04-控制流

一个声明

因为这个部分不太涉及java特有的属性or语法所以本人是用**c语言**写的（用惯了稍微方便一点），所以在输出语句上会有一点差别（前面没有`System.out`），请审核大大谅解T^T

Task1.if-else

Q1:判断闰年
（编程入门经典题目）
***判断闰年的条件：**

1. 被4整除且不被100整除
2. 被400整除

==**代码实现如下：**==
注：问题：题目用的是bool型但是返回值需要1or2（？），这里有一点没理解到。

```
 int isLeapYear(int n)
{

    if(n%4==0&&n%100!=0){
        return 1;
    }
    else if(n%400==0){
        return 1;
    }
    else{
        return 2;
    }
}
```

其他回答：
A1： switch-case的基础语法如下：

```
switch (表达式) {
    case 常量表达式1:语句系列1

        break;
    case 常量表达式2:语句系列2

        //break;
        ……
        ……
        ……
    default:

}
```

*break和default视情况可写可不写。

A2:

- 啥是语法糖(syntax suger)：语法糖是指一种语法上的便利，它并没有引入新功能，只是让代码更易写、易读。
- `switch-case`的底层原理：**跳转表** (Jump Table) 或 **二分查找** (Binary Search)。
    - **跳转表**：这个"跳转表"就是一个**数组**，每个case的值直接对应一个地址（要执行的代码的地址）。

      使用时，CPU先计算switch表达式的值。然后，CPU 将这个值减去最小的 case值，得到一个偏移量。最后，CPU 直接通过这个偏移量索引跳转表，从中取出目标地址，并立即跳转到该地址执行。
    - **二分查找**：当 case值非常稀疏时，构建一个庞大的跳转表会浪费大量内存，此时编译器会采用更高效的比较策略。

      如其名，二分查找从字面意思理解比较简单，就像我们猜数游戏一样。比如我们要猜1-16之间的一个数，先猜16/2=8，再根据比8大 or 比8小来继续猜测（比如比8小，那么我们猜4，后续以此类推）。

      在工作时，编译器会将所有 case 常量值排序。然后生成一系列的比较和跳转指令，但这些指令的组织方式不再是线性的（像`if-else`），而是**树形结构**。接着，CPU 会首先与中间的一个值进行比较，根据比较结果（大于、小于）决定下一次比较的方向，从而逐步缩小范围，最终定位到正确的 case。
- `if-else`的底层原理：**线性比较**
    - 打个比方，就像敲门找人一样，if-else会从上到下按顺序询问你是谁从而找到你要找的人。
    - 编译器会将其编译为一系列**线性的**比较和条件跳转/无条件跳转指令。CPU 必须**按顺序**执行每一个比较，直到找到匹配的条件。

结论：这样来看，某些情况下（尤其是分支）`switch-case`是`if-else`的语法糖，因为其提升了代码效率，代码更简洁，可读性更高。

Task2.for-while

Q2：打印空心菱形

分析过程如下：

```
/*
假设n=7

将空格键替换成@符号便于观察，先打印上半部分

行数          两边@    中间@
1  @@@*@@@  3        0
2  @@*@*@@  2        1
3  @*@@@*@  1        3
4  *@@@@@*  0        5

行数max=(n+1)/2=M(定值)  令行数=i  两边@=M-i
中间@=(i-1)*2-1=2*i-3 [if i==1,中间@=0]

再继续打印下半部分
行数          两边@    中间@
1  @@@*@@@  3        0
2  @@*@*@@  2        1
3  @*@@@*@  1        3
```

```
4  *@@@@@*  0       5

5  @*@@@*@  1       3
6  @@*@*@@  2       1
7  @@@*@@@  3       0

5=3 6=2 7=1 即关于M对称，其和即为n+1=N
此时行数从i=M+1开始，该行数对应上半部分行数=N-i,令N-i=l
*/
```

代码实现如下：

```
void print(int n)
{
    int M=(n+1)/2;
    int N=n+1;
    int i,j,k;

        for(i=1;i<=M;i++){
            for(j=1;j<=M-i;j++){
                //printf("@");
                printf(" ");
            }

        printf("*");

        if(i>1){
            for(k=1;k<=2*i-3;k++){
                //printf("@");
                printf(" ");
            }
            printf("*");
        }

        printf("\n");
    }
        //下半部分开始
        for(i=M+1;i<=n;i++){
            int l=N-i;
            for(j=1;j<=M-l;j++){
                //printf("@");
                printf(" ");
        }

        printf("*");

        if(i<n){
            for(k=1;k<=2*l-3;k++){
                //printf("@");
                printf(" ");
            }
            printf("*");
        }

        printf("\n");
```

```
    }
}
```

Task3.递归和迭代

Q3：斐波拉契数列

- 啥是**递归**：**一个过程或函数在其定义或说明中直接或间接地调用自身。**
  主要思考方式：**大事化小。**
  - 递归的两个必要条件：
  - 1.存在限制条件，当满足限制条件时，递归便不再继续。
    2.每次递归调用之后越来越接近这个限制条件（最简单情况，最初始情况）。
- 啥是**迭代**：迭代就是重复，类似“**循环**”。

用两种方式实现斐波拉契数列的第n项输出：

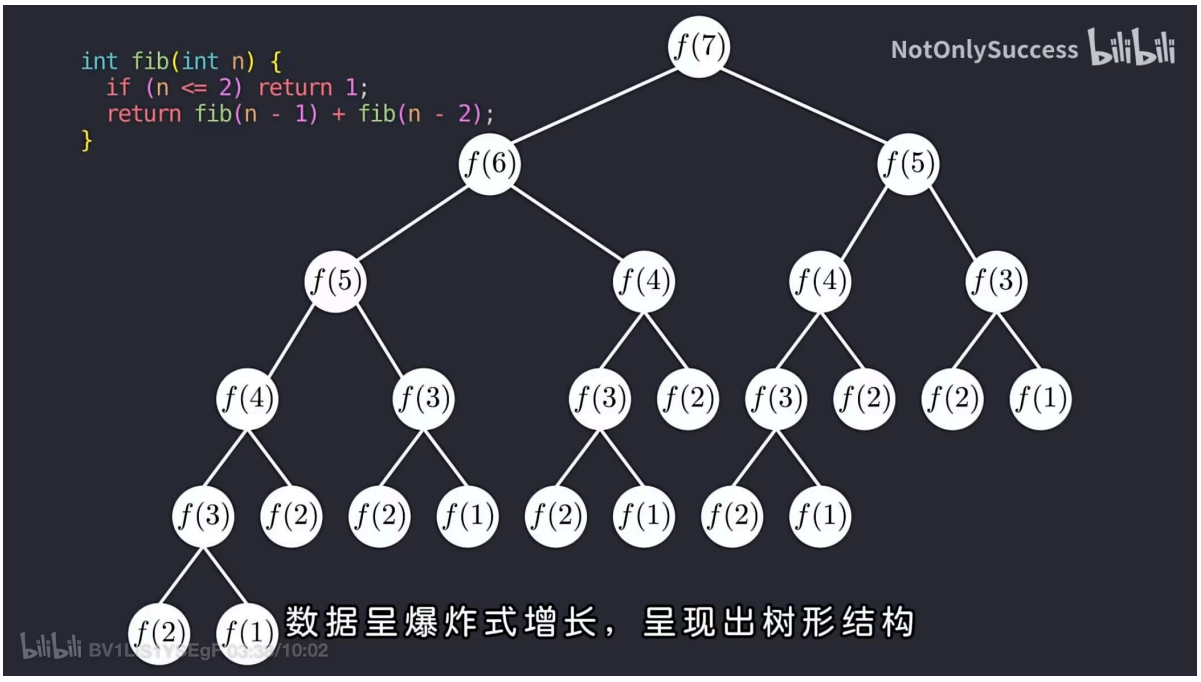
Edition 1：递归

```c
int fib(int n)
{
    if(n<=2){
        return 1;
    }
    else{
        return fib(n-1)+fib(n-2);
    }
}
```

Edition 2：迭代

```c
int fib2(int n)
{
    int a=1,b=1;
    int c;
    if(n<=2){
        return 1;
    }
    while(n>=3){
        c=a+b;
        a=b;
        b=c;
        n--;
    }
    return c;
}
```

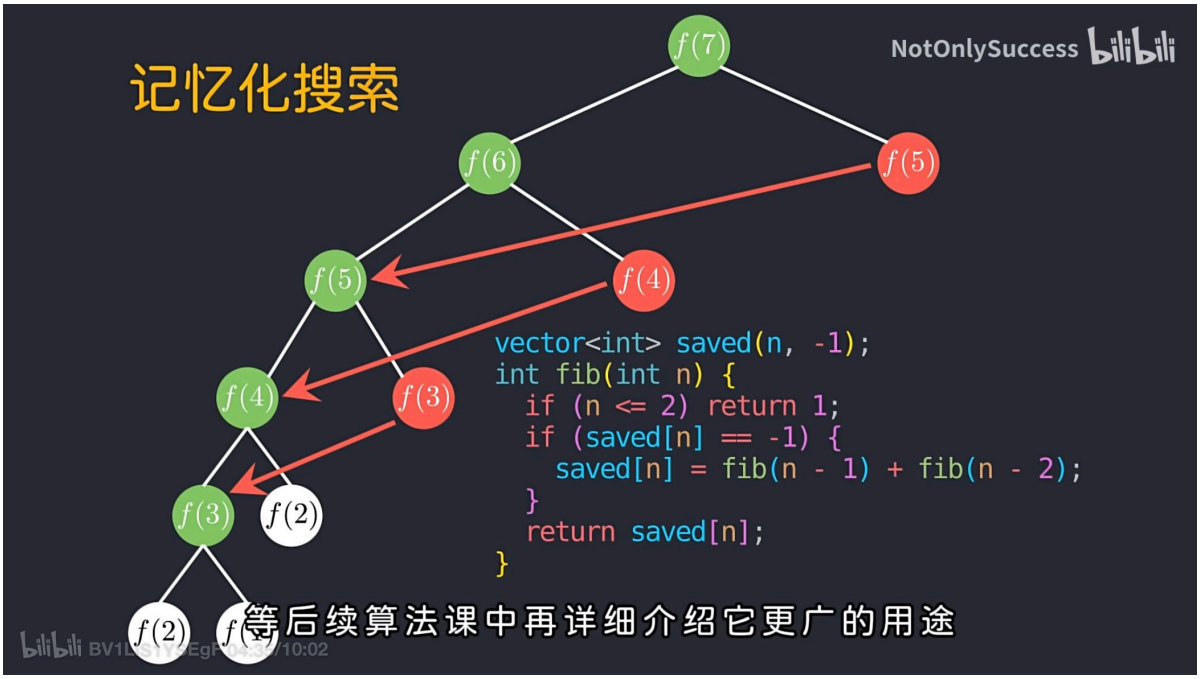
但是在这里用迭代会有一个**问题**：当迭代层数够深的时候，数据呈爆炸式增长，呈现出树状结构，所以某些层会被反复调用，导致计算速度减缓，代码效率降低，容易栈溢出。



所以，有时我们**更偏好迭代**，是因为其简单粗暴，思路展现直观（至少本人这样认为），容易调试，且不容易栈溢出。

虽然呢，理论上循环（迭代）能完全用递归来取代，但应该没有人会这样做吧……（？）能简单做的问题何必"宰羊焉用牛刀"？以本人尚浅的认知来看，递归更适用于汉诺塔（下题）、树的遍历这样为递归而生的的东西。（？）

*注：这道题的递归写法可以进行改进来避免代码效率过低和栈溢出问题，解决方案如下：
~~（不过我并没有看懂）~~



Task4.汉诺塔

最基本情况（此处暂不讨论n=1和n=2的情况）：n=3时，需要7步。

```
//A->C   (A->C的意思是把A柱最上方的铁饼移动到C柱上，以下同理)
//A->B
//C->B
//A->C
//B->A
//B->C
//A->C
```

接下来进行递归推理：

- n=4时，先移动最上面3个铁饼到B，再将最下面一个移动到C，最后将这3个铁饼移动到C。
  则需要：7+1+7=15步。
- n=5时，先移动最上面4个铁饼到B，再将最下面一个移动到C，最后将这4个铁饼移动到C。
  移动4个铁饼时 -> 即回到n=4的情况。
  则需要：15+1+15=31步。
- 由此可得递推关系：移动n个 -> (n-1)个的情况 -> (n-2)个的情况 -> ..... -> 3个的情况。
  ~~(可以得到：移动n个铁饼时需要(2^n-1)步。)~~

解题步骤：

先定义一个move函数来打印出移动方向

```
void move(char p1,char p2)
{
    //打印出该步骤的移动方向
    printf(" %c -> %c ",p1,p2);
}
```

我们需要输入四个东西：铁饼数n，三个柱子ABC，将其分别作为起始柱、中转柱和目标柱。

```
void hanoi(int n,char pos1,char pos2,char pos3)
//pos1:起始 pos2：中转 pos3：目的
```

先考虑递归的限制条件，也就是最基本条件：n=1时，此时直接移动就可以了。

```
if(n==1){
        move(pos1,pos3);
    }
```

接下来列举n=2和n=3的情况：

```
/*
*n=2
move(pos1,pos2)
move(pos1,pos3)
move(pos2,pos3)

*n=3
move(pos1,pos2)【上面2层整体，但实际是不能这样移动的】
    move(pos1,pos3)【细分操作】
    move(pos1,pos2)
    move(pos2,pos3)
```

```
move(pos1,pos3)【最下面一层】
move(pos2,pos3)【上面2层整体，但实际是不能这样移动的】
    move(pos2,pos1)【细分操作】
    move(pos2,pos3)
    move(pos1,pos3)
*/
```

可以发现：柱子并不是“固定不变”的。给柱子命名为“ABC”只是为了方便称呼，真正移动柱子时只需要知道**铁饼从哪里来，中转经过谁，要到哪里去**即可。

递归部分代码如下：

```
        hanoi(n-1,pos1,pos3,pos2);
        //将上层n-1个看成整体，把pos3看成中转柱，将其从pos1移动到pos2。即：从源柱子 -> 中转柱
        move(pos1,pos3);
        //最下面的最大铁饼从pos1 -> pos3
        hanoi(n-1,pos2,pos1,pos3);
        //将上层n-1个看成整体，把pos1看成中转柱，将其从pos2移动到pos3。即：从中转柱 -> 目标柱
```

完整的函数部分代码如下：

```
void hanoi(int n,char pos1,char pos2,char pos3)
{
    //pos1:起始 pos2：中转 pos3：目的
    if(n==1){
        move(pos1,pos3);
    }
    //最简单情况，直接移动即可
    else{
        hanoi(n-1,pos1,pos3,pos2);
        //将上层n-1个看成整体，把pos3看成中转柱，将其从pos1移动到pos2。即：从源柱子 -> 中转柱
        move(pos1,pos3);
        //最下面的最大铁饼从pos1 -> pos3
        hanoi(n-1,pos2,pos1,pos3);
        //将上层n-1个看成整体，把pos1看成中转柱，将其从pos2移动到pos3。即：从中转柱 -> 目标柱
    }
}

void move(char p1,char p2)
{
    //打印出该步骤的移动方向
    printf("%c -> %c ",p1,p2);
}

```

注：参考学习视频和部分截图来源

绝对没有纯抄！！！是自己看了好几遍才看懂之后重新整理思路打的T^T

- Task1：

    - 带你走进Switch的源码世界
- Task3：
    - 有个说法：“「递归」是检验编程天赋的试金石”；而本视频打破天赋壁垒，助你快速掌握递归。
- Task4：
    - 递归经典问题汉诺塔动画演示+代码讲解