Java-04

一、Task1.if-else

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

int a = isLeapYear(year) ? 1 : 2;
//true返回1 false返回2
System.out.println(b);
```

1. **适用范围**: **switch-case** 只能用于 **离散数据** 的判断

   而 **if** 则可以用于更加复杂的条件判断

2. **执行原理与效率:**

   从 **switch** 字节码可以看出其逻辑

```
L7
 LINENUMBER 22 L7
 ILOAD 1
 LOOKUPSWITCH
   0: L9
   5: L9
   9: L9
   default: L9
L9
 LINENUMBER 28 L9
 ILOAD 1
 TABLESWITCH
   0: L10
   1: L10
   2: L10
   default: L10
L10
 LINENUMBER 36 L10
 RETURN
```

   值得注意的是 **switch** 实际上有两种逻辑

   - 一种是对离散数据生成 **LOOKUPSWITCH**

     生成一个键值对表

并使用 **二分查找算法** 查询对应的 **case**

时间复杂度O(log n)

- 另一种是对密集连续数据生成 **TABLESWITCH**

  生成一个数组并通过每个 **case** 对应索引

  时间复杂度O(1) 效率更高

而 **if** 则是顺序执行扫描

显然 **时间复杂度O(N)** 效率更低

二、Task2.for-while

```
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("请输入层数：");
        int n = sc.nextInt();
        int a = 0;
        if (n%2 != 1){
            System.out.println("不是奇数，请重新输入：");
            n = sc.nextInt();
        }

        for (int i = 0;i < n/2+1;i++) {
            a = 0;
            while(a < n/2-i){
                System.out.printf(" ");
                a++;
            }
            System.out.printf("*");
            int j;
            for (j = 0;j < i*2-1 && i != 0;j++){
                System.out.printf(" ");
            }
            if (i != 0){
                System.out.printf("*");
            }
            System.out.printf("\n");
        }
        //以下逻辑完全一致
        //考虑到文件本身较小就选择了直接复制并做少量修改
        for (int i = n/2 - 1;i < n/2+1 && i >= 0;i--) {
            a = 0;
            while(a < n/2-i){
                System.out.printf(" ");
                a++;
            }
            System.out.printf("*");
            int j;
            for (j = 0;j < i*2-1 && i != 0;j++){
                System.out.printf(" ");
            }
            if (i != 0){
                System.out.printf("*");
            }
            System.out.printf("\n");
        }
```

```
    }
```

三、Task3.递归和迭代

1. **不同之处:**

   迭代: 循环过程可见，性能开销通常更小

   递归: 代码简洁美观，但可能导致重复计算有大量性能开销

2. **偏好迭代的原因:** 如上所述，递归有两个较严重的问题，

   一是循环过程不可见导致不够直观，检索错误更难

   二是性能开销大，每一次递归会都从下到上重新计算一遍，层数多时甚至肯导致 **栈溢出** 的问题

3. 递归从功能上是可以取代循环的

   但是性能损耗的问题是不可忽视的，

   因此只有在适合于递归的

   代码演示:

```java
    //迭代演示
    static void Iteration(int n){
        int[] arr = new int[n];
        for (int i = 2;i < n;i++) {
            arr[0] = 1;
            arr[1] = 1;
            arr[i] = arr[i-1] + arr[i-2];
        }
        System.out.println(arr[n-1]);
    }
    //递归演示
    static int Recursion(int n){
        if (n == 1){
            return 1;
        }
        else if (n == 2){
            return 1;
        }
        else {
            return Recursion(n-1)+Recursion(n-2);
        }
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("请输入n值:");
        int n = sc.nextInt();
         Iteration(n);
        int a = Recursion(n);
        System.out.println(a);
    }
```

四、Task4.汉诺塔

以下简称移动n层的操作为 **f(n)** [from,to]

可知 **f(n)** = **f(n)** [from,help] + (from->to) + **f(n)** [help,to]

以这个逻辑完成程序编写即可，详细代码如下:

```
    static void Hanno(String from,String to,String help,int layers){
        if (layers == 1){
            System.out.println(from + "->" + to);
        }
        else {
            Hanno(from,help,to,layers-1);
            System.out.println(from + "->" + to);
            Hanno(help ,to,from,layers-1);
        }
    }

    public  static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("请输入层数:");
        int n = sc.nextInt();
        Hanno("A","C","B",n);
    }
```