中国大学生计算机系统与程序设计竞赛

CCF-CCSP-2017

时间：2017 年 10 月 26 日 08:30 ~ 22:00

题目名称	五子棋	Lisp 语言解释器	串行调度	图的探索	简单图数据库
题目类型	传统型	传统型	传统型	传统型	传统型
输入	标准输入	标准输入	标准输入	标准输入	标准输入
输出	标准输出	标准输出	标准输出	标准输出	标准输出
每个测试点时限	1.0 秒	1.0 秒	3.0 秒	60.0 秒	300.0 秒
内存限制	128 MB	128 MB	128 MB	750 MB	2048 MB
子任务数目	20	10	20	5	0
测试点是否等分	是	是	是	是	是

五子棋（wuzi）

【题目描述】

五子棋是世界智力运动会竞技项目之一，是一种两人对弈的纯策略型棋类游戏。通常双方分别使用黑白两色的棋子，下在棋盘直线与横线的交叉点上，先形成五子连珠者获胜。

五子连珠是在**横线，纵线，斜线，反斜线**四个方向上形成**五子及以上**的连线，当出现多于五子的连珠时，也只记为**一次**五子连珠。

五子连珠总数等于棋局中的所有方向上的五子连珠连线的数量之和。

我们想知道，给定一个长宽皆为 n 的棋局，**白棋**落在哪些点可以**增加白棋五子连珠总数**？

对**增加白棋五子连珠总数**的举例说明 (A 点为我们选择的落点):

1

```
wwwwAbbbb
```

落白棋之前未形成五子连珠，落入白棋之后，五子连珠总数加一，满足要求。

2

```
wwwwwAbbbb
```

落白棋之前已经形成五子连珠，落白棋之后，五子连珠总数不变，不满足要求。

3

```
wwwwwAwwwww
```

落白棋之前五子连珠总数为二，落入白棋之后，两边连成一线，五子连珠总数减一，不满足要求。

4

```
*w***w*****
**w**w*****
***w*w*****
****ww*****
wwwwwAwwwww
```

落白棋之前五子连珠总数为二。落入白棋之后，两边连成一线，斜向和纵向形成新的五子连珠，总数为三。五子连珠总数加一，这个点满足要求。

【输入格式】

从标准输入读入数据。

输入为第一行为一个数字 n（$n \le 40$），表示棋盘大小。

接下来的 n 行，每行为 n 个字符，可能有三种字符，* 表示无棋子，b 表示黑棋，w 表示白棋。

输入棋局中可能已经有五子连珠的情况，我们需要计算能增加白棋五子连珠总数的白棋落点。

【输出格式】

输出到标准输出。

输出为 k 行，包括 k 个满足要求的落点，k 个点按照从左至右，从上至下的顺序输出，即先按行排序，再按列排序输出。

每行为一个点坐标 x y，分别表示列坐标，行坐标，以空格分隔，坐标序号从 0 开始，棋盘左上角为原点。

```
*a***
*****
*****
*****
***b*
```

图中 a 点的坐标为 1 0，b 点的坐标为 3 4。

【样例输入】

```
8
****b*bb
*****b*b
bb*bbbw*
w*wbwwww
bwwbwwbw
ww**wbbw
*bww***w
***bwb*b
```

【样例输出】

```
7 2
2 5
3 5
4 6
2 7
```

【子任务】

测试点	说明
1,2	没有满足要求的点
3,4,5,6,7,8	只需要计算横纵方向, 棋局中不存在旧的五子连珠
9,10,11,12,13,14	需要计算所有方向, 棋局中不存在旧的五子连珠
15,16,17,18,19,20	盘面上存在旧的五子连珠

Lisp 语言解释器（lisp）

【题目背景】

Lisp 语言由 John McCarthy 教授（1927-2011，1971 年图灵奖得主）发明于 1958 年，是现今还在使用的第二古老的高级语言（最古老的是 Fortran，比 Lisp 发明还早一年）。Lisp 的语法与常见的编程语言很不一样，Lisp 程序由很多列表组成，Lisp 也得名于**列表处理器（List Processor）**。

本题的任务是编写一个列表处理器，实现 Lisp 语言的基本功能（当然你可以把它看作是 Lisp 语言的一个简化版本）。输入是一段 Lisp 程序，输出是这个程序的运行结果。下面将详细的介绍需要实现的功能。

【题目描述】

Lisp 是一种函数式编程语言，每一个表达式都可以计算出一个结果。Lisp 表达式有两种形式——**原子**或**列表**。所谓原子，简单的讲就是一个字符串，可以使用的字符有大小写英文字母、数字和 `+-*/!?=<>_` ；而列表则是由若干个表达式和一对括号在两侧括起构成。

原子的例子：

```
12
+
John
Burger
```

上面给出了一些原子的例子。`12` 就是一个整数，就像其他编程语言中的 `int` 一样，但后面三个却略有不同。在 Lisp 中，这些含有非数字字符的原子被称作**标识符**。实际上每一个标识符都指代了一个值，不过有些标识符对应的值 Lisp 已经自动绑定好了，比如 `+` 指代加法函数，而更多的标识符需要在程序中人为地指定一个值。下面则是几个列表的例子。在书写格式上，列表中的每两个表达式由一个空格分隔，左右再加一对圆括号括起。

列表的例子：

```
(f a b c)
(define x (+ 2 3))
(+ 1 1)
```

在 Lisp 中，函数的使用是通过列表来完成的。通常使用圆括号时，会把列表中的第一个表达式的值默认为是一个函数，后面表达式的值则被视为传入该函数的参数，而函数的返回值就是整个表达式的值。所以 `(+ 1 2)` 就代表了 1 和 2 做加法运算，该表达式的结果显然为 3 。

```
(+ 1 2)      ; 表示 1 和 2 做加法运算，结果为 3。
(+ (* 5 2) 3)    ; 先计算 5 乘 2，结果为 10；再与 3 相加，最终结果为 13。
```

在本题中，基本的函数有四个：加减乘除（整除），分别用标识符表示 `+` 、`-` 、`*` 、`/` 指代。而我们要处理的数字（包括运算的中间结果），也都是小于等于 10^9 的自然数，且保证整除运算第二个参数不为 0 。我们的列表处理器同样支持布尔类型，分别用标识符 `True` 和 `False` 指代逻辑值真和假。此外还有一些 Lisp 预先定义好的函数，我们将在下面逐一介绍。

eq?

判断两个整数或两个逻辑值是否相等。如果相等返回 `True` ，否则返回 `False` 。参数的值只能是整数。

```
(eq? 1 1)        ;True
(eq? 4 (+ 1 2))      ;False
(eq? (eq? 1 2) False)   ;True
```

define

这是唯一一个没有返回值的函数，所以它不能被嵌套在列表之中。它的作用是为标识符绑定一个值。要求每个标识符只能被绑定一次，Lisp 自带的标识符相当于已经被绑定过一次，所以不能再成为 `define` 函数的第一个参数。

```
(define a 5)         ;a 的值绑定为 5
(define add +)       ;add 的值绑定为加法函数
```

lambda

返回一个新定义的函数。第一个参数是一个由若干个（至少一个）标识符构成的列表，表示新函数的参数列表，这些标识符做为新函数的参数仅在第二个参数中有效。需要注意的是，这里虽然也使用了列表的形式，但无需进行函数运算。

第二个参数表示新函数的返回值。在第二个参数中使用的标识符有两种情况，一种是通过 `define` 定义的标识符，一种是做为新函数参数的标识符。因为在定义函数时并不需要进行具体计算，我们要做的只是把它先存储下来，所以在第二个参数中可以使用暂时还没有绑定过值的标识符，只要保证在使用该函数进行计算时每个标识符都已经绑定了值即可。另一方面，`lambda` 表达式可以嵌套使用，这就导致了在第二个参数中使用的标识符也可能是外层函数的参数。为避免歧义，Lisp 对标识符的取值采取就近原则，即优先解释为较近层的函数参数；如果在任何一层的参数列表中都找不到这个标识符，就采用 `define` 表达式绑定的值。

此外，一个单独的 lambda 表达式没有任何意义。在 Lisp 中，一个值为函数的表达式一定被嵌套在列表之中：或者在 define 函数中使用，或者做为一个匿名函数直接参与运算。

```
(define add3 (lambda (x y z) (+ (+ x y) z)))
;;; 定义了一个将三个参数相加的函数 add3;
((lambda (+ -) (* + -)) 2 3)
;;; 这里用匿名函数的形式定义了乘法函数，然后计算 2 乘 3;
(add3 2 3 4)
;;; 使用刚刚定义的 add3 函数，结果为 9。
```

cond

选择函数，参数个数不定但至少为 2 ，其中每个参数都是一个由两个表达式构成的列表，且第一个表达式的值一定是逻辑类型（真或假）。类似地，这里的列表也不起函数计算作用。

cond 函数会依次检查每个参数的第一个表达式，如果值为真，则将第二个表达式的值返回，并且不再继续检查后面的参数。

```
(cond ((eq? a 2) 0) ((eq? a 3) 1) (True 2))
;;; a 等于 2 时返回 0，等于 3 时返回 1，其它情况返回 2。
```

保证每个 cond 函数至少存在至少一个参数，其第一个表达式为真。

【输入格式】

从标准输入读入数据。

一段 Lisp 程序，其中每行是一个表达式。

保证程序格式正确无误，运行时不会出现任何异常（每个函数都一定会有返回值、进行计算时不会遇到无法解释的标识符等等）。

【输出格式】

输出到标准输出。

对于 Lisp 程序中每一行的表达式，相应输出一行。

如果该表达式使用了 define 函数则输出 define ，否则输出该表达式的值。

【样例 1 输入】

```
(define y 10)
(define f (lambda (x y) (+ x ((lambda (x) (* x y)) y))))
(f 1 2)
y
```

【样例 1 输出】

```
define
define
5
10
```

【样例 1 解释】

定义了一个二元函数：$f(x,y) = x + y \times y$。

【样例 2 输入】

```
(define y 10)
(define sqr+y (lambda (x) (+ y (* x x))))
(define f (lambda (x y) (sqr+y x)))
(sqr+y 5)
(f 5 1)
```

【样例 2 输出】

```
define
define
define
35
35
```

【样例 2 解释】

`f` 函数的参数 `y` 并不能在 `sqr+y` 函数中起作用，即使 `f` 调用了 `sqr+y` 。

【样例 3 输入】

```
(define fact (lambda (n) (cond ((eq? n 1) 1) (True (* n (fact (- n 1)))))))
(fact 1)
(fact 5)
(fact 10)
(define sum (lambda (n) (cond ((eq? n 1) 1) (True (+ n (sum (- n 1)))))))
(sum 50)
```

【样例 3 输出】

```
define
1
120
3628800
define
1275
```

【样例 4 输入】

```
(define fun1 (lambda (x) (cond ((eq? x 0) 1) (True (fun2 (- x 1))))))
(define fun2 (lambda (x) (cond ((eq? x 0) 2) (True (fun1 (/ x 2))))))
(fun1 2)
(fun2 2)
(fun1 5)
(fun2 5)
```

【样例 4 输出】

```
define
define
1
2
1
1
```

【提示】

仔细审题并深入分析样例可能会事半功倍哦 ~

【子任务】

前 30% 的数据，没有 `lambda` 函数；

前 60% 的数据，`lambda` 函数不会嵌套出现；

对于 100% 的数据，输入程序的行数小于等于 200 ，每一行不多于 200 个字符（换行符不计），且函数调用时深度不会超过 50 。调用 Lisp 自带函数和匿名定义的函数不计入函数调用深度，样例 3 中计算 `sum(50)` 时函数调用深度恰好为 50 。

串行调度（serial）

【题目描述】

在数据库中，我们通常把能访问并可能更新各种数据项的一个程序执行单元称作“事务”。事务运用以下两个指令访问数据：

1. `READ(x)`：从数据库把数据项 x 传送到执行 `READ` 指令的事务的局部缓冲区；
2. `WRITE(x)`：从执行 `WRITE` 的事务把数据项 x 传回数据库。

当事务执行 `READ(x)` 指令时，将从数据库中得到数据项 x 的值，备份在该事务的局部缓冲区内。接下来经过对获得数据的计算处理，可以再使用 `WRTIE(x)` 指令去更新数据项 x 在数据库中的值。

我们做出以下几点约定：

1. 所有可以由指令访问的数据项有 n 个，依次记做 $x[1], \ldots, x[n]$ ；
2. 系统不会同时执行两个指令，一定是在执行完一个后再执行另一个，并且指令不会执行失败；
3. `WRITE(x)` 指令不会延迟更新，即该指令执行结束意味着数据库中数据项 x 的值已更新。

这里举一个简单的例子，假设 $n = 3$，三个数据项 $x[1], x[2], x[3]$ 依次代表 Alice、Bob 和 Dundun 的账户余额。那么事务 Alice 转账 50 元给 Bob 可以表示为如下形式：

事务 1：Alice 转账 50 元给 Bob

```
Tmp = READ(x[1]);
Tmp -= 50;
WRITE(x[1], Tmp);
Tmp = READ(x[2]);
Tmp += 50;
WRITE(x[2], Tmp);
```

这里 `Tmp` 是事务 1 缓冲区的一个临时变量，辅助完成对更新后余额的计算。出于某种考虑，现在我们只关心一个事务对哪些数据项执行了读写指令，而具体的数值我们可以忽略不管，所以事务 1 又可以简化为下面的形式：

事务 1（简）：Alice 转账给 Bob

```
READ(x[1]);
WRITE(x[1]);
READ(x[2]);
WRITE(x[2]);
```

类似地，这里再给出另一个简单的例子：

事务 2：Dundun 给 Bob 存入自己的余额那么多的钱（这笔钱并不从 Dundun 账户中出，所以无须 `WRITE(x[3])`）

```
READ(x[3]);
READ(x[2]);
WRITE(x[2]);
```

对于任意一个事务，其中的指令均是按顺序执行的。但是，数据库系统通常会并发地执行多个事务。通俗地说，就是执行完事务 1 的一个指令后，接下来可能会去执行事务 2 的下一个指令，然后继续执行事务 2 或者回来执行事务 1 都是完全有可能的。

这样并发管理事务的好处这里就不提了，我们还是来看看坏处吧。对于给定的 m 个事务，记做 $T[1],\ldots,T[m]$，显而易见的是这 m 个事务会有很多种不同的执行顺序，我们把这个执行顺序称为调度。对于上面的事务 1 和 2，下面是两种不同的调度。

调度 1：串行

```
T[1]: READ(x[1]);
T[1]: WRITE(x[1]);
T[1]: READ(x[2]);
T[1]: WRITE(x[2]);
T[2]: READ(x[3]);
T[2]: READ(x[2]);
T[2]: WRITE(x[2]);
```

调度 2：交错执行

```
T[2]: READ(x[3]);
T[1]: READ(x[1]);
T[1]: WRITE(x[1]);
T[2]: READ(x[2]);
T[1]: READ(x[2]);
T[2]: WRITE(x[2]);
T[1]: WRITE(x[2]);
```

仔细分析的话不难发现，这两个调度的执行结果可能会不同，即按照这两种不同的顺序执行完所有事务，数据项最终的值可能会不同。观察调度 2 的最后两行，$T[1]$ 的结果会把 $T[2]$ 的覆盖掉，导致 Bob 账户只多了 50 块钱，Dundun 的馈赠就这样因为一个糟糕的调度不翼而飞了。

为了保证数据完整性，数据库系统维护事务要保证隔离性（isolation）：尽管多个事务并发执行，但对于任何一对事务 $T[i]$ 和 $T[j]$，在 $T[i]$ 看来，$T[j]$ 或者在 $T[i]$ 开始前

已经完成执行，或者在 $T[i]$ 完成之后开始执行。这样，每个事务都感觉不到系统中有其他事务在并发地执行。因为调度的不确定性，在不加额外控制手段（比如加锁）的情况下，有些事务并不能同时在系统中并发执行。

在一个调度中，如果某个事务的所有指令是连续执行的（即其中没有穿插其他事务的指令），我们就称该事务被**串行执行**。像调度 1 这样所有事务都被串行执行的，我们称之为**串行调度**。串行调度显然是最理想的，所有事务不会互相干扰。对于一个给定的事务集，虽然有些调度并不是串行调度，但不同事务也不会产生冲突。接下来我们在调度上定义一个等价关系，来明确哪些调度和串行调度一样可靠。

考虑一个调度上**连续**的两条属于**不同事务**的指令，如果它们**针对不同的数据项**或者**同为 READ 指令**，则交换这两条指令可以得到一个等价的调度。不断交换两条满足上述要求的指令，则可以得到一系列等价的调度（即具有传递性）。如果一个调度等价于任意一个串行化调度，则称该调度是可串行化的，执行该调度时不会破坏数据完整性。

现在 Dundun 有一个在 n 个数据项、m 个事务上的**可串行化的**调度方案，希望你能帮他完成下面两个任务：

1. 找到一个与给定调度等价的串行调度，以证明 Dundun 的调度方案确实是可串行化的。如果有多个等价的串行调度，请输出字典序最小的那种，即优先执行编号较小的事务。
2. 解决 q 个如下的问题：对于给定的两个事务 $T[i]$ 和 $T[j]$，判断是否存在某个等价的串行调度，满足事务 $T[i]$ 可以在事务 $T[j]$ 之前完成。

【输入格式】

从标准输入读入数据。

第一行四个正整数 n、m、p 和 q，分别表示数据项的个数、事务个数、指令的个数及任务 2 的问题个数，数据项和事务编号均从 1 开始。

接下来 p 行，每行描述一个指令，包含三个整数：

- 第一个数为 op，表示指令的类型，其中 0 表示读指令，1 表示写指令，保证不会有其他的取值；
- 第二个数为 k，表示该指令的数据项为 $x[k]$，保证 $1 \le k \le n$；
- 第三个数为 w，表示该指令属于事务 $T[w]$，保证 $1 \le w \le m$，且每个事务至少包含一条指令。

这 p 条指令的顺序即为 Dundun 给出的这个可串行化的调度方案。

接下来 q 行，每行用空格分隔的两个整数 i 和 j，表示任务 2 的一个问题，保证 $1 \le i, j \le m$。

上述各行的整数之间用一个空格隔开。

【输出格式】

输出到标准输出。

第一行输出任务一的答案，输出一个 1 到 m 的排列，表示字典序最小的串行调度方案中 m 个事务的执行顺序，每两个整数用空格分隔。

接下来 q 行，每行对应任务 2 的一个问题，如果存在输出 `YES`，否则输出 `NO`。

【样例输入】

```
3 2 7 2
0 1 1
1 1 1
0 3 2
0 2 2
1 2 2
0 2 1
1 2 1
1 2
2 1
```

【样例输出】

```
2 1
NO
YES
```

【数据生成方式】

这里描述了本题目所有测试点的生成方式。你不需要自己生成数据，这部分只是为了方便你分析数据的性质。

下文中所有称“随机生成”的数，都是调用 python 语言的 `random.randint` 生成符合范围的随机整数，可以近似认为每个可能的取值等概率。

生成数据的步骤如下：

1. 使用 `random.shuffle`（近似于所有排列等概率）生成一个 1 到 m 的排列，作为初始串行调度方案中各事务的排列顺序。
2. 对于每个事务，生成恰好 $\frac{p}{m}$ 条指令，每条指令的 op 和 k 随机生成。
3. 不断地重复下列步骤，直到所有指令都被执行：
    - 在还没被执行的指令中找到所有可以通过等价调度变换到最靠前执行的集合 S；

- 从 S 中随机抽取一条执行。

4. 随机生成 q 对不同的整数 i 和 j，作为任务 2 的问题。

【子任务】

测试点	$n=$	$m=$	$p=$	$q=$
1	3	3	6	60
2	5	5	15	10^2
3	10	10	40	0
4	15	15	75	300
5	20	20	120	0
6	150	150	1,500	3,000
7	200	200	2,000	0
8	300	300	3,300	6,000
9	500	500	6,000	0
10	2,000	2,000	3×10^4	4×10^4
11	3,000	3,000	48,000	0
12	5,000	5,000	85,000	10^5
13	10^4	10^4	1.8×10^5	0
14				2×10^5
15	12,000	12,000	216,000	2.4×10^5
16	15,000	15,000	285,000	0
17				3×10^5
18	18,000	18,000	342,000	3.6×10^5
19	2×10^4	2×10^4	4×10^5	0
20				4×10^5

图的探索（memory）

【题目背景】

在社交网络中，你和任何一个陌生人之间所间隔的人不会超过六个，也就是说，最多通过六个人你就能够认识任何一个陌生人。

这就是著名的六度分隔现象，我们可以将社交网络模拟为图 。

【题目描述】

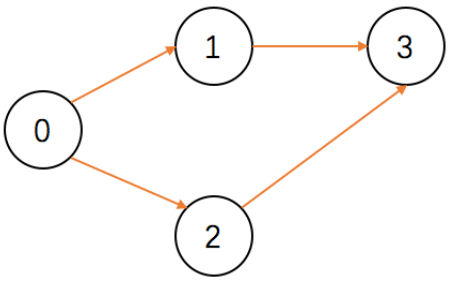
有向图 G 是由点与有向边构成的一种数据结构，记为 $G = (V, E)$ 。其中：V 为所有结点的集合，E 为所有有向边的集合。

点的总数为 $|V|$ （点的 id 范围为 0 到 $|V| - 1$），边的总数为 $|E|$ 。

有向边 $e = (u, v)$ 为一条由点 u 出发到达点 v 的边，u 为源结点，v 为目标结点，表明 u 可以通过边 e 到达 v ，我们规定每条边的距离为 1 ，即 u 与 v 的距离为 1 。

对应到现实的社交网络中，结点对应到社交网络中的成员，有向边 $e = (u, v)$ 对应成员之间的关系：u 认识 v 。

下图是用有向图描述社交网络认识关系的例子：



其中：

$V = \{0, 1, 2, 3\}$

$E = \{(0, 1), (0, 2), (1, 3), (2, 3)\}$

在社交网络中，一个人能通过多层关系找到最远的人和这个人通过任何方式都无法找到的人数能很好地刻画这个人的社交关系。在这个题目中，我们需要对指定的几个人求出这些数据。

具体地，我们会给出若干个社交网络成员对应的结点 s，你需要从 s 出发沿着有向边探索，以最近的距离，找到能探索到的所有结点中，距离最远的结点 d ，即最短距离的最大者。你需要输出点 d 的 id，点 d 与出发结点 s 的距离。当存在多个符合标准的结点时，输出 id 最大的。此外，你还要统计从点 s 出发进行探索之后，探索不到的结点的数量，并输出。

在上图的例子中，假设我们关注的结点 s 为点 0。从点 0 出发，最远能探索到点为 3，距离为 2。该图中所有点均能从 0 出发被探索到。因此本例输出的结果为: `3 2 0`。

【C/C++ 解题框架】

你需要在 `solve.c/cpp` 中实现预处理的 `my_init()` 函数与探索的 `my_solve()` 函数。

程序开始运行时，会先调用你的预处理函数，此时你可以读写文件、修改变量或进行其他操作，但你并不能获知需要探索的结点 s；之后会调用 5 次 `my_solve()` 函数，每次调用会给你一个结点 s，你需要在函数中调用一次且仅一次 `MAIN_output()` 函数来报告此次探索的答案。

具体的，在所给的框架中，包含以下一些文件、文件夹：

- `Makefile`：用于编译，生成可执行文件 `my_run` 。
- `main.c/cpp`：程序的入口，其中：
  - 会调用预处理的 `my_init()` 函数与进行探索的 `my_solve()` 函数。
  - `MAIN_output()` 函数用于输出每次探索的结果：最远的结点、距离、未探索到的结点总数。
  - 实际测评使用的 `main.c/cpp` 会有所不同。
- `solve.h`：声明了 `my_init()` 、`my_solve()` 、`MAIN_output()` 。
- `solve.c/cpp`：你所需要填写的代码文件，其中：
  - 你需要实现 `my_init()` 、`my_solve()` 函数。
  - 你可以在该文件中任意添加你要用到的变量、函数和其他代码。
- `run.sh`：运行程序的脚本。
- `case0`：文件夹。参赛者用于测试自己程序的一个较小的测试样例。其中包含图的信息文件、图数据的文本文件（**实际测评时不提供文本格式的文件**）、图数据的二进制文件、参考答案。

程序会自动统计你的预处理的用时、之后的 5 次探索的总用时。你各部分所用的时间会影响你的得分，将在下文的“评分方法”中介绍。

在答题中，有以下**注意事项**：

- 你只能修改并提交 `solve.c/cpp` 文件，修改其他文件是没有意义的。
- 你只能使用 `MAIN_output()` 函数告诉框架探索结果，提交的代码中不能用其他任何方式输出任何额外信息。
- 你的 `my_solve()` 函数每次被调用时，都必须调用恰好一次 `MAIN_output()`，不能多次调用，也不能不调用。

`make` 所使用的编译命令为 `gcc -O2 main.c solve.c -o my_run` / `g++ -O2 main.cpp solve.cpp -o my_run`

run.sh 中的运行命令为 ./my_run info.txt graph_binary_little_endian graph_binary_big_endian

三个参数都是文件路径，具体意义见“输入格式”。

【Java 解题框架】

你需要在 MainSolve.java 中实现预处理的 myInit() 函数与探索的 mySolve() 函数。

程序开始运行时，会先调用你的预处理函数，此时你可以读写文件、修改变量或进行其他操作，但你并不能获知需要探索的结点 s；之后会调用 5 次 mySolve() 函数，每次调用会给你一个结点 s，你需要在函数中调用一次且仅一次 mainOutput() 函数来报告此次探索的答案。

具体的，在所给的框架中，包含以下一些文件、文件夹：

- Makefile：用于编译，生成 .class 文件。
- MyMain.java：程序的入口，其中：
  - 会调用预处理的 myInit() 函数与进行探索的 mySolve() 函数。
  - 实际测评使用的 MyMain.java 会有所不同。
- MainOutput.java：实现了 mainOutput() 函数。
  - mainOutput() 函数用于输出每次探索的结果：最远的结点、距离、未探索到的结点总数。
- MainSolve.java：你所需要填写的代码文件，其中：
  - 你需要实现 myInit()、mySolve() 函数。
  - 你可以在该文件中任意添加你要用到的变量、函数和其他代码。
- run.sh：运行程序的脚本。
- case0：文件夹。参赛者用于测试自己程序的一个较小的测试样例。其中包含图的信息文件、图数据的文本文件（**实际测评时不提供文本格式的文件**）、图数据的二进制文件、参考答案。

程序会自动统计你的预处理的用时、之后的 5 次探索的总用时。你各部分所用的时间会影响你的得分，将在下文的“评分方法”中介绍。

在答题中，有以下**注意事项**：

- 你只能修改并提交 MainSolve.java 文件，修改其他文件是没有意义的。
- 你只能使用 mainOutput() 函数告诉框架探索结果，提交的代码中不能用其他任何方式输出任何额外信息。
- 你的 mySolve() 函数每次被调用时，都必须调用恰好一次 mainOutput()，不能多次调用，也不能不调用。

make 所使用的编译命令为 javac MyMain.java MainSolve.java MainOutput.java
run.sh 中的运行命令为 java MyMain info.txt graph_binary_little_endian graph_binary_big_endian

三个参数都是文件路径，具体意义见“输入格式”。

【输入格式】

你编译生成的程序 my_run 可以从如下文件中读入：

- info.txt 为测试所用到的图的信息，两行，分别为：点数 $|V|$ 、边数 $|E|$ 。
- graph_binary_little(big)_endian 为测试所用到的图的数据，以小端（大端）模式二进制形式存放所有的边，每条边 (u,v) 中的每个点均使用一个 int 来表示。该文件中的边，均已经升序排好，先按照边的源结点排序，对每一个点的所有出边，也已按照目标结点升序排好。

如上面的图中，一共有 4 条边 $\{(0,1),(0,2),(1,3),(2,3)\}$
因此在 graph_binary_big_endian 会是:

```
0x00000000 0x00000001 0x00000000 0x00000002
0x00000001 0x00000003 0x00000002 0x00000003
```

在 graph_binary_little_endian 会是:

```
0x00000000 0x01000000 0x00000000 0x02000000
0x01000000 0x03000000 0x02000000 0x03000000
```

你实现的函数中，可以根据需要读取这两个文件。
一共有 5 个测试点，每个测试样例进行 5 次探索。保证 $|V| < 5 \times 10^6$，$|E| < 3 \times 10^8$。

【样例】

在 case0 文件夹中含有以下文件：

- info.txt 图的信息文件，两行，分别为：点数 $|V|$ 、边数 $|E|$ 。
- graph.txt 图数据的文本文件 (实际测评时不提供文本格式的文件)，已升序排好。
- graph_binary_little_endian 图数据的二进制文件，将 graph.txt 中的每条边，以小端模式二进制形式存放。
- graph_binary_big_endian 图数据的二进制文件，将 graph.txt 中的每条边，以大端模式二进制形式存放。
- answer.txt 参考答案。

case0 所使用的图数据为使用 R-MAT 生成的符合 power-law 的图。

【调试方法】

如果你想测试自己的图：

1. 按照要求自己生成 `info.txt`、`graph.txt`、`graph_binary_little_endian`、`graph_binary_big_endian` 等文件；
2. 相应修改 `run.sh` 运行命令中的文件路径。

如果你想测试自己的点 s：

- C/C++：修改 `main.c/cpp` 中传给 `my_solve()` 函数的第 3 个参数，重新编译生成新的可执行文件。
- Java：修改 `MyMain.java` 中传给 `mySolve()` 函数的第 2 个参数，重新编译生成新的 `class` 文件。

【子任务】

测试点	$\|V\|$	$\|E\|$	基准时间 (s)	来源或生成方式
1	9,996	99,933	-1	R-MAT 生成图
2	4,847,571	68,993,773	8	LiveJournal social network
3	3,072,626	117,185,083		Orkut social network(部分)
4		234,370,166	12	Orkut social network(全部)
5	3,999,983	266,903,057		R-MAT 生成图

其中 R-MAT 生成图均与样例为同一代码生成，边的分布服从 power-law。

【评分方法】

100 = 25（正确性）+25（基准性能）+50（全场性能评比）

其中 25 分为正确性得分：每个测试点满分为 5 分，每个测试点一共 5 次查询，一次 1 分，即：$25 = 5 \times 5 \times 1$。你的每个测试点需要在 $60.0s$ 的时间内完成，包括框架用到的时间（$1s$ 内）、预处理用的时间、5 次查询用的时间。

对于 case2、3、4、5，若该测试点的 5 次查询全部正确，就会进行基准性能评估，基准性能满分时，就会进行全场性能评比。

每个测试点会统计 5 次查询的总用时 t，注意：这里不算入预处理所用时间。

25 分为基准性能得分：每个测试点满分为 6.25 分，当这个测试点的用时 t 小于该测试点的基准时间时，一次性得到满分 6.25 分。

50 分为性能评比得分：每个测试点满分为 12.5 分，设 $t_{\min}$ 为所有选手本测试点的 t 的最小值，该测试点的性能得分为：

$$12.5 \times \frac{t_{\min}}{t}$$

【提示】

- 本题中，一些测试点的图数据会非常大，请注意内存的使用。
- 社交网络中，两个人大多时候是相互认识或相互不认识，但也存在 u 认识 v 但 v 不认识 u 的情况。
- 社交网络中，有些人会认识很多人，有些人认识很少的人，甚至谁都不认识。
- 社交网络中，相邻的人经常会形成一些大大小小的团体，团体中的人往往相互认识。
- 社交网络中，一个人所认识的人可能分布比较广泛，如一个人的大学同学往往来自于全国不同城市，一个城市的人也可能生活在不同地区。
- 你可以使用 `"rb"` 参数来打开二进制文件，进而读取数据。

简单图数据库（graphdb）

【题目背景】

北岛曾经写过一首名为《生活》的一字诗：网。这一个字便概括出了这个世界上各种事物之间错综复杂的关联。没有什么事物是孤立的，关联无处不在。而图正是描述事物之间关联性的一个强有力的工具。

例如，在社交网络上，对于给定的一个用户，找出他/她的朋友、他/她的朋友的朋友、他/她的朋友的朋友的朋友中，有多少人拥有和他同样的生肖。这类查询在实际应用中非常常见。

传统的关系型数据库在处理图上的查询时往往效率不高，而图数据库将图作为第一公民，在数据结构的组织和对查询的处理等方面有很多针对图的专门的优化，这使得图数据库在一些特定应用场合表现出远超传统关系型数据库的效率，从而成为了业内新宠。

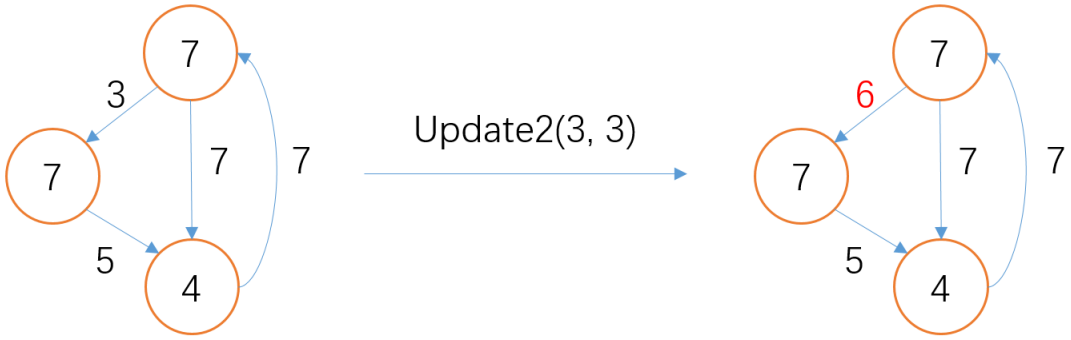
【题目描述】

本题要求你实现一个单机图数据库上的几个简单功能。

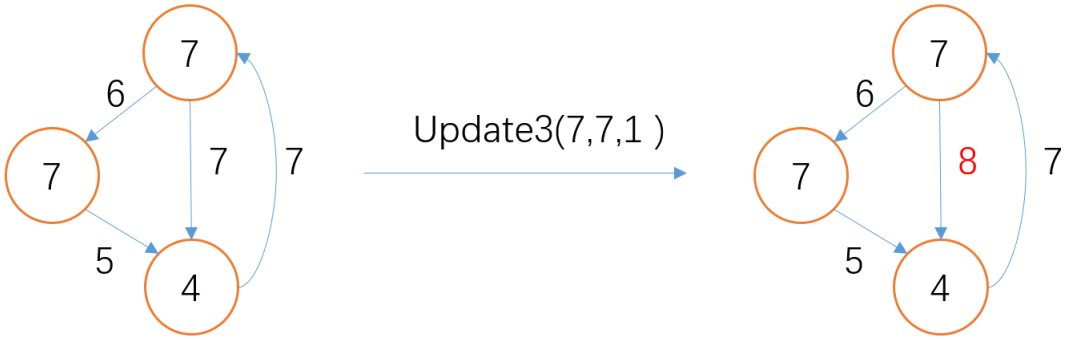
假设数据库中存在一张有向图，图上有 n 个点，每个点有唯一的编号，编号依次为 $0, 1, 2...n-1$ ，图上的每个点都有个属性值，每条边也都有个属性值，你需要实现如下 5 个针对此图的操作：

操作 1：$update1(a, b)$：表示将属性值为 a 的整数倍的点的属性值加上 b。

操作 2：$update2(a, b)$：表示将属性值为 a 的整数倍的边的属性值加上 b。

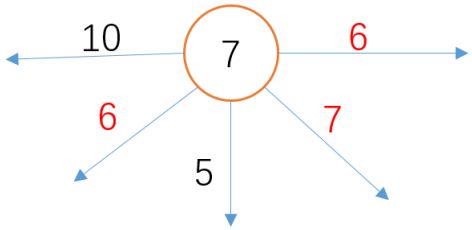


操作 3：$update3(a, b, c)$：表示将属性值为 a 的整数倍的点的出边中，属性值为 b 的整数倍的边的属性值加上 c



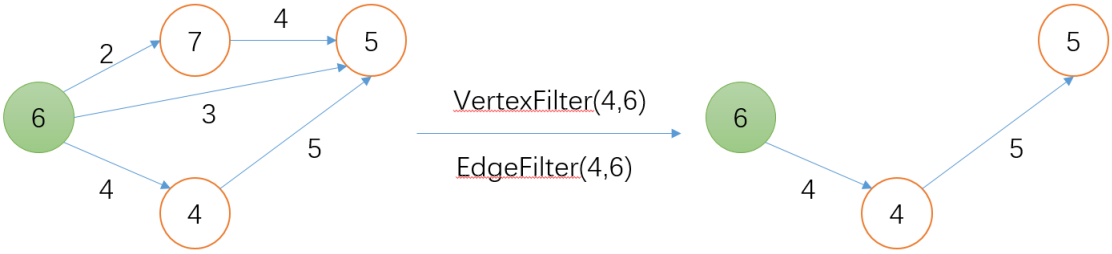
注意：在操作 1、2 和 3 中，如果一个属性加上另一个数后得到的新的值大于 M，则需要将其减去 M。例如，对于 $update1(3, M-1)$，如果一个点的属性值为 3，则执行此操作后新的属性值等于 2。

操作 4：$query1(u, v, x, y)$，表示查询点的**编号**在 u 到 v 之间的所有点的出边中，**属性范围**在 x 到 y 的边的数量之和。



假设上图中是 0 号节点及其所有出边，7 是其点的属性值，$10, 6, 5, 7, 6$ 是其出边的属性值，那么 $query1(0, 0, 6, 7)$ 的返回值就是 3。

操作 5：$query2(s, d, u, v, x, y)$，表示查询点的**属性范围**在 u 到 v，边的**属性范围**在 x 到 y 的子图中，点 s 在 d 跳以内可以达到的点的数目，如果 s 本身不在该子图中，则返回 0



假设上图绿色的点是起点，编号为 0，那么 $query2(0,1,4,6,4,6)$ 的返回值是 2。对于属性值为 7 的那个点，因为该点本身不包含在子图中，所以其所有的邻边，不论属性值如何，都不包含在子图中。所以符合的要求的点有两个，一个是 0 点本身，一个是其相邻的属性为 4 的点。

【输入格式】

读入有向图

你需要从指定的文本文件中读取有向图的数据。正式测试时的文件路径为 `./graph.data`，文件的格式如下：

第 1 行是一个正整数 n，表示图中点的个数。点的编号为 0 到 $n-1$。

第 2 行是 n 个正整数，第 i 个数表示编号为 $i-1$ 的点的初始属性值。保证这 n 个正整数均不超过 M。

第 3 行是一个整数，表示图中总的边数。

第 4 行到第 $n+3$ 行，第 $4+i$ 行表示编号为 i 的点的所有出边。其中每行开始有一个非负整数 x 表示该点有 x 条出边，接下来有 x 对整数，第 j 对整数 (d,l) 表示该点的第 j 条边的终点编号为 d，该边的初始属性值为 l。保证 $0 \le d < n$，且 $0 < l \le M$。

读入操作序列

你需要从标准输入流中读取你需要完成的操作。格式如下：

第 1 行是 1 个正整数 q，表示操作的个数。

第 2 行到 $n+1$ 行中，第 $i+1$ 行表示第 i 个操作。每行第一个字符串表示操作的类型，`u1`，`u2`，`u3` 分别表示 $update1,2,3$，`q1`，`q2` 别表示 $query1,2$。接下来的若干整数则是该操作的参数。每个操作的格式如下：

- $update1(a,b)$：`u1 a b`
- $update2(a,b)$：`u2 a b`
- $update3(a,b,c)$：`u3 a b c`
- $query1(u,v,x,y)$：`q1 u v x y`
- $query2(s,d,u,v,x,y)$：`q2 s d u v x y`

上述操作各自参数的含义参见题目描述。

本题中 M 等于 10^6。

*update*1 和 *update*2 中的 a，*update*3 中的 a 和 b，在大多数情况下不超过 10。

*update*1 和 *update*2 中的 a,b，*update*3 中的 a,b,c，*query*1 中的 x,y，*query*2 中的 u,v,x,y 均为不超过 M 的正整数。

对于同一个操作中的 u,v 或 x,y，有 $u \le v, x \le y$。特别地，*query*1 中 $0 \le v-u \le 10000$。

*query*1 中的 u,v 和 *query*2 中的 s,d 均为小于 n 的非负整数。

【输出格式】

你需要输出到标准输出流。

对于 *update*，你不需要输出任何东西。对于每个 *query*1 或 *query*2，你需要输出一个整数，表示所求的答案。每个整数单独占据一行，请不要输出多余的空格和空行。

【样例 1 输入】

有向图文件：

```
7
6 7 4 5 6 6 4
9
3 1 2 2 4 3 3
1 3 4
1 3 5
0
2 5 6 6 7
1 6 5
1 4 7
```

操作序列：

```
7
q2 0 1 4 6 4 6
q1 4 6 3 6
u1 3 1
u2 3 3
u3 7 7 1
q1 4 6 3 5
```

```
q1 4 6 8 8
```

【样例 1 输出】

```
2
2
1
1
```

【评分方式】

测试点：一共有 2 个测试点，每个测试点有多个操作。

实际评测中用到的有向图和参赛选手本机的有向图完全相同，各类操作的数目和参赛选手本机的测试数据大致相同。

计分方法：总分 100 分，每个测试点 50 分。对于每个测试点，如果该测试点的输出有错、超时、超内存限制或运行错误等则该测试得 0 分；若全部正确则首先得到 10 分的正确分，剩余 40 分是性能得分，通过和其他参赛者的计算时间的对比算出（用 40 分乘上最快的参赛者的程序耗时，再除以该参赛者的程序耗时）。

【提示】

评测机器为多核机器，所以你可以用并行化的方法来加速计算。但是你输出的结果必须和串行依次执行每个操作的输出结果完全相同。