约瑟夫问题 (Josephus Problem) 的求解与算法复杂度和数据结构

马欢飞*

2020 年 3 月

约瑟夫问题 据说著名犹太历史学家 Josephus 有过以下的故事: 在罗马人占领乔塔帕特后，39 个犹太人与 Josephus 及他的朋友躲到一个洞中，39 个犹太人决定宁愿死也不要被敌人抓到，于是决定了一个自杀方式，41 个人排成一个圆圈，由第 1 个人开始报数，每报数到第 3 人该人就必须自杀，然后再由下一个重新报数，这个过程沿着圆圈一直进行，直到最终只剩下一个人留下，这个人就可以继续活着。问题是，一开始要站在什么地方才能避免被处决？[1] 这个问题可以被归纳为如果有 n 个人，依次排成一圈报数，每报数报到 k 的人出列，下一个人重新开始报数，问最后剩下的一人是第几个人？

算法一 要解决这个问题，有很多种办法，最简单的办法是编写程序模拟整个过程。对于初学者来说，用 C 语言来编写模拟该过程的算法，有两个问题需要解决：第一是如何实现“排成一圈”，因为一般 n 个人的数据会采用数组，但数组是排成一列的。第二个问题是如何实现“出列”。

对于第一个问题，可以使用循环数组，即认为数组最后一个元素的下一个元素是第一个元素，这样首尾相接就形成了环形数组，而要实现这一点，注意到 n 个元素的数组的下标为 $0, 1, 2, \ldots, n-1$，所以如果用 i 表示当前下标的话，那么下一个元素的下标用 $(i+1)\%n$ 来表示的话，下标为 $n-1$ 的元素，其下一个元素自动变为了下标为 0，而其他下标则不受影响。

对于第二个问题，一个直观的方式是用非零来标识尚未出列的人，用零来标识已经出列的人。这样，我们就可以得到第一个算法，代码如下

```
============================================================
void number1(int n,int k)            //算法一
{
    int data[n];
    int count;
    int i;
```

*hfma@suda.edu.cn

[1]以上故事来自于 wikipedia 的介绍，未必可信。

```
    for(i=0;i<n;i++)
        data[i]=1;                  //初始化
    count=n;                        //count记录剩余人数
    int j=0;                        //j记录报数
    i=n-1;                          //i记录当前位置
    while(count>1)                  //模拟整个过程
    {
        i=(i+1)%n;                  //下一个的位置
        if(data[i])                 //若在队列中，报数加一
            j++;
        if(j==k)                    //若报数为k
        {   j=0;                    //报数归零
            data[i]=0;              //出列
            printf("No. %d is out\n", i+1); //输出出列者的逻辑序号
            count--;                //剩余人数减一
        }
    }
     i=0;
     while(data[i]<1)               //遍历数组寻找最后剩下者
        i++;
     printf("the last survivor is: %d",i+1);
}
=========================================================
```

对于该算法，我们可以看到，每扫描整个数组一遍（n 个元素），剩余人数在上一遍的基础上减少 $(1/k)$，所以假设总共扫描了 l 遍，则有 $n*(1-1/k)^l=1$, 所以 $l\sim O(\frac{\log n}{\log k-\log(k-1)})$，总的复杂度为 $O(\frac{n\log n}{\log k-\log(k-1)})$, 若 k 比较小看成常数，则复杂度为 $O(n\log n)$。空间复杂度则由于借助了一个长度为 n 的数组，所以为 $O(n)$。

算法二 仔细阅读上述算法，我们发现在不断扫描的过程中，整个数组里 0 越来越多，但却每个都要被不断扫描到，事实上这是无用功，如果能避免这种对出列者的扫描，则复杂度能降低。这就产生了第二种算法，在这种算法里，我们假设所有人排成一队，报数到某人时，如果不出列的话，则该人跑到队尾去继续排在队列里，如果出列的话，则出列后不再排到队列中 (从而模拟一个圈)。这样报数始终在剩余者中进行，并保持环形队列。可以得到以下第二个算法。

```
=========================================================
void number2(int n,int k)              //算法二
{
    int data[n];
int start=-1,rear=n-1;            //队头和队尾标识
```

```
int i,e;
for (i=0;i<n;i++)      //构建初始队列
{
data[i]=i+1;
}
while ((start+1)%n!=rear)  //队列不空循环
{
    for(i=1;i<k;i++)             //报数1到k-1
    {   start=(start+1)%n;
        e=data[start];
        rear=(rear+1)%n;
         data[rear]=e;          //将刚出队的元素进队
    }
       start=(start+1)%n;
        printf("No. %d is out\n",data[start]); //报数k出列
    }
    printf("The last suvivor is: %d",data[rear]);
}
============================================================
```

很明显，该算法复杂度为 $O(kn)$，如果 k 比较小看成常数时，则为 $O(n)$。事实上，算法二在算法上和算法一没有本质的区别，区别在于使用了队列这样一种数据结构，用入队和出队来模拟过程，减少了无用功。空间复杂度仍然是借助了一个长度为 n 的数组，所以为 $O(n)$。

算法三 事实上，同样的线性结构，如果使用链式存储而非顺序存储，同样可以减少无用功：即我们使用不带头节点的循环单链表来存储所有人员，然后依次扫描链表节点并报数，当报数为 k 时，将该节点从链表上删除，直至最后只剩一个节点。从而我们可以得到以下算法：

```
============================================================
void number2b(lnode *L,int k)
{
    lnode *p=L,*q=L->next;              //q指向当前报数节点，p为q前一个节点
    int c=2;                            //从第二个节点开始循环
    while(p->next!=p){                  //循环链表中只要多于一个节点，则继续循环
        if(c==k){                       //报数为k时，将当前节点q删除
            p->next=q->next;
            printf("No. %d is out\n",q->data);
            free(q);
            q=p->next;
            c=1;
        }
```

```
        else{                           //报数非k时，p和q均指向下一个节点，报数加一
            p=q;
            q=q->next;
            c++;
        }
    }
    printf("the last survivor is: %d",p->data);
}
============================================================
```

显然，该算法每扫描 k 次删除一个节点，所以总共扫描 $k(n-1)$ 次，即复杂度为 $O(kn)$，与用顺序存储方式实现的环形队列算法相当。空间复杂度方面这里开辟了一个长度为 n 的链表，所以为 $O(n)$。但是这里需要注意的是，环形队列里涉及到大量元素的拷贝和移动，虽然时间复杂度是相当的，但是总体开销而言，链表的开销会更小。

算法四 进一步观察，如果约瑟夫问题只关心最后留下的一个人的逻辑序号，而不关心中间出列的顺序的话，那么可以进一步首先从数学上进行归纳和递推，得到递推公式

$$g(n,k) = (g(n-1,k)+k) \mod n$$

其中 $g(n,k)$ 表示 n 个人,报数到 k 出列的话最后幸存者的编号,n 个人的编号为 $0,1,2,\ldots,n-1$。递推的起始条件显然为 $g(1,k)=0$。该公式的推导过程此处略去。这样我们可以得到算法三，

```
============================================================
int number3(int n,int k)          //算法三
{
    int i,f=0;
    for (i=1;i<=n;i++)
        f=(f+k)%i;
    return f+1;
}
============================================================
```

显然，该算法时间复杂度为 $O(n)$, 且与 k 的大小无关。空间复杂度则显然为 $O(1)$。