Task1

我认为并不正确，switch-case的底层原理：当case值是连续或可优化的离散值时，底层会使用跳转表（Jump Table） 来实现。

跳转表是一个地址数组，数组的索引对应case的值，程序通过计算索引直接跳转到对应的分支代码，执行效率比多次if-else判断更高

如果case值是离散且不连续的，编译器可能会将其优化为类似if-else if的结构，但本质上和纯粹的if-else判断在指令层面还是有区别的。

if-else：是通过条件跳转指令来实现的，程序会依次判断每个条件，满足则执行对应分支，不满足则继续判断下一个条件，属于线性的判断流程

Task2

```
1. void print(int n) {
2. int mid = n / 2;
3. // 打印上半部分（包括中间行）
4. for (int i = 0; i <= mid; i++) {
5. // 打印前导空格
6. for (int j = 0; j < mid - i; j++) {
7. System.out.print(" ");
8. }
9. // 打印第一个星号
10. System.out.print("*");
11. // 如果是第一行，只需要一个星号
12. if (i == 0) {
13. System.out.println();
14. } else {
15. // 打印中间的空格
16. for (int j = 0; j < 2 * i - 1; j++) {
17. System.out.print(" ");
18. }
19. // 打印最后一个星号
20. System.out.println("*");
21. }
22. }
23. // 打印下半部分
24.     for (int i = mid - 1; i >= 0; i--) {
25.         // 打印前导空格
26.         for (int j = 0; j < mid - i; j++) {
27.             System.out.print(" ");
```

```
        }

        // 打印第一个星号
        System.out.print("*");

        // 如果是最后一行，只需要打印一个星号
        if (i == 0) {
            System.out.println();
        } else {
            // 打印中间的空格
            for (int j = 0; j < 2 * i - 1; j++) {
                System.out.print(" ");
            }
            // 打印最后一个星号
            System.out.println("*");
        }
    }
```

Task3

递归版本：

```
int Fibonacci(int n) {
if (n >= 1) {
if (n == 1 || n == 2) {
return 1;
} else {
return Fibonacci(n - 1) + Fibonacci(n - 2);
}
}
return 0;
}
```

迭代版本：

```
int Fibonacci(int n) {
if (n <= 0) return 0;
if (n == 1 || n == 2) return 1;

    int prev1 = 1;  // F(n-1)
    int prev2 = 1;  // F(n-2)
    int current = 0;

    for (int i = 3; i <= n; i++) {
        current = prev1 + prev2;
        prev2 = prev1;
```

```
        prev1 = current;
    }

    return current;
}
```

递归和迭代迭代不同之处:
执行方式不同
递归：函数调用自身，通过系统调用栈来保存每次调用的状态
迭代：使用循环结构，在同一个函数内重复执行代码块
内存使用差异
递归：每次函数调用都会在调用栈上创建新的栈帧，存储局部变量和返回地址
迭代：通常只使用固定的内存空间，不会随着问题规模增长而增加栈空间
时间复杂度表现
递归：可能存在大量重复计算（如原始斐波那契实现）
迭代：通常能避免重复计算，按顺序执行
迭代的优势：避免内存浪费和更简便易懂
循环理论上可以由递归取代
Task4：
```
void hanoi(int n, char A, char B, char C) {
if (n == 1) {
// 基础情况：只有一个盘子，直接移动
System.out.println(A + “->” + C);
} else {
// 1. 将前n-1个盘子从A借助C移动到B
hanoi(n - 1, A, C, B);
// 2. 将第n个盘子从A移动到C
System.out.println(A + “->” + C);
// 3. 将n-1个盘子从B借助A移动到C
hanoi(n - 1, B, A, C);
}
}
}
```