数据结构实验报告

姓名： 杨家玺 学号：U201717007 班级：软工 1703 班

2018. 12. 03

实验三 约瑟夫问题和多项式相乘问题的求解

一、实验描述

1.约瑟夫问题：用游标方式的循环链表实现 Josephus(n, m)问题的求解。
2. 多项式乘法：用链表表示多项式，分别在对指数排序和不排序的情况下，写出求两个给定多项式的乘法的函数。其计算复杂度分别是多少？

二、实验环境

1. 开发环境： OS X
2. IDE： CLion
3. 编译器： Clang 9.1.0 Apple LLVM
4. C 标准： C11

三、问题分析

1. 约瑟夫问题

这是一个历史悠久的问题，解决这个问题的方法也层出不穷，本次实验我采用了 1. 基于游标的循环链表方法与 2. 基于数学推导的递归/迭代方法。

在代码中，规定了人的编号从 0 开始，报数从 1 开始，这样做使得三个算法的结果输出形式保持一致。

a. 对于基于游标的链表，首要解决的问题是 malloc 与 free 函数的数组模拟，此部分利用书中代码实现。
b. 对于基于数学推导的递归/迭代方法，推导如下：
   a) 对于一给定数列[0,1,2,3,4...n-1]与每轮划去的元素 m-1（从 1 开始报数，所以对应从 0 开始的 m-1）
   b) 假设 n>m，第一轮后：[0,1...m-2,m,m+1,...n-1]
   c) 将 m 置 0，也就是[-m,-m+1,...-2,0,1,...n-1-m]
   d) 由于出现负数，根据环状报数规则重新映射序列
   e) -> [n-m,n-m+1,...n-2,0,1,...n-1-m] (list+n)%n
   f) 对于这 n-1 个数组成的新序列使用同样的方法，构成递归
   g) 将上述过程转换为自底向上的搜索，可得到转移方程与边界条件：
   h) f(k, m)=(f(k-1, m)+m)%k, 1<=k<=n
   i) f(1, m)=0

2. 多项式乘法

利用链表实现多项式乘法，可以避免非稀疏情况下的内存浪费问题。

a. 对于没有进行指数排序的多项式：

a) 对于多项式 1 中的每个项
   i. 使用多项式 2 中的每个项与之相乘
   ii. 对于两元素相乘结果，在已有的结果多项式中寻找该指数是否已存在
   iii. 若存在，把当前得到的系数加上去，否则，新建节点

b. 对于已进行指数降序排列的两个多项式

a) 得到结果的最高次项系数，这一步可以通过加和两个多项式的首项次数得到，因为他们都是按照指数降序排列的
b) 将多项式 2 反向
c) 对于每一个可能出现的多项式系数 i：
   i. 找到多项式 1 中第一个指数不大于 i 的项 p1
   ii. 找到多项式 2 中第一个满足 p1.exp+p2.exp>i 的 p2
   iii. 进入循环，直到 p1 与 p2 有一个为空指针
      1. 如果两项次数和恰好等于 i，直接纳入结果，两个指针同时向后移动 1
      2. 如果两次数和小于 i，则将 p2 指针后移，因为多项式 2 在反向后为升序，只有更高的次数才能满足次数和为 i
      3. 否则，将 p1 后移，原理同上

四、算法实现

1. 约瑟夫问题

```
void Josephus_1(List list, int size, int death)      // 链表模拟
{
    printf("---------------------\n");
    printf("Algorithm 1:\n");
    Position tmp;
    int counter = 0;
    int left = size;
    tmp = First(list);
    while (left > 1)
    {
        if (counter % death == death - 1)
        {
            Position del = tmp;
            tmp = RollAdv(tmp, list);                // RollAdv(Position, List) 将链表循环
            Delete(Retrieve(del), list);
            --left;
            counter = 0;
        }
        else
        {
            counter++;
            tmp = RollAdv(tmp, list);
        }
    }
    printf("Left : %d\n", Retrieve(tmp));
}

int j2_rec(int n, int m)
{
    if (n == 1)
    {
        return 0;
    }
    else
    {
        return (j2_rec(n - 1, m) + m) % n;
    }
}

void Josephus_2(int size, int death)           // 递归实现
{
    printf("---------------------\n");
    printf("Algorithm 2:\n");
    printf("Left : %d\n", j2_rec(size, death) + 1);
```

```
}

void Josephus_3(int size, int death)     // 自底向上的记忆化搜索
{
    printf("---------------------\n");
    printf("Algorithm 3:\n");
    int f[size + 1];
    f[0] = f[1] = 0;
    for (int i = 2; i <= size; i++)
    {
        f[i] = (f[i - 1] + death) % i;
    }
    printf("Left : %d\n", f[size] + 1);
}
```

2. 多项式乘法

```
void MultPolynomial(Polynomial Poly1, Polynomial Poly2, Polynomial Res)
{
    Item p1 = FirstElement(Poly1);       // Pointer to Poly1
    Item p2 = NULL;                      // Pointer to Poly2
    Item tmp = NULL;                     // Tmp pointer, for Insert() to Res
    int Coef = 0;                        // Multiply of two items' coef
    int Exp;                             // Sum of two items' exp
    while (p1)                           // for each Item in Poly1
    {
        p2 = FirstElement(Poly2);        // Reset pointer p2, which point to Poly2
        while (p2)                       // for each item in Poly2, do p1 * item
        {
            Coef = p1->coef * p2->coef;
            Exp = p1->exp + p2->exp;
            p2 = Advance(p2);
            tmp = FindByExp(Exp, Res);   // Does this Exp already exist?
            if (tmp)                     // -> yes, add to its coef
            {
                tmp->coef += Coef;
            }
            else                         // -> no, create one
            {
                InsertAfter(Exp, Coef, Res, Res);
            }
        }
        p1 = Advance(p1);
    }
}

void MultPolynomial_Sorted(const Polynomial Poly1, const Polynomial Poly2, Polynomial Res)
{
    int MaxExp = FirstElement(Poly1)->exp + FirstElement(Poly2)->exp;
    InverseList(Poly2);
    Item p1 = NULL;
    Item p2 = NULL;
    Item pres = Header(Res);
    int CoefSum = 0;
    for (int i = MaxExp; i >= 0; i--)
    {
        p1 = FirstElement(Poly1);
        p2 = FirstElement(Poly2);
        while (p1 && p1->exp > i)
            p1 = Advance(p1);
        while (p1 && p2 && p1->exp + p2->exp < i)
            p2 = Advance(p2);
        while (p1 && p2)
        {
            if (p1->exp + p2->exp == i)
            {
                CoefSum += p1->coef * p2->coef;
                p1 = Advance(p1);
                p2 = Advance(p2);
            }
            else
            {
                if (p1->exp + p2->exp < i)
                    p2 = Advance(p2);
                else
                    p1 = Advance(p1);
            }
        }
        if (CoefSum != 0)
        {
            pres = InsertAfter(i, CoefSum, Res, pres);
            CoefSum = 0;
        }
    }
    InverseList(Poly2);
}
```

五、实验结果与分析

1．约瑟夫问题

```
(1) N=5, M=3
---------------------
Algorithm 1:
Left : 4
---------------------
Algorithm 2:
Left : 4
---------------------
Algorithm 3:
Left : 4

(2) N=41, M=3
---------------------
Algorithm 1:
Left : 31
---------------------
Algorithm 2:
Left : 31
---------------------
Algorithm 3:
Left : 31
```

2．多项式相乘

```
1. Mutiply of Exp-Sorted Polynomial:
Multiply of
     (5) * x^[2] + (3) * x^[1] + (6) * x^[0]
and
     (3) * x^[3] + (5) * x^[2] + (7) * x^[0]
is:
     (15) * x^[5] + (34) * x^[4] + (33) * x^[3] + (65) * x^[2] + (21) * x^[1] + (42) * x^[0]
---------------------------------------------
2. Mutiply of Non-Sorted Polynomial:
Multiply of
     (6) * x^[0] + (5) * x^[2] + (3) * x^[1]
and
     (5) * x^[2] + (7) * x^[0] + (3) * x^[3]
is:
     (21) * x^[1] + (15) * x^[5] + (34) * x^[4] + (33) * x^[3] + (42) * x^[0] + (65) * x^[2]
```

可见，对于输入(5*x^2+3*x+6)(3*x^3+5*x^2+7)，两种算法均给出了15*x^5+34*x^4+33*x^3+65*x^2+21*x+42 的结果，因此知算法正确

算法分析：

设两序列分别有 M 和 N 个元素，其最高次为 A 与 B，则

（1） 朴素算法。对于多项式 1 中的每一个元素，需要乘以 N 个元素，每次相乘后需要在已经生成的结果序列中寻找插入位置，保存结果的链表最长为 A+B 项，故最坏的时间复杂度为 O(M*N*(A+B))

（2） 指数排序后的算法。对于每一个可能出现在结果中的次数，最坏情况是两个多项式中没有符合条件的对，故需要循环 M+N 次，所以该算法的最坏时间复杂度为 O((M+N)*(A+B))