北京郵電大學

实验报告

实验名称: 加里森的任务

班级 : 2019211309 姓名 : 陈悦 学号 : 2019211413 分工 : 文档

班级 : 2019211309 姓名 : 马晓亮 学号 : 2019211400 分工 : 代码

2020 年 *10* 月 *13* 日

目录

一　需求分析

需求中要求程序求出 n, x, y 的关系，使加里森成为最后一个队员。所以我们可以设计一个程序，利用它来求出一定范围内所有三元组 (n, x, y) 满足 1 号队员最后一个执行任务。由于确定了 n 与 y 后，x 是唯一的，所以我们可以设置上界 N_{max} 与 Y_max，输出所有符合 $1 \leqslant n \leqslant N_{max}$ 与 $1 \leqslant y \leqslant Y_{max}$ 的三元组。

这样，我们的输入仅有一行，为 N_{max} 与 Y_{max}。同时，我们对输入作出要求，$N_{max} \geqslant 1$ 且 $Y_{max} \geqslant 1$

然后我们会按顺序输出 (n, y, x), 每个占一行，先从 $n = 1$ 开始输出到 $n = N_{max}$，才能递增 y。

例如输入

2 2

输出为

1. (1, 1, 1)
2. (2, 1, 1)
3. (1, 2, 1)
4. (2, 2, 2)

同时如果输入非法，例如 -1 2

输出为

输入错误 (Input Error)

二　概要设计

在 link.h 中定义链表数据结构:

1. solve 函数为链表对象唯一对外可以操作的函数，返回特定 n，y 下的 x;
2. delete_node 函数删去链表结点的下一个结点;
3. insert 使用头插插入一个新的结点;

main.cpp:

1. 调用 link 的 solve 得到特定 n 与 y 的结果。

三　详细设计

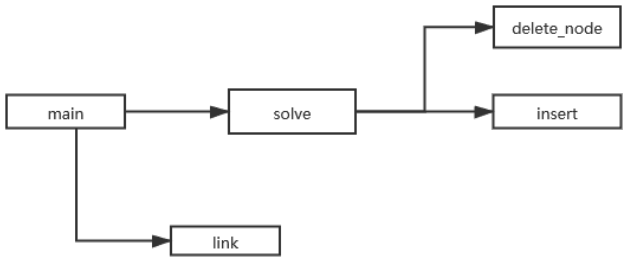

图 1: 总体流程图

Algorithm 1 Solve

Input: 队员总数 n 与间隔 y
Output: 加里森的位置

初始化链表，用链表模拟队员的顺序
用 *cur* 指针指向头指针
while *cur* 不是最后一个队员 **do**
    cur 向后移动 $y-1$ 次
    删除 *cur* 指向的下一个队员
end while
利用最后一个队员的编号计算 x 的值
return x;

Algorithm 2 insert

Input: 队员的编号

初始化结点，结点的编号为队员的编号
结点的后驱指向 head 的后驱
head 的后驱指向结点

Algorithm 3 delete_node

Input: 前驱结点 pre
Output: 删除结点的后驱

pre 的后驱设置为要删除结点的后驱
释放原 pre 的后驱

四　调试分析报告

在代码编写过程中，笔者曾经遇到过使用 delete_node 函数删除过头指针的情况。这是因为当 cur 指向最后一名队员时，没有直接过渡到头指针造成的。

对于这个过程的算法分析较为简单，我们模拟一组 (n, x, y) 操作的复杂度为 $O(ny)$, 因为我们每 y 个人删去一个人，总共要删去 n 个人。随后我们计算所有情况下的计算次数

$$\sum_{y=1}^{Y_{max}} \sum_{n=1}^{N_{max}} ny = O(N_{max}^2 Y_{max}^2)$$

同时，在分析固定 y 情况下的结果时，我们得到了一些有趣的关系图片。在固定 y 且 $x = 1$ 的情况下，设 $result_n$ 为在此情况下最后出界的队员编号。在已知 $result$ 的情况下，我们可以反推出在何种情况下可以使加里森成为最后一个队员。

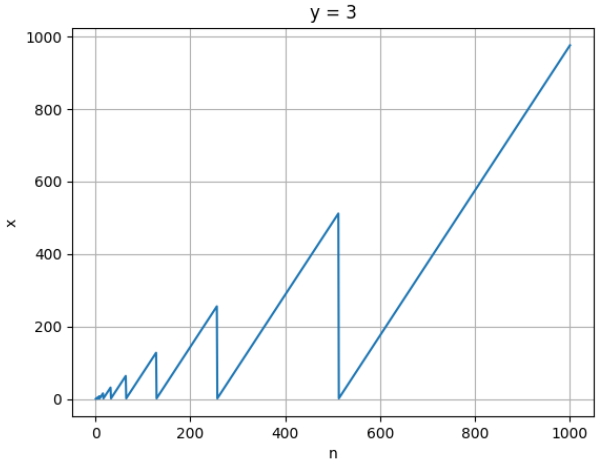

图 2: y = 3 时 n 与 x 的关系

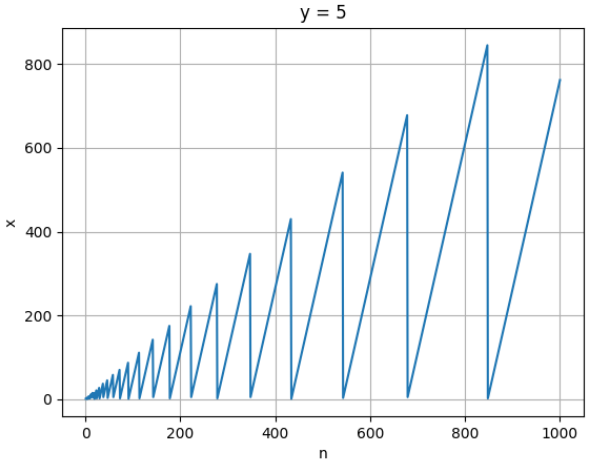

图 3: y = 5 时 n 与 x 的关系

我们惊奇的发现在固定 y 的情况下,n 与 x 的关系是一个前后之差为 y 的等差数列，并且有一定的周期性。根据多次测试后，我们得出了一个可靠的规律。

$$result_n = (result_{n-1} + y - 1)\%n + 1$$

利用这个关系,我们可以不使用模拟,而使用递推的方式得出结果,递推算法的复杂度为 $O(N_{max}Y_{max})$

五　用户使用说明

用户可以使用 IDE 或者手动编译源代码 Joseph.cpp，获得可执行文件。
笔者使用的 gcc 版本为 gcc version 8.1.0 ($x86_64-posix-seh-rev0, BuiltbyMinGW-W64project$)

六　测试结果

测试数据保存在 test.in 中，而测试的正确结果保存在 test.out 中。这里的 test.out 经手工计算获得，用于测试程序的正确性。代码中设定了测试相关的宏，在定义 TEST 的情况下，编译的程序会直接从 test.in 中读入数据
接下来对所有测试结果进行说明:

1. $(1, 1, 1)$ 只有一个人

2. (2, 1, 2) 2(x)

3. (3, 1, 2) 2(x) -> 3(x)

4. (1, 2, 1) 只有一个人

5. (2, 2, 1) 1->2(x)

6. (3, 2, 2) 2->3(x)->1->2(x)

7. (1, 3, 1) 只有一个人

8. (2, 3, 2) 2->1->2(x)

9. (3, 3, 3) 3->1->2(x)->3->1->3(x)

再使用 fc output.out test.out 比较程序输出的结果。