Java04

Task1

补全如下：

```
boolean isLeapYear(int year){
   if((year%100!=0 && year%4==0) || year%400==0){
        return true;
    } else{
        return false;
    }
}
```

虽然switch确实可以用一系列if-else if语句来实现相同逻辑，**但是**switch-case**不仅仅**是语法糖，因为：1、编译器会对其进行特殊优化；2、在某些情况下性能明显优于if-else；3、有更严格的语法约束（如case必须是常量表达式）。

if-else的底层实现原理

1. **基础**：依赖于 CPU 的条件码寄存器（FLAGS）。
2. **关键操作**：通过 CMP（比较）或 TEST（测试）指令来设置条件码。
3. **流程控制**：使用条件跳转指令（如 JE, JNE, JG, JL 等）来根据条件码决定程序的执行路径。这些指令是 CPU 级别实现的，效率极高。
4. **性能**：if-else 链的性能取决于条件的顺序。最可能为真的条件应该放在最前面，这样可以减少不必要的比较次数。在最坏情况下（条件都在最后），它是一个 O(n) 的操作。
5. **与 switch 的区别**：正如上一个问题所讨论的，if-else 是线性扫描，而 switch 在可能的情况下会被编译器优化为跳转表（O(1)）或二分查找（O(log n)），这在分支非常多时性能差异巨大。

switch-case的底层实现原理

编译器会根据 case 值的数量、分布和连续性来选择最优的实现方式，主要有三种策略：

1. **跳转表（Jump Table） - O(1) 时间复杂度** *（这是 switch 最高效的实现方式，当 case 值连续且密集时使用。）*： 编译器会创建一个地址表（跳转表），每个表项对应一个 case 值的跳转地址。执行时直接通过索引计算找到目标地址。
2. **二分查找（Binary Search） - O(log n) 时间复杂度** *（当 case 值数量较多但不连续时使用。）*： 编译器将 case 值排序，然后生成二分查找代码。
3. **线性比较（If-Else 链） - O(n) 时间复杂度** *（当 case 值非常少（通常3-4个以下）或者极其稀疏时使用。这种情况下，switch 的实现确实类似于 if-else，但编译器仍然可能优化比较顺序。）*

Task2

补全如下：

```
void print(int n){
    int m = (n/2);
    char[][] ch = new char[n+1][n+1];
    for(int i =1; i<=n;i++){
        for(int j=1;j<=n;j++){
            ch[i][j]=' ';
        }
    }
    ch[m+1][1]='*';
    ch[m+1][n]='*';
    for(int i =1;i<=m;i++){
        ch[i][(m+1)+(i-1)]='*';
        ch[i][(m+1)-(i-1)]='*';
        ch[n-(i-1)][(m+1)+(i-1)]='*';
        ch[n-(i-1)][(m+1)-(i-1)]='*';
    }
    for(int i =1;i<=n;i++){
        for(int j =1;j<=n;j++){
            System.out.print(ch[i][j]);
        }
        System.out.print('\n');
    }
}
```

Task3

（这里应该还暂时不用考虑高精度算法吧）

第一个版本（递归）：

```
int Fibonacci(int n){
    if(n==1 || n==2){
        return 1;
    }
    return Fibonacci(n-1) + Fibonacci(n-2);
}
```

第二个版本（迭代）：

```
int Fibonacci(int n){
    int[] a = new int[50];
    //实测n到47时int就爆了
    a[1]=1;
    a[2]=1;
    for(int i =3;i<=n;i++){
        a[i]=a[i-1]+a[i-2];
    }
```

```
    return a[n];
}
```

迭代 vs 递归：

特性	迭代	递归
实现机制	使用循环结构	函数调用自身
性能	高效，无函数调用开销	较低，有函数调用开销
可读性	相对直接，但可能冗长	更优雅，更符合数学思维
栈溢出	无风险	深度过大时可能发生
调试难度	相对容易	相对困难

一般偏好迭代的原因：

1. 性能优势（迭代：O(n)时间复杂度，O(1)空间复杂度； 递归：O(n)时间复杂度，O(n)空间复杂度）。
2. 内存安全，避免栈溢出。
3. 实际工程考虑
    1. 可维护性：迭代代码更容易被其他开发者理解
    2. 调试便利：循环的调试比递归调用栈调试简单
    3. 资源控制：迭代可以更好地控制内存使用

循环不能完全用递归取代

虽然迭代和递归在计算能力上是等价的，但是考虑到栈深度限制、性能要求限制、实时系统和高并发环境限制*（递归的时间不确定性不可接受）*，循环不能完全用递归取代。

Task4

较简单直观的实现：通过递归实现

```
import java.util.*;
public class java04 {
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        Hanoi(n,'A','C','B');
```

```
    }
    static void Hanoi(int n ,char src ,char tgt ,char tmp){
        //四个变量分别代表：移动的圆盘个数、移动的起点、移动的终点、临时柱子
        if(n==1){//当移动的圆盘个数为1时，直接输出移动步骤
            System.out.println(src + "->" + tgt);
            return;
        }
        Hanoi(n-1,src,tmp,tgt);//将（n-1）个圆盘移动到临时柱子上，以便最底下那个圆盘能移动到目标柱子上
        Hanoi(1,src,tgt,tmp);//将最底下那个圆盘移动到目标柱子上
        Hanoi(n-1,tmp,tgt,src);//将（n-1）个圆盘叠放到目标柱子上
    }
}
```