数据结构 ： 加里森的任务 实验报告

毛子恒 李臻 张梓靖

2020 年 10 月 11 日

小组成员

班级：2019211309	姓名：毛子恒	学号：2019211397	分工：代码 文档
班级：2019211310	姓名：李臻	学号：2019211458	分工：测试 文档
班级：2019211308	姓名：张梓靖	学号：2019211379	分工：可视化 文档

目录

1 需求分析

1.1 题目描述

在由序号为 1 至 n 的 n 个元素依次排列并且首尾相接而组成的环中，规定初始时从序号 1 开始依次经过 $2, 3, ...$ 元素走到第 n 个元素的方向为正方向。

初始时以第 x 个元素为起点 st，重复以下过程 $n-1$ 次：以 st 为第 1 个元素，沿正方向找到第 y 个元素 del，从环中删除 del 元素，再将原 del 的下一个元素作为新的 st。

求经过 $n-1$ 次操作之后，环中仅剩的一个元素的序号是否是 1。

1.2 输入描述

程序从标准输入中读入数据。输入一行三个整数，用空格分隔，分别表示 n, x, y。

其中各个值的范围需要满足 $1 < n \leq 10^4 \quad 0 < x \leq n \quad 0 < y \leq 5 \times 10^4$。

1.3 输出描述

程序向标准输出中输出结果。

输出分为三种情况：

1. 输入合法，程序正常运行结束。此时输出两行，第一行一个字符串"Yes" 或者"No"（不带引号），分别表示最后一个元素是/不是 1，第二行一个数字，表示最后一个元素的序号。
2. 输入不合法。此时输出一行一个字符串"Please check your input."（不带引号）。
3. 程序发生运行时错误，比如内存分配失败。此时程序没有输出。

1.4 样例输入输出

1.4.1 样例输入输出 1

【输入】

```
10 1 3
```

【输出】

```
No
4
```

1.4.2 样例输入输出 2

【输入】

```
10 3 7
```

【输出】

```
Yes
1
```

1.4.3 样例输入输出 3

【输入】

```
100 87 305
```

【输出】

```
No
50
```

1.4.4 样例输入输出 4

【输入】

```
1000 725 801
```

【输出】

```
No
798
```

1.4.5 样例输入输出 5

【输入】

```
1 1 3
```

【输出】

```
Please check your input.
```

1.4.6 样例输入输出 6

【输入】

```
5 6 3
```

【输出】

```
Please check your input.
```

1.5 程序功能

程序通过给定的 n, x, y 计算出最后环中仅剩的元素序号，并且与 1 比较。

2 概要设计

2.1 问题解决的思路

使用单循环链表维护此约瑟夫环，首先在链表中依次插入 n 个结点表示 n 名队员，以 now 指针模拟计数过程。从头结点找到第 x 个结点，此后执行以下操作 $n-1$ 次：找到当前结点之后的第 $y-1$ 个结点，删除这个结点。此题中单循环链表实现了初始化、判空、在指定位置增加节点、删除指定位置的节点、释放空间这五种操作。

2.2 链表的定义

```
// 数据对象
typedef struct node
{
   int item;
   struct Node * next;
} Node;

typedef Node * List;

// 基本操作
/*
 * 操作：初始化链表
 * 后件：plist指向一个循环链表的头结点
 */
void initList(List * plist);

/*
 * 操作：判断链表是否为空
 * 前件：list是循环链表的头结点
 * 后件：如果该链表为空，返回true，否则返回false
 */
bool isEmpty(const List list);

/*
 * 操作：向链表的某个节点后插入一个节点
 * 前件：pnode是链表中的某一个节点
 * 后件：如果成功，pnode之后添加一个新节点，item属性为传入的第二个参数
 */
void addNode(List pnode, int item);

/*
 * 操作：删除链表中指定的节点
 * 前件：pnode是需要删除的节点的前驱且不是头结点
 * 后件：删除链表中的pnode节点的后继
 */
void delNode(List pnode);

/*
 * 操作：找到链表中某一节点的后继
 * 前件：pnode指向链表中的某一个节点
 * 后件：函数返回pnode的后继，并且跳过头结点
 */
List nextNode(const List pnode);

/*
 * 操作：释放链表空间
 * 前件：plist指向需要释放空间的链表的头结点
```

```
48  * 后件：释放plist指向链表的空间，plist重置为空指针
49  */
50 void destroyList(List * plist);
```

2.3 主程序的流程

1. 输入
2. 初始化链表
3. 在链表中依次插入 n 个结点
4. 找到第 x 个结点
5. 循环 $n-1$ 次：找到当前节点之后的第 $y-1$ 个结点，删除这个结点
6. 输出
7. 释放空间

2.4 各程序模块之间的层次关系

函数调用关系图如图 1。

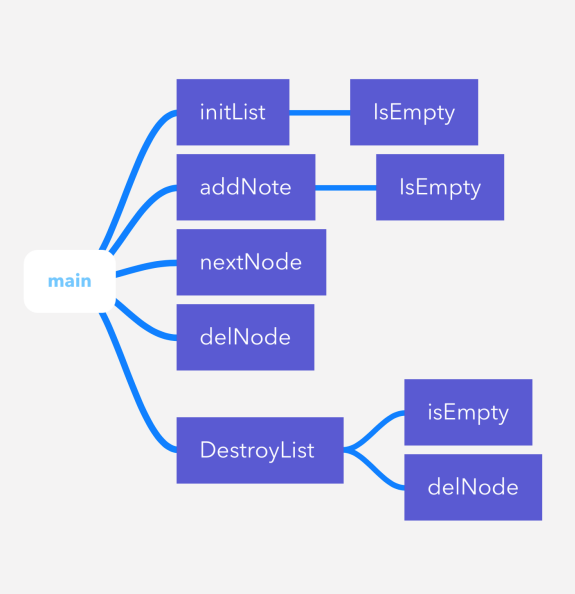

图 1：函数的调用关系

3 详细设计

3.1 链表的实现

链表设计种基本操作的伪代码算法如下：

```
void initList(List * plist) // 初始化链表
{
   给*plist分配内存
   if (*plist内存分配失败)
      异常退出
   (*plist)->item <- 0 // 创建空的头结点
   (*plist)->next <- *plist
}

bool isEmpty(const List list) // 判断链表是否为空
{
   if (List的后继为自身) 返回1
   else 返回0
}

void addNode(List pnode, int item) // 向链表的某个节点后插入一个节点
{
   创建newNode结点，分配内存
   if (newNode内存分配失败)
      异常退出
   newNode->item <- item
   newNode->next <- pnode->next // 将newNode插入链表内
   pnode->next <- newNode
}

List nextNode(const List pnode) // 找到链表中某一节点的后继
{
   定义nItem为pnode的后继
   if (nItem是头节点)
      nItem指向它的后继
   返回 nItem
}

void delNode(List pnode) // 删除链表中指定的节点
{
   定义delNode为pnode的后继
   if (delNode是头节点)
      pnode <- delNode, delNode <- delNode->next // pnode和delNode都指向他们的后继
   pnode->next <- delNode->next // 从链表中移除delNode结点
   释放delNode
}

void destroyList(List * plist) // 释放链表空间
{
   while (*plist不为空)
      删除*plist的后继
   释放*plist
   *plist <- NULL
```

```
}
```

3.2 函数的调用关系图

如 2.4 所示。

4 调试分析报告

4.1 调试过程中遇到的问题和思考

初步实现完成后代码即通过样例测试。之后程序多次对边界条件、不合法输入和异常情况进行优化。

4.2 设计实现的回顾讨论

由于链表的删除操作实现是删除给定结点的后继，所以 now 指针始终指向当前正在计数元素的前驱。

由于单循环链表中存在一个特殊的头结点，所以另实现一个函数，返回某个结点的后继（跳过头结点）。

删除操作的细节：由于 now 指向正在计数结点的前驱，删除某个结点之后 now 仍然指向原来被删节点的前驱，之后执行 $y-1$ 次寻找后继操作，now 便指向下一个待删除结点的前驱。

由于主函数对函数的调用足够严密，所以链表的实现没有考虑不符合前件的情况。

由于链表元素均为 int 类型，所以链表的实现中没有对元素类型进行抽象，并且多次使用赋值运算符更改元素值。

4.3 算法复杂度分析

initList，isEmpty，addNode，nextNode，delNode 函数的时间复杂度均为 $O(1)$。

destroyList 函数的时间复杂度为 $O(n)$，但是由于主程序调用该函数时链表中一定只有 2 个结点，故时间复杂度为 $O(1)$。

主程序复杂度为 $O(n^2)$，整体时间复杂度为 $O(n^2)$。

4.4 改进设想的经验和体会

4.4.1 改进 1

在主程序的这一部分：

```
for (int i = 1; i <= n; ++i) // 逐个添加元素
{
   addNode(now, i);
   now = nextNode(now);
}
now = list;
for (int i = 1; i < x; ++i) // 找到第x个元素的前驱
   now = nextNode(now);
```

可以另用一个指针变量在向链表逐个添加元素的同时记录第 $x-1$ 个元素的位置，以省去第二个循环。优化后的实现如下：

```
List temp = NULL;
for (int i = 1; i <= n; ++i)
{
   addNode(now, i);
   now = nextNode(now);
   if (i == x - 1) temp = now;
}
now = temp;
```

4.4.2 改进 2

在主程序的这一部分：

```
for (int i = 1; i < n; ++i)
{
   for (int j = 1; j < y; ++j)
      now = nextNode(now);
   delNode(list, now);
}
```

对于有 $n-i+1$ 个元素的环，找到当前元素之后的第 $y-1$ 个元素和找到当前元素之后的第 $(y-1) \bmod (n-i+1)$ 个元素并无区别。优化后的实现如下：

```
for (int i = 1; i < n; ++i)
{
   for (int j = 1; j <= (y - 1) % (n - i + 1); ++j)
      now = nextNode(now);
   delNode(list, now);
}
```

当 y 比 n 大的时候对时间复杂度有很可观的优化。

4.4.3 改进 3

约瑟夫问题有时间复杂度为 $O(n)$ 的递归解法，现论述如下：

假设对于有 n 个元素的环，序号为 0 至 $n-1$，以序号为 0 的元素为起点，删去第 y 个元素，即序号为 $y-1$ 的元素，之后进行下一次删除。

而根据题意，下一次删除应该从被删除元素的下一个元素开始计数，所以可以将整体序号减去 y 再对 n 取余数，得到新的序号，范围是 0 至 $n-2$，然后再以 0 为起点重复删除操作。

最后一次删除和序号变化之后，剩余一个序号为 0 的元素。可以根据上述操作的逆过程推出这个元素在初始状态下的序号。

由于题目规定了起点的序号为 x，所以还要再进行一次类似的整体序号位移，另外题目中序号为 1 至 n，给求得答案 $+1$ 得到题目要求的答案。

该解法的实现如下：

```
#include <stdio.h>

int main()
```

```
{
   int n, x, y;
   scanf("%d%d%d", &n, &x, &y);
   int ans = 0;
   for (int i = 2; i <= n; ++i)
      ans = (ans + y) % i;
   printf("%d\n", (ans + x - 1) % n + 1);
   return 0;
}
```

这个程序 (main1.c) 被用于测试环节，用来验证原解法的正确性。

5 用户使用说明

使用 gcc 编译生成可执行文件。

```
gcc -o main -std=c11 main.c list.c
```

执行可执行文件：

```
./main
```

在 Windows cmd 下：

```
main
```

之后通过标准输入输入数据，输入格式参考 1.2 节的输入描述，结果通过标准输出返回。如果输入合法并且程序正常运行结束，主函数返回值为 0。

6 测试结果

测试环节分为四个步骤。

6.1 测试第一部分

对 1.4 节给出的样例进行测试。

6.2 测试第二部分

在 delNode 函数中添加输出语句，输出每一轮计数时的第 y 个元素，输入小样例，将输出与手动模拟结果比对。

【输入】

```
10 1 3
```

【输出】

```
3
6
9
2
7
```

```
1
8
5
10
No
4
4
```

此样例中环中删除的元素依次为 3,6,9,2,7,1,8,5,10,4，与模拟结果相符。

6.3 测试第三部分

测试非法输入和边界条件。

【输入】

```
5 -1 2
```

【输出】

```
Please check your input.
```

【输入】

```
5 2 -1
```

【输出】

```
Please check your input.
```

【输入】

```
1000000 256 512
```

【输出】

```
Please check your input.
```

【输入】

```
10000 5723 4627
```

【输出】

```
No
5180
```

6.4 测试第四部分

将原解法与 4.4.3 中的改进解法比对。

测试在 macOS Catalina 10.15.6 下进行。

在 $n <= 10$，$n <= 1000$，$n <= 10000$ 的范围下分别随机生成 1000 组测试数据，分别传入 main 和 main1，并且比对两程序的输出。

3000 组数据中两程序的输出均相同。

数据生成程序 (data.cpp) 如下：

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

const int LIMIT = 98;

int main()
{
   srand(time(0));
   int n, x, y;
   n = rand() % LIMIT + 2;
   x = rand() % n + 1;
   y = rand() % (n * 3) + 2;
   printf("%d %d %d\n", n, x, y);
   return 0;
}
```

比对脚本 (chk.cpp) 如下：

```cpp
#include <bits/stdc++.h>
using namespace std;
int main()
{
   int T = 1000;
   while (T--)
   {
      system("./data >in.in");
      system("./main <in.in >out.out");
      system("./main1 <in.in >out1.out");
      if (system("diff out.out out1.out"))
         break;
      puts("Correct");
   }
   return 0;
}
```

7 可视化

7.1 过程可视化

使用 JavaScript 实现过程可视化。

7.1.1 实现细节

程序运行后，读取用户输入的 n, x, y 参数，进行参数类型转换。通过 document.getElementById 获取画布。创建双链表。以上三者存放在然后存放到 simulator 的 store 成员中。

通过 2π 均分获取角度差，通过画布的长宽确定中心点。一中心点为圆心，通过 center.x + radius * Math.cos(deltaDegree * i); 和 center.y + radius * Math.sin(deltaDegree * i); 分别计算各个圆的位置。使用 20 作为半径绘制圆。分别在个个

圆上绘制文本。通过定时器模拟人员活动，每 100 毫秒执行。根据存活状况更新各个圆的颜色，通过 currentNode 和 flag 局部变量存放当前节点和跳过人数。当剩余一个人时停止计时。

使用 JavaScript 实现双向链表模拟环中元素的关系。

7.1.2 用户使用说明

使用现代浏览器打开 Animation/index.html，根据提示输入 n, x, y，要求输入合法且 $1 < n < 40$，之后页面展示一段动画，内容为约瑟夫问题的过程。

7.1.3 示例

见图 2。

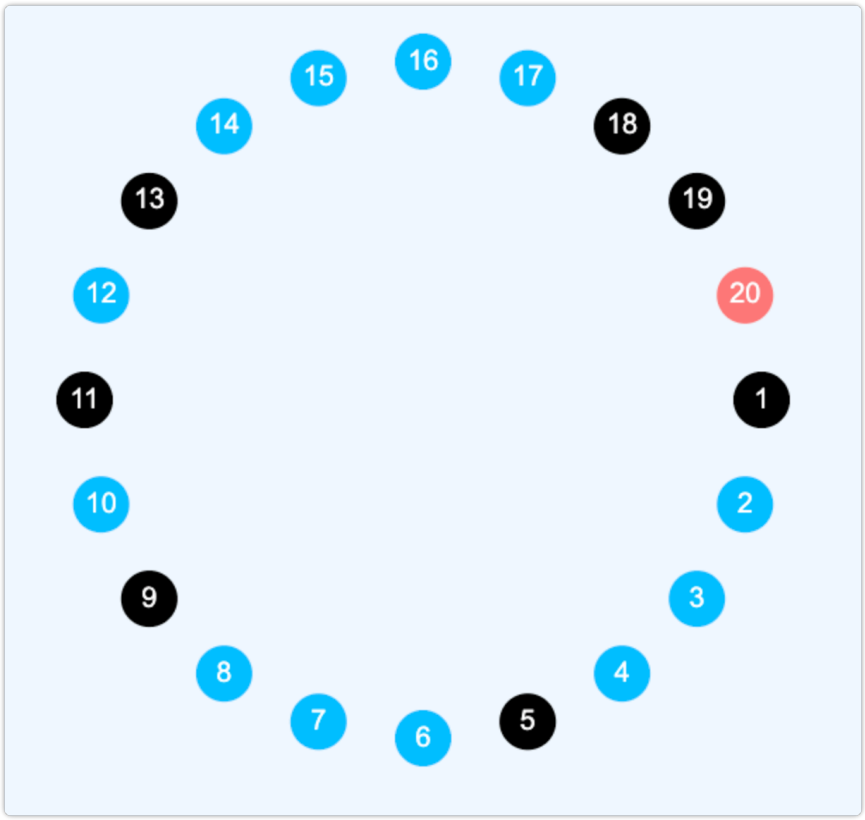

图 2: 过程可视化示例

7.2 结果可视化

使用 JavaScript 实现结果可视化。

7.2.1 实现细节

程序运行后，读取用户输入的 n 参数，之后利用使用 JavaScript 实现的 4.4.3 节改进算法计算 $1 \leq x \leq n, 1 \leq y \leq n$ 中的所有解，导出为点集。

使用 Highcharts 库绘制 3D 散点图，展示所求出的点集。

7.2.2 用户使用说明

使用现代浏览器打开 Scatter/index.html，根据提示输入 n，要求 $1 < n <= 25$，之后页面展示一张散点图。拖动该散点图可以旋转，将鼠标指针放到某个点上可以看到该点的值。

7.2.3 示例

见图 3。

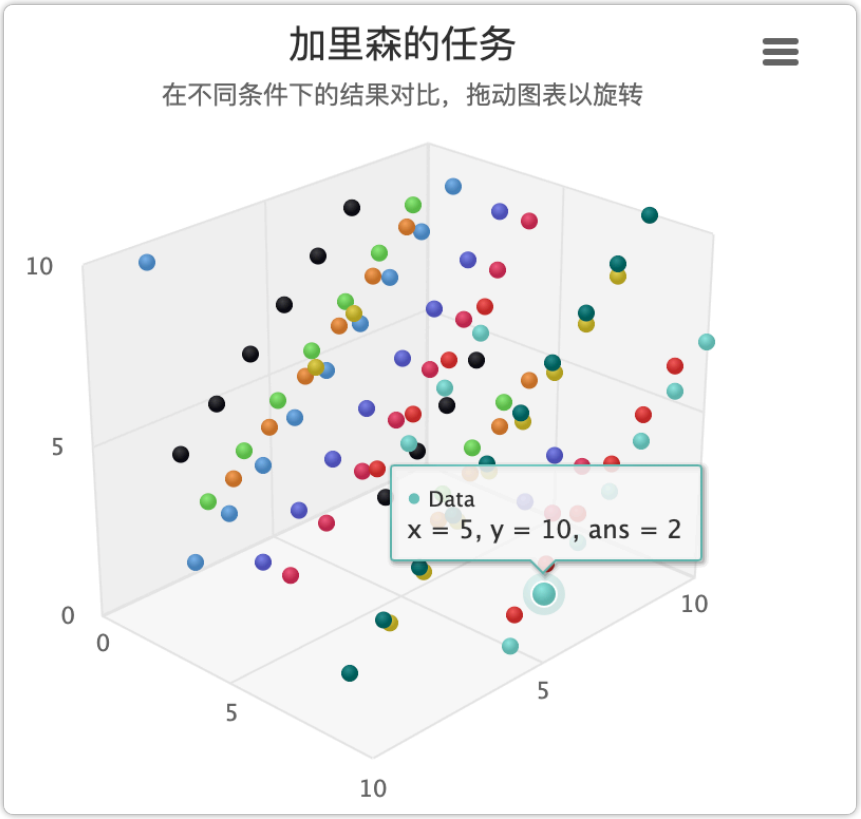

图 3: 结果可视化示例

7.2.4 结论

可以明显地看出，当 n 一定，y 一定时，由于 x 增加 1 时 ans 也对应地增加 1，所以这些点分布在一条直线上。