一、软件测试

1. 什么是软件测试
   提高软件质量的重要手段
   确认是否达到可用级别
   关注系统的某一侧面的质量特性
2. 测试级别
   Unit testing 单元测试
   Integration testing 集成测试
   System testing 系统测试
   Regression testing 回归测试
   Acceptance testing 验收测试
3. 静态测试与动态测试
   Static testing
      不执行程序
      Reviews
      walkthroughs 预排/演练/走查
      inspections 视察
   Dynamic testing
      执行程序，有测试用例
      Debugger
4. 测试的困难性
   穷举+暴力=不可能
   靠偶然测试没有意义
   基于样本的统计数据对软件测试意义不大
   软件行为在离散输入空间中差异巨大
   无统计分布规律可循

二、测试用例

Test case={test inputs+exception conditions+excepted results}
测试用例=输入+执行条件+期望结果

三、测试优先的编程

步骤：
1. 为函数写一个规格说明(specification)
2. 写符合规格说明的测试用例
3. 编写实际代码，执行测试，直到通过它

写测试用例，就是理解、修正、完善你的 spec 设计的过程。
先写测试会节省大量的调试时间。

四、单元测试

Unit testing：针对软件的最小单元模型开展测试，隔离各个模块，容易定位错误和调试。

JUnit test case：

在测试方法前使用@Test annotation 来表明这是一个 JUnit 测试方法。如果要在测试开始之前做一些准备则在准备方法前添加@Before annotation，如果要在测试结束后做一些收尾工作则在收尾方法前添加@After annotation。

JUnit 使用的是断言机制来完成测试，常用的有三种测试方法：assertEquals()、assertTrue()、assertFalse()。

```
AssertEquals(2,Math.max(1,2));
```

要注意的是 assertEquals 的参数顺序很重要。它的第一个应该是我们期望的值，通常是一个我们算好的常数，第二个参数就是我们要进行的测试。

五、黑盒测试

黑盒测试：用于检查代码的功能，不关心内部实现细节

检查程序是否符合规约

用尽可能少的测试用例，尽快运行，并尽可能的发现程序的错误

5.1 通过分区的方法选择测试用例

Example 1：BigInteger.multiply()

从正负的角度对二维空间进行等价类划分

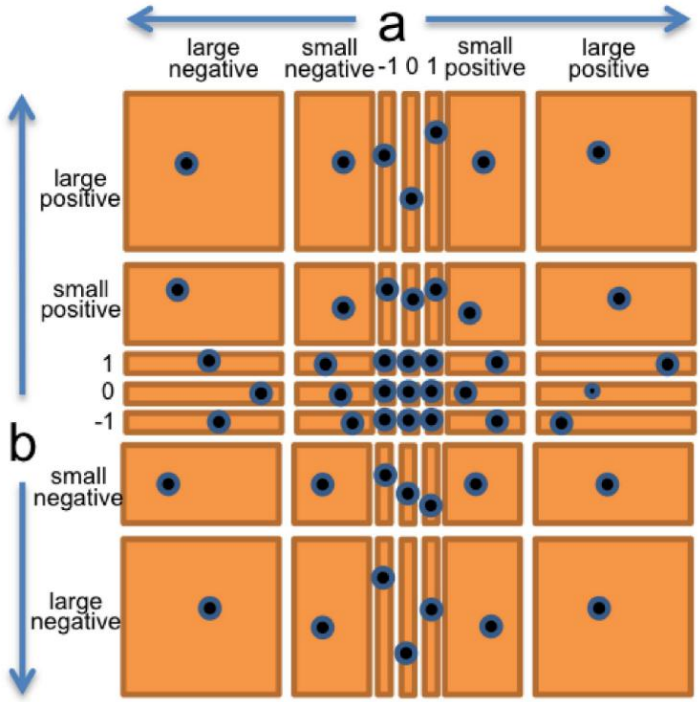
- a and b are both positive
- a and b are both negative
- a is positive, b is negative
- a is negative, b is positive

需要考虑输入数据的特殊情况

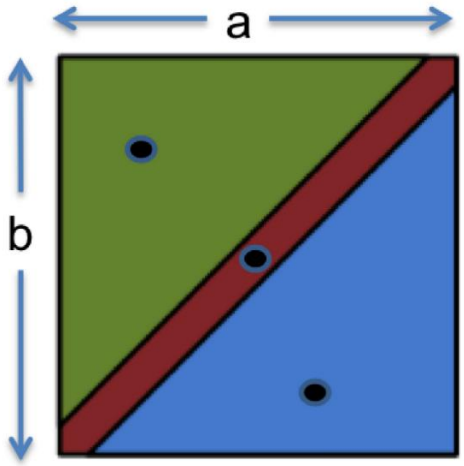
- a or b is 0, 1, or -1

考虑输入的上限：很大的数是否仍然正确

- Small/big




```
Example 2:max()
    Max:int*int->int
    a<b
    a=b
    a>b
```



5.2 包含分区的边界

bug 经常会在各个分区的边界处发生，例如：

（1）在正整数和负整数之间的 0

（2）数字类型的最大值和最小值，例如 int 和 double

（3）空集，例如空的字符串，空的列表，空的数组

（4）集合类型中的第一个元素或最后一个元素

为什么这些边界的地方经常产生 bug 呢？一个很重要的原因就是程序员经常犯"丢失一个（off-by-one mistakes）"的错误。例如将<=写成<，或者将计数器用 0 来初始化而不是 1。另外一个原因就是边界处的值可能需要用特殊的行为来处理，例如当 int 类型的变量达到最大值以后，再对其加正整数反而会变成负数。

redo max:int×int→int

- Value of a
  - a = 0
  - a < 0
  - a > 0
  - a = minimum integer
  - a = maximum integer
- Value of b
  - b = 0
  - b < 0
  - b > 0
  - b = minimum integer
  - b = maximum integer

(1, 2) covers a < b, a > 0, b > 0
(-1, -3) covers a > b, a < 0, b < 0
(0, 0) covers a = b, a = 0, b = 0
(Integer.MIN_VALUE, Integer.MAX_VALUE) covers a < b, a = minint, b = maxint
(Integer.MAX_VALUE, Integer.MIN_VALUE) covers a > b, a = maxint, b = minint

5.3 覆盖分区的两个极限情况

笛卡尔积：全覆盖

多个划分维度上的多个取值，要组合起来，每个组合都要有一个用例

测试完备，但用例数量多，测试代价高

覆盖每个取值：最少 1 次即可

每个维度的每个取值至少被 1 个测试用例覆盖一次即可

测试用例少，代价低，但测试覆盖度未必高

六、代码覆盖度

代码覆盖度：已有的测试用例有多大程度覆盖了被测程序

代码覆盖度越低，测试越不充分；但要做到很高的代码覆盖度，需要更多的测试用例，测试代价高。

Function coverage 函数覆盖

Statement coverage 语句覆盖

Branch coverage 分支覆盖
Condition coverage 条件覆盖
Path coverage 路径覆盖

测试效果：路径覆盖>分支覆盖>语句覆盖
测试难度：路径覆盖>分支覆盖>语句覆盖

100%的声明覆盖率一个普遍的要求，但是这有时也是不可能实现的，因为会存在一些"不可能到达的代码"（例如有一些断言）。
100%的分支覆盖率是一种很高的要求，对于军工/安全关键的软件可能会有此要求。
100%的路径覆盖率是不可能的，因为这会让测试用例空间以指数速度增长。

一个标准的方法就是不断地增加测试用例直到覆盖率达到了预定的要求。在实践中，声明覆盖通常用覆盖率工具进行计数。你只需要不断地调整覆盖的地方，直到所有重要的声明都被覆盖到。

在 Eclipse 中有一个好用的代码覆盖率工具 EclEmma。EclEmma 会将被执行过的代码用绿色标出，没有被执行的代码用红色标出。对于一个分支语句，如果它的一个分支一直没有被执行，那么这个分支判断语句会被标为黄色。

七、编写测试策略

测试策略（根据什么来选择测试用例）非常重要，需要在程序中显式记录下来。目的：在代码评审过程中，其他人可以理解你的测试，并评判你的测试是否足够充分。

我们应该在测试时记录下我们的测试策略，例如我们是如何分区的，有哪些特殊值、边界值等等：

```
/*
 * Testing strategy
 *
 * Partition the inputs as follows:
 * text.length(): 0, 1, > 1
 * start:         0, 1, 1 < start < text.length(),
 *                text.length() - 1, text.length()
 * text.length()-start: 0, 1, even > 1, odd > 1
 *
 * Include even- and odd-length reversals because
 * only odd has a middle element that doesn't move.
 *
 * Exhaustive Cartesian coverage of partitions.
 */
```

另外，每一个测试方法都要有一个小的注解，告诉读者这个测试方法是代表我们测试策略中的哪一部分，例如：

```
// covers test.length() = 0,
//        start = 0 = text.length(),
//        text.length()-start = 0
@Test public void testEmpty() {
    assertEquals("", reverseEnd("", 0));
}
```