

C语言程序设计

Programming in C

第十一讲

问题求解方法

How to Solve

问题求解方法

学习要点

- 程序设计的基本过程
- 如何进行问题分析
- 如何进行方案设计
- 如何选择数据结构
- 如何设计算法
- 从算法到编码实现



武功心法

程序设计的基本过程

问题分析

- 功能描述
- 输入输出分析
- 性能要求
- 错误处理
- 测试数据

方案设计

- 数学建模（问题抽象）
- 算法设计与描述
- 数据结构选择
- 自顶向下
- 时间复杂度分析
- 空间复杂度分析

编码

- 语言选择
- 根据算法流程编码
- 代码检查与优化
- 代码注释

调试

- 调试工具
- 调试方法
- 问题定位
- 标准输入输出重定向
- 调试日志



大规模测试 (alpha, beta,..., preview)



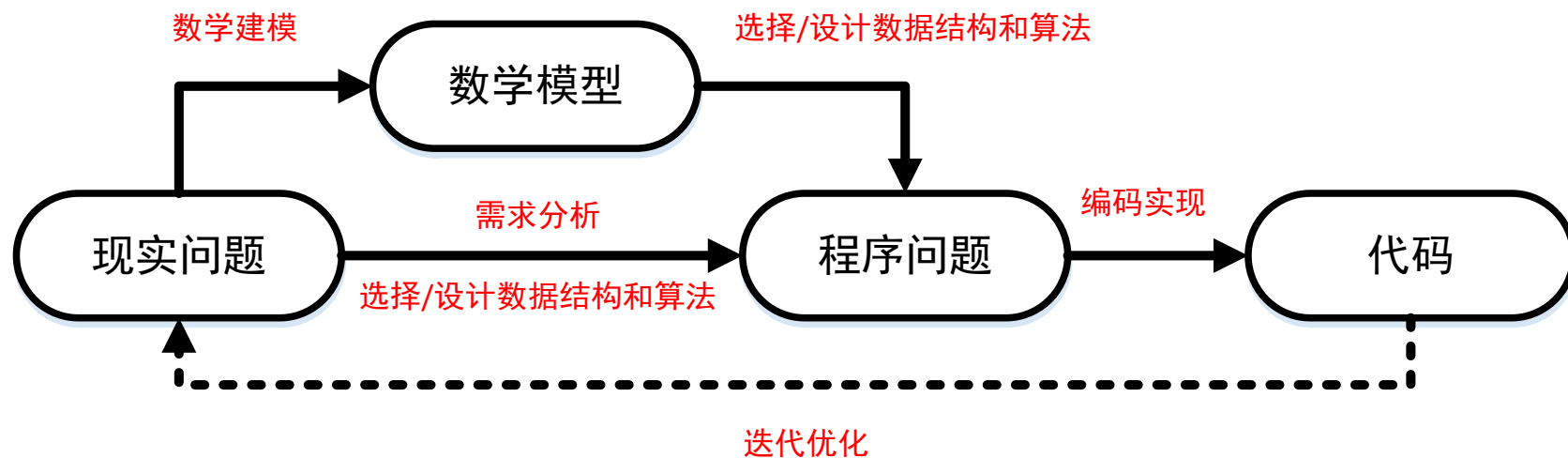
Release! 😊

程序设计的基本过程

- **编程非常复杂！绝不仅仅是写代码的问题！**
- **科学、工程、管理、艺术、伦理**
- **当然，写好代码是一项基本功！**

问题分析

- 程序设计的目的是什么？
 - 发挥计算机的高速计算能力解决现实世界中的问题
- 是不是所有的现实问题都能被程序解决？
 - 只有那些在限定范围内被人类的自然科学证明有解的问题才可以通过程序来解决
 - 程序的特点是执行速度快，判断一个问题在给定范围内是否可解以及如何去解，需要自然科学尤其是数学知识
- 如何将实际问题转化为程序问题，并通过代码实现解决，是新工科学生需要具备的基本能力



方案设计

- **解题思路：**用自然语言对计算过程的框架性描述，说明解题过程所需要的步骤以及各步骤之间的相互关系，或者进行数学建模

功能 已知条件 结果 算法性质 数据或控制结构概要 数学公式

- **算法：**设计代码结构，解题思路到程序之间的桥梁

解题思路 ~~——~~ 算法

- **数据结构：**根据数据行为特点选择合适的数据结构

绝对坐标、相对坐标、随机存储、先进先出、先进后出...

方案设计一定围绕目标问题的主要矛盾展开！

数学模型？数据结构？算法？

在解决主要矛盾之前，次要矛盾一律先不考虑！

**优秀的代码不只追求功能实现，还要考虑多快好省
主要矛盾直接影响最终程序水平**

数学模型的建立

[例7-9]母牛的数量

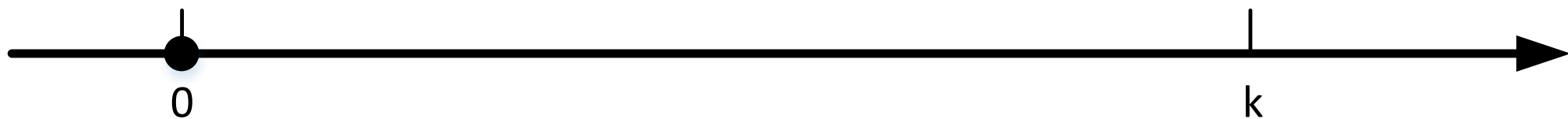
一头 x 年出生的母牛从 $x+m$ 年到 $x+n$ 年间每年生出一头母牛，并在 $x+p$ 年被淘汰。写一个程序，从标准输入上按顺序读入整数 m, n, p, k ($3 < m < n < p < 60$, $0 < k < 60$)，设第0年有一头刚出生的母牛，计算第 k 年时共存有多少头未被淘汰的母牛。

主要矛盾是什么？

这是一个典型的数学建模问题，所以主要矛盾一定是数学模型。

数学模型的建立

1. 首先画出问题的时间轴，并给出数学假设



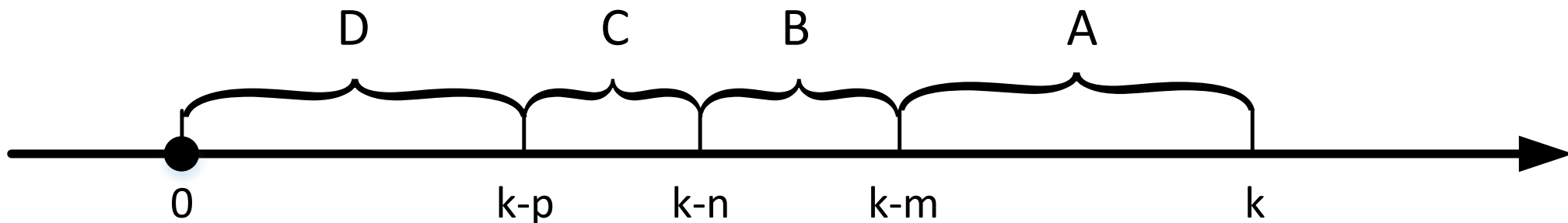
假设：第 x 年的牛总数为 $T(x)$
第 x 年出生的牛数为 $N(x)$
第 x 年死去的牛数为 $D(x)$

那么， $T(k)$ 就是我们的目标问题，显然：

$$T(k) = T(k - 1) + N(k) - D(k)$$

数学模型的建立

2. 求解 $N(k)$ 和 $D(k)$, 用 $k-p, k-n, k-m$ 三个关键时间将 $[0 \sim k]$ 区间划分成四个区间: A、B、C、D



考察在不同时间区间出生的牛在第k年的生牛情况:

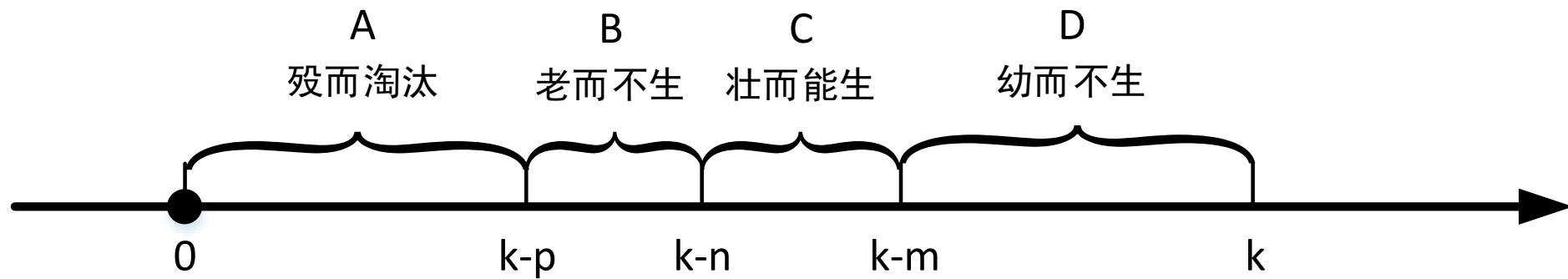
A: 在这个区间出生的牛太小了, 幼而不生;

B: 在这个区间出生的牛刚刚好, 壮而能生;

C: 在这个区间出生的牛太老了, 老而不生;

D: 在这个区间出生的牛寿终了, 殁而淘汰;

数学模型的建立



3. 计算 $N(k)$, 第 k 年的新牛是历史能生的新牛数之和:

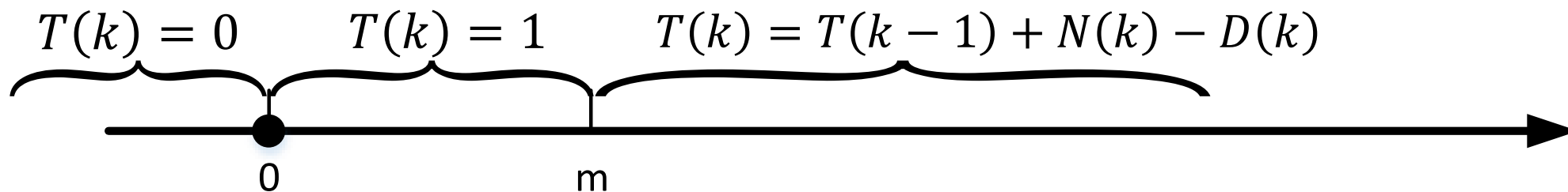
$$N(k) = \sum_{i=m}^n N(k-i)$$

4. 计算 $D(k)$, 第 k 年死去的牛数是第 $k-p$ 年的新牛数:

$$D(k) = N(k-p)$$

数学模型的建立

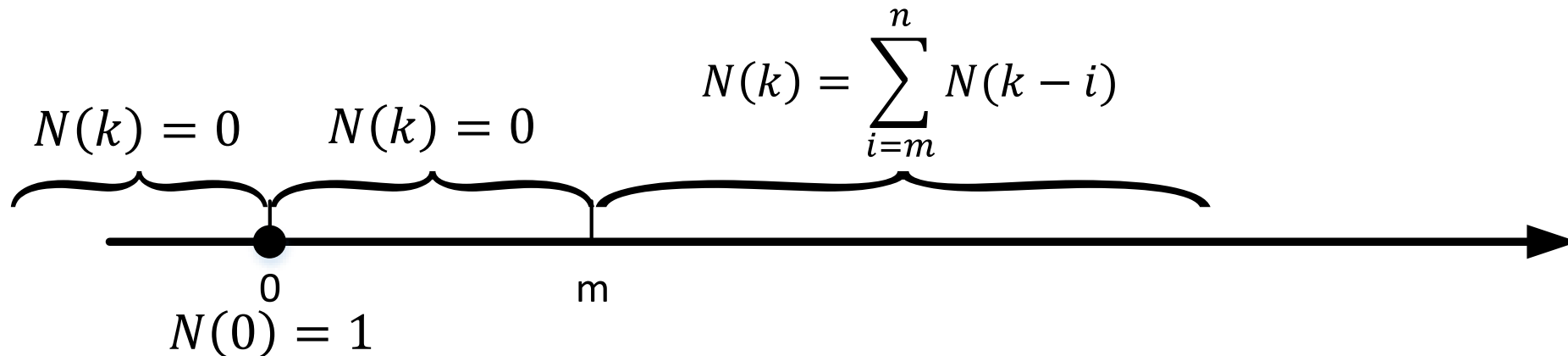
第0年之前没有牛，第 m 年之前1头牛



$$\begin{cases} T(k) = 0, k < 0 \\ T(k) = 1, 0 \leq k < m \\ T(k) = T(k - 1) + N(k) - D(k), k \geq m \end{cases}$$

数学模型的建立

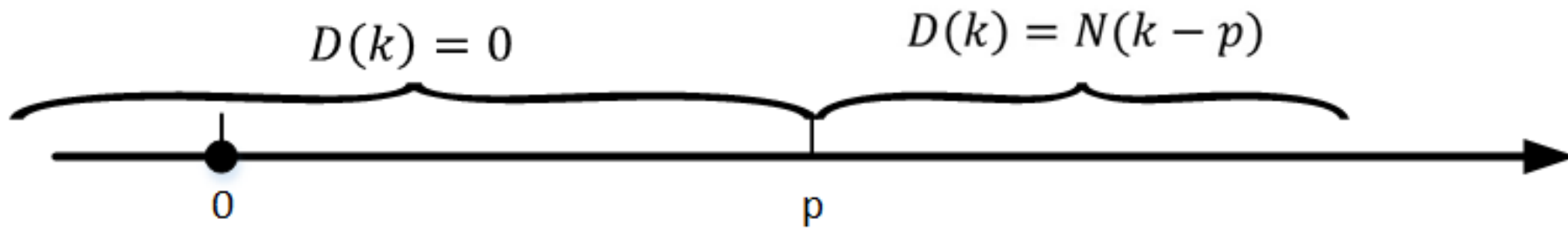
第0年有1头新牛,第 m 年之前(除第0年)没有新牛



$$\begin{cases} N(0) = 1 \\ N(k) = 0, k < m \text{ \& } k \neq 0 \\ N(k) = \sum_{i=m}^n N(k-i), k \geq m \end{cases}$$

数学模型的建立

第 p 年之前没有死牛



$$\begin{cases} D(k) = 0, k < p \\ D(k) = N(k - p), k \geq p \end{cases}$$

数学模型的建立

至此，我们已经获得了所有的数学模型，可以开始设计算法了！

$$\begin{cases} T(k) = 0, k < 0 \\ T(k) = 1, 0 \leq k < m \\ T(k) = T(k-1) + N(k) - D(k), k \geq m \end{cases}$$

$$\begin{cases} N(0) = 1 \\ N(k) = 0, k < m \text{ \& } k \neq 0 \\ N(k) = \sum_{i=m}^n N(k-i), k \geq m \end{cases} \quad \begin{cases} D(k) = 0, k < p \\ D(k) = N(k-p), k \geq p \end{cases}$$

数学模型的建立

首先，声明三个函数，分别对应 $T(k)$, $N(k)$, $D(k)$

```
#include<stdio.h>
int Tcows(int k);
int Ncows(int k);
int Dcows(int k);
int m, n, p, k;
void main()
{
    scanf("%d%d%d%d", &m, &n, &p, &k);
    printf("%d", Tcows(k));
}
```

数学模型的建立

参照 $T(k)$ 公式，实现Tcows函数实体：

$$\begin{cases} T(k) = 0, k < 0 \\ T(k) = 1, 0 \leq k < m \\ T(k) = T(k-1) + N(k) - D(k), k \geq m \end{cases}$$

```
int Tcows(int k)
{
    if (k < 0)
        return 0;
    else if (k < m)
        return 1;
    else
        return Tcows(k-1) + Ncows(k) - Dcows(k);
}
```


数学模型的建立

参照 $N(k)$ 公式，实现Ncows函数实体：

$$\begin{cases} N(0) = 1 \\ N(k) = 0, k < m \text{ \& } k \neq 0 \\ N(k) = \sum_{i=m}^n N(k-i), k \geq m \end{cases}$$

```
int Ncows(int k) {  
    int i, cows;  
    if( k == 0 ) return 1;  
    else if( k < m ) return 0;  
    else {  
        cows = 0;  
        for(i = m; i <= n; i++) cows += Ncows(k - i);  
        return cows;  
    }  
}
```

数学模型的建立

参照 $D(k)$ 公式，实现Dcows函数实体：

$$\begin{cases} D(k) = 0, k < p \\ D(k) = N(k - p), k \geq p \end{cases}$$

```
int Dcows(int k)
{
    return ( k < p ) ? 0 : Ncows(k - p);
}
```

至此，程序完成！运行成功！是不是很简单？

数学模型的建立

真的完成了吗？

上述代码的功能虽然已经实现，但是还有一个严重的问题：

当 k 比较大时，算不动！

这是一个典型的时间复杂度问题，我们需要分析一下。

这个程序用了很多递归来实现隐函数的功能。

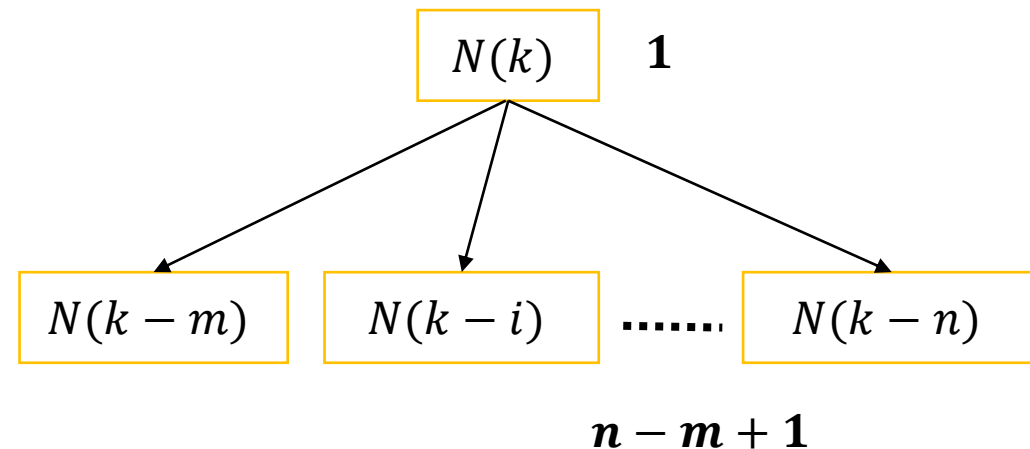
递归的问题是，消耗计算时间和存储空间（详见，第5讲）

所以，递归调用的地方是产生算不动的根源！

数学模型的建立

检查代码后，我们发现递归调用最为频繁的函数是Ncows函数：

```
int Ncows(int k) {  
    int i, cows;  
    if( k == 0 ) return 1;  
    else if( k < m ) return 0;  
    else {  
        cows = 0;  
        for(i = m; i <= n; i++) cows += Ncows(k - i);  
        return cows;  
    }  
}
```

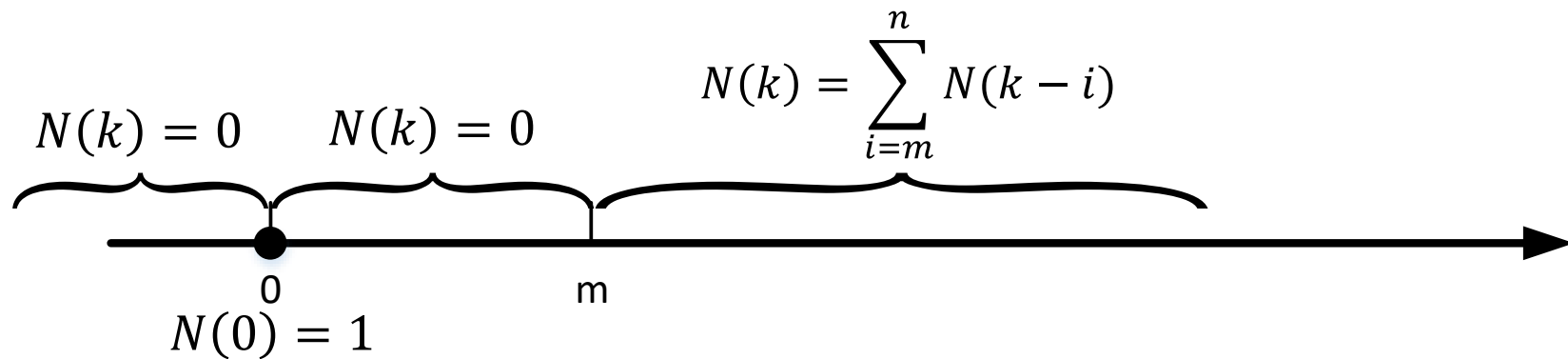


*每次递归调用了 $n-m+1$ 次， $n-m$ 较大时复杂度较高。

数学模型的建立

回到公式，还有一件事可以做！

$$\begin{cases} N(0) = 1 \\ N(k) = 0, k < m \text{ \& } k \neq 0 \\ N(k) = \sum_{i=m}^n N(k-i), k \geq m \end{cases}$$



上下做差运算，可得：

$$\begin{cases} N(k) = \sum_{i=m}^n N(k-i) \\ N(k-1) = \sum_{i=m}^n N(k-1-i) \end{cases}$$

$$N(k) - N(k-1) = N(k-m) - N(k-n-1) \Rightarrow$$

$$\begin{cases} N(0) = 1 \\ N(k) = 0, k < m \text{ \& } k \neq 0 \\ N(k) = N(k-1) + N(k-m) - N(k-n-1), k \geq m \end{cases}$$

数学模型的建立

调整Ncows的代码:

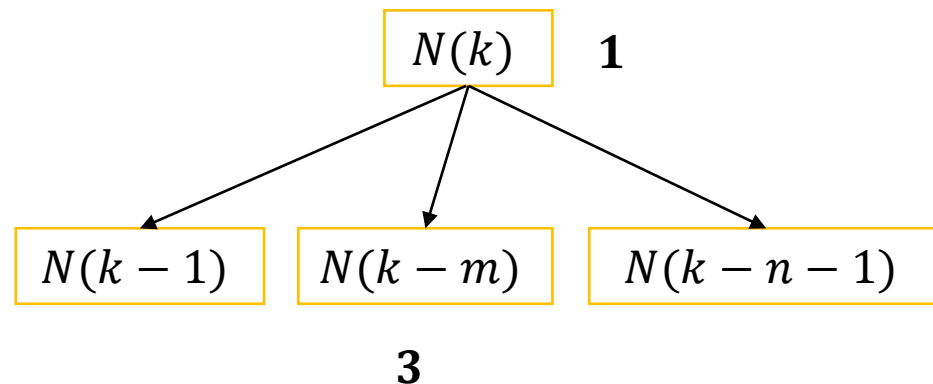
$$\begin{cases} N(k) = 0, k < m \text{ \& } k \neq 0 \\ N(0) = 1, N(m) = 1 \\ N(k) = N(k-1) + N(k-m) - N(k-n-1), k \geq m+1 \end{cases}$$

```
int Ncows(int k) {  
    if( k == 0 ) return 1;  
    else if( k < m ) return 0;  
    else return Ncows(k-1) + Ncows(k-m) - Ncows(k-n-1);  
}
```

$$N(k) = \sum_{i=m}^n N(k-i)$$



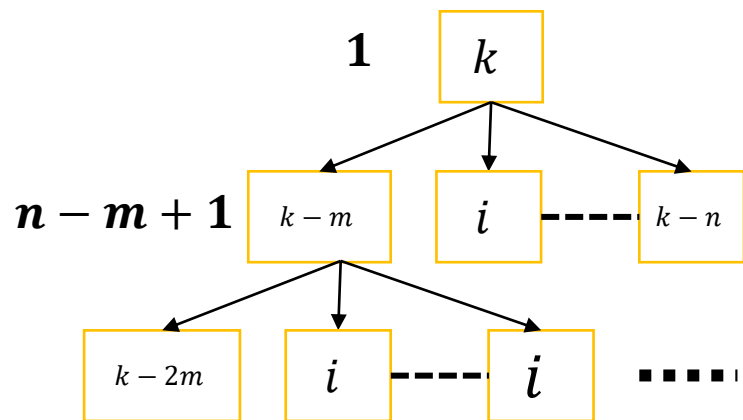
$$N(k) = N(k-1) + N(k-m) - N(k-n-1)$$



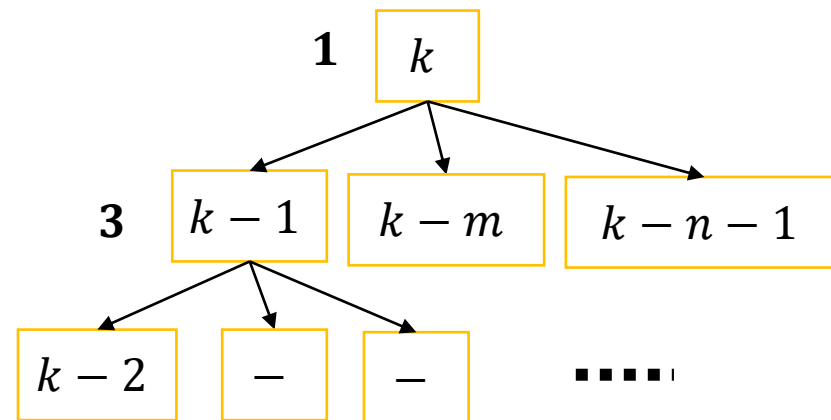
*每次递归调用了3次, $n - m > 2$ 时, 理论上单次递归速度更快

数学模型的建立

虽然有一些改善，但是为什么还是**算不动**！



高度小，但分支多



分支少，但高度大

左树宽度更大，深度更浅；右树宽度更窄，深度更深；

最差情况，迭代重复计算，左树时间复杂度为 $O(2^n)$ ，右树时间复杂度为 $O(3^n)$ ；

都是指数级的！

最好情况，非迭代计算，左树时间复杂度为 $O(n-m+1)$ ，右树时间复杂度为 $O(n)$ ；

需要将小于k的新牛预存下来，以避免重复计算。=> 空间换时间！

注：这就是动态规划算法的思想。

数学模型的建立

空间换时间：存下已经算好的新牛，以避免重复计算。

```
#define N 64
int newcows[N];
int main() {
    ...
    for(i=0; i < N; i++)
    {
        newcows[i] = -1;
    }
    ...
}
```

```
int Ncows(int k) {
    int i;
    if(k == 0) return 1;
    else if(k < m) return 0;
    else if(newcows[k] >= 0) return newcows[k];
    else {
        newcows[k] = Ncows(k-1) + Ncows(k-m) - Ncows(k-n-1);
        return newcows[k];
    }
}
```

时间复杂度降下来了，自然算得动了！

数学模型的建立

上面的分析给了我们提出了一个重要的问题：循环和递归，到底哪个更高效？
显然，从树的角度看，循环的宽度更广，递归的深度更深。
所以上述问题等效于：树广和树深哪个更高效？

为了简化问题，我们首先假设：树在深度与广度方向遍历一步的成本是一样的。
从 $T(x)$ 和 $N(x)$ 的公式可以看出，Tcows和Ncows函数都存在树的效率问题。

遍历方式	$T(k)$	$N(k)$
广度/循环	$T(k) = \sum_{i=0}^{p-1} N(k-i)$	$N(k) = \sum_{i=m}^n N(k-i)$
深度/递归	$T(k) = T(k-1) + N(k) - D(k)$	$N(k) = N(k-1) + N(k-m) - N(k-n-1)$

书上的解法是 $T(x)$ 和 $N(x)$ 都用了广度/循环遍历。
那么，到底用那个更好呢？简单枚举原则！全都实现，然后视情况而定！

数学模型的建立

```
int Tcows(int k)
{
    int i, cows;
    if (k < 0) return 0;
    else if (k < m) return 1;
    else if (k < m + p)
    {
        return Tcows(k-1) + Ncows(k) - Dcows(k);
    }
    else
    {
        cows = 0;
        for(i = k-p+1; i <= k; i++)
        {
            cows += Ncows(i);
        }
        return cows;
    }
}
```

$k < m + p$ 时递归高效

$k \geq m + p$ 时循环高效

```
int Ncows(int k) {
    int i;
    if( k == 0 ) return 1;
    else if( k < m ) return 0;
    else if(newcows[k] >= 0) return newcows[k];
    else if(n - m < 3) {
        newcows[k] = 0;
        for(i = k - n; i <= k - m; i++) newcows[k] += Ncows(i);
        return newcows[k];
    }
    else {
        newcows[k] = Ncows(k-1) + Ncows(k-m) - Ncows(k-n-1);
        return newcows[k];
    }
}
```

$n - m < 3$ 时循环高效

$n - m \geq 3$ 时递归高效

这里只是简化情况！实际的递归效率问题会更为复杂。

数学模型的建立

例7.9给了我们以下编程启示：

- (1) **数学建模**是数学问题，与算法和数据结构无关；
- (2) 在完成数学模型的推导之前，**不要**进行算法设计和编码；
- (3) 数学模型不等于算法，**理论可行不代表工程可行**；
- (4) 算法的设计过程与编程过程**交替进行**；
- (5) **迭代式改进**数学模型可以帮助算法的优化；
- (6) **空间换时间**（时间换空间）是解决时空复杂度的大招，最后才能用！
- (7) 问题的解决方法**不唯一**，要根据实际情况判读优劣；
- (8) 数学建模有利于**深入分析**程序的功能和效率问题。

数学模型的建立

还有一个问题：

从例7-9中，我们可以观察到数学建模、算法、数据结构三者被引入到设计的顺序为：

数学建模=>算法=>数据结构

其中，数学建模要先于算法，这一点我们已经理解了。

那么，算法与数据结构的关系呢？是否算法一定要先于数据结构考虑呢？

下面我们重点讨论数据结构与算法的关系！

数据结构的选择

- 规范化的数据管理操作
 - ◆ 散列表（随机存取）、堆栈（先进后出）、队列（先进先出）
- 前人总结的通用经验
 - ◆ 合适的数据结构可以大大提升编程效率
 - ◆ 数据结构相关代码可以直接照搬，提高代码复用率
 - ◆ 算法设计的得力助手，可分离出非核心的算法设计要求
 - ◆ 更容易设计开发出普适性强的算法和代码
- 数据结构的选择的主要考量
 - ◆ 数据结构的选择在算法设计之前完成：工欲善其事，必先利其器！
 - ◆ 根据数据的特点和行为，选择数据结构
 - ◆ 严格遵照数据结构的标准定义进行函数操作

数据结构的选择

【例8.8】组合的生成

- 设计一个程序，从标准输入上读入正整数 m ($0 < m < 10$) 和 n ($0 < n \leq m$)，在标准输出上输出所有包含 n 个 $1 \sim m$ 的整数的组合。

解题思路：主要矛盾是数据结构

- 1) 从可用的数据结构（散列表、队列、堆栈）中选择一种合适的。
- 2) 之后，再根据数据和数据结构的行为特点设计算法。

数据结构的选择

选择哪种数据结构（散列表、队列、堆栈）更为合适？
需要根据数据特点（而不是程序员的喜好）来判断。

假设：m = 5, n = 2

1	2	3	4	5
---	---	---	---	---

1	2
1	3
1	4
1	5

2	3
2	4
2	5

3	4
3	5

4	5
---	---

数据特点：

1. 后加数频繁变=>需要遍历
2. 遍历的range由历史决定=> $[\max+1, m]$



数据结构的选择：

- ☐ 散列表？×（不关心顺序）
- ☐ 队列？×（先进先出）
- ☒ 堆栈？✓（后进先出，需要频繁操作后进数）

提示：成熟的数据结构是编程的得力助手！

工欲善其事（算法），必先利其器（数据结构）！

数据结构的选择

对数据的进一步观察，我们发现，可以把数字组成一棵树：

只要实现了树的遍历，就可以找到所有的目标数。

栈的行为：后进先出

栈的操作：压栈弹栈

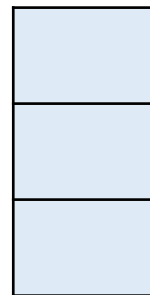
现在，例8.8问题已经转化为了：
如何利用压栈弹栈解决树的遍历问题？

这个问题比原问题要好得多：

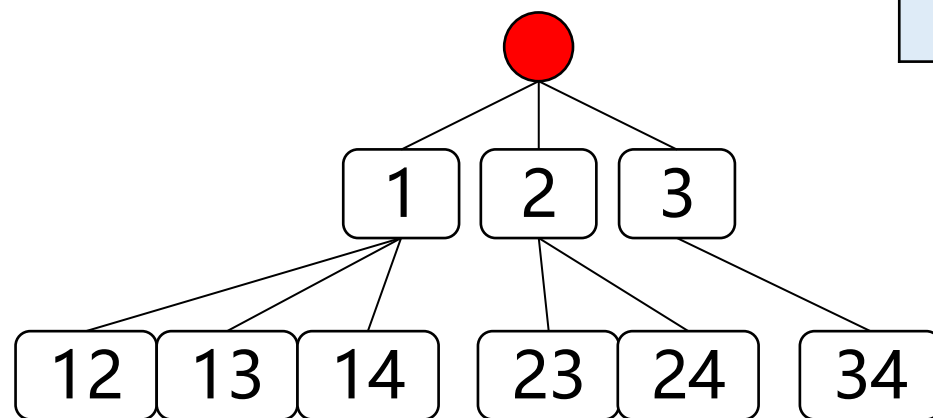
1. 在树和栈上更容易找规律；
2. 这是个通用问题，更容易借鉴和被借鉴；



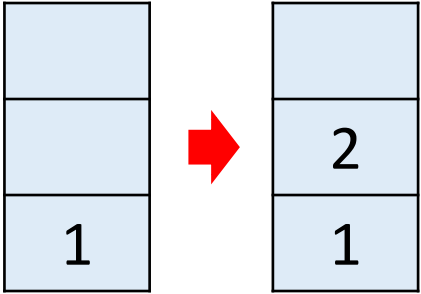
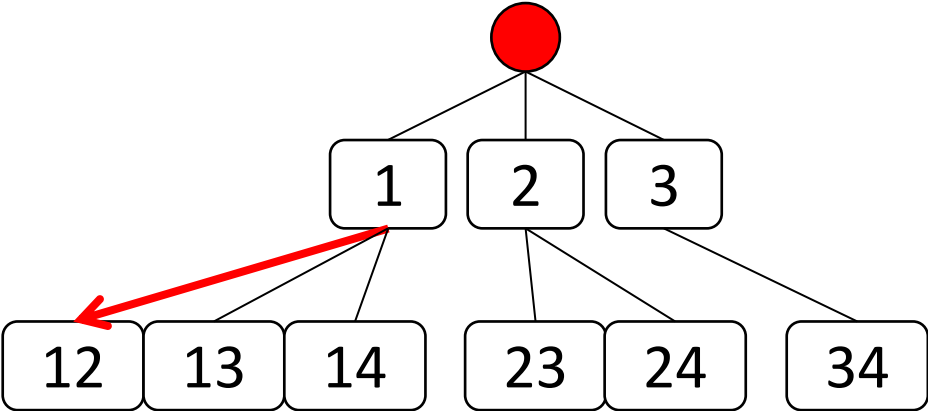
1~5挑2个不重复数



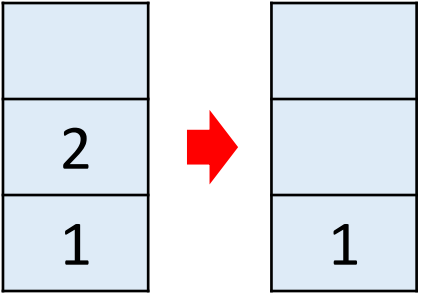
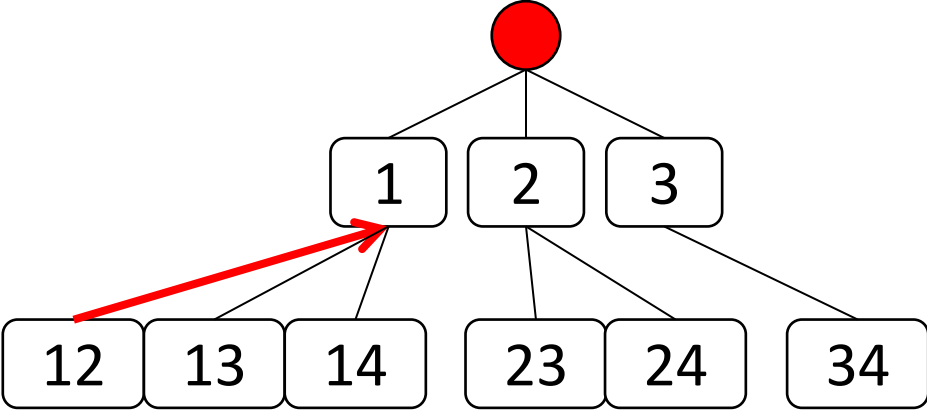
栈



数据结构的选择



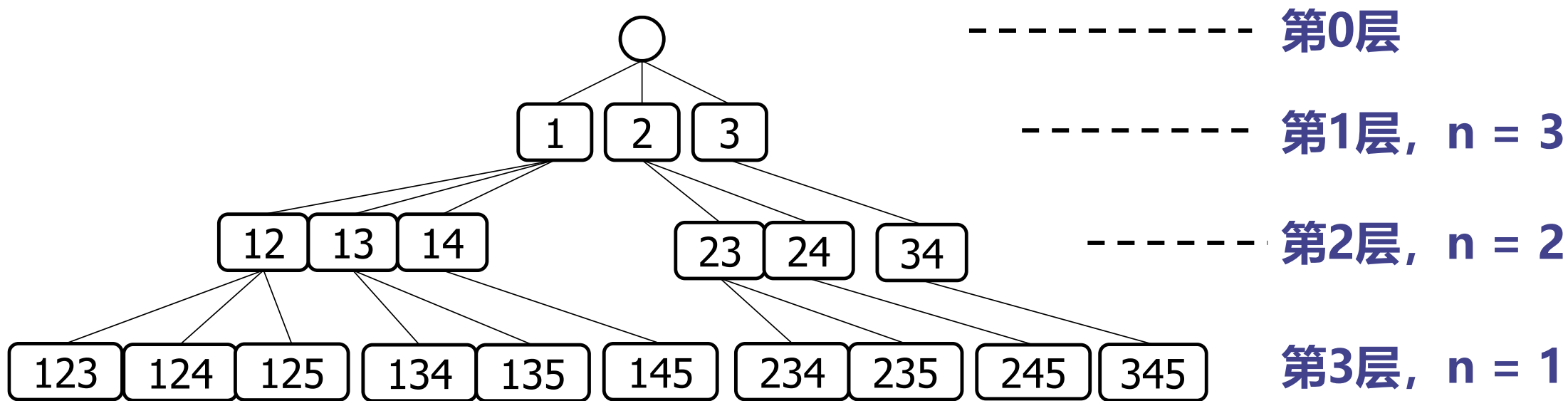
树的下行就是压栈



树的上行就是弹栈

数据结构的选择

进一步验证，假设： $m = 5, n = 3$ ，压栈递归多叉树



上边界问题：从树可以看出，每一层的上边界为 $m-n+1$

数据结构的选择

循环(压栈=>递归=>弹栈)是遍历一棵树的标准做法 (每个组合都是一个叶节点)

代码实现:

```
void comb(int m, int n)
{
    int i, lb, ub;
    if (n > 0)
    {
        lb = stackTop() + 1;
        ub = m - n + 1;
        for(i = lb; i <= ub; i++)
        {
            stackPush(i);
            comb(m, n-1);
            stackPop();
        }
    }
    else
    {
        stackPrint();
    }
}
```

Range一定要控制好! 栈顶为最大值

压栈, 树的下行

带栈递归, 压栈剩下的n-1个数, 遍历子树

弹栈, 树的上行

递归的基本情况, 组合生成, 放在栈中!

数据结构的选择

代码优化：接口问题

```
void comb(int m, int n) // 从1~m中，挑出n个数
{
    int i, lb, ub;
    if (n > 0)
    {
        lb = stackTop() + 1;
        ub = m - n + 1;
        for(i = lb; i <= ub; i++)
        {
            stackPush(i);
            comb(m, n-1); // 从1~m中，挑出n-1个数? 错误!
            stackPop();
        }
    }
    else
    {
        stackPrint();
    }
}
```

函数接口的定义和调用不一致!
造成理解和调试困难。

从lb~ub中，挑出n-1个数!

数据结构的选择

函数接口优化带来的更大好处在例8-10中体现!

代码优化：接口与实现分离

```
void getComb(int m, int n) // 从1~m中, 挑出n个数
{
    stackInit(n);
    comb(1, m - n + 1); //从1~(m-n+1)中, 挑出剩余压栈的数
}
```

函数接口与题意一致
——对人友好：便于理解

```
void comb(int lb, int ub) // 从lb~ub中, 挑出剩余压栈的数
{
    if ( !stackFull() )
    {
        for(; lb <= ub; lb++)
        {
            stackPush(lb);
            comb(lb+1, ub+1); // 从(lb+1)~(ub+1)中, 挑出剩余压栈的数
            stackPop();
        }
    }
    else
    {
        stackPrint();
    }
}
```

函数接口与调用一致
——对计算机友好：需要递归!

函数重在接口!
递归重在调用!

数据结构的选择

其他代码

```
void stackInit(int n)
{
    top = 0;
    top_max = n;
}
void stackPrint()
{
    int i;
    for(i = 0; i < s_top; i++) {
        printf("%d", stack[i]);
    }
    printf("\n");
}
```

```
void main()
{
    int m, n;
    scanf("%d%d", &m, &n);
    getComb(m, n);
}
```

注：省略了栈的基本操作函数（stackPush, stackPop, stackTop, etc.）, 同学们可以参考C6的课件和附件代码。

数据结构的选择

【例8.10】花朵数

- 一个n位的十进制正整数，如果它的每个位上的数字的n次方的和等于这个数本身，则被称为花朵数。从标准输入上读入一个正整数n($n \leq 9$)，求所有的n位花朵数。
- 例如153就是一个长度为3的花朵数，因为 $1^3 + 5^3 + 3^3 = 153$

思路：

对于这道题，最直观的方法就是枚举所有长度为n的正整数，检查被枚举的数的各位的n次方之和是否等于该数。

数据结构的选择

【例8.10】花朵数：方法1

```
int main()
{
    int i, min_v = 1, n;
    scanf("%d", &n);
    for(i = 0; i < n-1; i++)
    {
        min_v *= 10;
    }
    for(i = min_v; i < min_v * 10; i++)
    {
        if(is_flower(i,n))
            printf("%d\n",i);
    }
    return 0;
}
```

• 读取数字n

• 生成最小的n位数min_v

• 遍历所有n位数，若存在花朵数，则输出该花朵数

数据结构的选择

【例8.10】花朵数：方法1

```
#define N 20
int is_flower(int num, int len)
{
    int i, j, s, t, dig[N], m = num;
    for (i = 0; i < len; i++)
    {
        dig[i] = m % 10;
        m /= 10;
    }
    for (s = i = 0; i < len; i++)
    {
        for (t = 1, j = 0; j < len; j++)
            t *= dig[i];
        s += t;
    }
    return s == num;
}
```

- 判断长度为len数num是否为花朵数的子程序
is_flower:

• 分离num的各个十进制位

• 计算各位len次方之和

• 返回逻辑量：各位len次方之和s是否与num相等，即num是否为花朵数

数据结构的选择

【例8.10】花朵数：方法1

```
int is_flower(int num, int len)
{
    int i, s, dig[N], m = num;
    for (i = 0; i < len; i++)
    {
        dig[i] = m % 10;
        m /= 10;
    }
    for (s = i = 0; i < len; i++)
        s += digit_n[dig[i]];
    if (s == num)
        return 1;
    return 0;
}
```



- 判断长度为len数num是否为花朵数的子程序is_flower:
- 每次需要计算各位的len次方，属于重复计算
- 用空间换时间

```
int digit_n[10] = {0, 1};
int main(int argc, char **argv)
{
    int i, j, n;
    ...
    for (i = 2; i < 10; i++)
    {
        digit_n[i] = 1;
        for (j = 0; j < n; j++)
            digit_n[i] *= i;
    }
    ...
}
```

数据结构的选择

【例8.10】花朵数：方法2

- 观察：9474、4974、7494、4497，仅交换顺序，每位数字的4次方和是不变
- 列举0-9在4位数中出现的次数，得到求和后的数，花朵数一定在这些数中（如果有）！

0	1	2	3	4	5	6	7	8	9
0	0	0	0	2	0	0	1	0	1

$$4^4 + 4^4 + 7^4 + 9^4 = 9474$$

- rule1：如果和是一个可能的花朵数，那么一定是由4，4，7，9排列得到
- rule2：统计和9474数字出现次数，4出现2次，7和9各一次，因此是花朵数

0	1	2	3	4	5	6	7	8	9
0	1	0	0	0	0	0	2	0	0

$$1^3 + 7^3 + 7^3 = 687$$

- 如果和是一个可能的花朵数，那么一定是由1，7，7排列得到
- 判断687中的每一位，与1，7，7不符合，因此不是花朵数

数据结构的选择

【例8.10】花朵数：方法2

主要矛盾是数据结构！

假设 $n=3$ ，则要遴选出3个数来（可重复）组合。

```
3
0 0 0
0 0 1
0 0 2
0 0 3
0 0 4
0 0 5
0 0 6
0 0 7
0 0 8
0 0 9
0 1 1
0 1 2
0 1 3
0 1 4
0 1 5
0 1 6
0 1 7
0 1 8
0 1 9
```

← 这些都有可能！

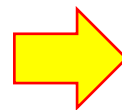
但是1 0 0、1 3 2、7 8 3是不可能的！

因为0 0 1、1 2 3、3 7 8等效于上述组合

推论：后一位必须 \geq 之前数字的最大值！

也就是说，需要知道：

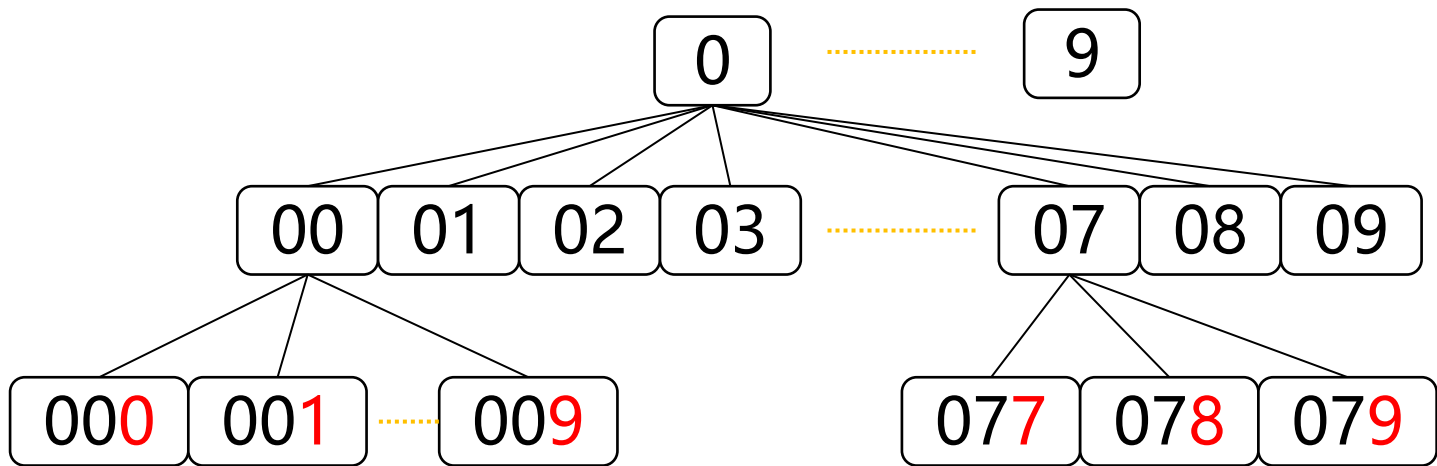
1. 历史选出的数；
2. 选数的顺序；
3. 后选的数变的更频繁；



栈！

数据结构的选择

假设：n = 3，压栈递归多叉树



循环栈递归本质是在做树的遍历！（注意上下界）
递归的退出基本条件：叶节点！

例8-8与例8-10的上下界（区别在于可否重复）

	例8-8	例8-10
下界	[stackTop] + 1	[stackTop]
上界	m - n + 1	m

数据结构的选择

【例8.10】花朵数：方法2

```
int powers[10] = {0, 1};  
int min_v = 0;  
void main()  
{  
    int n;  
    scanf("%d", &n);  
    initPowers(n);  
    adjustMinV(n);  
    generateFlowers(n);  
}
```

初始化0-9的n次方powers
和最小的可能值min_v!

核心函数!

注：正确使用的数据结构是编程效率倍增器

数据结构的选择

核心函数generateFlower: 用堆栈!

```
void generateFlowers(int n)
```

```
{
```

```
    int i, sum;
```

```
    if (n > 0)
```

```
    {
```

```
        for (i = stackTop(); i < 10; i++)
```

```
        {
```

```
            stackPush(i);
```

```
            generateFlowers(n - 1);
```

```
            stackPop();
```

```
        }
```

```
    }
```

```
    else
```

```
    {
```

```
        sum = stackPowSum();
```

```
        if (sum >= min_v && sum < min_v * 10 && stackCheckSum(sum))
```

```
        {
```

```
            printf("%d\n", sum);
```

```
        }
```

```
    }
```

```
}
```

允许重复

压栈! 树的下行

带栈递归, 树的遍历!

弹栈! 树的上行

算幂和

基本情况: 叶节点!

校验和

至此, 我们成功的将例8-8的经验引入了例8-10!

数据结构的选择

计算幂和函数

```
int stackPowSum()
{
    int i, sum = 0;
    for(i = 0; i < s_top; i++)
    {
        sum += powers[stack[i]];
    }
    return sum;
}
```

遍历全栈

利用之前算好的幂

```
void initPowers(int n)
{
    int i, j;
    for (i = 2; i < 10; i++)
    {
        powers[i] = 1;
        for (j = 0; j < n; j++)
        {
            powers[i] *= i;
        }
    }
}
```

不能用pow函数，会损失精度！

```
void adjustMinV(int n)
{
    int i;
    min_v = 1;
    for(i = 0; i < n - 1; i++)
    {
        min_v *= 10;
    }
}
```


数据结构的选择

校验和函数

```
int compare(const char *p1, const char *p2) { return *p1 - *p2; }
int stackCheckSum(int sum)
{
    int i = 0, n = 0;
    char digits[10] = {0};
    while (sum > 0)
    {
        digits[n++] = sum % 10;
        sum /= 10;

        qsort(digits, n, sizeof(char), compare);
        for (i = 0; i < s_top; i++)
        {
            if (stack[i] != digits[i])
                return 0;
        }
        return 1;
    }
}
```

比较函数

将幂和转为数组

按升序排序

stack已经是升序了，直接比较

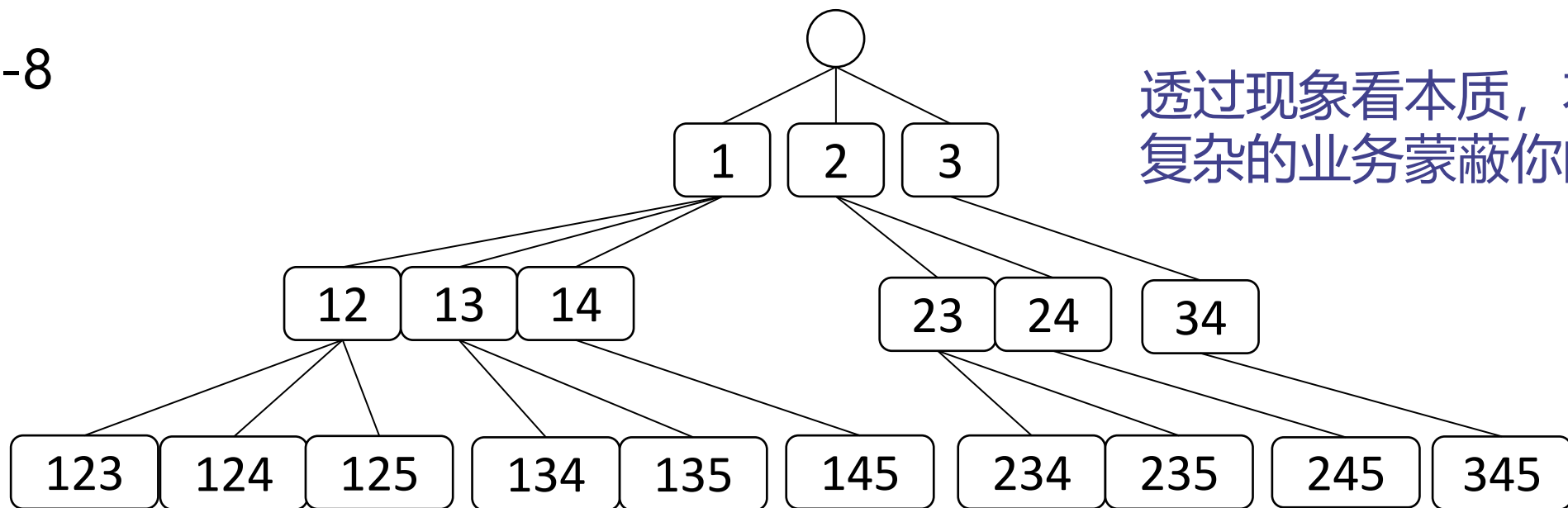
有不同，返回0，不是花朵数

都相同，返回1，是花朵数

数据结构的选择

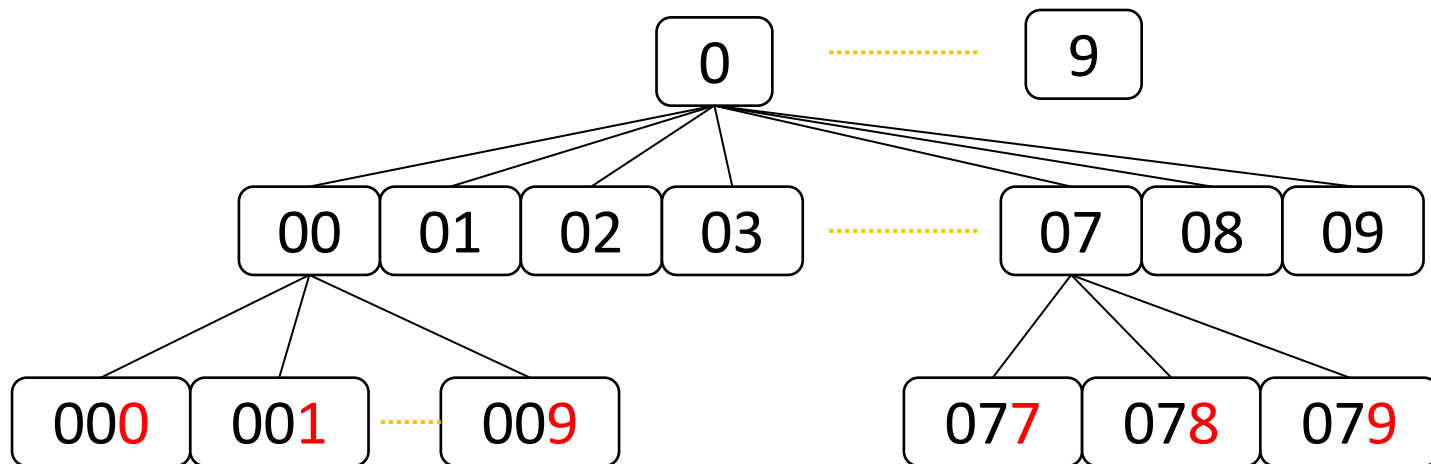
都是树的遍历问题！

例8-8



透过现象看本质，不要被复杂的业务蒙蔽你的双眼！

例8-10



数据结构的选择

【例8.10】花朵数：方法3

从方法2我们可以看到，将业务问题和算法框架问题分离所带来的好处非常明显：大大节约了算法设计的复杂度。

但是方法2的代码还是有一点遗憾：既然可以复用，为何不直接复用例8.8的代码？至少是框架部分。

吃一堑长一智，万一以后又遇到数字的组合问题呢？直接可以少写很大一块。

例8.8与例8.10的框架代码的唯一区别就是允不允许重复。

数据结构的选择

如果例8-8的函数接口没有优化，这里改起来就没那么方便了！

【例8.10】花朵数：方法3

能不能将例8.8的框架改写成通用版本？

```
int repeat = 0; // 0: 不允许重复; 1 : 允许重复
void getComb(int m, int n, void (*printer)(void)) )
{
    stackInit(n);
    repeat ? comb(0, m, printer) : comb(1, m - n + 1, printer);
}
```

算法框架函数

```
void comb(int lb, int ub, void (*printer)(void))
{
    if (!stackFull())
    {
        for (; lb <= ub; lb++)
        {
            stackPush(lb);
            repeat ? comb(lb, ub) : comb(lb + 1, ub + 1);
            stackPop();
        }
    }
    else
    {
        printer();
    }
}
```

将业务抽象成函数指针参数

	例8-8	例8-10
下界	1	0
上界	m-n+1	m
递增	1	0

区别在于允不允许重复

数据结构的选择

【例8.10】花朵数：方法3

```
void printFlower() // 业务函数
{
    int sum;
    sum = stackPowSum();
    if (sum >= min_v && sum < min_v * 10 && stackCheckSum(sum))
    {
        printf("%d\n", sum);
    }
}
void generateFlowers(int n)
{
    repeat = 1; //花朵数的各位允许重复
    getComb(9, n, printFlower);
}
int main()
{
    int n;
    scanf("%d", &n);
    initPowers(n);
    adjustMinV(n);
    generateFlowers(n);
}
```

业务函数实体

函数指针

数据结构的选择

例8.8和例8.10给了我们以下编程启示：

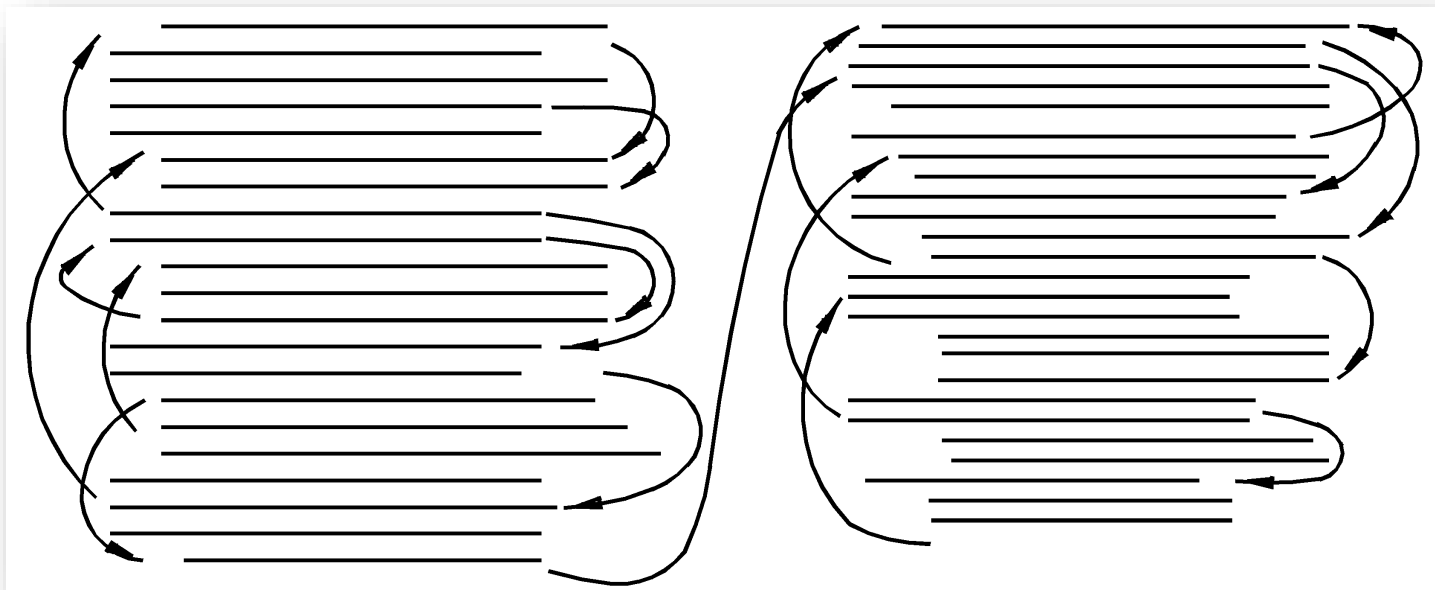
- (1) 善于抓住问题的主要矛盾，透过现象看本质；**
- (2) 通过观察数据特点来选择合理的数据结构；**
- (3) 设计算法时应综合考虑目标问题及数据结构的行为；**
- (4) 函数的接口设计很重要，要重视形参与实参的意义一致性；**
- (5) 注重业务与框架的分离，可以增加代码的复用性；**
- (6) 代码的优化工作最后再做（Knuth定律：过早优化是万恶之源）。**

算法的设计

- 算法是使用计算机求解问题的有限步骤
 - 表示为运算的序列
- 算法评估和比较的主要考虑
 - 运行速度/资源消耗（时间、空间复杂度）
 - 实现的复杂度/难度
- 算法设计的主要考量
 - 算法设计应尽量遵循一些编程规则：**无以规矩，不能成方圆！**
 - 自顶向下，往往**代码的结构设计**重于函数内部设计
 - 关注运行效率，分析时间/空间复杂度
 - 关注开发效率，确保函数功能的单一性和接口的易读性

算法的设计

- 算法设计过程中，重点在于设计**代码的结构**。



缺点：难以阅读、修改，程序的可靠性和可维护性难以保证。

- 良好的程序结构可以提高代码的效率、可读性、可维护性，并减少错误。
- 通常采用**自顶向下的分层描述策略**和**对互相独立任务的水平分解策略**。

算法的设计

● 自顶而下

- ◆ 出发点：从问题的总体目标开始，抽象低层的细节，先专心构造高层的结构，然后再一层一层地分解和细化，直至所有操作都可以转化为一系列子问题。
- ◆ 使设计者把握主题，高屋建瓴，避免一开始就陷入复杂的细节中，使复杂的设计过程变得简单明了，过程的结果也容易做到正确可靠。

● 水平分解

- 对复杂问题，设计一些子目标作为过渡，把复杂的大任务分割为若干相互独立的子任务，并分别完成各子任务的描述和综合。

编码策略的共同点：把复杂问题不断分解为大的、中的、小的、超小的相对简单的子问题，直至问题可以直接求解。

实现方法：通过函数定义和调用实现。

算法的设计

【例8.12】序列的第N项

程序说明:

- 在序列 a_1, a_2, \dots, a_n 中, 对于任意 $i > 1$, a_i 是满足下面两个性质的最小正整数:
 - ① $a_i > a_{i-1}$
 - ② a_i 的各位数字之和与 $k * a_{i-1}$ 的各位数字之和相等。
- 从标准输入上读入正整数 a_1 、 k 、 n ($0 < a_1 \leq 1000$, $0 < k < 300000$, $0 < n < 6000$), 计算该序列第 n 项 a_n 的值并写到标准输出上。

例如:输入 1 2 12 时,序列为

1 2 4 8 16 23 28 29 49 89 97 149

算法的设计

【例8.12】序列的第N项

- 顶层设计
 - I. 输入数据 a_1, k, n
 - 数据类型、数据结构
 - II. 根据输入数据生成最终结果
 - 设计算法, 根据 a_i , 生成 a_{i+1}
 - III. 根据生成的数据输出

算法的设计

【例8.12】序列的第N项

- 顶层设计

- I. 输入数据 a_1, k, n ($0 < a_1 \leq 1000, 0 < k < 300000, 0 < n < 6000$)
 - 确定数据的大小范围, 选择数据类型和结构

简单分析:

取 $k=1, a_1=1$

$a_2=10$

$a_3=100$

取 $k=1, a_1=10$

$a_2=100$

$a_3=1000$

取 $k=1, a_1=1000$

$a_n =$ 计算结果位6003位

当 k 和 a_1 取其他值时, 结果的位数随 n 的增加而增长的速度相对较慢。

算法的设计

【例8.12】序列的第N项

- 设计算法，生成 a_{i+1}



对于任意 k ， 能不能找到 a_i 和 a_{i+1} 之间的关系？

换一种思路：

1. 计算出 a_i 的各位数字之和 $ds(a_i)$;
2. 计算 a_{i+1} 的各位数字之和 $ds(k*a_i)$;
3. 根据第1步、第2步数字之和间的关系，对 a_i 的各位进行调整得到 a_{i+1} ，使其是满足 $a_{i+1} > a_i$ ， a_{i+1} 的各位数字之和与 $k*a_i$ 的各位数字之和相等这两个条件最小正整数;
4. 依次得到 a_2 、 a_3 、 a_4 、...、 a_n

算法的设计

在我们解决复杂问题之前，需要认清以下一个事实：

代码的实现依赖算法，然而问题到算法之间存在一个巨大的鸿沟，叫作**智商**！



存在两种手段来填满这个智商的鸿沟：

1. 最强大脑
2. 借助代码架构降低对智商的要求



除非故意炫技，后者当然是我们期望的了！

算法的设计

代码结构设计的推荐原则：

- (1) 单一责任——德国菜刀
- (2) 接口简洁——庖丁解牛
- (3) 行为多态——阿凡达

不以规矩不能成方圆！



单一责任



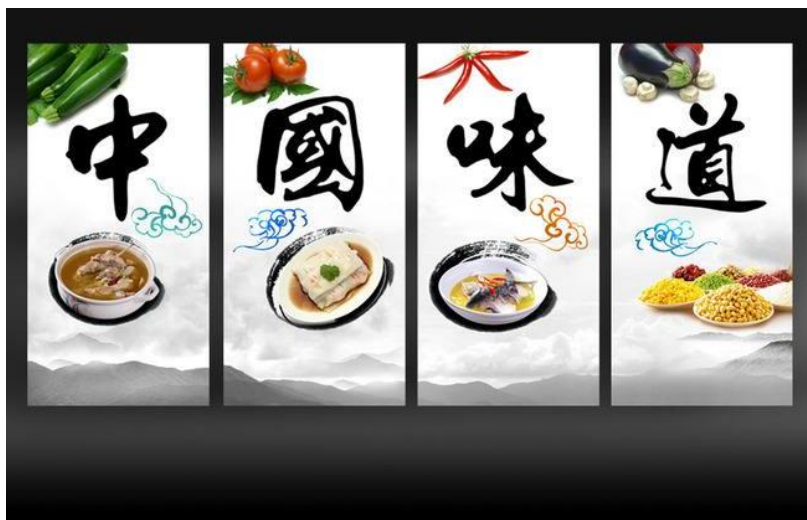
接口简洁



行为多态

算法的设计

舌尖上的中国



算法的设计

舌尖上的德国



地狱里最好的厨师是德国人!

算法的设计

舌尖上的德国



面包刀：刀长、刀刃带锯齿。面包、烤肉切片。薄而均匀；

切片刀：防粘、防滑。切菜、片鱼，切肉。轻巧锋利，得心应手；

主厨刀：适合用力处理的食材；

厨师刀、去皮刀.....

德国菜刀
多刀多用
单一责任

算法的设计

单一责任——德国菜刀



中国菜刀，复合责任！
做菜可以，编程不行！



德国菜刀，单一责任！
做菜不行，编程可以！

算法的设计

核心目的还是算法，大数只是干扰项。
我们还是应该以算法为主要矛盾，先不考虑大数！

```
int main()
{
    int i, ai = 0, k = 0, n = 0;
    scanf("%d%d%d", &ai, &k, &n);
    for (i = 1; i < n; i++)
    {
        ai = generateNext(ai, k);
        printf("%d\n", ai);
    }
    return 0;
}
```

单一责任，输入用int

for循环外面做，分解掉参数n

生成下一个数，输入参数ai和k，没有n！

水平分解

注：幻灯片顺序即算法设计顺序，自顶向下的做法是先设计顶层函数，同时水平分解掉非核心问题；

算法的设计

进一步采用单一责任：

该分开的，再简单也要分开；不该分开的，再复杂也不能分开！

```
int generateNext(int ai, int k)
{
    int digitsum;
    digitsum = computeDigitSum(ai * k);
    return adjustNumber(ai, digitsum);
}
```

水平分解

计算数字和，乘法单独做！进一步分解掉参数k

调整数到目标和，数的调整是一件事，输入ai和digitsum，不再考虑参数k。

出现核心函数：**adjustNumer**

目标问题得以分解和转化：如何将一个**正整数向上调整到目标和**？

有人会问：这不是啥也没做吗？

稍安勿躁~~

自顶向下的原则是：**首先设计顶层，而不考虑具体实现！**

算法的设计

回顾一下当前设计好的顶层函数：

```
//先考虑函数单一责任和接口设计，一开始不要考虑如何实现！  
//函数接口（参数）设计很重要！一定要简单易读，理解不依赖于内部实现！  
int generateNext(int ai, int k); // 根据ai和k生成下一个数  
int adjustNumber(int v, int digitsum); // 将数v向上调整到目标和digitsum
```



始臣之解牛之时，所见无非牛者。
三年之后，未尝见全牛也。
方今之时，臣以神遇而不以目视，官知止而神欲行。
良庖岁更刀，**割**也；
族庖月更刀，**折**也；
今臣之刀十九年矣，所解数千牛矣，而刀刃若新发于硎。

彼节者有间，而刀刃者无厚；以无厚入有间，恢恢乎其于游刃必有余地矣！

既不能砍骨头也不能割肉，找缝隙下手
——函数接口设计很重要！

算法的设计

adjustNumber的等效钱包问题：

`int adjustNumber(int v, int digitsum);` // v: 现有钱数; digitsum: 目标张数

假设，ai有136元：



1



3



6

钱数：136元

张数：1+3+6 = 10张

目标问题的简化得益于顶层函数的水平分解和核心函数的单一责任与接口设计！

136是随意假定的！因为核心函数跟n和k一点关系都没有。对于adjustNumber来说，v当然可以是任意数！

算法的设计

adjustNumber的等效钱包问题：**钱数要增，张数也要增！**
即 $Ds(k \cdot a_i) > Ds(a_i)$ ，假设需要从10张加到16张：

16也是任意假定的！



1



3



6

需要加 $16 - 10 = 6$ 张钱：

- (1) 先把1块加满，到9
- (2) 剩下再加10块
- (3) 不够再加100...



3



3

算法的设计

adjustNumber的等效钱包问题：钱数要增，张数也要增！
 $Ds(k \cdot a_i) > Ds(a_i)$ （10张加到16张）的调整结果：



1



6



9

钱数：169元 > 136元
张数：1+6+9 = 16张

算法的设计

adjustNumber的等效钱包问题：**钱数要增，张数要减！**
即 $Ds(k*ai) < Ds(ai)$ ，假设需要从10张减到8张：



1



3



6

零钱换整钱：

- (1) 1块的换成10块的
- (2) 6张变1张
- (3) 不够再10块换成100块的...

思路：张数要减，硬塞是不行的！零钱换整钱（拿钱的人肯定愿意^_^）的目的是先把钱包做薄，然后再寻找机会。



4



1

算法的设计

adjustNumber的等效钱包问题：**钱数要增，张数要减!**
 $Ds(k*ai) < Ds(ai)$ (10张减到8张) 的初步调整结果:

还记得**数的调整是一件事**的单一责任原则吗?
幸好当时没把加钱和减钱分开, 否则怎么能想到递归的方法? !



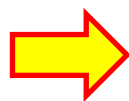
1

4

0

钱数: 140元

张数: $1 + 4 = 5$ 张 < 8 张!



转换为 $Ds(k*ai) > Ds(ai)$ 的问题, **递归!**

算法的设计

adjustNumber的等效钱包问题：钱数要增，张数要减！
 $Ds(k*ai) < Ds(ai)$ （10张减到8张）的最后调整结果：



1

钱数：143元 > 136元
张数：1+4+3 = 8张



4



再加3张1块的！

3

思路：这一步的加钱动作不需要考虑换整钱之前的钱数和张数，抛却烦扰，重新上路！

算法的设计

adjustNumber的等效钱包问题: **钱数要增, 张数不变!**
即 $Ds(k \cdot a_i) = Ds(a_i)$, 假设需要10张不变:



1



3



4



6



1



必须至少换一次零钱:

- (1) 1块的换成10块的
- (2) 没有1块的, 10块换成100块的...
- (3) 递归前两种情况

算法的设计

adjustNumber的等效钱包问题: **钱数要增, 张数不变!**
 $Ds(k \cdot a_i) = Ds(a_i)$ (10张不变) 的初步调整结果:



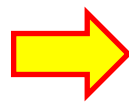
1



4

0

钱数: 140元
张数: $1 + 4 = 5$ 张 < 10 张!



转换为 $Ds(k \cdot a_i) > Ds(a_i)$
的问题, 递归!

算法的设计

adjustNumber的等效钱包问题: **钱数要增, 张数不变!**
 $Ds(k*ai) = Ds(ai)$ (10张不变) 的最后调整结果:



1



4



再加5张1块的!

5

钱数: 145元 > 136元

张数: 1+4+5 = 10张

算法的设计

回到之前的智商鸿沟设定。



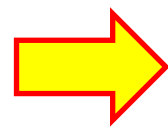
从哪个函数出发，更容易联想到等效钱包问题？

main



根据a1、n和k，生成第n个数an

adjustNumber



将一个数调整到目标数字和

将一笔钱调整到目标张数

尽管核心代码还没开始写，代码结构的设计已经大大降低了目标问题对智商的要求！

算法的设计

核心代码其实很简单:

```
int adjustNumber(int v, int digitsum, int flag)
{
    int dsx;
    dsx = computeDigitSum(v);
    if (dsx < digitsum)
    {
        v = digitAdd(v, digitsum - dsx);
    }
    else if (dsx > digitsum || flag == 0)
    {
        v = carryLowest(v);
        v = adjustNumber(v, digitsum, 1);
    }
    return v;
}
```

增加一个flag参数来区分张数不变情况

计算钱的张数

钱数要增，张数要增的情况

加钱函数

钱数要增，张数要减的情况

钱数要增，张数不变的情况

零钱换整钱函数

算法的设计

我们还需要三个函数：计算数位和（钱张数）函数

```
int computeDigitSum(int v)
{
    int sum = 0;
    while (v > 0)
    {
        sum += v % 10;
        v = v / 10;
    }
    return sum;
}
```

注：进行adjustNumber函数的编码时，并不需要考虑计算钱张数、加钱和、零钱换整钱等函数的具体实现。但在编写完adjustNumber函数之后，其余的函数也得到了大大的简化。这也是自顶向下和单一责任的具体体现。

算法的设计

我们还需要三个函数：加钱函数

```
int digitAdd(int v, int add)
{
    int i, p = 1, m;
    for (i = 0; add > 0; i++)
    {
        m = 9 - ((v / p) % 10);
        if (add > m)
        {
            v += m * p;
            add -= m;
        }
        else
        {
            v += add * p;
            add = 0;
        }
        p *= 10;
    }
    return v;
}
```

从个位开始，依次加！

加不完就到9，继续

加得完就结束

算法的设计

我们还需要三个函数：换钱函数

```
int carryLowest(int v)
{
    int m, p = 1;
    m = v;
    while((m % 10) == 0)
    {
        m = m / 10;
        p *= 10;
    }
    m = 10 - ((v / p) % 10);
    v += m * p;
    return v;
}
```

算最小的零钱

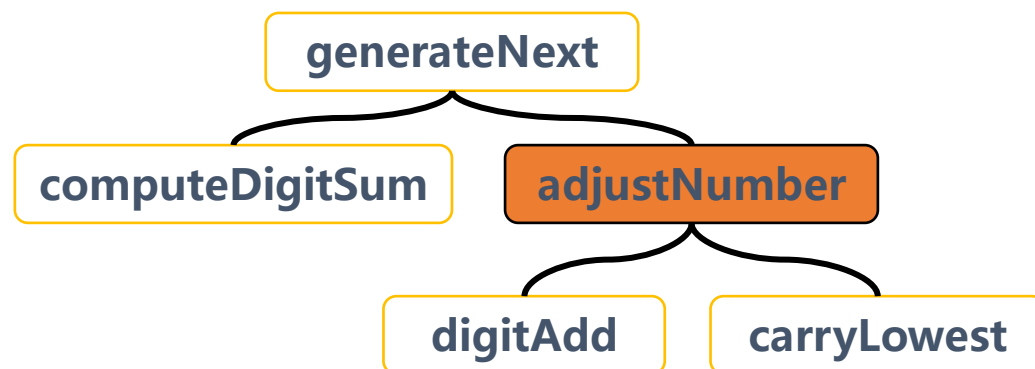
加满到10张，零钱换整钱

算法的设计

总结一下:

```
//第一层函数
int generateNext(int ai, int k);
//第二层函数
int computeDigitSum(int v);
int adjustNumber(int v, int digitsum, int flag); //核心函数!
//第三层函数
int digitAdd(int v, int add);
int carryLowest(int v);
```

核心函数1个，辅助函数4个：
典型的二八定律——
20%核心代码，80%辅助代码

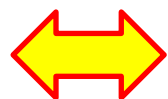


算法的设计

大数怎么办？



地球



潘多拉

主要矛盾（算法）已经解决了，现在需要考虑次要矛盾（数据结构）了！

卡梅隆没去过潘多拉星，如何构思《阿凡达》的剧情？

参照地球生物，为每个潘多拉生物设计相应的属性行为，造一个可与地球对照的生态！

——多态

然后，在地球上设计好的剧本（算法）就可以照搬到潘多拉星球上去了！

算法的设计

定义参照的属性行为，造一个类似的生态——多态！
大数的算法和小数版本是一样的，只是大数的属性行为不一样。

```
char ai[LEN] = {0};
```

大数用数组

VS

```
int a0 = 0;
```

小数用INT

属性/行为	int	char[]
存储	YES	YES
位数	YES	NO
传递方式	传值	传址

需要参照小数int的属性行为来设计大数的数据结构：
大数的数据结构应该可存大数、位数，以及支持传值！

算法的设计

设计一个高效的传递大数的值和位数的类型——用struct+指针!

```
typedef struct SuperLong
{
    int len;
    char* buf;
} slong;

char ai_buf[LEN], kai_buf[LEN];
```

定义一个结构体类型slong，存超大数：

- (1) 用len存位数，用buf存每一位的数字；
- (2) 为了传值高效，采用char* buf而不是char buf[N]；
- (3) 在外部定义ai_buf用以存 a_i 和kai_buf用以存 $k \cdot a_i$ ；


自己动手，丰衣足食！

算法的设计

参照小数版本的函数，设计大数版本的函数

```
int computeDigitSum(int v);  
int digitAdd(int v, int add);  
int carryLowest(int v);  
  
int adjustNumber(int v, int digitsum, int flag);  
int generateNext(int ai, int k);
```

```
int computeDigitSum_sl(slong v);  
slong digitAdd_sl(slong v, int add);  
slong carryLowest_sl(slong v);  
  
slong adjustNumber_sl(slong v, int digitsum, int flag);  
slong generateNext_sl(slong ai, int k);
```



算法的设计

参照int的属性行为，设计大数的基础函数：

```
slong int2sl(int n)
{
    slong ai;
    int i = 0;
    while (n > 0)
    {
        ai_buf[i++] = n % 10;
        n /= 10;
    }
    ai.len = i;
    ai.buf = ai_buf;
    return ai;
}
```

int转slong

```
void printf_sl(slong v)
{
    int i;
    for (i = v.len - 1; i >= 0; i--)
    {
        printf("%d", v.buf[i]);
    }
    printf("\n");
}
```

打印slong

算法的设计

参照小数的底层函数，设计大数的计算数字和函数：

```
int computeDigitSum_sl(slong v)
{
    int i, sum = 0;
    for (i = 0; i < v.len; i++)
    {
        sum += v.buf[i];
    }
    return sum;
}
```

算法的设计

大数的乘法比小数复杂得多：

```
slong timesInt(slong ai, int k)
{
    int i, tmp = 0;
    slong r;
    r.len = ai.len;
    r.buf = kai_buf;

    for (i = 0; i < ai.len; i++)
    {
        tmp += k * ai.buf[i];
        tmp = overflow(&(r.buf[i]), tmp);
    }

    while (tmp)
    {
        r.len++;
        tmp = overflow(&(r.buf[r.len - 1]), tmp);
    }
    return r;
}
```

还记得**乘法单独做**的单一责任原则吗？
幸好当时分开了，否则就要修改架构了！

```
int overflow(char *digit, int load)
{
    *digit = load % 10;
    return load / 10;
}
```

计算溢出

算法的设计

大数的加钱函数

```
slong digitAdd_sl(slong v, int add) {  
    int i, m;  
    for (i = 0; add > 0; i++)  
    {  
        if (i > v.len)  
        {  
            v.len = i;  
            v.buf[i] = 0;  
        }  
        m = 9 - v.buf[i];  
        if (add > m)  
        {  
            v.buf[i] = 9;  
            add -= m;  
        }  
        else  
        {  
            v.buf[i] += add;  
            add = 0;  
        }  
    }  
    return v;  
}
```

溢出了！数组不能自动增位

目的一样，实现方法不一样！

多态的意思是：八仙过海，各显神通。
大数的加钱实现方法可以和小数大有不同。

算法的设计

大数的零钱换整钱函数

```
slong carryLowest_sl(slong v)
{
    int i;
    for (i = 0; i < v.len; i++) 获得最小的零钱
    {
        if (v.buf[i] != 0)
            break;
    }

    v.buf[i] = 0;
    return carry(v, i + 1); 换整钱+递归进位
}
```

大数的零钱换整钱函数比小数版本更复杂！
得益于单一责任，这个复杂度可以被封闭于函数内部，而不影响其他代码。

```
slong carry(slong v, int i)
{
    if (i >= v.len)
    {
        v.len = i + 1;
        v.buf[i] = 1;
        return v;
    }
    if (v.buf[i] + 1 < 10)
    {
        v.buf[i]++;
        return v;
    }
    v.buf[i] = 0;
    return carry(v, i + 1);
}
```

算法的设计

大数的顶层函数

```
slong adjustNumber_sl(slong v, int digitsum, int flag)
{
    int dsx;
    dsx = computeDigitSum_sl(v);
    if (dsx < digitsum)
    {
        v = digitAdd_sl(v, digitsum - dsx);
    }
    else if (dsx > digitsum || flag == 0)
    {
        v = carryLowest_sl(v);
        v = adjustNumber_sl(v, digitsum, 1);
    }
    return v;
}
```

函数架构和int版本一致

这里多用了kai_buf，如何优化？
(简单枚举原则，前后并不矛盾)

```
slong generateNext_sl(slong ai, int k)
{
    int digitsum;
    digitsum = computeDigitSum_sl(timesInt(ai, k));
    return adjustNumber_sl(ai, digitsum, 0);
}
```

自顶向下设计的好处在于，顶层结构几乎不变：代码的架构不变。

算法的设计

大数版本的main函数:

```
int main()
{
    int a1 = 1, k = 0, n = 1, i;
    slong ai;
    scanf("%d%d%d", &a1, &k, &n);
    ai = int2sl(a1);
    printf_sl(ai);
    for (i = 1; i < n; i++)
    {
        ai = generateNext_sl(ai, k);
        printf_sl(ai);
    }
    return 0;
}
```

函数架构和int版本类似

大数版本的参照设计方法也是一种自顶向下的设计：从一个int衍生出一大堆的底层函数，却基本不会增加算法本身的复杂度。

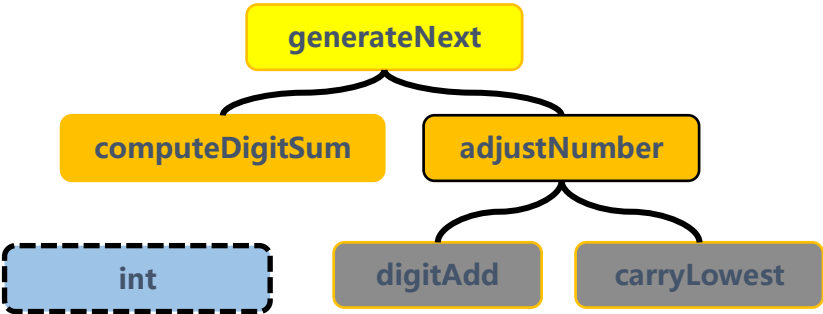
算法的设计

小数版本 vs 大数版本

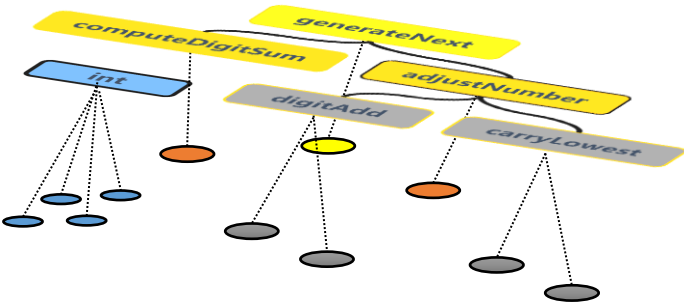
generateNext	generateNext_sl
adjustNumber	adjustNumber_sl
computeDigitSum	computeDigitSum_sl
digitAdd	digitAdd_sl overflow
carryLowest	carryLowest_sl carry
int	int2sl printf_sl timesInt
	typedef struct SuperLong { int len; char* buf; } slong; char ai_buf[LEN], kai_buf[LEN];

动态行为

静态属性



一维自顶向下



二维自顶向下

为什么要先设计小数版本？
——降维打击！

算法的设计

例8-12给我们的启示：

- (1) 要根据问题的主要矛盾来分离出**算法设计问题**；
- (2) 重视代码结构的设计可以实现对目标问题的**分解和转化**；
- (3) 一些基本的**编程原则**是降低算法设计难度的得力助手；
- (4) PPT代码比教材上的代码产生的代码量要多很多，这一点可以引起我们对**“过度设计”**的辩证思考；
- (5) 考试、竞赛时，推荐用教材上的设计方案，因为**时间**紧迫；
- (6) 项目开发时，推荐用PPT的设计方案，因为代码的**可读性、复用性和可扩展性**要高很多。

算法的设计

可扩展性和复用性验证：代码的进一步优化

我们用数组存大数是为了解决int超限的问题。但是，大数版本的函数具有更高的时间/空间复杂度，如果数字不超限，也会受到牵连，这当然不是我们想要的。

“鱼与熊掌兼得” 的解决方案：**分而治之**
数小的时候用小版本函数，数大的时候用大数版本函数。

在教材上的代码中，这一解决方案几乎无法实施，原来的代码结构只好被推倒重来；

在PPT的代码中，情况却大不一样。小数版本和大数版本的绝大多数函数可以得以保留，**只需要改一下顶层代码即可。**

算法的设计

可扩展性和复用性验证：代码的进一步优化

首先，将buf数组修改成指针，以实现内存动态分配，同时不影响顶层架构。

```
// char ai_buf[LEN], kai_buf[LEN];  
char *ai_buf, *kai_buf;  
  
void initBufs()  
{  
    ai_buf = (char *)malloc(LEN);  
    kai_buf = (char *)malloc(LEN);  
}  
  
void freeBufs()  
{  
    if (ai_buf)  
        free(ai_buf);  
    if (kai_buf)  
        free(kai_buf);  
}
```

需要使用的时候再分配内存；
内存是资源，借了就要还。

算法的设计

然后，修改顶层，实现**大小数函数兼容**，同时**不影响底层架构**：

```
#include<limits.h> //获得INT_MAX
void main()
{
    int k = 0, n = 1, i;
    int small;
    slong big = {0, NULL};
    scanf("%d%d%d", &small, &k, &n);
    printf("int: %d\n", small);
    for (i = 1; i < n; i++)
    {
        if (big.buf == NULL && small <= INT_MAX / k)
        {
            small = generateNext(small, k); // 数比较小时，调用小数函数
            printf("int: %d\n", small);
            continue;
        }
        else if (big.buf == NULL)
        {
            initBufs();
            big = int2sl(small);
        }
        big = generateNext_sl(big, k); // 数比较大时，调用大数函数
        printf("slong: ");
        printf_sl(big);
    }
    freeBufs();
}
```

函数动态调用；
内存动态分配；
鱼与熊掌可以兼得！

小结

- 掌握自顶而下、水平分解等问题求解方法
- 学会如何分析问题的主要矛盾
- 理解例7.9的数学建模方法
- 理解例8.8、例8.10的数据结构选择方法
- 理解例8.12的算法设计方法
- 理解编程开发的单一责任原则
- 理解接口简洁的函数设计原则
- 理解行为多态的架构设计方法

课程总结

《画》王维 唐
远看山有色，
近听水无声。
春去花还在，
人来鸟不惊。



让我们的代码像
诗歌一样优美！

远看山有色，
近听水无声。
王维诗
丁酉年
林其
公画

全课完