

因而我们可以得到对应的状态转移方程：

$$dp[i][j] = \max(dp[i - 1][j], dp[i][j - w[i]] + v[i]) // j >= w[i]$$

我们来看一道例题：

例 12.3.4 (luogu-P1616 疯狂的采药). *LiYuxiang* 是个天资聪颖的孩子，他的梦想是成为世界上最伟大的医师。为此，他想拜附近最有威望的医师为师。

医师为了判断他的资质，给他出了一个难题。

医师把他带到一个到处都是草药的山洞里对他说：“孩子，这个山洞里有一些不同种类的草药，采每一种都需要一些时间，每一种也有它自身的价值。我会给你一段时间，在这段时间里，你可以采到一些草药。如果你是一个聪明的孩子，你应该可以让采到的草药的总价值最大。”

如果你是 *LiYuxiang*，你能完成这个任务吗？

此题和原题的不同点：

1. 每种草药可以无限制地疯狂采摘。
2. 药的种类眼花缭乱，采药时间好长好长啊！师傅等得菊花都谢了！

输入格式

输入第一行有两个整数，分别代表总共能够用来采药的时间 *t* 和代表山洞里的草药的数目 *m*。

第 2 到第 $(m + 1)$ 行，每行两个整数，第 $(i + 1)$ 行的整数 a_i, b_i 分别表示采摘第 *i* 种草药的时间和该草药的价值。

```
70 3
71 100
69 1
1 2
```

输出格式

输出一行，这一行只包含一个整数，表示在规定的时间内，可以采到的草药的最大总价值。

140

例题12.3.4就是一个标准的完全背包问题，题中明确表明每种药可以**无限地**采摘。

```
#include <bits/stdc++.h>

using namespace std;

int main() {
    int t, m;
    cin >> t >> m;

    vector<int> time(m + 1);
    vector<int> value(m + 1);
    for (int i = 1; i <= m; i++) {
        cin >> time[i] >> value[i];
    }

    vector<long long> dp(t + 1, 0);
    for (int i = 1; i <= m; i++) {
        for (int j = time[i]; j <= t; j++) {
            dp[j] = max(dp[j], dp[j - time[i]] + value[i]);
        }
    }
    cout << dp[t] << endl;
    return 0;
}
```

12.3.3 多重背包问题

多重背包也是 0-1 背包的一个变种，与 0-1 背包的区别在于每种物品 k 有 k_i 个，而非一个或无限个。我们同样修改问题12.3.1的描述即可得到多重背包问题：

例 12.3.5 (多重背包问题). 有 n 种物品和一个容量为 W 的背包，第 i 个物品的数量为 m_i 且每个物品包含重量 w_i 和价值 v_i 两种属性，要求选若干物品放入背包使背包中物品的总价值最大且背包中物品的总重量不超过背包的容量。

为了解决这个问题，我们有一个朴素的想法：由于第 i 个物品的数量为 m_i ，我们可以将之转换为有 m_i 个相同的物品，每个物品选一次，这样就将问题转换为了 0-1 背包问题。

此时我们得到状态转移方程：

$$dp[i][j] = \max(dp[i - 1][j - k * w[i]] + k * v[i])$$

其中， $k \leq \min(m[i], j / w[i])$ ，即当前剩余容量中该物品数量和全部装入该物品所能装下的最大个数中较小的那个值，我们需要从 0 开始枚举每一个这样的 k 。

我们还是通过一个例题来体会多重背包问题：

例 12.3.6 (luogu-1776 宝物筛选). 终于，破解了千年的难题。小 FF 找到了王室的宝物室，里面堆满了无数价值连城的宝物。

这下小 FF 可发财了，嘎嘎。但是这里的宝物实在是太多了，小 FF 的采集车似乎装不下那么多宝物。看来小 FF 只能含泪舍弃其中的一部分宝物了。

小 FF 对洞穴里的宝物进行了整理，他发现每样宝物都有一件或者多件。他粗略估算了下每样宝物的价值，之后开始了宝物筛选工作：小 FF 有一个最大载重为 W 的采集车，洞穴里总共有 n 种宝物，每种宝物的价值

为 v_i , 重量为 w_i , 每种宝物有 m_i 件。小 FF 希望在采集车不超载的前提下, 选择一些宝物装进采集车, 使得它们的价值和最大。

输入格式

第一行为一个整数 n 和 W , 分别表示宝物种数和采集车的最大载重。

接下来 n 行每行三个整数 v_i, w_i, m_i 。

```
4 20
3 9 3
5 9 1
9 4 2
8 1 3
```

输出格式

输出仅一个整数, 表示在采集车不超载的情况下收集的宝物的最大价值。

```
47
```

很显然, 例题12.3.6中, 对每个物品的数量就做了限制, 那么对于每个物品的取法就和完全背包问题有所区别, 我们直接来看做法:

```
#include <bits/stdc++.h>

using namespace std;

int main() {
    int n, W;
    cin >> n >> W;
    vector<int> v(n), w(n), m(n);
    for (int i = 0; i < n; i++) {
        cin >> v[i] >> w[i] >> m[i];
    }

    vector<int> dp(W + 1, 0);
```

```

// 对于每种物品
for (int i = 0; i < n; i++) {
    // 遍历该物品的数量，从 1 到 m[i]
    for (int k = 1; k <= m[i]; k++) {
        // 使用 0-1 背包的方法更新 dp 数组
        for (int j = W; j >= w[i]; j--) {
            dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
        }
    }
    cout << dp[W] << endl;
    return 0;
}

```

这里直接使用三重循环来解决多重背包问题，对于每种物品，我们遍历其可能的数量，然后使用 0-1 背包的方法更新 dp 数组。由于这种方法的时间复杂度是 $O(n \times W \times \max(m))$ ，其中 $\max(m)$ 是物品数量的最大值，在数据范围较大时可能会超时。

因此，除了一个个枚举外，我们还可以使用**二进制分组**进行优化，二进制分组优化基于一个结论：**任何正整数都可以表示为若干个 2 的幂的和**。在多重背包问题中，我们可以将物品划分为若干个 2 的幂，这样可以减少对 k 枚举的次数。

特别地，若 $k_i + 1$ 不是 2 的整数次幂，则需要在最后添加一个由 $k_i - 2^{\lfloor \log_2(k_i+1) \rfloor - 1}$ 个单个物品组合的大物品用于补足。

```

#include <bits/stdc++.h>

using namespace std;

int main() {
    int n, W;
    cin >> n >> W;

```

```
vector<int> v(n), w(n), m(n);
for (int i = 0; i < n; i++) {
    cin >> v[i] >> w[i] >> m[i];
}

vector<int> dp(W + 1, 0);
for (int i = 0; i < n; i++) {
    int k = 1;
    while (m[i] > 0) {
        int num = min(k, m[i]);
        for (int j = W; j >= num * w[i]; j--) {
            dp[j] = max(dp[j], dp[j - num * w[i]] + num * v[i]);
        }
        m[i] -= num;
        k *= 2;
    }
}
cout << dp[W] << endl;
return 0;
}
```

这种方法的好处在于，它将时间复杂度从 $O(n \times W \times \max(m))$ 降低到了 $O(n \times W \times \log(\max(m)))$ ，这在很多情况下都是一个很大的改进。

12.3.4 混合背包

混合背包就是将前面三种的背包问题混合起来，有的只能取一次，有的能取无限次，有的只能取有限次。

由于多重背包是三种问题的组合，我们也需要根据不同的问题给出解决方法，例如下面这道题：

例 12.3.7 (luogu-1833 樱花). 爱与愁大神后院里种了 n 棵樱花树，每棵都有美学值 $C_i(0 \leq C_i \leq 200)$ 。爱与愁大神在每天上学前都会来赏花。

爱与愁大神可是生物学霸，他懂得如何欣赏樱花：一种樱花树看一遍过，一种樱花树最多看 $P_i(0 \leq P_i \leq 100)$ 遍，一种樱花树可以看无数遍。但是看每棵樱花树都有一定的时间 $T_i(0 \leq T_i \leq 100)$ 。

爱与愁大神离去上学的时间只剩下一小会儿了。求解看哪几棵樱花树能使美学值最高且爱与愁大神能准时（或提早）去上学。

输入格式

共 $n + 1$ 行：

第 1 行：现在时间 T_s （几时：几分），去上学的时间 T_e （几时：几分），爱与愁大神院子里有几棵樱花树 n 。这里的 T_s, T_e 格式为：`hh:mm`，其中 $0 \leq hh \leq 23$, $0 \leq mm \leq 59$ ，且 hh, mm, n 均为正整数。

第 2 行到第 $n + 1$ 行，每行三个正整数：看完第 i 棵树的耗费时间 T_i ，第 i 棵树的美学值 C_i ，看第 i 棵树的次数 P_i ($P_i = 0$ 表示无数次， P_i 是其他数字表示最多可看的次数 P_i)。

```
6:50 7:00 3
2 1 0
3 3 1
4 5 4
```

输出格式

只有一个整数，表示最大美学值。

```
11
```

例题12.3.7中每棵树给定的参数 P 可以理解为常规背包问题描述中的物品个数，当 $P = 0$ 时，这是**完全背包问题**，可以无限的获取；当 $p = 1$ 时，这是**0-1 背包问题**，只能获取一次；当 $p = k$ 时，这是**多重背包问题**，只能获取 k 次。因此，我们可以进行判断分别处理：

```
for (int i = 1; i <= n; i++) { // n为物品个数
    if (P == 0) {
        // 完全背包处理方法...
    } else if (P == 1) {
        // 0-1背包处理方法...
    } else {
        // 多重背包处理方法...
    }
}
```

但是，我们在分析多重背包问题时选择的做法是将第 i 个物品的数量 m_i 视为有 m_i 个相同的物品，每个物品选一次，从而将问题转化为了 0-1 背包问题，因此我们可以将 $P \geq 1$ 合并为一种情况，并通过二进制分组进行优化：

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

const int N = 10005;
int dp[N];

int main() {
    int h1, m1, h2, m2, n;
    // 直接获取时间
    scanf("%d:%d %d:%d %d", &h1, &m1, &h2, &m2, &n);
    int total_time = (h2 - h1) * 60 + (m2 - m1);

    // 对于每棵樱花树，根据其观赏次数进行处理并更新dp数组
    for(int i = 1; i <= n; i++) {
        int T, C, P;
        cin >> T >> C >> P;

        // P = 0, 完全背包问题
        if(P == 0) {
```

```

    for(int j = T; j <= total_time; j++) {
        dp[j] = max(dp[j], dp[j - T] + C);
    }
} else {
    // 否则，进行二进制分组优化
    for(int k = 1; k <= P; k *= 2) { // k每次乘2
        P -= k; // 减去已处理的观赏次数
        for(int j = total_time; j >= k * T; j--) {
            dp[j] = max(dp[j], dp[j - k * T] + k * C);
        }
    }
    // 处理剩余的观赏次数
    for(int j = total_time; j >= P * T; j--) {
        dp[j] = max(dp[j], dp[j - P * T] + P * C);
    }
}
cout << dp[total_time] << endl;
return 0;
}

```

12.3.5 二维费用背包

二维费用背包问题也是 0-1 背包的一个变种，不同在于除了空间的消耗外还有一个消耗，我们仍然修改问题12.3.1的描述即可得到二维费用背包问题：

例 12.3.8 (二维费用背包问题). 有 n 种物品和一个容量为 W 的背包，你有 S 元的活动经费，每个物品包含重量 w_i 和价值 v_i 两种属性，将第 i 个物品买入需要 s_i 元，要求选若干物品放入背包使背包中物品的总价值最大，背包中物品的总重量不超过背包的容量且不能超过给定的活动经费。

对于这类问题，通常我们定义状态为： $dp[i][j][k]$ ，代表在前 i 个物品都考虑过后，在剩余 j 元和 k 的剩余空间的情况下能得到的最大价值。

紧接着我们考虑状态转移方程，对于第 i 件物品，只有取与不取两种选择，在剩余的时间和空间都足够的情况下考虑取与不取两种情况的较大值，从而得到本次的状态转移方程：

```
dp[i][j][k] =
    max(dp[i - 1][j][k], dp[i - 1][j - s[i]][k - w[i]] + v[i])
```

于是我们可以得到相应的解决方法：

```
for(int i = 1; i <= n; i++) {
    for(int j = 0; j <= S; j++) {
        for(int k = 0; k <= W; k++) {
            if(j >= s[i] && k >= w[i]) {
                dp[i][j][k] = max(dp[i - 1][j][k], dp[i - 1][j - s[i]][k - w[i]] + v[i]);
            }
        }
    }
}
```

然而当数据量很大时，多开一维用于存储物品编号的情况往往会导致超出内存限制。因此，更为常用的解决方法是使用**二维数组的两个维度分别代表两种消耗**，例如在本题中我们可以设 $dp[j][k]$ 代表在总费用不超过 j 且背包容量不超过 k 的情况下可以获得的最大价值。

这种情况下，我们可以得到新的状态转移方程：

```
dp[j][k] = max(dp[j][k], dp[j - s[i]][k - w[i]] + v[i])
```

亦即只要当前的空间与经费足够，我们就可以比较第 i 个物品取或不取两种情况下的收益较大值，于是这种情况下的解决方法如下：

```
for(int i = 1; i <= n; i++) {
    // 注意这里要逆序循环，避免物品被重复选取
```

```

for(int j = S; j >= s[i]; j--) {
    for(int k = W; k >= w[i]; k--) {
        dp[j][k] =
            max(dp[j][k], dp[j - s[i]][k - w[i]] + v[i]);
    }
}
}

```

我们以一道例题来体会二维费用背包问题：

例 12.3.9 (luogu-1855 榨取 kkksc03). 有 n 个任务需要完成，完成第 i 个任务需要花费 t_i 分钟，消耗 m_i 元。

现在一共有 T 分钟时间， M 元钱来处理这些任务，求最多能完成多少任务。

输入格式

第一行三个整数 n, M, T ，表示一共有 $n(1 \leq n \leq 100)$ 个愿望，kkksc03 的手上还剩 $M (0 \leq M \leq 200)$ 元，他的暑假有 $T (0 \leq T \leq 200)$ 分钟时间。

第 $2 n + 1$ 行 m_i, t_i 表示第 i 个愿望所需要的金钱和时间。

```

6 10 10
1 1
2 3
3 2
2 5
5 2
4 3

```

输出格式

一行，一个数，表示 kkksc03 最多可以实现愿望的个数。

我们来分析例题12.3.9，对于每一个任务，我们有两个选择：完成或不完成，这是0-1背包问题的特征，于是我们参考0-1背包问题的状态定义方法：定义二维数组 $dp[i][j]$ 表示在有*i*元钱和*j*分钟时间的条件下，最多可以完成的任务数量。

如果我们选择完成这个任务，那么我们需要从 $i - m_i$ 元钱和 $j - t_i$ 分钟时间的状态转移到 i 元钱和 j 分钟时间的状态，并增加一个任务数量；如果我们选择不完成这个任务，那么状态不变。于是我们也不难得出状态转移方程：

$$dp[i][j] = \max(dp[i][j], dp[i - cost[i]][j - time[j]] + 1)$$

于是我们也得出了本题的解决方案：

```

    }
}

}

cout << dp[M][T] << endl;
return 0;
}

```

12.3.6 分组背包

分组背包同样也是 0-1 背包的变种，我们直接来看描述：

例 12.3.10 (分组背包). n 种物品和一个容量为 W 的背包，每个物品包含重量 w_i 和价值 v_i 两种属性，此外每个物品还有一个代表其组号的编号 g_i ，同组内最多只能选择一个物品。要求选若干物品放入背包使背包中物品的总价值最大，背包中物品的总重量不超过背包的容量且不能超过给定的活动经费。

这类问题与 0-1 背包问题的主要区别在于从在所有物品中选择一件变成了从当前组中选择一件，于是我们只需要根据组号，对每一组采用 0-1 背包问题的解决方法。具体如下面这个题目：

例 12.3.11 (luogu-1757 通天之分组背包). 一天，小 A 去远游，却发现他的背包不同于 01 背包，他的物品大致可分为 k 组，每组中的物品相互冲突。现在，他想知道最大的利用价值是多少。

输入格式

两个数 m, n ，表示一共有 n 件物品，总重量为 m 。

接下来 n 行，每行 3 个数 a_i, b_i, c_i ，分别表示物品的重量，利用价值，所属组数。

```

45 3
10 10 1

```

```
10 5 1  
50 400 2
```

输出格式

一个数，最大的利用价值。

```
10
```

这里我们直接给出例题12.3.11的解决方案：

```
#include<bits/stdc++.h>  
  
using namespace std;  
  
const int MAXN = 205, MAXM = 10001;  
  
int main(){  
    int W, n;  
    cin >> W >> n;  
    int w[MAXM], v[MAXM], gruop[MAXM]; // gruop[i] 代表第 i 组有一  
    共多少个物品  
    int G; // 一共有 G 组物品  
    int item[MAXN][MAXN]; // item[i][j] 代表第 i 组第 j 个物品  
    for(int i = 1; i <= n; i++){  
        int x; // 当前物品的组号  
        cin >> w[i] >> v[i] >> x;  
        G = max(G, x); // 更新最大组号  
        gruop[x]++;
        item[x][gruop[x]] = i; // 将物品 i 加入到组 x
    }
    int dp[MAXM];
    for(int i = 1; i <= G; i++){ // 按组号遍历
        for(int j = W; j >= 0; j--){
            for(int k = 1; k <= gruop[i]; k++){
                if(j >= w[item[i][k]]){
```

```

        dp[j] = max(dp[j], dp[j - w[item[i][k]]] + v
                     [item[i][k]]);
    }
}
}
}

cout << dp[W] << endl;
return 0;
}

```

不难发现，对于分组背包，我们处理每个物品属性变得繁杂了许多，但核心的动态规划部分与 0-1 背包几乎一样，区别仅仅在于遍历的条件从物品编号变成了组号下的物品编号。

12.4 区间 DP

区间 DP 是指在考虑动态规划的状态定义的时候定义的是某个区间的整体状态，是线性 DP 的扩展。

我们直接来看一道例题：

例 12.4.1 (acwing-282 石子合并). 设有 N 堆石子排成一排，其编号为 $1, 2, 3, \dots, N$ 。每堆石子有一定的质量，可以用一个整数来描述，现在要将这 N 堆石子合并成为一堆。每次只能合并相邻的两堆，合并的代价为这两堆石子的质量之和，合并后与这两堆石子相邻的石子将和新堆相邻，合并时由于选择的顺序不同，合并的总代价也不相同。

例如有 4 堆石子分别为 $1, 3, 5, 2$ ，我们可以先合并 1、2 堆，代价为 4，得到 $4\ 5\ 2$ ，再合并 1、2 堆，代价为 9，得到 $9\ 2$ ，再将之合并得到 11，总代价为 $4 + 9 + 11 = 24$ ；如果第二步是先合并 2、3 堆，则代价为 7，得到 $4\ 7$ ，最后一次合并代价为 11，总代价为 $4 + 7 + 11 = 22$ 。

问题是：找出一种合理的方法，使总的代价最小，输出最小代价。

输入格式

第一行一个数 N 表示石子的堆数 N 。

第二行 N 个数，表示每堆石子的质量 (均不超过 1000)。

4

1 3 5 2

输出格式

输出一个整数，表示最小代价。

22

我们来看例题12.4.1，题意要求我们每次合并相邻的两堆石子，我们可以换种理解方式，即每次选择相邻的两个数相加，最终得到最小的和。作为动态规划问题，我们需要从两个角度来思考：1. 状态定义，2. 状态转移。

针对这个问题，我们首先需要搞清楚如何合理的定义状态的表示，在这里，我们通常使用二维数组 $\text{dp}[i][j]$ 表示区间 $[i, j]$ 内所有元素合并的最大值 (最小值，根据题意选择)。对于例题12.4.1，我们就定义 $\text{dp}[i][j]$ 为将区间内石子两两合并、最终合并为一堆所消耗的最小成本。

紧接着我们考虑状态转移方程，由于我们定义的是区间 $[i, j]$ 内所有元素合并的最小值，我们考虑合并过程：

- 两堆石子， $\{i, j\}$ 的最小值 = $s_i + s_j$ (其中， s_i 代表第 i 堆石子的重量，下同)；
 - 三堆石子， $\{i, k, j\}$ 的最小值 = $\min((s_i + s_k) + s_j, s_i + (s_k + s_j))$ ；
 - 四堆石子， $\{i, k, l, j\}$ 的最小值 = $\min(((s_i + s_k) + s_l) + s_j, (s_i + s_k) + (s_l + s_j), (s_i + (s_k + s_l)) + s_j, s_i + ((s_k + s_l) + s_j), s_i + (s_k + (s_l + s_j)))$ ；
- 这个表达式很长，但其实际意义很简单：任取除左右端点外的任意一点将石子分为两组，如果组中只有一堆石子，那么不需要合并，成本为 0；如果组中有两堆石子，直接合并，成本为两堆石子的质量和；如

果组中有三堆石子，则需要再次进行分组，直至组中只有一堆或两堆石子。

•

总结这个过程，我们可以得到状态转移方程：

```
every k from i to j - 1:  
dp[i][j] = min(dp[i][k] + dp[k + 1][j] + cost)
```

其中，`cost`为合并当前两堆需要的成本。

于是我们不难得出本题的解决方法：

```
#include <bits/stdc++.h>  
  
using namespace std;  
  
const int N = 310, INF = 1e9;  
  
int main() {  
    int n;  
    cin >> n;  
    int s[N];  
    for (int i = 1; i <= n; i++) {  
        cin >> s[i];  
        s[i] += s[i - 1]; // 前缀和  
    }  
    int dp[N][N];  
    for (int len = 2; len <= n; len++) {  
        for (int i = 1; i + len - 1 <= n; i++) {  
            int j = i + len - 1;  
            dp[i][j] = INF;  
            for (int k = i; k < j; k++) {  
                dp[i][j] = min(dp[i][j], dp[i][k] + dp[k + 1][j]  
                               + s[j] - s[i - 1]);  
            }  
        }  
    }  
    cout << dp[1][n] << endl;  
}
```

```
        }
    }
}

cout << dp[1][n] << endl;
return 0;
}
```

程序中我们使用前缀和来简化区间求和，外层遍历每一个长度 len ，从 1 到 n ，内层遍历每一个起点 i ，计算终点 $j = i + len - 1$ ，然后遍历 k 从 i 到 $j - 1$ ，在 k 处分割并计算。

此外，我们来逐层分析循环，外层循环：

```
for (int len = 2; len <= n; len++)
```

这个循环是为了枚举所有可能的区间长度，由于合并至少需要两堆石子，所以长度从 2 开始。中层循环：

```
for (int i = 1; i + len - 1 <= n; i++)
```

这个循环是为了枚举区间的起始位置， i 是区间的起始位置， $i + len - 1$ 是区间的结束位置，确保整个区间在 1 到 n 的范围内。内层循环：

```
for (int k = i; k < j; k++)
```

这个循环是为了枚举所有可能的分割点，我们尝试在每一个位置 k 将区间 $[i, j]$ 分割成两个子区间 $[i, k]$ 和 $[k + 1, j]$ ，然后计算合并这两个子区间的成本。这么做的目的在于：由于我们每次只能合并相邻的石子，这自然地引入了一个区间的概念，我们需要考虑所有可能的区间和合并顺序。通过从小到大枚举区间长度，我们可以确保在计算一个区间的代价时，所有更小的子区间的代价都已经被计算过了。在此基础上，我们将区间 $[i, i + len - 1]$ 分割为 $[i, k]$ 和 $[k + 1, i + len - 1]$ 两份， k 为 $[i, i + len - 1]$ 中的每个点，计算合并的最小值，以此完成所有区间的计算。

例题12.4.1的题目描述比较简单，明确了石子拍成一排，但存在一类题目，描述的区间是成环的，例如下面这道题：

例 12.4.2. 有一串环形的能量项链，每个项链上有 N 颗能量珠。每颗能量珠都有一个头标记和一个尾标记，这些标记都是正整数。相邻的两颗珠子可以合并成一颗新的珠子，并释放出能量。合并的能量计算方式是：前一颗珠子的头标记 \times 前一颗珠子的尾标记 \times 后一颗珠子的尾标记。新珠子的头标记和尾标记分别是前一颗珠子的头标记和后一颗珠子的尾标记。

目标是找到一个合并顺序，使得释放的总能量最大。

输入格式

第一行是一个正整数 N ($4 \leq N \leq 100$)，表示项链上珠子的个数。

第二行是 N 个用空格隔开的正整数，所有的数均不超过 1000。第 i 个数为第 i 颗珠子的头标记 ($1 \leq i \leq N$)，当 $i < N$ 时，第 i 颗珠子的尾标记应该等于第 $i + 1$ 颗珠子的头标记。第 N 颗珠子的尾标记应该等于第 1 颗珠子的头标记。

```
4
```

```
2 3 5 10
```

输出格式

一个正整数 E ($E \leq 2.1 \times 10^9$)，为一个最优聚合顺序所释放的总能量。

```
710
```

对于这类区间成环的问题，我们可以将环断裂为一个链表，但由于环上的每一个位置都可以将其作为断点，从而得到一个线性序列，我们就需要重复执行 n 次。当环断开之后，就变成了一个链表，其处理方式就与之前的问题几乎相同，这里给出解决方案：

```
#include <bits/stdc++.h>

using namespace std;
```

```
const int N = 210;
int ring[N], dp[N][N];

int solve(int start, int n) {
    for (int len = 2; len <= n; len++) {
        for (int i = start; i < start + n; i++) {
            int j = i + len - 1;
            if (j >= start + n) break;
            for (int k = i; k < j; k++) {
                dp[i][j] = max(dp[i][j], dp[i][k] + dp[k+1][j] +
                    ring[i] * ring[k + 1] * ring[j + 1]);
            }
        }
    }
    return dp[start][start+n-1];
}

int main() {
    int n;
    cin >> n;
    for (int i = 1; i <= n; i++) {
        cin >> ring[i];
        ring[i + n] = ring[i]; // 复制项链
    }

    int res = 0;
    for (int i = 1; i <= n; i++) {
        res = max(res, solve(i, n));
    }
    cout << res << endl;
    return 0;
}
```

在这里我们将求能量的方案抽成了一个函数，枚举每个点为起点的情况，可以发现**solve**函数与例题12.4.1基本相似。但由于我们枚举了每个可能的起点，一共需要枚举 n 次导致时间复杂度变成了 $O(n^4)$ ，在数据量大的情况下可能会存在超时，因此我们再来看一种解决方案。

回顾之前的做法，我们将区间复制了一份衔接在原区间的后面，并从第一个点开始向后枚举断裂点，其实我们可以利用好这个复制的区间，因为第 i 处的数值与第 $i + n$ 初的数值是一样的，我们只需要正常使用用动态规划求解，选取 $dp[1][n]、dp[2][n + 1]、\dots、dp[n - 1][2 * n - 2]$ 中的最大值即是最终结果，具体实现如下：

```
#include <bits/stdc++.h>

using namespace std;

const int MAXN = 205;

int main() {
    int n;
    cin >> n;
    int ring[MAXN];
    for (int i = 1; i <= n; i++) {
        cin >> ring[i];
        ring[i + n] = ring[i]; // 为了处理循环，我们复制一份到
                               // 后面
    }
    long long dp[MAXN][MAXN];
    for (int len = 2; len <= n; len++) { // len表示区间长度
        for (int i = 1; i <= 2 * n - len + 1; i++) {
            int j = i + len - 1;
            for (int k = i; k < j; k++) {
```

```

        dp[i][j] = max(dp[i][j], dp[i][k] + dp[k + 1][j]
                      + ring[i] * ring[k + 1] * ring[j + 1]);
    }
}
}

long long ans = 0;
for (int i = 1; i <= n; i++) {
    ans = max(ans, dp[i][i + n - 1]);
}
cout << ans << endl;
return 0;
}

```

其余不变，中层循环：`for (int i = 1; i <= 2 * n - len + 1; i++)`用来枚举线性序列中所有可能的子区间的起始位置，起始位置为 1，因为珠子的编号从 1 开始，结束位置为 $2 \times n - len + 1$ ，这是为了确保子区间的长度为 len 并且完整地位于线性序列中。考虑到我们已经将原始的项链复制了一遍，所以线性序列的总长度为 $2 \times n$ ，而为了保证子区间 $[i, i + len - 1]$ 能完整地位于这个线性序列中， i 的最大值应该是 $2 \times n - len + 1$ 。

最后通过一个 `for` 循环遍历比较最大值，避免了循环嵌套，从而保持时间复杂度仍是原本的 $O(n^3)$ 。

12.5 数位统计 DP

数位统计 DP，顾名思义，即关注数据的每个数位上的值，一种特定的动态规划方法，主要用于解决与数位数有关的问题。

数位是指将一个数字按照个、十、百、千等位拆开，关注它每一位上的数字。例如，数字 20489 的数位为 {9, 8, 4, 0, 2}。通常的数位统计 DP 问题包含以下特点：

- 目标是统计满足某些条件的数字的数量。

- 这些条件可以使用数位的思想来判断。
- 输入通常提供一个数字区间作为统计的限制。
- 上界很大，使得暴力枚举不可行。

回顾人类计数的方式，最朴素的计数就是从小到大开始依次加一。但对于多位数加法，这种计数中有许多重复的部分，例如，从 7000 到 7999、从 8000 到 8999 和从 9000 到 9999 的计数过程非常相似，这些过程都是后三位从 000 变到 999。而动态规划思想的特点就是将这些相似的过程合并，将答案存储在一个通用的数组中，以便后续使用，这也是数位统计 DP 的基本原理。我们通过一道题体会这个过程：

例 12.5.1 (luogu-2602 数字计数). 给定两个整数 a 和 b ，求 a 和 b 之间的所有数字中 $0 \sim 9$ 的出现次数。

输入格式

仅包含一行两个整数 a, b ，含义如上所述。

```
1 99
```

输出格式

包含一行十个整数，分别表示 $0 \sim 9$ 在 $[a, b]$ 中出现了多少次。

```
9 20 20 20 20 20 20 20 20 20
```

问题12.5.1很简洁，计算给定范围内 $[a, b]$ 的所有整数中，每个数码 ($0 \sim 9$) 各出现了多少次。为了解决这个问题，我们来分析数字的构成，数字由多个数位组成的，例如数字 20489 由五个数位 $\{2, 0, 4, 8, 9\}$ 组成，这种结构使我们可以分别考虑每一位数字，而不是整个数字。同时，由于计数方式决定了后面的位数相对于前面的位数变动次数会更多，因此考虑数字的高位比考虑低位更为简单。例如，考虑数字 20489 的前三位 204 比考虑整个数字更为简单。

此外，对于每一位数字，我们可以计算小于当前位的数字出现的次数，然后计算当前位数字出现的次数。这种方法利用了数字的结构和性质，使我们可以快速统计每一位数字出现的次数。假设我们要统计数字 1 到 204 中每个数码出现的次数：

1. 考虑个位

- 数字 1 到 9：每个数码 1 ~ 9 各出现 1 次；
- 数字 10 到 19：0 出现 1 次 (10)，1 ~ 9 各出现 1 次；
- 数字 20 到 29：0 出现 1 次 (20)，1 ~ 9 各出现 1 次；
-
- 数字 200 到 204：0 出现 1 次 (200)，1 ~ 4 各出现 1 次。

综上，个位上 1、2、3、4 出现 21 次，0、5、6、7、8、9 出现 20 次。

2. 考虑十位

- 数字 0 到 9，一位数，不考虑；
- 数字 10 到 19：1 出现 10 次；
- 数字 20 到 29：2 出现 10 次；
-
- 数字 100 到 109：0 出现 10 次；
-
- 数字 200 到 204：0 出现 5 次。

综上，十位上 0 出现 15 次，1 ~ 9 各出现 20 次。

3. 考虑百位

- 数字 100 到 199：1 出现 100 次；

- 数字 200 到 204: 2 出现 5 次。

综上，百位上 1 出现 100 次，2 出现 5 次。

根据这个过程，我们不难发现除 200 ~ 204 仅有五个数外，出现在第 i 数位上的数字将出现 10^i 次（从个位开始），最后我们可以得出本轮统计的结果：

0	1	2	3	4	5	6	7	8	9
35	141	46	41	41	40	40	40	40	40

这个过程相较于我们直接讨论 20489 中个十、百、千、万五个数位简化了不少，同时也体现出从高到低位讨论的优越性，即在计算过程中会遇到许多重复的子问题，例如数字 20489 和 20478 的前三位都是 204。因此，我们就可以存储已经计算过的结果，避免重复计算。

当然，对于动态规划问题，我们需要解决一个首要问题，即**状态数组如何定义**。在这个题目中，状态是通过数位来表示的，具体地说，我们关心的是数字的每一位上的数码，以及这一位之前的所有数码。本题中，我们设 $dp[i]$ 表示的是**考虑到第 i 位时，小于当前位的数字出现的次数**。以数字 20489 为例，当我们考虑到数位 3 时，实际上我们关心的是数字 204，而不是整个数字 20489。另一方面，当我们考虑数位 3 时，当前位的数字是 4，因此，我们关心的是在这一位上数字 0 ~ 3 出现了多少次。同时，我们还需要一些数组来辅助：使用 $power[i]$ 代表 10 的 i 次方，用于计算每一位上数码实际次数；使用 $ans[j]$ 存储在给定范围内数码 j 出现的次数；使用 $numbers[i]$ 存储数字的每一位。

既然与状态相关的是数位，那么状态的转移自然就是如何从当前数位转到下一个数位，在此给出状态转移方程：

$$dp[i] = dp[i - 1] * 10 + power[i - 1];$$

相信并不难理解，其中 $dp[i - 1] * 10$ 表示当我们从第 $i - 1$ 位转移到第 i 位时，之前的每个数码都会出现 10 倍的次数， $+ power[i - 1]$ 则表示，当我们考虑到第 i 位时，小于当前位的每个数字都会额外出现 10^{i-1} 次。

至此，给出问题12.5.1的解决方案：

```
#include <bits/stdc++.h>

using namespace std;

const int N = 15;

long long a, b, dp[N];

// 10的i次方。
vector<long long> power(N);
// 存储在两个数字范围内每个数码出现的次数。
vector<long long> ans1(N), ans2(N);
// 存储数字的每一位。
vector<long long> nums(N);

void solve(long long n, vector<long long>& ans) {
    long long tmp = n;
    int len = 0;

    while (n) {
        nums[++len] = n % 10;
        n /= 10;
    }

    for (int i = len; i >= 1; --i) {
        // 对于每个数字，考虑其在比当前位更高位上的出现次数
        for (int j = 0; j < 10; j++) {
            ans[j] = ans[j] + dp[i - 1] * nums[i];
        }
        // 对于当前位数字小于nums[i]的每个数字，加上它们在当前位
        // i上的出现次数
        for (int j = 0; j < nums[i]; j++) {
            ans[j] = ans[j] + power[i - 1];
```

```
    }

    // 处理当前位数字出现的次数
    tmp = tmp - power[i - 1] * nums[i];
    ans[nums[i]] = ans[nums[i]] + tmp + 1;
    // 处理前导零
    ans[0] = ans[0] - power[i - 1];
}

int main() {
    cin >> a >> b;
    power[0] = 1;
    for (int i = 1; i <= 13; ++i) {
        dp[i] = dp[i - 1] * 10 + power[i - 1];
        power[i] = 10 * power[i - 1];
    }
    // 计算在给定范围内每个数字出现的次数
    solve(b, ans1);
    solve(a - 1, ans2);
    for (int i = 0; i < 10; ++i) {
        cout << ans1[i] - ans2[i] << " ";
    }
    return 0;
}
```

例 12.5.2 (hdu-2089 不要 62). 杭州人称那些傻乎乎粘嗒嗒的人为 62 (音: *laoer*)。

杭州交通管理局经常会扩充一些的士车牌照，新近出来一个好消息，以后上牌照，不再含有不吉利的数字了，这样一来，就可以消除个别的士司机和乘客的心理障碍，更安全地服务大众。不吉利的数字为所有含有 4 或 62 的号码。例如：62315, 73418, 88914 都属于不吉利号码。但是，61152 虽

然含有 6 和 2，但不是 62 连号，所以不属于不吉利数字之列。

你的任务是，对于每次给出的一个牌照区间号，推断出交管局今次又要实际上给多少辆新的士车上牌照了。

输入格式

输入的都是整数对 $n, m (0 \leq n \leq m < 1000000)$ ，如果遇到都是 0 的整数对，则输入结束。

```
1 100
0 0
```

输出格式

对于每个整数对，输出一个不含有不吉利数字的统计个数，该数值占一行位置。

```
80
```

例题12.5.2是一个数位统计 DP 的模板题，在这个问题中，我们只需要关注数字 4 是否出现，或者当 2 出现时上一位数字是否是 6，因此我们定义状态数组为： $dp[pos][pre]$ ，其中 pos 代表当前位数， pre 代表上一位的数字是什么。

同样由高位向低位考虑，例如我们要求出 20489 以内符合条件的数，我们只需要考虑形如 0????、1????、2???? 且小于等于原数的符合条件的数的数量即可。对于本题来说，显然 104?? 和 204?? 范围内的符合条件的数是一样的，因此直接递归地搜索就可以遍历所有范围内的值。我们直接给出解决方案：

```
#include <bits/stdc++.h>

using namespace std;

const int maxn = 505;
```

```
int nums[maxn];
int dp[20][20];

int DFS(int pos, int pre, int limit) {
    // 搜索结束
    if (pos == 0) {
        return 1;
    }
    if (limit == 0 && dp[pos][pre] != -1) {
        return dp[pos][pre];
    }
    //是否有上界，如果有上界那么不能超过上界
    int top = limit ? nums[pos] : 9;
    int ans = 0;
    for (int i = 0; i <= top; i++) {
        if (i == 4) {
            continue;
        }
        if (pre == 6 && i == 2) {
            continue;
        }
        ans += DFS(pos - 1, i, limit && nums[pos] == i);
    }
    if (limit == 0) {
        dp[pos][pre] = ans;
    }
    return ans;
}

int solve(int x) {
    int len = 0;
    memset(dp, -1, sizeof(dp));
    while (x) {

```

```
        nums[++len] = x % 10;
        x /= 10;
    }
    return DFS(len, 0, 1); //从高位向低位枚举
}

int main() {
    int n, m;
    while (cin >> n >> m) {
        if (n == 0 && m == 0) {
            break;
        }
        cout << solve(m) - solve(n - 1) << endl; //即为 [n, m] 区间
    }
    return 0;
}
```

12.6 计数类 DP

计数类 DP 也是线性 DP 的一种扩展，其通常的目的是求出某一举动的所有方案数，例如在第五章中列举出来的青蛙跳台阶、机器人走迷宫都可以视为计数类 DP。

计数类 DP 的一道经典例题如下：

例 12.6.1 (acwing-900 整数拆分). 一个正整数 n 可以表示成若干个正整数之和，形如： $n = n_1 + n_2 + \dots + n_k$ ，其中 $n_1 \geq n_2 \geq \dots \geq n_k, k \geq 1$ 。

我们将这样的一种表示称为正整数 n 的一种划分。

现在给定一个正整数 n ，请你求出 n 共有多少种不同的划分方法。

输入格式

共一行，包含一个整数 n 。

5

输出格式

共一行，包含一个整数，表示总划分数量。

由于答案可能很大，输出结果请对 $10^9 + 7$ 取模。

7

对于这个问题，我们可以视作一个完全背包问题，即一个容量为 n 的背包，有 $1, 2, \dots, n$ 种体积的物品，每种物品有无限个，求多少种方案使得背包恰好能够装满。在这种思路下，我们定一状态 $\text{dp}[i][j]$ 为只从前 i 个数中选，数的和恰好为 j 的所有集合的数量。那么显然，由于每种物品的数量无限，我们从小到大考虑这些物品时，最后一次的操作必然是选择了第 i 件物品 x 次。即 $\text{dp}[i][j] = \text{dp}[i - 1][j - x * i]$ ，到这里相信不难发现，在前 $i - 1$ 个物品选定的情况下，第 i 个物品的数量也被决定完，因此宏观来说，我们需要的状态转移如下：

```
dp[i][j] = dp[i - 1][j] + dp[i - 1][j - i] + dp[i - 1][j - 2 * i]
        + ... + dp[i - 1][j - x * i]
```

而另一方面：

```

dp[i][j - i] = dp[i - 1][j - i] + dp[i - 1][j - 2 * i] + ... +
               dp[i - 1][j - x * i]

```

于是我们可以合并得到 $dp[i][j] = dp[i - 1][j] + dp[i][j - 1]$ 。

此外，有一点需要强调，对于计数类 DP，不选也是一种选取的方案，即 $dp[i][0] = 1$ 。至此，我们得出这个问题的解决方案：

```
#include <bits/stdc++.h>

using namespace std;

const int N = 1010, mod = 1e9 + 7;
```

```
int main() {
    int n;
    cin >> n;
    int dp[N][N];
    for (int i = 0; i <= n; ++i) {
        dp[i][0] = 1;
    }
    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j <= n; ++j) {
            dp[i][j] = dp[i - 1][j];
            if (j >= i) {
                dp[i][j] = (dp[i][j] + dp[i][j - i]) % mod;
            }
        }
    }
    cout << dp[n][n] << endl;
    return 0;
}
```

在这里同样可以把第一位去掉，用滚动数组来存答案。

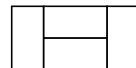
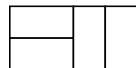
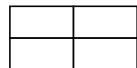
12.7 状态压缩 DP

状态压缩类 DP 通常会将状态压缩为整数来达到优化转移的目的。例如下面这个题目：

例 12.7.1 (acwing-291 蒙德里安的梦想). 求把 $N \times M$ 的棋盘分割成若干个 1×2 的长方形，有多少种方案。

例如当 $N = 2, M = 4$ 时，共有 5 种方案。当 $N = 2, M = 3$ 时，共有 3 种方案。如下图所示：

$N = 2$
 $M = 4$



$N = 2$
 $M = 3$



输入格式

输入包含多组测试用例。

每组测试用例占一行，包含两个整数 N 和 M 。

当输入用例 $N = 0, M = 0$ 时，表示输入终止，且该用例无需处理。

```
1 2
1 3
1 4
2 2
2 3
2 4
2 11
4 11
0 0
```

输出格式

每个测试用例输出一个结果，每个结果占一行。

```
1
0
1
2
3
5
144
51205
```

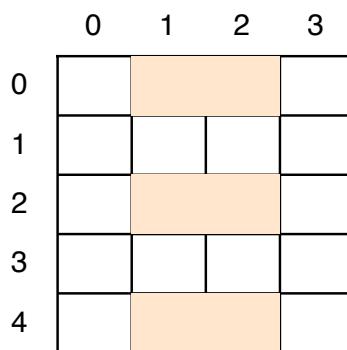
很显然，对于例题12.7.1，其含义为用若干个 1×2 的长方形覆盖满整个棋盘，那么如果我们先放横着的方块，那么我们可以确保每一行都被完

整地覆盖 (方块宽度为 1)，不会留下任何空隙。这样，我们可以将问题简化为：**如何在每一行中放置横着的方块**，当我们已经放置了所有的横着的方块后，剩下的空隙只能由竖着的方块来填充。因此，我们只需要计算放置横着的方块的方案数，就可以得到总的方案数。

为了验证方案是否能够达到预期，关键在于看所有剩余的位置能否用竖着的方块填充满。那么验证的方法就是：遍历每一列，确保每一列中空着的 1×1 的方块是**连续的偶数个**。

这类问题通常会采用**二进制压缩**的方式解决，例如本题中，我们定义状态数组 $\text{dp}[i][j]$ 代表已经将前 $i - 1$ 列摆好，且从第 $i - 1$ 列，伸出到第 i 列的状态是 j 的所有方案数，并且其中 j 是一个二进制数，用来表示哪一行的小方块是**横着放的**，其位数和棋盘的行数一致。

到这里为止或许难以理解，我们来看一个例子：设 $i = 2$, $j = 10101_2$ ，所以这里的 $\text{dp}[2][21]$ (21 是 10101 的十进制) 表示的是前 i 列摆完之后，从第 $i - 1$ 列延伸到第 i 列的状态是 10101 的方案数。具体如下图所示：



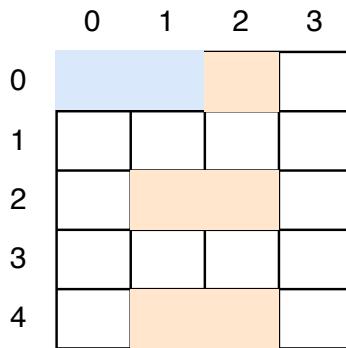
根据图片，对于第 2 列来说， 10101_2 中，二进制中 1 的位置代表第 0、2、4 行存在从第 0 列因横置而延伸过来的方块，二进制中 0 的位置则代表这一列没有从第 0 行延伸过来的方块。

确定了状态，就需要考虑状态如何转移，既然第 i 列固定了，则看第 $i - 2$ 列是怎么转移到到第 $i - 1$ 列的。假设此时对应的状态是 k (第 $i - 2$ 列到第 $i - 1$ 列伸出来的二进制数，设为 00100_2)。此时对应的方案数

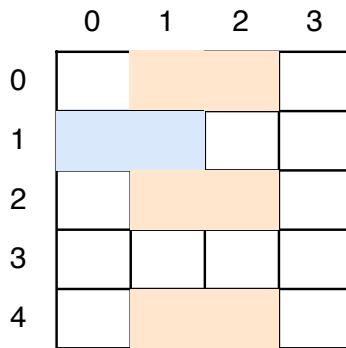
是 $\text{dp}[i - 1][k]$ ，即前 $i - 2$ 列都已摆完，且从第 $i - 2$ 列伸到第 $i - 1$ 列的状态为 k 的所有方案数。

那么，这个 k 需要满足什么条件呢？

首先如图所示， k 不能和 j 在同一行，因为从 $i - 1$ 列到第 i 列是横着摆放的 1×2 的方块，那么 $i - 2$ 列到 $i - 1$ 列就不能是横着摆放的，否则会存在方块堆叠的情况，与题意矛盾，所以 k 和 j 不能位于同一行。



既然不能同一行伸出来，我们用逻辑与运算 $(k \& j) == 0$ 来检查，如果 $(k \& j) == 0$ 表示 k 和 j 没有冲突。那么显然，合法的情况如下图所示：



第 $i - 1$ 列到第 i 列是横着摆的，和第 $i - 2$ 列到第 $i - 1$ 列横着摆的都确定了，那么第 $i - 1$ 列空着的格子就确定了，这些空着的格子将来用作竖着放。那么当某一列有空着的位置时，该列所有连续的空着的位置长度必须是偶数。综上可以得到解决方案：

```
#include <bits/stdc++.h>

using namespace std;

const int N = 12, M = 1 << N;

long long dp[N][M];

// 合法的状态
vector<int> state[M];
// 判断状态是否合法
bool st[M];

int main() {
    int m, n;
    while (cin >> n >> m) {
        if (n == 0 && m == 0) {
            break;
        }
        for (int i = 0; i < 1 << n; ++i) {
            int count = 0;
            bool isValid = true;
            for (int j = 0; j < n; ++j) {
                if (i >> j & 1) { // 当前位是1
                    // 判断之前的0是否是奇数个
                    if (count % 2 != 0) {
                        isValid = false;
                        break;
                    }
                    count = 0;
                } else {
                    count++;
                }
            }
            if (!isValid) {
                cout << "NO" << endl;
            } else {
                cout << "YES" << endl;
            }
        }
    }
}
```

```
        }

    }

    if (count % 2 != 0) {
        isValid = false;
    }

    st[i] = isValid;
}

// 枚举所有行的状态

for (int i = 0; i < 1 << n; ++i) {
    state[i].clear();
    // 暴力枚举每种方案
    for (int j = 0; j < 1 << n; ++j) {
        // i & j == 0 两列没有冲突
        if ((i & j) == 0 && st[i | j]) {
            state[i].push_back(j);
        }
    }
}

memset(dp, 0, sizeof dp);
dp[0][0] = 1;

for (int i = 1; i <= m; ++i) {
    // 对于每个状态
    for (int j = 0; j < 1 << n; ++j) {
        // 遍历合法状态
        for (int k = 0; k < state[j].size(); k++) {
            dp[i][j] = dp[i][j] + dp[i - 1][state[j][k]];
        }
    }
}

cout << dp[m][0] << endl;
```

```

    }
    return 0;
}

```

其中, $st[i \mid j]$ 用以进行状态的预处理, $st[i \mid j]$ 是一个位运算, 它表示 i 和 j 中任何一个为 1 的位, 这实际上是合并了 i 和 j 的状态。 $st[i \mid j]$ 检查合并后的状态是否有效, 即不存在奇数个连续 0 的情况。

我们不妨举一个简单的例子, 有一个 3×3 的棋盘, 每一列都有 2^3 种可能, 假设我们想要知道状态 $i = 101_2$ (即第 1 和第 3 行有瓷砖的一部分) 和状态 $j = 010_2$ (即第 2 行有瓷砖的一部分) 是否兼容。那么我们首先要判断 $i \& j = 101 \& 010 = 000$, 即方块没有重叠。接下来使用 $j \mid k$ 进行位运算, 结果是 $101 \mid 010 = 111$, 这表示当我们考虑从状态 i 转移到状态 j 时, 整个列的状态是 111_2 , 即这一列完全被瓷砖覆盖。而当 $i = 100_2, j = 010_2$ 时, $i \mid j = 100 \mid 010 = 110$, 这就表示当前状态合并后存在奇数个 0(第三行), 因此不是合法的状态。

综上所述, $st[j \mid k]$ 的作用是检查合并后的状态是否有效, 在上述例子中, $st[111] = \text{true}$ 但 $st[110] = \text{false}$ 。

12.8 树形 DP

树形 DP 指的是在树上进行的 DP 操作, 考虑到树的构建及树的性质, 树形 DP 通常采用递归进行。

接下来是一道例题:

例 12.8.1 (luogu-1352 没有上司的舞会). 某大学有 n 个职员, 编号为 $1 \sim N$ 。他们之间有从属关系, 也就是说他们的关系就像一棵以校长为根的树, 父结点就是子结点的直接上司。

现在有个周年庆宴会, 宴会每邀请来一个职员都会增加一定的快乐指数 a_i , 但是呢, 如果某个职员的直接上司来参加舞会了, 那么这个职员就

无论如何也不肯来参加舞会了。所以，请你编程计算，邀请哪些职员可以使快乐指数最大，求最大的快乐指数。

输入格式

输入的第一行是一个整数 n 。

第 2 到第 $(n + 1)$ 行，每行一个整数，第 $(i + 1)$ 行的整数表示 i 号职员的快乐指数 r_i 。

第 $(n + 2)$ 到第 $2n$ 行，每行输入一对整数 l, k ，代表 k 是 l 的直接上司。

```
7
1
1
1
1
1
1
1
1
1 3
2 3
6 4
7 4
4 5
3 5
```

输出格式

输出一行一个整数代表最大的快乐指数。

```
5
```

数据规模与约定

对于 100% 的数据，保证 $1 \leq n \leq 6 \times 10^3$, $-128 \leq r_i \leq 127$, $1 \leq l, k \leq n$ ，且给出的关系一定是一棵树。

对于例题12.8.1很像前文提到的**打家劫舍 III**, 都可以将问题简化为: 一棵二叉树, 树上的每个点都有对应的权值, 每个点有选中和不选中两种状态, 问在不能同时选中有父子关系的点的情况下, 能选中的点的最大权值和是多少。

对于**打家劫舍 III** 这个问题, 由于所有的节点都是非负数, 所以对于每棵子树来说都只有两种情况: 1、取根节点则相邻的两个字节点必须舍弃, 2、舍弃根节点则相邻的两个字节点 (如果有) 需要纳入考虑。因此在这种情况下我们可以用 $f[node]$ 表示选择 $node$ 节点的情况下, 节点的子树上被选择的节点的最大权值和; 用 $g[node]$ 表示不选择 $node$ 节点的情况下, 节点的子树上被选择的节点的最大权值和; 用 $left$ 和 $right$ 代表 $node$ 节点的左右孩子。这种情况下的状态转移为:

```
// 对于每个节点 node
f[node] = node.val + g[node.left] + g[node.right];
g[node] = max(f[node.left], g[node.left]) +
          max(f[node.right], g[node.right]);
value[node] = max(f[node], g[node]);
```

那么对于例题12.8.1, 尽管存在节点的值为负, 我们在考虑的时候仍然是当节点 $node$ 被选取的情况下, 必须舍弃其相邻的两个字节点, 当节点 $node$ 不被选入, 我们则需要继续考虑两个字节点作为子树的根节点是否需要考虑, 因此我们仍可以用同样的方式完成。

```
#include "bits/stdc++.h"

using namespace std;

struct Node {
    int val;
    Node *left;
    Node *right;
}
```

```
Node() : val(0), left(nullptr), right(nullptr) {}

Node(int x) : val(x), left(nullptr), right(nullptr) {}

Node(int x, Node *left, Node *right) : val(x), left(left),
    right(right) {}

};

unordered_map <Node*, int> f, g;

void dfs(Node* node) {
    if (!node) {
        return;
    }
    dfs(node->left);
    dfs(node->right);
    f[node] = node->val + g[node->left] + g[node->right];
    g[node] = max(f[node->left], g[node->left]) + max(f[node->
        right], g[node->right]);
}

int solve(Node* root) {
    dfs(root);
    return max(f[root], g[root]);
}

int main() {
    int n;
    cin >> n;
    vector<Node*> nodes(n + 1);
    vector<bool> hasParent(n + 1, false);
    nodes[0] = NULL; // 0号位不用
    for (int i = 1; i <= n; ++i) {
```

```
int value;
cin >> value;
nodes[i] = new Node(value);
}

for (int i = 0; i < n - 1; ++i) {
    int son, fa;
    cin >> son >> fa;
    if (!nodes[fa]->left) {
        nodes[fa]->left = nodes[son];
    } else {
        nodes[fa]->right = nodes[son];
    }
    hasParent[son] = true;
}
Node* root = NULL;
for (int i = 1; i <= n; ++i) {
    if (!hasParent[i]) {
        root = nodes[i];
        break;
    }
}
dfs(root);

cout << max(f[root], g[root]) << endl;
}
```

但是这个程序存在一个问题，那就是在定义的时候限定了只能是二叉树，例如输入为：

```
8
6
1
1
```

```
1  
1  
1  
1  
1  
2 1  
3 1  
4 1  
5 1  
6 1  
7 1  
8 1
```

预期的结果是 7 但实际得到的是 6，这是因为在建树时，当第一次遇到 1 的子节点会直接将其设置为左子节点，随后遇到的 1 的子节点都将被视为 1 的右子节点，且这个过程是覆盖，因而出错。修改也很简单，例如我们在不知道每个根节点到底有多少个子节点的情况下，我们不妨在结构体中定义可变数组vector，修改后的程序如下：

```
#include "bits/stdc++.h"  
  
using namespace std;  
  
struct Node {  
    int val;  
    std::vector<Node*> son;  
  
    Node(int x) : val(x) {}  
};  
  
unordered_map <Node*, int> f, g;  
  
void dfs(Node* node) {
```

```
if (!node) {
    return;
}
for (int i = 0; i < node->son.size(); i++) {
    Node* child = node->son[i];
    dfs(child);
    f[node] += g[child];
    g[node] += max(f[child], g[child]);
}
f[node] += node->val;
}

int solve(Node* root) {
    dfs(root);
    return max(f[root], g[root]);
}

int main() {
    int n;
    cin >> n;
    vector<Node*> nodes(n + 1);
    vector<bool> hasParent(n + 1, false);
    nodes[0] = NULL; // 0号位不用
    for (int i = 1; i <= n; ++i) {
        int value;
        cin >> value;
        nodes[i] = new Node(value);
    }

    for (int i = 0; i < n - 1; ++i) {
        int son, fa;
        cin >> son >> fa;
        nodes[fa]->son.push_back(nodes[son]);
    }
}
```

```
    hasParent[son] = true;
}

Node* root = NULL;
for (int i = 1; i <= n; ++i) {
    if (!hasParent[i]) {
        root = nodes[i];
        break;
    }
}
dfs(root);

cout << max(f[root], g[root]) << endl;
}
```

此外，我们还可以通过定义`dp[i][j]`来定义状态，其中`i`表示第`i`个节点，`j`有两个值`0/1`表示取与不取。

同时，我们给出12.8.1的另一种做法，即常规的用数组来构建树的做法：

```
#include <bits/stdc++.h>

#define ll long long

using namespace std;

const int MAXN = 20007;

ll dp[MAXN][2], happy[MAXN];
// 子节点
vector<ll> children[MAXN];
bool has_parent[MAXN];

// 深度优先搜索，用于计算每个节点的最大快乐值
void dfs(ll node) {
    dp[node][0] = 0; // 不参加聚会时的快乐值
```

```
dp[node][1] = happy[node]; // 参加聚会时的快乐值

for (ll child : children[node]) {
    dfs(child);
    dp[node][0] += max(dp[child][0], dp[child][1]); // 当前
    // 节点不参加
    dp[node][1] += dp[child][0]; // 当前节点参加
}
}

int main() {
    ll n;
    scanf("%lld", &n);

    for (ll i = 1; i <= n; i++) {
        scanf("%lld", &happy[i]);
    }

    for (ll i = 1; i < n; i++) {
        ll parent, child;
        scanf("%lld%lld", &child, &parent);
        children[parent].push_back(child);
        has_parent[child] = true;
    }
}

// 找到根节点（没有父节点的节点）
ll root = -1;
for (ll i = 1; i <= n; i++) {
    if (!has_parent[i]) {
        root = i;
        break;
    }
}
```

```

dfs(root);

printf("%lld\n", max(dp[root][0], dp[root][1]));

return 0;
}

```

实现过程中，我们用 $dp[i][j]$ 来存储当前节点的处理情况，每个节点仅有两种状态，用 $dp[i][0]$ 表示不选当前节点，用 $dp[i][1]$ 表示选当前节点，问题则变成了当前状况累加子节点的最优状况：

```

// 当前节点不参加
dp[node][0] += max(dp[child][0], dp[child][1]);
// 当前节点参加
dp[node][1] += dp[child][0];

```

还有另一类也是树形结构的问题，但这类问题通常不是一棵树，而是多棵树组成的森林，对于这类问题可以设置一个哨兵节点作为每棵树根节点的父节点，以此将森林转化为树。例如：

例 12.8.2 (P2014 选课). 在大学里每个学生，为了达到一定的学分，必须从很多课程里选择一些课程来学习，在课程里有些课程必须在某些课程之前学习，如高等数学总是在其它课程之前学习。

现在有 N 门功课，每门课有各自学分，每门课有一门或没有直接先修课 (若课程 a 是课程 b 的先修课即只有学完了课程 a ，才能学习课程 b)。

一个学生要从这些课程里选择 M 门课程学习，问他能获得的最大学分是多少？

输入格式

第一行有两个整数 N, M 用空格隔开。

接下来的 N 行，每行包含两个整数 k_i 和 s_i ， k_i 表示第 i 门课的直

接先修课, s_i 表示第 i 门课的学分。其中若 $k_i = 0$ 表示没有直接先修课 ($1 \leq k_i \leq N$, $1 \leq s_i \leq 20$)。

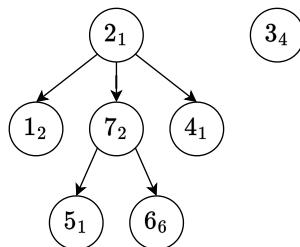
7	4
2	2
0	1
0	4
2	1
7	1
7	6
2	2

输出格式

只有一行, 选 M 门课程的最大得分。

13

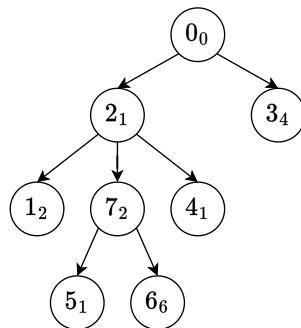
根据题目12.8.2中的输入, 我们可以得到如下的课程学习图:



由于要选择四个课程, 我们选择课程 2、课程 7、课程 6 和课程 3 共修道 13 学分达到最大。

对于树型 DP 问题来说, 我们每次考虑问题只考虑当前节点和其子节点之间的关系, 这道题实际上是将 0-1 背包问题与树进行了结合。同时, 上图中节点 2 有三个子节点, 也就构成了三棵子树, 如果我们将每课子树视为一个整体, 将选课数量视作背包容量, 那么每棵子树就有体积(课程数)和价值(学分数)两个属性, 于是我们不妨将之视作分组背包问题, 即每次从子树中选出一个出来。

在解决问题之前，我们先要对上图做一点修改，如前文所说，我们增加一个度数为 0 的哨兵节点，将所有独立的树的根节点指向它，从而将森林转化为树，且这个哨兵节点必选，于是问题就从原来的选 M 节课变成了选 $M + 1$ 节课：



因此，例题12.8.2的解决方案如下：

```

#include <bits/stdc++.h>

using namespace std;

const int MAXN = 407;

int n, m;
vector<int> tree[MAXN];
int score[MAXN], dp[MAXN][MAXN];

void solve(int x) {
    // 从1号点开始，0号点作为哨兵节点
    for (int i = 1; i <= m; ++i) {
        dp[x][i] = score[x];
    }
    for (int i = 0; i < tree[x].size(); i++) {
        int val = tree[x][i];
        solve(val); // 树形，往下去处理子树
    }
}
  
```

```
// 逆序遍历容量，更新dp，背包问题的技巧
for (int j = m; j > 0; j--) {
    for (int k = 0; k < j; k++) {
        // 状态转移怎么转移，把子树视为一个整体，整体就有体积
        // 对于当前这个课程，以及当前的容量
        dp[x][j] = max(dp[x][j], dp[x][j - k] + dp[val][k]);
    }
}
}

int main(){
    cin >> n >> m;
    int parent;
    for (int i = 1; i <= n; i++) {
        cin >> parent >> score[i];
        // 存入编号
        tree[parent].push_back(i);
    }
    m += 1; // 多选一门哨兵节点
    solve(0);
    cout << dp[0][m] << endl;
    return 0;
}
```

第十三章 计算几何

13.1 计算几何基础

工欲善其事，必先利其器，做计算几何的题目前也需要有一个模板，在这里我们遵守几个原则：

- 写全局函数而非类方法，结构体只存数据；
- 每个函数标注依赖于哪些函数，且尽量减少依赖；
- 用简略的名字，同时传值而非 const 引用。

13.1.1 几何对象

常用的几何对象包括点，向量，直线，线段和圆，我们通常采取如下的定义方式：

```
// 点
struct Point {
    double x, y;
};

// 向量
using Vec = Point;

// 直线
```

```
struct Line {
    Point P; Vec v;
};

// 线段(存两个端点)
struct Seg {
    Point A, B;
}

// 圆(存圆心和半径)
struct Circle {
    Point O; double r;
};
```

当然，向量和点并不是完全一样的，但很多时候这两者可以混用。

13.1.2 常用常量

```
const Point O = {0, 0}; // 原点

const Line Ox = {O, {1, 0}}, Oy = {O, {0, 1}}; // 坐标轴

const double PI = acos(-1);
```

13.1.3 基础操作

```
// 逆时针旋转90度的向量
Vec r90a(Vec v) {
    return {-v.y, v.x};
}

// 顺时针旋转90度的向量
```

```
Vec r90c(Vec v) {
    return {v.y, -v.x};
}

// 向量加向量
Vec operator+(Vec u, Vec v) {
    return {u.x + v.x, u.y + v.y};
}

// 向量减向量
Vec operator-(Vec u, Vec v) {
    return {u.x - v.x, u.y - v.y};
}

// 数乘
Vec operator*(double k, Vec v) {
    return {k * v.x, k * v.y};
}

// 点乘
double operator*(Vec u, Vec v) {
    return u.x * v.x + u.y * v.y;
}

// 叉乘
double operator^(Vec u, Vec v) {
    return u.x * v.y - u.y * v.x;
}

// 向量长度
double len(Vec v) {
    return sqrt(v.x * v.x + v.y * v.y);
}
```

```
// 斜率
double slope(Vec v) {
    return v.y / v.x;
}
```

13.1.4 向量

```
// 两向量的夹角余弦
// DEPENDS len, V*V
double cos_t(Vec u, Vec v) {
    return u * v / len(u) / len(v);
}

// 归一化向量(与原向量方向相同的单位向量)
// DEPENDS len
Vec norm(Vec v) {
    return {v.x / len(v), v.y / len(v)};
}

// 与原向量平行且横坐标大于等于0的单位向量
// DEPENDS d*V, len
Vec pnorm(Vec v) {
    return (v.x < 0 ? -1 : 1) / len(v) * v;
}

// 线段的方向向量
// DEPENDS V-V
Vec dvec(Seg l) {
    return l.B - l.A;
}
```

13.1.5 直线

```
// 两点式直线
// DEPENDS V-V
Line line(Point A, Point B) {
    return {A, B - A};
}

// 斜截式直线
Line line(double k, double b) {
    return {{0, b}, {1, k}};
}

// 点斜式直线
Line line(Point P, double k) {
    return {P, {1, k}};
}

// 线段所在直线
// DEPENDS V-V
Line line(Seg l) {
    return {l.A, l.B - l.A};
}

// 给定直线的横坐标求纵坐标
// NOTE 请确保直线不与y轴平行
double at_x(Line l, double x) {
    return l.P.y + (x - l.P.x) * l.v.y / l.v.x;
}

// 给定直线的纵坐标求横坐标
// NOTE 请确保直线不与x轴平行
double at_y(Line l, double y) {
```

```
    return l.P.x - (y + l.P.y) * l.v.x / l.v.y;
}

// 点到直线的垂足
// DEPENDS V-V, V*V, d*V
Point pedal(Point P, Line l) {
    return l.P - (l.P - P) * l.v / (l.v * l.v) * l.v;
}

// 过某点作直线的垂线
// DEPENDS r90c
Line perp(Line l, Point P) {
    return {P, r90c(l.v)};
}

// 角平分线
// DEPENDS V+V, len, norm
Line bisec(Point P, Vec u, Vec v) {
    return {P, norm(u) + norm(v)};
}
```

13.1.6 线段

```
// 线段的方向向量
// DEPENDS V-V
// NOTE 直线的方向向量直接访问属性v
Vec dvec(Seg l) {
    return l.B - l.A;
}

// 线段中点
Point midp(Seg l) {
```

```
    return {(l.A.x + l.B.x) / 2, (l.A.y + l.B.y) / 2};  
}  
  
// 线段中垂线  
// DEPENDS r90c, V-V, midp  
Line perp(Seg l) {  
    return {midp(l), r90c(l.B - l.A)};  
}
```

13.1.7 几何对象之间的关系

```
// 向量是否互相垂直  
// DEPENDS eq, V*V  
bool verti(Vec u, Vec v) {  
    return eq(u * v, 0);  
}  
  
// 向量是否互相平行  
// DEPENDS eq, V^V  
bool paral(Vec u, Vec v) {  
    return eq(u ^ v, 0);  
}  
  
// 向量是否与x轴平行  
// DEPENDS eq  
bool paral_x(Vec v) {  
    return eq(v.y, 0);  
}  
  
// 向量是否与y轴平行  
// DEPENDS eq  
bool paral_y(Vec v) {
```

```
    return eq(v.x, 0);
}

// 点是否在直线上
// DEPENDS eq
bool on(Point P, Line l) {
    return eq((P.x - l.P.x) * l.v.y, (P.y - l.P.y) * l.v.x);
}

// 点是否在线段上
// DEPENDS eq, len, V-V
bool on(Point P, Seg l) {
    return eq(len(P - l.A) + len(P - l.B), len(l.A - l.B));
}

// 两个点是否重合
// DEPENDS eq
bool operator==(Point A, Point B) {
    return eq(A.x, B.x) && eq(A.y, B.y);
}

// 两条直线是否重合
// DEPENDS eq, on(L)
bool operator==(Line a, Line b) {
    return on(a.P, b) && on(a.P + a.v, b);
}

// 两条线段是否重合
// DEPENDS eq, P==P
bool operator==(Seg a, Seg b) {
    return (a.A == b.A && a.B == b.B) || (a.A == b.B && a.B
        == b.A);
}
```

```
// 以横坐标为第一关键词、纵坐标为第二关键词比较两个点
// DEPENDS eq, lt
bool operator<(Point A, Point B) {
    return lt(A.x, B.x) || (eq(A.x, B.x) && lt(A.y, B.y));
}

// 直线与圆是否相切
// DEPENDS eq, V^V, len
bool tangency(Line l, Circle C) {
    return eq(abs((C.O ^ l.v) - (l.P ^ l.v)), C.r * len(l.v));
}

// 圆与圆是否相切
// DEPENDS eq, V-V, len
bool tangency(Circle C1, Circle C2) {
    return eq(len(C1.O - C2.O), C1.r + C2.r);
}
```

13.1.8 距离

```
// 两点间的距离
// DEPENDS len, V-V
double dis(Point A, Point B) {
    return len(A - B);
}

// 点到直线的距离
// DEPENDS V^V, len
double dis(Point P, Line l) {
    return abs((P ^ l.v) - (l.P ^ l.v)) / len(l.v);
```

```
}
```

```
// 平行直线间的距离
// DEPENDS d*V, V^V, len, pnorm
// NOTE 请确保两直线是平行的
double dis(Line a, Line b) {
    return abs((a.P ^ pnorm(a.v)) - (b.P ^ pnorm(b.v)));
}
```

13.1.9 平移和旋转

```
// 平移
// DEPENDS V+V
Line operator+(Line l, Vec v) {
    return {l.P + v, l.v};
}

Seg operator+(Seg l, Vec v) {
    return {l.A + v, l.B + v};
}

// 旋转
// DEPENDS V+V, V-V
Point rotate(Point P, double rad) {
    return {cos(rad) * P.x - sin(rad) * P.y, sin(rad) * P.x
        + cos(rad) * P.y};
}

// DEPENDS ~1
Point rotate(Point P, double rad, Point C) {
    return C + rotate(P - C, rad);
}

// DEPENDS ~1, ~2
Line rotate(Line l, double rad, Point C = 0) {
```

```
        return {rotate(l.P, rad, C), rotate(l.v, rad)};  
    }  
    // DEPENDS ~1, ~2  
    Seg rotate(Seg l, double rad, Point C = 0) {  
        return {rotate(l.A, rad, C), rotate(l.B, rad, C)};  
    }
```

13.1.10 对称

```
// 对称  
// 关于点对称  
Point reflect(Point A, Point P) {  
    return {P.x * 2 - A.x, P.y * 2 - A.y};  
}  
// DEPENDS ~1  
Line reflect(Line l, Point P) {  
    return {reflect(l.P, P), l.v};  
}  
// DEPENDS ~1  
Seg reflect(Seg l, Point P) {  
    return {reflect(l.A, P), reflect(l.B, P)};  
}  
  
// 关于直线对称  
// DEPENDS V-V, V*V, d*V, pedal  
// 向量和点在这里的表现不同，求向量关于某直线的对称向量需要  
// 用reflect_v  
// DEPENDS ~1  
Point reflect(Point A, Line ax) {  
    return reflect(A, pedal(A, ax));  
}  
// DEPENDS ~1, ~4
```

```

Vec reflect_v(Vec v, Line ax) {
    return reflect(v, ax) - reflect(0, ax);
}

// DEPENDS ^1, ^4, ^5
Line reflect(Line l, Line ax) {
    return {reflect(l.P, ax), reflect_v(l.v, ax)};
}

// DEPENDS ^1, ^4
Seg reflect(Seg l, Line ax) {
    return {reflect(l.A, ax), reflect(l.B, ax)};
}

```

需要注意的是，这里的点和向量表现是不同的。

13.1.11 交点

```

// 直线与直线交点
// DEPENDS eq, d*V, V*V, V+V, V^V
vector<Point> inter(Line a, Line b) {
    double c = a.v ^ b.v;
    if (eq(c, 0))
        return {};
    Vec v = 1 / c * Vec{a.P ^ (a.P + a.v), b.P ^ (b.P + b.v)}
    ;
    return {{v * Vec{-b.v.x, a.v.x}, v * Vec{-b.v.y, a.v.y}
    }};
}

// 直线与圆交点
// DEPENDS eq, gt, V+V, V-V, V*V, d*V, len, pedal
vector<Point> inter(Line l, Circle C) {
    Point P = pedal(C.O, l);
    double h = len(P - C.O);

```

```

if (gt(h, C.r))
    return {};
if (eq(h, C.r))
    return {P};
double d = sqrt(C.r * C.r - h * h);
Vec vec = d / len(l.v) * l.v;
return {P + vec, P - vec};
}

// 圆与圆的交点
// DEPENDS eq, gt, V+V, V-V, d*V, len, r90c
vector<Point> inter(Circle C1, Circle C2) {
    Vec v1 = C2.O - C1.O, v2 = r90c(v1);
    double d = len(v1);
    if (gt(d, C1.r + C2.r) || gt(abs(C1.r - C2.r), d))
        return {};
    if (eq(d, C1.r + C2.r) || eq(d, abs(C1.r - C2.r)))
        return {C1.O + C1.r / d * v1};
    double a = ((C1.r * C1.r - C2.r * C2.r) / d + d) / 2;
    double h = sqrt(C1.r * C1.r - a * a);
    Vec av = a / len(v1) * v1, hv = h / len(v2) * v2;
    return {C1.O + av + hv, C1.O + av - hv};
}

```

13.1.12 三角形的四心

```

// 三角形的重心
Point barycenter(Point A, Point B, Point C) {
    return {(A.x + B.x + C.x) / 3, (A.y + B.y + C.y) / 3};
}

// 三角形的外心

```

```
// DEPENDS r90c, V*V, d*V, V-V, V+V
// 给定圆上三点求圆，要先判断是否三点共线
Point circumcenter(Point A, Point B, Point C) {
    double a = A * A, b = B * B, c = C * C;
    double d = 2 * (A.x * (B.y - C.y) + B.x * (C.y - A.y) +
        C.x * (A.y - B.y));
    return 1 / d * r90c(a * (B - C) + b * (C - A) + c * (A -
        B));
}

// 三角形的内心
// DEPENDS len, d*V, V-V, V+V
Point incenter(Point A, Point B, Point C) {
    double a = len(B - C), b = len(A - C), c = len(A - B);
    double d = a + b + c;
    return 1 / d * (a * A + b * B + c * C);
}

// 三角形的垂心
// DEPENDS V*V, d*V, V-V, V^V, r90c
Point orthocenter(Point A, Point B, Point C) {
    double n = B * (A - C), m = A * (B - C);
    double d = (B - C) ^ (A - C);
    return 1 / d * r90c(n * (C - B) - m * (C - A));
}
```

第十四章 组合数学

14.1 生成函数

生成函数 (Generating Function, 也叫**母函数**) 是**组合数学**中一种重要的方法，它把**离散数列**与**形式幂级数**对应了起来。

接下来我们来看**普通生成函数** (Ordinary Generating Function, OGF) 的定义：

定义 14.1.1 (普通生成函数). 对于有限数列 $\{a_i\}(i = 0, 1, \dots, n)$, 我们定义其生成函数为 $\sum_{i=1}^n a_i x^i$, 对于无限数列 $\{a_i\}(i = 0, 1, \dots)$, 我们则定义其生成函数为 $\sum_{i=1}^{\infty} a_i x^i$ 。

根据定义14.1.1, 我们不难发现对于有限数列 $1, 3, 5$, 其生成函数为 $1 + 3x + 5x^2$, 对于无限数列 $1, 1, 1, \dots$ 其生成函数为 $1 + x + x^2 + \dots + x^n + \dots$, 我们不难发现, 数列的**相加**对应生成函数的**相加**, 数列的**卷积**对应生成函数的**相乘**。

上面的生成函数是用**级数**形式给出的, 但生成函数更有用的是它的**封闭**形式。例如, 生成函数 $1 + x + x^2 + \dots + x^n + \dots$, 考虑收敛域 $|x| < 1$, 则根据求和公式可以表示为 $\frac{1}{1-x}$ 。由于生成函数是形式幂级数, 我们不会往里面带入 x 的具体的值, 所以我们**不用考虑级数的敛散性**。

生成函数被广泛应用于组合计数中, 例如接下来一个场景:

例 14.1.2. 有若干种物品 I_1, I_2, \dots, I_m , 每种物品可以取的件数受到一定限制, 求取 n 件物品的总方案数。

我们令每个物品的集合对应一个数列 $\{a_i\}$, 表示取 i 件集合内物品的方案数为 a_i 。对于单个物品的集合, 它对应的数列中只存在 0 或 1, 表示该件数可选与否。

如果现在是两个集合的并集情况, 问题就相当于: 在集合 A 中取 i 件物品的方案有 a_i 种, 在集合 B 中取 j 件物品的方案有 b_j 种, 那么在 $A \cup B$ 中取 k 件物品的方案数为多少?

我们根据**乘法原理**和**加法原理**, 可以知道有 $\sum_{i+j=k} a_i x b_j$ 。这就是一个标准的卷积形式, 因此我们可以求出每件物品对应的生成函数¹, 然后把它们乘起来得到全集对应的生成函数。

我们来看具体的例子:

例 14.1.3. 有若干种物品 I_1, I_2, \dots, I_m , 每种物品只有 1 件, 求取 n 件物品的总方案数。

例中, 每种物品的生成函数都是 $1 + x$, 那么 m 种物品的生成函数就是 $(1 + x)^m$, 由二项式定理展开得到 $\sum_{i=0}^m C_m^i x^i$, 因此 n 次项系数为 C_m^n , 与组合数意义一致。

例 14.1.4. 有若干种物品 I_1, I_2, \dots, I_m , 每种物品可以取任意件, 求取 n 件物品的总方案数。

本例中, 每种物品的生成函数都是 $1 + x + x^2 + \dots = \frac{1}{1-x}$, 那么 m 种

¹这里生成函数的定义为: 如果有 k 种方法取 n 件物品, 则 n 次项系数为 k 。

物品的生成函数是 $(\frac{1}{1-x})^m$, 通过广义二项式定理展开:

$$\begin{aligned}
 (\frac{1}{1-x})^m &= \frac{1}{(1-x)^m} = (1-x)^{-m} \\
 &= \sum_{i=0}^{\infty} \frac{-m \cdot (-m-1) \cdots (-m-(i-1))}{i!} (-x)^i \\
 &= \sum_{i=0}^{\infty} \frac{(-1)^i \cdot m \cdot (m+1) \cdots (m+(i-1))}{i!} (-1)^i x^i \\
 &= \sum_{i=0}^{\infty} \frac{(m+i-1)!}{i!(m-1)!} x^i \\
 &= \sum_{i=0}^{\infty} C_{m+i-1}^{m-1} x^i
 \end{aligned}$$

因此 n 次项系数为 C_{m+n-1}^{m-1} 。

例 14.1.5 (砝码称重问题). 给你 1 克、2 克、3 克、4 克的砝码各一枚, 称出 1-10 克的方案分别有多少种?

在这里, 我们可以把 ω 克的砝码当作一种物品, 它只能取 0 件或 1 件, 那么它的生成函数就是 $1 + x^\omega$, 所以全集对应的生成函数为 $(1+x)(1+x^2)(1+x^3)(1+x^4)$, 展开得到:

$$x^{10} + x^9 + x^8 + 2x^7 + 2x^6 + 2x^5 + 2x^4 + 2x^3 + x^2 + x + 1$$

称出 k 克的方案数为该生成函数的 k 次项系数。

例 14.1.6 (整数划分问题). 一个正整数可以被用多少种方式划分为其他正整数的和?

例如, 4 可以被划分为 4、2+2、1+3、1+1+2、1+1+1+1。

我们把每个数字 k 当作一种物品, 它只能取 0 件、 k 件、 $2k$ 件 …, 即生成函数为 $1 + x^k + x^{2k} + \cdots = \frac{1}{1-x^k}$, 所有正整数 n 对应生成函数的乘积为 $\prod_{i=1}^{\infty} \frac{1}{1-x^i}$, 那么这个式子如何展开呢?

定理 14.1.7 (五边形数定理). $\prod_{i=1}^{\infty}(1-x^i) = \sum_{j=-\infty}^{\infty}(-1)^j x^{\frac{j(3j-1)}{2}}$, 这个式子可以进一步合并成:

$$\sum_{j=0}^{\infty}(-1)^j x^{\frac{j(3j\pm 1)}{2}}$$

即 $(1-x)(1-x^2)(1-x^3)(1-x^4)(1-x^5)\cdots = 1-x-x^2+x^5+x^7-x^{12}-x^{15}+x^{22}+x^{26}+\cdots$ 。

展开后, 有些次方项被消去, 只留下次方项为 $1, 2, 5, 7, 12, \dots$ 的项次, 留下来的次方恰为广义五边形数, 即 $\frac{j(3j\pm 1)}{2}$ 。

根据14.1.7, 设 $\prod_{k=1}^{\infty} \frac{1}{1-x^k} = \sum_{i=0}^{\infty} p(i)x^i$, 则有 $\sum_{i=0}^{\infty} p(i)x^i \sum_{j=0}^{\infty} (-1)^j x^{\frac{j(3j-1)}{2}} = 1$, 根据多项式乘法的法则可知其 k 次项系数为 $\sum_{j=0}^{\infty} (-1)^j p(k - \frac{j(3j\pm 1)}{2})$, 但当 $k > 0$ 时, 该式的值为 0, 于是我们移项得到:

$$\begin{aligned} p(k) &= \sum_{j \neq 0} (-1)^{j+1} \cdot p(k - \frac{j(3j \pm 1)}{2}) \\ &= p(k-1) + p(k-2) - p(k-5) - p(k-7) + \cdots \end{aligned}$$

我们通常会用生成函数来解决**递推数列**通项的问题, 例如:

例 14.1.8 (递推数列 I). $a_n = 5a_{n-1} - 6a_{n-2}$, $n \geq 2$, $a_1 = -2$, $a_0 = 1$ 。

本题可以乘**形式变量**来升降阶数, 具体解法如下:

设 $F(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n + \cdots$, 于是我们有:

$$F(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n + \cdots$$

$$-5xF(x) = -5a_0x - 5a_1x^2 - \cdots - 5a^{n-1}x^n - \cdots$$

$$6x^2F(x) = 6a_0x^2 + \cdots + 6a^{n-2}x^n - \cdots$$

左右求和，我们可以得到：

$$(1 - 5x + 6x^2)F(x) = a_0 + (a_1 - 5a_0)x + \sum_{i=2}^{\infty} (a_i - 5a_{i-1} + 6a_{i-2})x^i$$

显然 $\sum_{i=2}^{\infty} (a_i - 5a_{i-1} + 6a_{i-2})x^i = 0$ ，代入得 $(1 - 5x + 6x^2)F(x) = 1 + (-2 - 5)x$ ，进而，我们有：

$$\begin{aligned} F(x) &= \frac{1 - 7x}{1 - 5x + 6x^2} \\ &= \frac{1 - 7x}{(1 - 2x)(1 - 3x)} \\ &= \frac{5}{1 - 2x} - \frac{4}{1 - 3x} \\ &= 5 \sum_{i=0}^{\infty} 2^i x^i - 4 \sum_{i=0}^{\infty} 3^i x^i \\ &= \sum_{i=0}^{\infty} (5 \times 2^i - 4 \times 3^i) x^i \end{aligned}$$

根据生成函数定义，我们得到数列通项为 $a_n = 5 \times 2^n - 4 \times 3^n$ 。

本题可以作为生成函数求解递推数列通项的一个模板。

例 14.1.9 (递推数列 II). $a_n = 3a_{n-1} + 3^n$, $n \geq 1$, $a_0 = 2$ 。

我们同样设 $F(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n + \cdots$ ，于是我们有：

$$\begin{aligned} F(x) &= a_0 + a_1x + a_2x^2 + \cdots + a_nx^n + \cdots \\ -3xF(x) &= -3a_0x - 3a_1x^2 - \cdots - 3a_{n-1}x^n - \cdots \end{aligned}$$

左右相加，我们得到：

$$\begin{aligned}
 (1 - 3x)F(x) &= a_0 + (a_1 - 3a_0)x + \cdots + (a_n - a_{n-1})x^n + \cdots \\
 &= 2 + 3x + 3^2x^2 + \cdots + 3^n x^n + \cdots \\
 &= 1 + 1 + 3x + (3x)^2 + \cdots + (3x)^n + \cdots \\
 &= 1 + \frac{1}{1 - 3x}
 \end{aligned}$$

进而我们得到 $F(x) = \frac{1}{1-3x} + \frac{1}{(1-3x)^2}$ ，为了进行级数转化，我们这里需要插入讲解一下泰勒级数。

定理 14.1.10 (泰勒级数). 设 $f(x)$ 在定义域内任意阶可导，则：

$$\begin{aligned}
 f(x) &= f(0) + \frac{f'(0)}{1!}x + \frac{f''(0)}{2!}x^2 + \cdots + \frac{f^{(n)}(0)}{n!}x^n + \cdots \\
 &= f(0) + \sum_{i=1}^{\infty} \frac{f^{(i)}(0)}{i!}x^i
 \end{aligned}$$

回到题目，我们令 $g(x) = (1 - 3x)^{-2}$ ，于是我们有：

$$\begin{aligned}
 g(x) &= (1 - 3x)^{-2}, g(0) = 1; \\
 g'(x) &= 2 \times 3 \times (1 - 3x)^{-3}, g'(0) = 2 \times 3; \\
 g''(x) &= 2 \times 3 \times 3^2 \times (1 - 3x)^{-4}, g''(0) = 2 \times 3 \times 3^2; \\
 g'''(x) &= 2 \times 3 \times 4 \times 3^3 \times (1 - 3x)^{-5}, g'''(0) = 2 \times 3 \times 4 \times 3^3; \\
 &\dots\dots \\
 g^{(n)}(x) &= (n+1)! \cdot 3^n (1 - 3x)^{-(n+2)}, g^{(n)}(0) = (n+1)! \cdot 3^n
 \end{aligned}$$

于是，我们可以得到 $F(x) = \sum_{i=0}^{\infty} 3^i x^i + \sum_{i=0}^{\infty} (i+1)3^i x^i = \sum_{i=0}^{\infty} (i+2)3^i x^i$ ，那么根据生成函数定义，我们得到数列通项为 $a_n = (n+2)3^n$ 。

例 14.1.11 (水果问题). 把 n 个水果放进一个篮子里面，求方法数。其中水

果取法满足：

- 橘子个数是偶数；
- 香蕉个数不大于 3 个；
- 菠萝个数是 4 的倍数；
- 最多一个西瓜；
- 水果只有上面这 4 种。

这道题我们逐个来进行分析：

1. 由于橘子个数是偶数， n 个水果中橘子取 $x, 2x, 4x, \dots, 2nx, \dots$ ，于是我们可以得到橘子的生成函数： $1 + x^2 + x^4 + \dots + x^{2n} + \dots = \frac{1}{1-x^2}$ ；
2. 香蕉的要求是不大于 3 个，即 $1 + x + x^2 + x^3 = \frac{1-x^4}{1-x}$ ；
3. 菠萝的要求是 4 的倍数，即 $1 + x^4 + x^{4\times 2} + \dots + x^{4n} + \dots = \frac{1}{1-x^4}$ ；
4. 西瓜最多取一个，即 $1 + x$ 。

由于各种选取相互独立，由乘法原理得到：

$$\begin{aligned} F(x) &= \frac{1}{1-x^2} \cdot \frac{1}{1-x^4} \cdot \frac{1-x^4}{1-x} \cdot (1+x) \\ &= \frac{1}{(1-x)^2} \\ &= \sum_{i=0}^{\infty} (i+1)x^i \end{aligned}$$

因而，所有放水果的方法数是 $a(n) = n + 1$ 。

例 14.1.12 (BZOJ-3028 食物). 明明这次又要出去旅游了，和上次不同的是，他这次要去宇宙探险！

我们暂且不讨论他有多么 NC ，他又幻想了他应该带一些什么东西。理所当然的，你当然要帮他计算携带 n 件物品的方案数。

他这次又准备带一些受欢迎的食物，如：蜜桃多啦，鸡块啦，承德汉堡等等。当然，他又有一些稀奇古怪的限制，每种食物的限制如下：

- 承德汉堡：偶数个；
- 可乐：0 个或 1 个；
- 鸡腿：0 个，1 个或 2 个；
- 蜜桃多：奇数个；
- 鸡块：4 的倍数个；
- 包子：0 个,1 个,2 个或 3 个；
- 土豆片炒肉：不超过一个；
- 面包：3 的倍数个。

注意，这里我们懒得考虑明明对于带的食物该怎么搭配着吃，也认为每种食物都是以‘个’为单位（反正是幻想嘛），只要总数加起来是 n 就算一种方案。因此，对于给出的 n ，你需要计算出方案数，并对 10007 取模。

我们直接写出每类食物的生成函数：

- 承德汉堡： $1 + x^2 + x^4 + \cdots + x^{2n} + \cdots = \frac{1}{1-x^2}$ ；
- 可乐： $1 + x$ ；
- 鸡腿： $1 + x + x^2 = \frac{1-x^3}{1-x}$ ；
- 蜜桃多： $x + x^3 + x^5 + \cdots + x^{2n+1} + \cdots = \frac{x}{1-x^2}$ ；
- 鸡块： $1 + x^4 + x^8 + \cdots + x^{4n} + \cdots = \frac{1}{1-x^4}$ ；

- 包子: $1 + x + x^2 + x^3 = \frac{1-x^4}{1-x}$;
- 土豆片炒肉: $1 + x$;
- 面包: $1 + x^3 + x^6 + \cdots + x^{3n} + \cdots = \frac{1}{1-x^3}$ 。

有了每一项的生成函数之后, 利用乘法原理, 我们可以得到:

$$\begin{aligned} F(x) &= \frac{1}{1-x^2} \cdot (1+x) \cdot \frac{1-x^3}{1-x} \cdot \frac{x}{1-x^2} \cdot \frac{1}{1-x^4} \cdot \frac{1-x^4}{1-x} \cdot (1+x) \cdot \frac{1}{1-x^3} \\ &= \frac{x}{(1-x)^4} \end{aligned}$$

现在我们将 $\frac{1}{(1-x)^4}$ 展开:

$$\frac{1}{(1-x)^4} = 1 + 4x + \frac{4 \times 5}{2!} x^2 + \cdots + \frac{(n+3)!}{3! \times n!} x^n + \cdots = \sum_{i=0}^{\infty} \frac{(i+3)!}{3! \times i!} x^i$$

于是生成函数 $F(x)$ 就变成了:

$$\begin{aligned} F(x) &= x \cdot \frac{1}{(1-x)^4} \\ &= x \cdot \sum_{i=0}^{\infty} \frac{(i+3)!}{3! \times i!} x^i \\ &= \sum_{i=0}^{\infty} \frac{(i+3)!}{3! \times i!} x^{i+1} \\ &= \sum_{i=0}^{\infty} C_{i+3}^i x^{i+1} \end{aligned}$$

于是我们得到通项 $a_n = C_{n+2}^{n-1} = C_{n+2}^3$ 。

第十五章 多项式

15.1 快速傅里叶变换

傅里叶变换可以把信号在时域和频域互相转化。打个比方，如果说时域是一首歌曲的波形，那么频域就是乐谱。最后进入我们耳朵的是复杂的声波，但它是由于一系列特定频率的简单波形按一定规律组合得到的。直接在时域上对波形处理可能是比较困难的，但经过傅里叶变换，我们可以把它转化到较好处理的频域上，处理后再通过相应的逆变换转化回去。

15.1.1 离散傅里叶变换

离散傅里叶变换 (Discrete Fourier Transform, DFT) 是傅里叶变换在时域和频域上都呈离散的形式。它在很多领域都有各种不同的应用，但在算法竞赛上，主要是用来解决多项式乘法 (卷积) 等问题。

我们通常会把 $N - 1$ 次多项式写成 $P = \sum_{n=0}^{N-1} a_n x^n$ ，其由 n 个系数 $[a_0, a_1, \dots, a_{N-1}]$ 来确定，这也称作系数表达法，这可以看作这个多项式的频域，多项式就是由若干简单的幂函数线性组合而成。

但还有另一种表达方法：点值表达法，即在多项式上取 N 个不同的点： $(b_0, P(b_0)), (b_1, P(b_1)), \dots, (b_{N-1}, P(b_{N-1}))$ ，利用这些点也可唯一确定一个多项式。这相当于在时域上采样。

现在我们的目的就是将系数表达法转化为点值表达法，这里就可以用上离散傅里叶变换，在这里我们先取一组特殊点 $e^{\frac{2k\pi i}{N}} (k = 0, 1, \dots, N - 1)$ ，

并设 $p_k = P(e^{\frac{2kn\pi i}{N}})$, 其中 i 是虚数单位, 于是有:

$$p_k = \sum_{n=0}^{N-1} e^{\frac{2kn\pi i}{N}} a_n$$

这里的 a_n 是多项式的系数, 那么逆变换就是:

$$a_n = \frac{1}{N} \sum_{k=0}^{N-1} e^{-\frac{2kn\pi i}{N}} p_k$$

当然, 系数也可能有点差别, 但是这个系数其实不重要, 反正都是要转化回来的, 只要保证两个系数相乘等于 $\frac{1}{N}$ 就好了。

15.1.2 快速傅里叶变换

快速傅里叶变换 (Fast Fourier Transform, FFT) 利用**分治**思想简化 DFT 的计算, 使得时间复杂度降到了 $O(n \log n)$ 。

我们回到公式 $p_k = \sum_{n=0}^{N-1} e^{\frac{2kn\pi i}{N}} a_n$, 这里不妨设 N 为偶数, 我们把**奇偶项分开**:

$$\begin{aligned} p_k &= \sum_{n=0}^{\frac{N}{2}-1} e^{\frac{2k(2n)\pi i}{N}} a_{2n} + \sum_{n=0}^{\frac{N}{2}-1} e^{\frac{2k(2n+1)\pi i}{N}} a_{2n+1} \\ &= \sum_{n=0}^{\frac{N}{2}-1} e^{\frac{4kn\pi i}{N}} a_{2n} + e^{\frac{2k\pi i}{N}} \cdot \sum_{n=0}^{\frac{N}{2}-1} e^{\frac{4kn\pi i}{N}} a_{2n+1} \end{aligned}$$

现在我们把原来的系数序列的奇偶项分别看作一个新的**系数序列**, 即令 $b = [a_0, a_2, \dots, a_{n-2}]$, $c = [a_1, a_3, \dots, a_{n-1}]$, 并分别对它们进行**离散傅里叶变换**,

分别设：

$$\begin{aligned} g_k &= \sum_{n=0}^{\frac{N}{2}-1} e^{\frac{2kn\pi i}{N}} b_n = \sum_{n=0}^{\frac{N}{2}-1} e^{\frac{4kn\pi i}{N}} b_n \\ h_k &= \sum_{n=0}^{\frac{N}{2}-1} e^{\frac{2kn\pi i}{N}} c_n = \sum_{n=0}^{\frac{N}{2}-1} e^{\frac{4kn\pi i}{N}} c_n \end{aligned}$$

注意到 b_n 和 c_n 分别是 a_{2n} 和 a_{2n+1} ，所以我们得到公式：

$$p_k = g_k + e^{\frac{2k\pi i}{N}} h_k$$

由于 b, c 的长度均为 $\frac{N}{2}$ ，即只有在 $k < \frac{N}{2}$ 时， b_n, c_n 才能分别用 a_{2n} 和 a_{2n+1} 替换，因此上式只有在 $k < \frac{N}{2}$ 时才成立。于是问题变成了如何去确定剩下 $\frac{N}{2}$ 个系数，我们可以发现，如果我们用 $k + \frac{N}{2}$ 代替 k 分别代入 g 和 h 中会发现 $g_{k+\frac{N}{2}}$ 和 g_k 是相等的，同理 h 也是如此。我们有：

$$e^{\frac{2(k+\frac{N}{2})\pi i}{N}} = e^{\frac{2k\pi i}{N}} \cdot e^{\pi i} = -e^{\frac{2k\pi i}{N}}$$

所以我们有：

$$p_{k+\frac{N}{2}} = g_k - e^{\frac{2k\pi i}{N}} h_k$$

所以我们只需要求出 g_k 和 $h_k (k < \frac{N}{2})$ ，就可以在 $O(n)$ 内求出 $p_k (k < N)$ ，也就是将问题规模缩小了一半，同时， g_k 和 h_k 也可以用同样的方法递归求下去，最终实现 $O(n \log n)$ 的时间内完成 DFT。

当然能一直递归下去的条件是 N 是 2 的整次幂，但这不成问题，若非如此，直接补成 2 的整次幂即可。

第十六章 线性代数

16.1 线性基

在线性代数中，基（基底）是描述、刻画向量空间的基本工具。向量空间的基是它的一个特殊的子集，基的元素称为基向量。向量空间中任意一个元素，都可以唯一地表示成基向量的线性组合。如果基中元素个数有限，就称向量空间为有限维向量空间，将元素的个数称作向量空间的维数。

在线性代数中，对于向量组 $\alpha_1, \alpha_2, \dots, \alpha_m$ ，我们把其张成空间的一组线性无关的基称为该向量组的线性基。

通常在算法竞赛中，我们主要讨论的是 \mathbb{F}_2^n 中的向量，这时我们可以把每个向量都当做一个二进制数，而向量间的加法也就是按位异或。即根据二进制数集合 $S = \{x_1, x_2, \dots, x_n\}$ ，得到另外一个二进制数集合 $S' = \{y_1, y_2, \dots, y_n\}$ ，我们保证 $\forall A \subseteq S, \exists A' \subseteq S'$, s.t. A 与 A' 的异或和相等；同时保证 S' 中任意元素都不能被 S' 中其他元素的组合异或出来，那么我们把满足这两个条件的 S' 称作 S 的线性基，利用它可以方便地求出原集合的 k 大异或和。

构建线性基通常采用**动态插入元素**的方法，即：不断地向已经得到的线性基里面加入新的元素，如果它可以被线性基内其他元素的组合异或出来则舍去；否则留下。

```
vector<ull> B;
void insert(ull x) {
```

```

for (auto b : B)
    x = min(x, b ^ x);
for (auto &b : B)
    b = min(b, b ^ x);
if (x)
    B.push_back(x);
}

```

这样构造出的集合，在满足定义“ S' 中任意元素都不能被 S' 中其他元素组的合异或出来”，还具有更好的性质：每个元素的**最高位 1 所在位**各不相同，且如果某一位是一个元素的最高位 1 所在的位置，则其他元素在这一位均为 0。

这里略去证明，直接来看一个例子，设原集合 $\langle 01001, 00100, 10011 \rangle$ ，现在插入 11110，则经过 $11110 \rightarrow 10111 \rightarrow 10011 \rightarrow 00000$ ，得到 0，说明可以被异或出来；插入 11111，则经过 $11111 \rightarrow 10110 \rightarrow 10010 \rightarrow 00001$ ，说明不能被异或出来，则我们重构原集合得到 $\langle 01000, 00100, 10010, 00001 \rangle$ ，仍满足线性基定义。

这样构造出来的线性基性质非常好。对于某个元素，选总是比不选更好，所以最大异或和就是整个集合的异或和。进一步地，排序后还可求出第 k 大的异或和，只需要按照 k 的二进制分解去选即可。例如，我们将上面得到的线性基排序后得到 $\langle 00001, 00100, 01000, 10010 \rangle$ ，那么第 $3 = (11)_2$ 小的就是第一个和第二个元素的异或 00101，第 $6 = (110)_2$ 小的就是第二个和第三个元素的异或 01100。

此外，还需要考虑 0 的情况，如果构造的线性基比原集合小，说明原集合可以异或出 0，那么我们需要将 k 减去 1：

```

sort(B.begin(), B.end());
ull ans = 0;
if (B.size() < n)
    k--;

```

```
for (auto b : B) {
    if (k & 1)
        ans ^= b;
    k >>= 1;
}
if (k == 0)
    cout << ans << endl;
else
    cout << -1 << endl;
```

如果要计算 a_1, a_2, \dots, a_n 有多少种方法可以异或出 x , 只需要求出 a_1, a_2, \dots, a_n 的线性基 S , 如果 x 可以被 S 表示出来, 则方案数为 $2^{n-|S|}$ 。

第十七章 杂项

17.1 摩尔投票法

摩尔投票是一种用来解决**绝对众数**问题的算法。

在一个集合中，如果一个元素的出现次数比其他所有元素的出现次数之和还多，那么就称它为这个集合的绝对众数，换言之，绝对众数的出现次数大于总元素数的一半。例如下面这道题：

例 17.1.1 (LeetCode-169 多数元素). 给定一个大小为 n 的数组 $nums$ ，返回其中的多数元素。多数元素是指在数组中出现次数 大于 $\lfloor \frac{n}{2} \rfloor$ 的元素。

你可以假设数组是非空的，并且给定的数组总是存在多数元素。

输入

```
nums = [1, 3, 3, 2, 3]
```

输出

```
3
```

对于这个问题，解决方法并不难，最常规的想法是统计每个元素的出现次数，最多的那个就是多数元素。

这里来用摩尔投票法解决：我们把找到绝对众数的过程想象成一次选举。通过维护一个 m 表示当前的候选人，然后维护一个 cnt 。对于每一张新的选票，如果它投给了当前的候选人，就把 cnt 加 1，否则就把 cnt 减 1。

特别地，计票时如果 $cnt = 0$ ，我们可以认为目前谁都没有优势，所以新的选票投给谁，谁就成为新的候选人。

轮次	1	2	3	4	5
m	1	1	3	3	3
cnt	1	0	1	0	1

这里给出实现的程序：

```
int m = 0, cnt = 0;
for (int i = 0; i < n; ++i) {
    if (cnt == 0)
        m = A[i];
    if (m == A[i])
        cnt++;
    else
        cnt--;
}
```

值得注意的是，如果我们要求的是**众数**，这样的做法并不能给出正确答案，但如果要求的是**绝对众数**，且绝对众数确实存在，则结果一定是正确的。

这个算法具有 $O(n)$ 的时间复杂度和 $O(1)$ 的空间复杂度，相当优秀。

17.1.1 摩尔投票法的拓展

先前讲述的算法解决的只是选出一个人的情形。但其实，如果要选出 N 个候选人，并且要求每个人的得票都超过总票数的 $\frac{1}{N+1}$ ，也可以用上面的算法，只需稍微修改即可：

```
int m[N], cnt[N];
for (auto e : nums) {
    int i = find(m, m + N, e) - m;
    if (i != N) { // 如果当前票投给了候选人之一
        cnt[i]++;
    }
}
```

```
        continue;
    }

    int j = find(cnt, cnt + N, 0) - cnt;
    if (j != N) { // 如果当前存在一个位置"虚位以待"
        m[j] = e;
        cnt[j] = 1;
        continue;
    }

    for (auto &c : cnt) {
        c--;
    }
}
```

现在我们来分析这个程序，我们用 `m[N]` 数组来存储当前被认为可能超过特定频率的候选元素，用 `cnt[N]` 数组来维护与 `m` 中每个候选元素对应的计数，算法的流程如下：

1. **遍历元素**: 遍历 `nums` 中的每个元素 `e`。
2. **查找候选人**: 检查元素 `e` 是否已经是候选人之一（即查看 `e` 是否在数组 `m` 中）。
 - 如果是，对应候选人的计数 `cnt[i]` 增加。
3. **查找空位**: 如果 `e` 不是现有的候选人，并且存在未被占用的候选位置（即 `cnt` 中有 0 的位置），将 `e` 设置为新的候选人，并初始化计数为 1。
4. **减少计数**: 如果没有空位可用（即所有 `cnt` 都大于 0），对所有候选人的计数减 1。

17.2 集合论

17.2.1 集合的基本定义

集合是现代数学的一个重要概念，它是由一些元素构成的，这些元素可以是数、符号、对象、事件等等。我们先来看一些定义：

定义 17.2.1 (集合). 具有某种性质的，确定的，有区别的事物的全体称为集合，集合中的事物称为元素。

定义 17.2.2 (元素). 若 x 是集合 A 中的元素，记作 $x \in A$ ，若 x 不是集合 A 中的元素，记作 $x \notin A$ 。

定义 17.2.3 (空集). 若 $\forall x$ ，都有 $x \notin A$ ，则 A 为空集，记作 $A = \emptyset$ 。

定义 17.2.4 (子集与真子集). 对于集合 A 和 B ， $\forall x \in A$ ，都有 $x \in B$ ，则称 A 是 B 的子集，记作 $A \subset B$ ；若 $A \subset B$ ， $\exists x \in B$ 且 $x \notin A$ ，则称 A 是 B 的真子集，记作 $A \subseteq B$ 。空集是任何集合的子集。

以上的定义共同构成了集合的基础，例如 $C = \{1, 2, 3, 4\}$ ， C 是集合，1, 2, 3, 4 是集合中的元素，根据上述定义，我们容易得到以下定理：

定理 17.2.5. 1. **自反性**: $A \subset A$ 。

2. **传递性**: 若 $A \subset B$ 且 $B \subset C$ ，则 $A \subset C$ 。

3. 若 $x \in A$ ，则 $\{x\} \subset A$ 。

以上几个定理证明不难，如 $A = \{1\}$, $B = \{1, 2\}$, $C = \{1, 2, 3, 4\}$ ，显然有 $A \subset B \subset C$ ，且显然有 $1 \in \{1\}$ ，所以 $\{1\} \subseteq A$ 。

定义 17.2.6 (集合相等). 若 $A \subset B$ 且 $B \subset A$ ，则称集合 A 和 B 相等记作 $A = B$ ，否则 A 和 B 不相等，记作 $A \neq B$ 。

17.2.2 集合的运算

集合的运算这里主要提及的是并、交、补和差运算，我们首先来看一些集合的基本运算交与并的定义、定理：

定义 17.2.7 (交集与并集). 对于集合 A 和 B ， $\{x|x \in A \text{ or } x \in B\}$ 称为 A 与 B 的并集，记作 $A \cup B$ ； $\{x|x \in A \text{ and } x \in B\}$ 称为 A 与 B 的交集，记作 $A \cap B$ 。

定理 17.2.8 (交换律). $A \cup B = B \cup A$ ， $A \cap B = B \cap A$ 。

定理 17.2.9 (结合律). $(A \cup B) \cup C = A \cup (B \cup C)$ ， $(A \cap B) \cap C = A \cap (B \cap C)$ 。

根据以上的定义，我们不难得出以下结论：

定理 17.2.10. 对于集合 A 和 B ， $A \cap B \subset A \subset A \cup B$ ， $A \cap B \subset B \subset A \cup B$ 。

定理 17.2.11 (分配律). $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$ ， $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$ 。

以上是集合交与并的基本操作，接下来我们来看补集和差集的定义与定理：

定义 17.2.12 (差集与补集). 对于集合 A 和 B 属于集合 A 而不属于集合 B 的元素称为 A 减 B 的差集，记作 $A \setminus B = \{x \in A \text{ and } x \notin B\}$ ；当 $B \subset A$ 时，则 $A \setminus B$ 也记作 $C_A B$ ；当我们讨论的集合都是 X 的子集时，称 X 为全集， $C_X B$ 一般记作 B^c ，称作 B 的补集。

需要注意的是，我们一般只使用 $A \setminus B$ 与 B^c 两种符号， $C_A B$ 一般不使用，因任何情况下都可以用 $A \setminus B$ 代替；此外， $A \setminus B$ 也可以记作 $A - B$ 。

定理 17.2.13. 设 S 是全集，集合 $A \subset S$ ，则有：

- (1) $A \cap A^c = \emptyset$ ， $A \cup A^c = S$ 。
- (2) 若 $x \in S$ ，则 $x \notin A \Leftrightarrow x \in A^c$ ， $x \in A \Leftrightarrow x \notin A^c$ 。
- (3) $(A^c)^c = A$ 。

对于这个定理，如果将 S 为全集的条件去掉，同时将 A^c 改为 $S \setminus A$ ，结论仍然成立。

定理 17.2.14. 若集合 $A, B \subset S$ ，则有：

- (1) $A \subset B \Leftrightarrow B^c \subset A^c$ 。
- (2) $A \setminus B = A \cap B^c$ 。

定理 17.2.15. $(A \setminus B) \cap C = A \cap C \setminus B \cap C$ ， $(A \setminus B) \setminus C = A \setminus (B \cup C)$ 。

定理 17.2.16 (De Morgan 定理). 若集合 A 和 $B \subset S$ ，则有 $(A \cup B)^c = A^c \cap B^c$ ， $(A \cap B)^c = A^c \cup B^c$ 。

两个集合的**对称差**是只属于其中一个集合，而不属于另一个集合的元素组成的集合，也就是不在交集中的元素组成的集合。

定义 17.2.17 (对称差). 对于集合 A, B ， $(A \setminus B) \cup (B \setminus A)$ 称为它们的对称差，记作 $A \Delta B$ 。

对称差描述了两个集合不相交的部分，因对称差是集合基本运算的复合，以下是对称差的一些性质：

- 定理 17.2.18.**
- (1) $A \Delta B = (A \cup B) \setminus (A \cap B) = (A \cap B^c) \cup (B \cap A^c) = A^c \Delta B^c$ 。
 - (2) $A \Delta B = \emptyset \Leftrightarrow A = B$ 。
 - (3) $A \Delta B = B \Delta A$ 。
 - (4) $(A \Delta B) \Delta C = A \Delta (B \Delta C)$ 。

17.3 离散化

离散化，就是当我们**只关心数据的大小关系时**，用**排名代替原数据进行处理**的一种预处理方法。

离散化本质上是一种哈希，在保持原序列大小关系的前提下映射其为**正整数**。当原数据很大或含有负数、小数时，难以表示为数组下标，一些算

法和数据结构（如二叉搜索树）无法运作，这时我们就可以考虑将其离散化。

例如，现在我们有序列 $A = [10, 23, 35, 3, -40, 3]$ 。我们先复制一个同样的序列：

```
int C[MAXN];
memcpy(C, A, sizeof(A));
```

排序，去重：

```
sort(C, C + n);
int l = unique(C, C + n) - C; // l 为不重复元素的数量
```

其中，`std::unique()`的返回值是一个迭代器，对于数组来说就是指针，它表示去重后容器中不重复序列的最后一个元素的下一个元素。所以可以通过作差求得不重复元素的数量。现在我们有 $C = [-40, 3, 10, 23, 35]$ 。再用一个数组 L ，储存 A 中每个元素在 C 中的排名：

```
int L[MAXN];
for (int i = 0; i < n; ++i) {
    L[i] = lower_bound(C, C + l, A[i]) - C + 1; // 二分查找
}
```

这样我们就实现了原序列的离散化。得到 $L = [3, 4, 5, 2, 1, 2]$ 。

因为排序和 n 次二分查找的复杂度都是 $O(n \log n)$ ，所以离散化的复杂度也是 $O(n \log n)$ 。完整的程序如下：

```
int C[MAXN], L[MAXN];
// 在 main 函数中...
memcpy(C, A, sizeof(A)); // 复制
sort(C, C + n); // 排序
int l = unique(C, C + n) - C; // 去重
for (int i = 0; i < n; ++i) {
    L[i] = lower_bound(C, C + l, A[i]) - C + 1; // 查找
}
```

散化也不一定要从小到大排序，有时候也需要从大到小。这时在排序和查找时相应地加上`greater<int>()`就可以了。

例 17.3.1 (LeetCode-2830 销售利润最大化). 给你一个整数 n 表示数轴上的房屋数量，编号从 0 到 $n - 1$ 。

另给你一个二维整数数组 offers ，其中 $\text{offers}[i] = [\text{start}_i, \text{end}_i, \text{gold}_i]$ 表示第 i 个买家想要以 gold_i 枚金币的价格购买从 start_i 到 end_i 的所有房屋。

作为一名销售，你需要有策略地选择并销售房屋使自己的收入最大化。
返回你可以赚取的金币的最大数目。

注意 同一所房屋不能卖给不同的买家，并且允许保留一些房屋不进行出售。

示例：

输入： $n = 5$, $\text{offers} = [[0, 0, 1], [0, 2, 2], [1, 3, 2]]$

输出： 3

解释：

有 5 所房屋，编号从 0 到 4，共有 3 个购买要约。

将位于 $[0, 0]$ 范围内的房屋以 1 金币的价格出售给第 1 位买家，
并将位于 $[1, 3]$ 范围内的房屋以 2 金币的价格出售给第 3 位买家。

可以证明我们最多只能获得 3 枚金币。

对于问题17.3.1，在区间 $[0, n)$ 的房子，如果我们选择 $[i, j, \text{gold}]$ 的 offer，那么原问题的解就变成 $\text{gold} + [0, i) + (j, n)$ 的两个子问题的解；定义 $\text{dp}[i]$ 表示到 i 为止可以收获的最大销售利润，则对于第 i 间房子有卖和不卖两种选择：

- 不卖，那么 $\text{dp}[i] = \text{dp}[i - 1]$

- 卖，那么 $dp[i] = dp[i - 1] + gold$

然而题目的销售 `offer` 是按照区间销售而不是按照单个房子销售，如果第 i 个房子没有处于 `offer` 的 `end` 端点的话，是不能卖出的。

因此，我们需要找到枚举 `start` 端点（从后往前遍历）或枚举 `end` 端点（从前往后遍历）的方法，并使用转移方程 $dp[i] = \max(dp[i], dp[start] + gold)$ 更新答案。

这里略过基本的线性 DP 过程，直接考虑到当 n 的非常大时，可能会超过空间和时间限制，且由于影响题目的关键点仅在 `offer` 的 `start` 端点和 `end` 端点，而中间空白的点或者被覆盖的点是无关紧要的，因此可以使用离散化的技巧，将所有 `offer` 的 `start` 端点和 `end` 端点去重后组合成新的坐标轴 `points`，将在 $[0, n)$ 上的线性 DP 转换为在 $[0, m)$ 上的离散化线性 DP，方案如下：

```
class Solution {

public:
    int maximizeTheProfit(int n, std::vector<std::vector<int>>& offers) {
        // 对 start 和 end 离散化
        std::unordered_set<int> pointSet;
        for (const auto& offer : offers) {
            pointSet.insert(offer[0]);
            pointSet.insert(offer[1]);
        }

        // 排序
        std::vector<int> points(pointSet.begin(), pointSet.end());
        std::sort(points.begin(), points.end());

        // 端点 -> id
        int m = points.size();
```

```
    std::unordered_map<int, int> ids;
    for (int id = 0; id < m; ++id) {
        ids[points[id]] = id;
    }

    // 分桶
    std::vector<std::vector<std::pair<int, int>>> buckets(m)
        ;
    for (const auto& offer : offers) {
        int start = offer[0];
        int end = offer[1];
        int gold = offer[2];
        buckets[ids[end]].push_back({ids[start], gold});
    }

    // 线性 DP
    std::vector<int> dp(m + 1, 0);
    for (int i = 1; i <= m; ++i) {
        // 不卖
        dp[i] = dp[i - 1];
        // 卖
        for (const auto& e : buckets[i - 1]) {
            dp[i] = std::max(dp[i], dp[e.first] + e.second);
        }
    }
    return dp[m];
}
};

};
```

- 时间复杂度: $O(m \log m + m)$, 预处理时间为 $O(m \log m)$, 瓶颈在排序, 线性 DP 时间为 $O(m)$;
- 空间复杂度: $O(m)$, 离散化节点空间、分桶空间和线性 DP 空间都是 $O(m)$ 的空间复杂度。

第十八章 初赛

18.1 计算机概述

18.1.1 发展历史

计算机发展的五个阶段

代别	年代	电子元件	应用范围
第一代	1946-1958	真空电子管	科学计算、军事研究
第二代	1959-1964	晶体管	数据处理、事务处理
第三代	1965-1970	集成电路	工业控制的各个领域
第四代	1971 至今	超大规模集成电路	各个领域
第五代	现代	智能计算机系统	人工智能

世界上第一台电子计算机

1946 年 2 月，世界上第一台电子计算机 ENIAC(Electronic Numerical Integrator And Computer) 在美国宾夕法尼亚大学诞生。

冯诺依曼理论

1944 年，美籍匈牙利数学家冯·诺依曼提出计算机基本结构和工作方式的设想，为计算机的诞生和发展提供了理论基础。时至今日，尽管计算机

软硬件技术飞速发展，但计算机本身的体系结构并没有明显的突破，当今的计算机仍属于冯·诺依曼架构。

冯·诺依曼结构也称普林斯顿结构，提出了计算机制造的三个基本原则，即采用二进制逻辑、程序存储执行以及计算机由五个部分组成，这套理论被称为冯·诺依曼体系结构。

- **二进制逻辑**: 使用只有 0 和 1 两种状态的系统来表示和处理所有信息。
- **程序存储执行**: 将程序和数据存储在存储器中，并通过控制器按顺序执行程序。
- **计算机的五个基本组成部分**: 运算器、控制器、存储器、输入设备、输出设备。

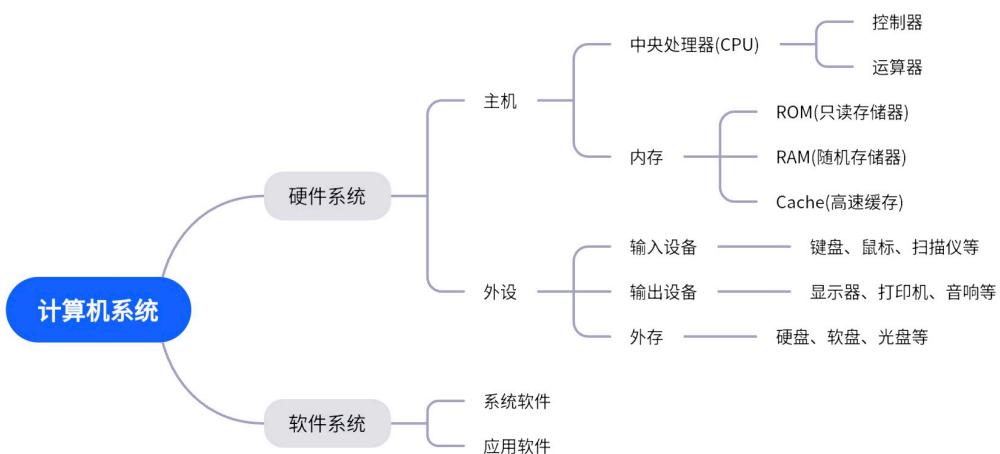
计算机领域三个重要人物

- 冯·诺依曼：电子计算机之父，提出了计算机体系结构。
- 艾伦·麦席森·图灵：计算机科学之父，人工智能之父，提出了一种判定机器是否具有智能的试验方法，即图灵测试。首次提出了计算机科学理论。计算机界的最高奖项“图灵奖”以他命名，被称为“计算机界的诺贝尔奖”。
- 克劳德·艾尔伍德·香农：现代信息论的著名创始人，信息论之父，创造了信息论，提出了某种信息从一处传送到另一处所需的全部设备所构成的系统。

18.1.2 计算机结构与硬件

计算机系统的构成大致如下图所示，一个完整的计算机系统由硬件和软件两大部分组成，硬件主要包含五大部分构成：运算器、控制器、存储

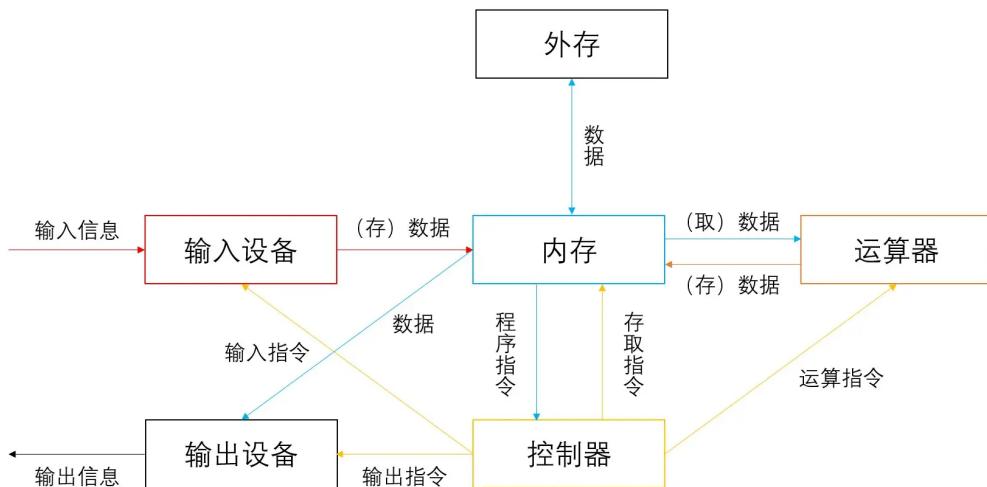
器、输入设备、输出设备；软件则是指运行在计算机上的各种程序，主要包含系统软件和应用软件两部分。



计算机硬件

- **中央处理器 (CPU)**: CPU 是计算机的核心，负责解释和执行计算机程序中的指令。在冯诺依曼体系中，CPU 通常由两个主要部分组成：
 - **运算器**: 运算器负责执行所有算术和逻辑运算，比如加、减、乘、除、与、或、非等操作。
 - **控制器**: 控制器负责从内存中获取指令（通过取指令、译码、执行三个步骤），并协调和控制计算机各个部分的工作。它通过控制信号来管理数据的流动和操作的顺序。
 - **寄存器**: 寄存器是 CPU 内部的一种高速存储单元，用于暂时存储指令、数据和地址等信息。寄存器的访问速度极快，远超系统主存储器 (RAM)。
- **存储器**: 存储器是计算机的重要组成部分，用于存储程序、数据和指令。存储器分为主存储器 (Main Memory) 和辅助存储器 (Secondary Storage)。

- 主存储器，也叫内存 (Random Access Memory, RAM)，是计算机中直接与 CPU 交互的存储器，用于存储当前正在执行的程序和数据。它的特点是速度快、易于 CPU 访问，但容量相对较小。
 - 只读存储器 (Read-Only Memory, ROM) 是一种非易失性存储器，它与 RAM (随机存取存储器) 不同，即使断电后，ROM 中的数据也不会丢失。ROM 的主要特点是数据只能被读取，通常在出厂时数据已经写入，用户不能随意修改或删除其中的内容。
 - 辅助存储器 (Secondary Storage)：辅助存储器是计算机中用于长期存储数据的设备。它相对于主存储器的访问速度较慢，但容量更大且数据在断电时不会丢失。常见的辅助存储设备包括硬盘 (HDD)、固态硬盘 (SSD)、光盘 (CD/DVD) 和 U 盘等。
- **输入设备：**输入设备包括键盘、鼠标、扫描仪、摄像头等，它们是计算机与外部世界进行信息交互的重要工具。
 - **输出设备：**输出设备包括显示器、打印机、声卡、网卡等，它们是计算机向外部世界输出信息的重要工具。



CPU 性能指标

CPU 的主要**性能指标**包括：**时钟主频、字长、高速缓存容量、指令合集和动态处理技术**：

- **时钟主频**：也称为**时钟频率或主频**，指的是 CPU 的时钟信号每秒钟的振荡次数，通常以赫兹 (Hz) 为单位，现代 CPU 的主频一般以千兆赫兹 (GHz) 为标准衡量。

例如，一款 CPU 主频为 3.0 GHz，意味着其每秒可进行 30 亿次时钟周期的操作。

- **字长**：字长指的是 CPU 一次能够处理的数据宽度，通常以位 (bit) 为单位。常见的字长有 8 位、16 位、32 位、64 位等。

字长决定了 CPU 在一次处理周期中可以处理的数据量，以及能够访问的内存地址范围。字长越大，CPU 一次能处理的数据越多，因此性能会更高。

例如，64 位处理器可以在一次运算中处理 64 位的数据，而 32 位处理器一次只能处理 32 位数据，64 位 CPU 的理论运算能力更强。

- **高速缓存容量**：高速缓存 (Cache) 是位于 CPU 和主存 (RAM) 之间的小型、高速存储器，分为一级缓存 (L1 Cache)、二级缓存 (L2 Cache) 和三级缓存 (L3 Cache)，它们负责临时存储经常使用的数据和指令。
- **指令集**：指令集是 CPU 能够理解并执行的指令的集合。它规定了 CPU 如何执行程序、处理数据和与内存及输入/输出设备交互。常见的指令集有 x86、x64(Intel、AMD 使用的架构) 和 ARM(用于移动设备的处理器)。

指令集决定了 CPU 如何处理指令、指令的种类、数据操作方式等。不同指令集架构影响了 CPU 的性能、效率和与操作系统及软件的兼容性。

- CISC(复杂指令集计算, Complex Instruction Set Computer): 例如 x86 架构, 拥有大量复杂的指令, 每条指令可以执行多种操作。
 - RISC(精简指令集计算, Reduced Instruction Set Computer): 例如 ARM 架构, 指令集简化, 每条指令执行单一任务, 但指令处理速度更快, 适合低功耗设备。
- **动态处理技术:** 动态处理技术是指现代 CPU 为了提高指令执行效率而引入的各种动态优化技术。它们使得 CPU 能够更智能地管理和处理指令, 而不仅仅是按顺序执行。

空间换算

8 bit = 1 Byte	1024 Byte = 1 KiB
1024 KiB = 1 MiB	1024 MB = 1 GiB
1024 GB = 1 TiB	1024 TiB = 1 PiB

我们常把这些单位中的 i 省去, 如: KB、MB。若无特殊说明, KB 默认代表 1024 字节, MB 默认代表 1024 KiB。值得注意的是, 在计算机中**最小存储单位是位 (bit)**, 而**基本单位是字节 (Byte)**。

18.1.3 计算机语言

计算机语言是人与计算机之间交流的语言, 是人类用来表达思想和指令的符号系统。计算机语言通常分为三类: 机器语言、汇编语言和高级语言。

机器语言

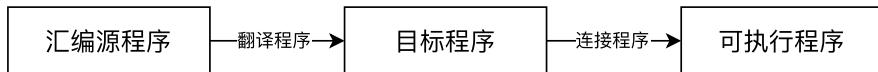
最早出现的语言就是机器语言, 它是**计算机能够直接识别的语言**, 而且**速度快**。机器语言用二进制代码来编写计算机程序的。因此又称二进制

语言。

例如 $8 + 4$ 对应的是一串二进制码 00001000 00000100 00000100。机器语言书写困难，记忆复杂，一般很难记忆。

汇编语言

由于机器语言的缺陷，人们开始用助记符编写程序，用一些符号代替机器指令所产生的语言称为汇编语言。但是，用汇编语言编写的程序不能被计算机所识别，必须使用某种特殊的软件用汇编语言写的源程序翻译和连接成能被计算机直接识别的二进制代码。

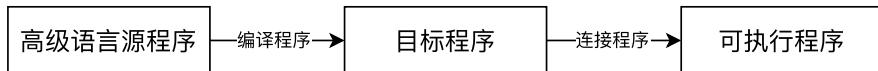


汇编语言虽然采用了助记符来编写程序，比机器语言简单，但是汇编语言仍属于低级语言，它与计算机的体系结构有关。工作量大，繁琐，而且程序可移植性差。

高级语言

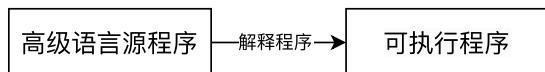
计算机并不能直接接受和执行高级语言编写的源程序，源程序在输入计算机时，通过“翻译程序”翻译成机器语言形式的目标程序，计算机才能识别和执行。这种“翻译”通常有两种方式：**编译**和**解释**。

编译方式：编译方式的翻译工作由“编译程序”来完成，它是先将整个源程序都转换成二进制代码，生成目标程序，然后把目标程序连接成可执行的程序，以完成源程序要处理的运算并取得结果。



解释方式：解释方式的翻译工作由“解释器”完成，它并不将整个源程序一次性翻译为目标代码或可执行程序，而是逐行读取源程序，逐行翻译并

立即执行相应的指令。在解释方式下，程序在运行过程中依赖于解释器进行实时翻译和执行，因此程序的执行速度通常比编译方式要慢。



高级语言的两种分类方式：

- 编译型：C、C++、Go 等；
- 解释型：Python、JavaScript、Ruby 等。
- 面向过程：C、Pascal；
- 面向对象：Python、C++、Java 等。

18.1.4 信息编码

计算机要处理的数据除了数值数据以外，还有各类符号、图形、图像和声音等非数值数据。而计算机只能识别两个数字。要使计算机能处理这些信息，首先必须将各类信息转换成 0 和 1 表示的代码，这一过程称为编码。

ASCII 编码

ASCII 码对英语字符与二进制位之间的关系，做了统一规定。

ASCII 码是由美国国家标准委员会制定的一种包括数字、字母、通用符号和控制符号在内的字符编码集，称为美国国家信息交换标准码 (American Standard Code for Information Interchange)。ASCII 码是一种 7 位二进制编码，能表示 $2^7 = 128$ 种国际上最通用的西文字符。是目前计算机中，特别是微型计算机中使用最普遍的字符编码集。

0 - 48
A - 65
a - 97

Unicode 编码

世界上存在着多种编码方式，同一个二进制数字可以被解释成不同的符号。因此，要想打开一个文本文件，就必须知道它的编码方式，否则用错误的编码方式解读，就会出现乱码，因此 Unicode 应运而生。

Unicode，就像它的名字都表示的，这是一种所有符号的编码，将世界上所有的符号都纳入其中，每一个符号都给予一个独一无二的编码。

汉字编码

汉字交换码是指不同的汉字处理功能的计算机系统之间在交换汉字信息时所使用的的代码标准。自国家标准 GB-2312 公布以来，我国一直延用该标准所规定的国标码作为统一的汉字信息交换码 GB5007-85 图形字符编码。

GB2312-80 标准包括了 6763 个汉字，按其使用频率分为一级汉字 3755 个和二级汉字 3008 个。一级汉字按拼音排序，二级汉字按部首排序。该标准还包括标点符号、数种西文字符、图形、数码等符号 682 个。

区位码的区码和位码采用从 01 到 94 的十进制，国标码采用十六进制的 21H 到 73H。区位码和国标码的换算关系是：区码和位码分别加上十进制 32。如“国”字在表中的 25 行 90 列，其区位码为 2590。国标码是 397AH。

18.1.5 机器数与真值

计算机中要处理的整数有“无符号”和“有符号”之分，“无符号”整数顾名思义就是不考虑正负的整数，可以直接用二进制表示，故只讨论“有符号”整数。

原码、反码、补码

原码：原码是最简单的表示方法，就是用最高位表示符号位，其余位表示数值位。

$$\text{原码} = \text{符号位} + \text{数值位}$$

反码：对于一个正数，反码就是其原码；对于一个负数，反码就是除符号位外，原码的各位全部取反。

例如： $x_{\text{原}} = 01000101$ ，则 $x_{\text{反}} = 01000101$ ； $x_{\text{原}} = 11000101$ ，则 $x_{\text{反}} = 10111010$ 。

补码：对于一个正数，补码就是其原码；对于一个负数，补码等于反码+1。

例如： $x_{\text{原}} = 01000101$ ，则 $x_{\text{补}} = 01000101$ ； $x_{\text{原}} = 11000101$ ，则 $x_{\text{反}} = 10111011$ 。

18.1.6 计算机网络

所谓计算机网络，就是利用通信线路和设备，把分布在不同地理位置上的多台计算机连接起来。

计算机网络是现代通信技术与计算机技术相结合的产物。网络中的计算机与计算机之间的通信依靠协议进行。协议是计算机收、发数据的规则。

计算机网络的主要功能

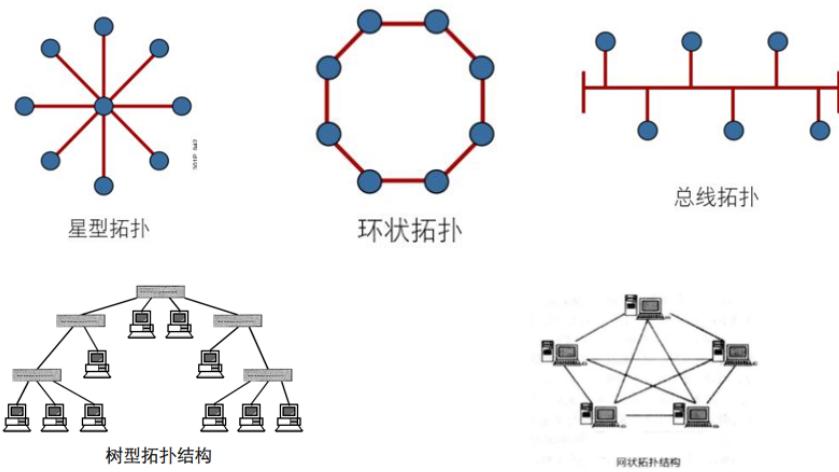
- 资源共享
- 数据传输
- 分布式处理
- 综合信息服务

网络的分类

- 局域网 (Local Area Network): 一般局限在 1km 范围内。
- 城域网 (Metropolitan Area Network): 城域网的范围为几千米到几十千米以内。
- 广域网 (Wide Area Network): 广域网的范围在几十千米到几千千米以上。

网络的拓扑结构

按网络的拓扑结构进行分类：星型、总线型、环型、树型、网状型。



- **星型:** 所有设备通过单独的连接线与中央设备 (如集线器或交换机) 相连，中央设备负责数据转发；易于管理，节点故障不会影响整个网络，中央设备故障时便于诊断，但中央设备故障会导致整个网络瘫痪，布线成本较高。
- **环型:** 传输速率高，传输距离远；各节点的地位和作用相同；各节点传输信息的时间固定；容易实现分布式控制，但任何一个节点故障都会影响整个网络。

- **总线型**: 结构简单, 可靠性高, 布线容易, 连线总长度小于星型结构; 总线任务重, 易产生瓶颈问题; 总线本身的故障对系统是毁灭性的。
- **树型**: 星型拓扑和总线型拓扑的结合, 节点通过分支方式连接, 形成层次结构; 易于扩展, 便于网络分层管理, 但分支和主干线故障可能影响较大范围的设备。
- **网状型**: 每个节点都与其他节点直接相连, 数据可以通过多条路径传输; 具有高度的冗余和可靠性, 任何一条链路故障不影响网络整体, 但布线复杂, 成本较高。

IP 地址

P 地址用于标识 Internet 网络上节点的 32 位地址 (以后可能使用 V6 版本是 128 位的, 分 8 组, 每组 16 位), 对于 Internet 网络上的每个节点, 都必须指派一个唯一的地址, 它由网络 ID 和唯一的主机 ID 组成。该地址通常用由点分隔的八位字节的十进制数表示 (例如: 192.168.7.27)。

IP 地址分成 A,B,C,D,E 五类, 其中 A、B、C 为常用类, 由**网络 ID** 和 **主机 ID** 两个部分组成, **网络 ID** 也叫**网络地址**, 标识大规模 TCP/IP 网际网络内的单个网络, 这个 ID 用于唯一地识别大规模的网际网络内部的每个网络; **主机 ID**, 也叫**作主机地址**, 识别每个网络内部的 TCP/IP 节点, 每个设备的主机 ID 唯一地识别所在网络内的单个系统。

	1	8	16	24	32
A类	0	网络号		主机号	
B类	1 0	网络号		主机号	
C类	1 1 0	网络号		主机号	
D类	1 1 1 0			组播地址	
E类	1 1 1 1 0			保留地址	

IP 地址的表示方法

IP 地址采用点分十进制记法，即将 32bit 的 IP 地址中的每 8 位用等效的十进制表示，并每 8 位之间加上一个点 . 隔开，每组数字的取值范围只能 0 ~ 255。

例如：

10000001	.	00001011	.	00000011	.	00011111
129	.	11	.	3	.	159

尽管 IP 地址的主机号的每个域的取值范围是 0 ~ 255，但主机 ID 所有域不能都为 0 或 255。

例如：若网络 ID 为 10，那么就不能把 10.0.0.0 和 10.255.255.255 两个 IP 地址分配给任何主机；若网络 ID 为 192.114.31，那么就不能把 192.114.31.0 和 192.114.31.255 两个地址分配给任何主机。

此外，经过 Internet 编号指派机构（IANA）协商同意，几个 IP 网络被保留下来用作企业内部私有。这些保留的号码是：

10.0.0.0 ~ 10.255.255.255	A 类
172.16.0.0 ~ 172.31.255.255	B 类
192.168.0.0 ~ 192.168.255.255	C 类

域名

域名 (Domain Name) 是 Internet 上主机的可读名称，它由一串用点分隔的单词组成，通常以 .com, .org, .net 等后缀结尾。域名的作用是将人类可读的主机名转换为 IP 地址，使得用户可以方便地访问互联网上的资源。例如，www.test.edu.cn，从右至左分别是：

- .cn 顶级域名；
- .edu 二级域名；
- .test 三级域名；
- www 四级域名。

18.1.7 计算机网络安全

计算机安全是中最重要的存储数据安全，其面临的主要威胁包括：计算机病毒、非法访问、计算机电磁辐射、硬件损坏等。

计算机病毒是附在计算机软件中的隐蔽的小程序，它和计算机其他程序一样，但会破坏正常的程序和数据文件。恶性病毒可使整个计算机软件系统崩溃，数据全毁。要防止病毒侵袭主要是加强管理，不访问不安全的数据，使用杀毒软件并及时升级更新。

计算机病毒的特性

计算机病毒是一种具有自我复制能力的**恶意程序**，旨在未经授权的情况下对计算机系统进行破坏、窃取信息或影响其正常功能。病毒通常附加到合法文件或程序中，一旦这些文件或程序被执行，病毒就会开始传播。

计算机病毒的主要特性有：

1. **自我复制性**：病毒能够通过复制自身来感染其他文件或程序，并传播到其他计算机或设备上。它们常常通过网络、外部存储设备或电子邮件附件传播。
2. **潜伏性**：病毒可以在系统中隐藏一段时间，悄悄地进行复制或窃取数据，等到触发特定条件或时间后再显现出来，开始执行破坏性操作。
3. **破坏性**：病毒可能会删除文件、损坏系统、修改数据，甚至导致计算机无法启动。它们也可能窃取敏感信息，如密码和财务数据。
4. **传播性**：病毒的传播方式多种多样，可以通过网络、U 盘、邮件附件、下载的文件或共享资源等方式传播给其他设备。
5. **触发性**：病毒通常会在特定条件下触发其恶意行为，如特定日期、用户执行某操作时，或者通过外部命令来激活。
6. **隐蔽性**：许多病毒会使用技术隐藏自己，使用户或防病毒软件难以发现。例如，它们可能通过伪装成合法程序、加密代码等方式来逃避检测。

测。

常见病毒类型：

1. **文件型病毒**：感染可执行文件或文档，随文件的打开或运行而传播。
2. **宏病毒**：嵌入在文档的宏中，通常通过办公软件文档传播。
3. **引导区病毒**：感染计算机的启动引导区，影响系统启动流程。
4. **蠕虫**：类似病毒，但可以独立传播，不依赖宿主程序。

常见的安全策略

1. 安装杀毒软件

对于一般用户而言，首先要做到就是为电脑安装一套杀毒软件，并定期升级所安装的杀毒软件，打开杀毒软件的实时监控程序。

2. 安装防火墙

安装个人防火墙 (Fire Wall) 以抵御黑客攻击，最大限度地阻止网络中的黑客来访问你的计算机，防止他们更改、拷贝、毁坏你的重要信息。防火墙在安装后根据需求进行详细配置。

3. 分类设置密码并使密码设置尽可能复杂

在不同的场合使用不同的密码，如网上银行、Email、聊天室以及一些网站的会员等应尽可能使用不同的密码，以免因一个密码泄露而导致所有资料外泄。对于重要的密码（银行）一定要单独设置，并且不要与其他密码相同。

设置密码时要尽量避免使用有意义的英文单词，姓名缩写及生日、电话号码等容易暴露的字符作为密码，最好采用字符数字和特殊符号混合的密码。建议定期修改自己的密码，这样可以确保即使原密码泄露，也能将损失减到最小。

4. 不下载不明来历的软件及程序

应选择信誉较好的下载网站下载软件，将下载的软件及程序集中放在非引导分区的某个目录，在使用的时候最好用杀毒软件查杀病毒。

不要打开来历不明的电子邮件及其附件，以免遭受病毒邮件的侵害，这些病毒邮件通常会以带有噱头的标题来吸引你打开附件，如果下载或运行了他的附件，就会遭受感染，同样也不要接收和打开来历不明的 QQ、微信等发过来的邮件。

5. 定期备份重要文件

数据备份的重要性毋庸置疑，无论你的防范措施做的多么严密，也无法完全防止。如果遭受到了致命攻击，操作系统和应用软件可以重装，而重要的数据只能靠你日常备份了。所以，无论你采取了多么严格的防范措施，也不要忘了随时备份你的重要数据，做到有备无患。

18.2 排列组合

定理 18.2.1 (加法原理). 完成一项任务有 n 种方法， $a_i(1 \leq i \leq n)$ 代表完成第 i 类方法的数目，则完成该任务共有 $S = a_1 + a_2 + \dots + a_n$ 种方法。

定理 18.2.2 (乘法原理). 完成一项任务有 n 个步骤， $a_i(1 \leq i \leq n)$ 代表完成第 i 个步骤的数目，则完成该任务共有 $S = a_1 \times a_2 \times \dots \times a_n$ 种方法。

18.2.1 排列

定义 18.2.3 (排列). 从 n 个不同元素中，任取 $m(m \leq n)$ 个元素按照一定的顺序排成一列，读做从 n 个不同元素中取出 m 个元素的一个排列，记为 A_n^m 或 P_n^m 。

计算公式为： $A_n^m = n(n-1)(n-2)\cdots(n-m+1) = \frac{n!}{(n-m)!}$ 。其中， $!$ 表示阶乘，例如 $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$ ，特别规定 $0! = 1$ 。

18.2.2 组合

定义 18.2.4 (组合). 从 n 个不同元素中, 任取 $m(m \leq n)$ 个元素, 不要求元素之间有任何先后顺序, 读做从 n 个不同元素中取出 m 个元素的一个组合, 记为 C_n^m 。

$$\text{计算公式为: } C_n^m = \frac{A_n^m}{m!} = \frac{n!}{m!(n-m)!}.$$

18.2.3 抽屉原理

定理 18.2.5 (第一抽屉原理). 如果将 m 个物件放入 n 个抽屉内, 那么必有一个抽屉内至少有 $\left\lceil \frac{m-1}{n} \right\rceil + 1$ 个物件。

针对第一抽屉原理, 有如下推论:

定理 18.2.6 (推论). 如果将 $\sum_{i=1}^n m_i + 1$ ($m_i \in Z^+, i = 1, 2, \dots, n$) 个物件放入 n 个抽屉内, 那么存在第 i 个抽屉内至少有 $m_i + 1$ 个物件。

定理 18.2.7 (第二抽屉原理). 如果将 m 个物件放入 n 个抽屉内, 那么必有一个抽屉内至多有 $\left\lfloor \frac{m}{n} \right\rfloor$ 个物件。

常规情况下, 将第一、第二抽屉原理合并, 表述如下:

定理 18.2.8 (抽屉原理). 把 m 个苹果放入 n 个抽屉里, 一定有一个抽屉至少有 $\left\lceil \frac{m}{n} \right\rceil$ 个苹果, 一定有一个抽屉至多有 $\left\lfloor \frac{m}{n} \right\rfloor$ 个苹果。

值得注意的是, 这里苹果的平均数即为 $\frac{m}{n}$, 而 $\left\lceil \frac{m}{n} \right\rceil$ 为这些抽屉里苹果的最大值, $\left\lfloor \frac{m}{n} \right\rfloor$ 为这些抽屉里苹果的最小值, 因此抽屉原理即为离散情况下的 $\max \geq \text{avg} \geq \min$ 不等式, 即最大值不小于平均值不小于最小值。

例 18.2.9 (第一抽屉原理). 一个班级有 35 名学生, 现在有 7 个小组, 每个小组可以容纳任意数量的学生。证明至少有一个小组的学生数不少于 6 名。

首先，学生数 $m = 35$ ，小组数 $n = 7$ 。

根据第一抽屉原理，如果我们将 35 个学生放入 7 个小组，那么至少有一个小组内的学生数不少于 $\lceil \frac{35}{7} \rceil = 5$ 名。

为了证明至少有一个小组有不少于 6 名学生，我们假设每个小组都有最多 5 名学生，那么总人数最多为 $7 \times 5 = 35$ 名。这时，所有小组刚好达到最大容量 5 名学生。因此，若再多出一名学生，必定有一个小组至少有 6 名学生。

因此，至少有一个小组的学生数不少于 6 名。

例 18.2.10 (第二抽屉原理). 某公司有 23 名员工，他们被分配到 5 个不同的办公室，证明至少有一个办公室的员工数不超过 5 人。

应用第二抽屉原理来解决这个问题，物件数 $m = 23$ ，抽屉数 $n = 5$ 。

根据第二抽屉原理，至少有一个抽屉的物件数至多为 $\lfloor \frac{m}{n} \rfloor$ 。在这里，
 $\left\lfloor \frac{23}{5} \right\rfloor = \lfloor 4.6 \rfloor = 4$ 。

因此，至少有一个办公室的员工数不超过 5 人。

例 18.2.11. 50 个苹果被放入 8 个抽屉里，证明至少有一个抽屉有不少于 7 个苹果，并且至少有一个抽屉的苹果数不超过 6 个。

分别应用第一抽屉原理和第二抽屉原理来解决这个问题。

1. 应用第一抽屉原理：

苹果数 $m = 50$ ，抽屉数 $n = 8$ 。

根据第一抽屉原理，至少有一个抽屉的苹果数不少于 $\lceil \frac{50}{8} \rceil = \lceil 6.25 \rceil = 7$ 。

因此，至少有一个抽屉有不少于 7 个苹果。

2. 应用第二抽屉原理：

根据第二抽屉原理，至少有一个抽屉的苹果数至多为 $\lfloor \frac{50}{8} \rfloor = \lfloor 6.25 \rfloor = 6$ 。

因此，至少有一个抽屉的苹果数不超过 6 个。

18.3 概率与期望

概率与期望作为概率论中常见的概念，在数理统计中有着重要的地位。

定义 18.3.1 (概率). 概率 (*Probability*)，亦称“或然率”，是数学中描述随机事件发生可能性的度量，反映的是随机事件出现的可能性大小，一般是 0 到 1 之间的一个数字。

更广泛的情况下，概率需要通过概率测度来定义，但在经典概率论中，如若每个可能结果是等可能的，则事件 A 的概率为：

$$P(A) = \frac{\text{事件 A 的结果数}}{\text{样本空间中的所有可能结果数}}$$

定义 18.3.2 (期望). 期望 (*Expectation*)，亦称“数学期望”，是随机变量取值在所有可能取值上的平均值，是最基本的数学特征之一。

举例来说，有一个骰子，这个骰子是正常的，均匀的，那么投掷到每一个面的概率都是 $\frac{1}{6}$ 。

点数	1	2	3	4	5	6
概率	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{6}$

那么这个骰子的数学期望就是：

$$\frac{1}{6} \times 1 + \frac{1}{6} \times 2 + \frac{1}{6} \times 3 + \frac{1}{6} \times 4 + \frac{1}{6} \times 5 + \frac{1}{6} \times 6 = 3.5$$

所以，期望其实有均值的意思，如果掷一次骰子并不会掷出 3.5，甚至也不一定就在 3 或者 4 上，但是根据大数定理，随着掷骰子的次数越多，投出来骰子的均值就会越接近 3.5 这个数字。

因此，期望的根本意义是在大数定理之下最终能够获得的收益。

若 X 是一个随机变量，其概率分布为 $P(X = x_i)$ ，则随机变量 X 的期望 $\mathbb{E}(X)$ 定义为：

$$\mathbb{E}(X) = \sum_{i=1}^n x_i P(X = x_i) \quad (i = 1, 2, \dots, n)$$

对于连续型随机变量 X ，期望定义为其概率密度函数 $f(x)$ 下的积分：

$$\mathbb{E}(X) = \int_{-\infty}^{\infty} x f(x) dx$$

例 18.3.3 (连续型随机变量的期望). 设随机变量 X 服从区间 $[0, 1]$ 上的均匀分布，求 X 的期望值。

例题中，随机变量 X 在区间 $[0, 1]$ 上的均匀分布，其概率密度函数为：

$$f(x) = \begin{cases} 1, & 0 \leq x \leq 1 \\ 0, & \text{其他情况} \end{cases}$$

根据期望 $\mathbb{E}(X)$ 的计算公式，这里积分区间是 $[0, 1]$ 且有 $f(x) = 1$ 在该区间，因此期望为：

$$\mathbb{E}(X) = \int_0^1 x \cdot 1 dx = \int_0^1 x dx$$

计算这个积分：

$$\mathbb{E}(X) = \left[\frac{x^2}{2} \right]_0^1 = \frac{1^2}{2} - \frac{0^2}{2} = \frac{1}{2}$$

所以，随机变量 X 的期望为 $\frac{1}{2}$ 。

18.4 杂项

18.4.1 图片与视频的大小计算

分辨率：分辨率就是屏幕上显示的像素个数，分辨率 160×128 的意思是水平方向含有像素数为 160 个，垂直方向像素数 128 个。屏幕尺寸相同，分辨率越高，显示效果就越精细和细腻。

问题一般为：给出一张图片的分辨率和色彩的位率（位率越高色彩越多），要算出这张图片所占的空间，计算公式为：

$$\text{水平方向像素数} \times \text{垂直方向像素数} \times \text{色彩位率} = \text{图片所占空间(bit)}$$

一个视频可以视为很多图片的集合，显然，图片的张数为时长乘帧率（每秒几张图片），计算公式为：

$$\text{图片所占空间} \times \text{视频时长} \times \text{视频帧数} = \text{视频所占空间(bit)}$$

18.4.2 音频的大小计算

计算一段音频文件的大小通常需要以下几个参数：

- 采样率：每秒钟采集的音频样本数，单位为赫兹 (Hz)。常见的采样率有 44.1kHz(44100Hz)、48kHz(48000Hz) 等。
- 量化位数：表示每个音频样本使用的位数，常见的有 16 位、24 位等。
- 声道数：音频的声道数，例如单声道、立体声 (2 声道)。
- 音频时长：音频的持续时间，单位为秒。

计算音频文件的大小的公式为：

$$\text{采样率} \times \text{量化位数} \times \text{音频时长} \times \text{声道数} = \text{音频所占空间(bit)}$$