CSE 464 – Homework 2
Bhavya Patel - ASU ID: 1225740997

Q1 a)
My approach is to use Equivalent Partitioning (EP) and some Boundary Value Analysis (BVA), focusing on both valid (normal) and invalid (robust) input conditions. EP helps divide the input domain (the ZIP code) into partitions where we expect the system to behave similarly.
Since ZIP codes are defined as a 5-digit number or a 9-digit ZIP+4, and the prompt specifically asks for the "Cities by ZIP Code" tool, which typically takes the 5-digit ZIP, I will partition based on the 5-digit structure and range of the ZIP code.

Equivalent Classes:

- **Valid EP**
  - EP1: Valid 5-digit ZIP with a single dominant city result, ensuring the tool lists at least the preferred city per USPS guidance.
  - EP2: Valid 5-digit ZIP with multiple acceptable names, verifying the UI shows the recommended/recognized city set rather than an arbitrary long alias list, consistent with USPS “recommended or recognized” guidance.
  - EP3: Valid 5-digit ZIPs spanning first-digit regions, including a leading-zero ZIP and a high 9-prefix ZIP. I specifically include the known lowest (00501) and highest (99950) ZIPs to hit numeric extremes with realistic data.
- **Invalid EP**
  - EP4: Invalid length should not return cities. The UI either shows a clear “You did not enter a valid ZIP Code” message aligned with the tool’s scope.
  - EP5: Non-existent/unassigned 5-digit ZIP should yield “You did not enter a valid ZIP Code” and match the page’s behavior when no match exists.
  - EP6: Non-numeric or mixed characters should be rejected because the tool expects a numeric ZIP (5 digits), per USPS standards.
  - EP7: Special-case numeric values that are structurally valid but invalid in context should not map to cities and should fail lookup.

b) I've selected specific ZIP codes to cover the equivalent partitions and incorporate Boundary Value Analysis (BVA) where appropriate, as BVA complements EP by testing values at the edges of the partitions.

Test Cases	ZIP Code Input	Expected Output
TC1	00501	City list present; includes Holtsville, NY, because 00501 is a valid ZIP and leading double zeros must be accepted, ending with 5-9 per USPS facts and tool scope.
TC2	99950	City list present; includes Ketchikan, AK, since 99950 is the highest-numbered ZIP, validating upper numeric handling.
TC3	85281	City list present; includes Tempe, AZ, as a normal single-preferred-city style case to verify straightforward mapping.
TC4	01813	City list present; verify display of recommended/recognized names where multiple acceptable names may exist in the 018 region.
TC5	00000	Error: Invalid ZIP Code.
TC6	1234	Error: Invalid ZIP Code.
TC7	123456	Error: Invalid ZIP Code.
TC8	ASU	Error: Invalid ZIP Code.
TC9	48!7@	Error: Invalid ZIP Code.
TC10	90650	City list present; includes Norwalk, CA as a straightforward preferred-city mapping.
TC11	90210-0133	City list present; same city set as 90210 since ZIP+4 should resolve to the same city list for this feature, or be guided to use 5 digits per tool scope.

c) I believe that USPS does not provide a sufficiently detailed flow for this use case. While the page copy is straightforward, the process lacks consistent clarity, especially regarding edge cases and special situations. The main issue is that the tool instructs users to "enter a ZIP to see the cities it covers," but it does not clearly explain how it handles partially valid numeric ranges within a prefix or how it processes ZIP+4 codes in this context.

For example, consider the range 09700–09799. Some codes in the lower part of this range (for instance, 09700–09756) may be valid and assigned in the USPS dataset, while the higher end (e.g., 09757–09799) may not be assigned at all. This inconsistency creates ambiguity that can affect users trying to validate ZIP codes.

Q2 a)

My approach is to combine two Equivalent Partitioning (EP) strategies: partitioning by output (BMR Categories) and partitioning by input validity (Valid/Invalid Ranges). Since the problem has four distinct numerical outputs based on the calculated BMR, and three input variables with defined valid ranges, this combined approach ensures comprehensive test coverage.

Equivalent Partitions:

Valid EP:

EP1: BMR<1200 kcal/day
EP2: 1200≤BMR<1800 kcal/day
EP3: 1800≤BMR<2500 kcal/day
EP4: BMR≥2500 kcal/day

Invalid EP:

EP5: Gender≠ "male" or "female"
EP6: Age≤0 or Weight≤0 or Height≤0
EP7: Age≥120 or Weight≥1000 or Height≥100

b)

Test Case #	Age	Weight	Height	Gender	Expected Output
WR1	-5	150	60	male	Invalid Inputs: Age cannot be negative.
WR2	30	150	0	male	Invalid Inputs: Height cannot be zero.
WR3	30	-1	60	female	Invalid Inputs: Weight cannot be negative.
WR4	122	150	60	male	Invalid Inputs: Age is out of range.
WR5	30	150	101	male	Invalid Inputs: Height is out of range.
WR6	30	1100	60	female	Invalid Inputs: Weight is out of range.
WR7	30	150	60	other	Invalid Inputs: Gender is invalid.
WR8	15	140	60	male	All Valid Inputs: BMR Calculations

c)

```java
import static org.junit.Assert.*;
import org.junit.Test;
import java.io.*;
import java.nio.charset.StandardCharsets;

public class WeakRobustTests {

   private String runWithInput(String input) throws Exception {
       InputStream backupIn = System.in;
       PrintStream backupOut = System.out;
       ByteArrayOutputStream buffer = new ByteArrayOutputStream();
       try {
           System.setIn(new
ByteArrayInputStream(input.getBytes(StandardCharsets.UTF_8)));
           System.setOut(new PrintStream(buffer, true,
StandardCharsets.UTF_8));
           Runnable task = () -> Assignment2.main(new String[0]);
           task.run();
           return buffer.toString(StandardCharsets.UTF_8);
       } finally {
           System.setIn(backupIn);
           System.setOut(backupOut);
       }
   }

   @Test
   public void WR1_ageNegative() throws Exception {
       String output = runWithInput("-5\n150\n60\nM\n");
       assertTrue(output.contains("Error: Age must be between 1 and 119."));
   }

   @Test
   public void WR2_heightZero() throws Exception {
       String output = runWithInput("30\n150\n0\nM\n");
       assertTrue(output.contains("Error: Height must be between 1 and 99
inches."));
   }

   @Test
   public void WR3_weightNegative() throws Exception {
       String output = runWithInput("30\n-1\n60\nF\n");
       assertTrue(output.contains("Error: Weight must be between 1 and 999
pounds."));
   }

   @Test
   public void WR4_ageTooHigh() throws Exception {
       String output = runWithInput("122\n150\n60\nM\n");
       assertTrue(output.contains("Error: Age must be between 1 and 119."));
```

```
    }

    @Test
    public void WR5_heightTooHigh() throws Exception {
        String output = runWithInput("30\n150\n101\nM\n");
        assertTrue(output.contains("Error: Height must be between 1 and 99
inches."));
    }

    @Test
    public void WR6_weightTooHigh() throws Exception {
        String output = runWithInput("30\n1100\n60\nF\n");
        assertTrue(output.contains("Error: Weight must be between 1 and 999
pounds."));
    }

    @Test
    public void WR7_genderInvalid() throws Exception {
        String output = runWithInput("30\n150\n60\nother\n");
        assertTrue(output.contains("Error: Invalid gender. Please enter 'M' or
'F'."));
    }

    @Test
    public void WR8_allValid_maleModerate() throws Exception {
        String output = runWithInput("15\n140\n60\nM\n");
        assertTrue(output.contains("Your BMR is:"));
        assertTrue(output.contains("Metabolism Category: Moderate Metabolism"));
    }
}
```

https://drive.google.com/file/d/1VN8VeOmh8FGeEgRfQwXUhOqXF7SpiXZq/view?usp=drive_link

d)

Input	Valid Range	Minimum Value	Maximum Value
Age (years)	0<age<120	1	119
Height (inches)	0<height<100	1	99
Weight (pounds)	0<weight<1000	1	999

e)

Test Case #	Age (Yrs)	Weight (Lbs)	Height (In)	Expected Output
1	0	500	50	Invalid Inputs: Age cannot be zero.

2	1	500	50	BMR Calculation
3	119	500	50	BMR Calculation
4	120	500	50	Invalid Inputs: Age is out of range.
5	30	500	0	Invalid Inputs: Height cannot be zero.
6	30	500	1	BMR Calculation
7	30	500	99	BMR Calculation
8	30	500	100	Invalid Inputs: Height is out of range.
9	30	0	50	Invalid Inputs: Weight cannot be zero.
10	30	1	50	BMR Calculation
11	30	999	50	BMR Calculation
12	30	1000	50	Invalid Inputs: Weight is out of range.

f)

```
import static org.junit.Assert.*;
import org.junit.Test;
import java.io.*;
import java.nio.charset.StandardCharsets;

public class BoundaryRobustTests {

   private String runWithInput(String input) throws Exception {
       InputStream backupIn = System.in;
       PrintStream backupOut = System.out;
       ByteArrayOutputStream buffer = new ByteArrayOutputStream();
       try {
           System.setIn(new
ByteArrayInputStream(input.getBytes(StandardCharsets.UTF_8)));
           System.setOut(new PrintStream(buffer, true,
StandardCharsets.UTF_8));
           Runnable task = () -> Assignment2.main(new String[0]);
           task.run();
           return buffer.toString(StandardCharsets.UTF_8);
       } finally {
           System.setIn(backupIn);
           System.setOut(backupOut);
       }
   }

   @Test
   public void B1_ageZeroInvalid() throws Exception {
       String output = runWithInput("0\n500\n50\nM\n");
       assertTrue(output.contains("Error: Age must be between 1 and 119."));
```

```
    }

    @Test
    public void B2_ageOneValid() throws Exception {
        String output = runWithInput("1\n500\n50\nM\n");
        assertTrue(output.contains("Your BMR is:"));
        assertTrue(output.contains("Metabolism Category:"));
    }

    @Test
    public void B3_age119Valid() throws Exception {
        String output = runWithInput("119\n500\n50\nM\n");
        assertTrue(output.contains("Your BMR is:"));
        assertTrue(output.contains("Metabolism Category:"));
    }

    @Test
    public void B4_age120Invalid() throws Exception {
        String output = runWithInput("120\n500\n50\nM\n");
        assertTrue(output.contains("Error: Age must be between 1 and 119."));
    }

    @Test
    public void B5_heightZeroInvalid() throws Exception {
        String output = runWithInput("30\n500\n0\nM\n");
        assertTrue(output.contains("Error: Height must be between 1 and 99
inches."));
    }

    @Test
    public void B6_heightOneValid() throws Exception {
        String output = runWithInput("30\n500\n1\nM\n");
        assertTrue(output.contains("Your BMR is:"));
        assertTrue(output.contains("Metabolism Category:"));
    }

    @Test
    public void B7_height99Valid() throws Exception {
        String output = runWithInput("30\n500\n99\nM\n");
        assertTrue(output.contains("Your BMR is:"));
        assertTrue(output.contains("Metabolism Category:"));
    }

    @Test
    public void B8_height100Invalid() throws Exception {
        String output = runWithInput("30\n500\n100\nM\n");
        assertTrue(output.contains("Error: Height must be between 1 and 99
inches."));
    }
```

```
    @Test
    public void B9_weightZeroInvalid() throws Exception {
        String output = runWithInput("30\n0\n50\nM\n");
        assertTrue(output.contains("Error: Weight must be between 1 and 999
pounds."));
    }

    @Test
    public void B10_weightOneValid() throws Exception {
        String output = runWithInput("30\n1\n50\nM\n");
        assertTrue(output.contains("Your BMR is:"));
        assertTrue(output.contains("Metabolism Category:"));
    }

    @Test
    public void B11_weight999Valid() throws Exception {
        String output = runWithInput("30\n999\n50\nM\n");
        assertTrue(output.contains("Your BMR is:"));
        assertTrue(output.contains("Metabolism Category:"));
    }

    @Test
    public void B12_weight1000Invalid() throws Exception {
        String output = runWithInput("30\n1000\n50\nM\n");
        assertTrue(output.contains("Error: Weight must be between 1 and 999
pounds."));
    }
}
```

https://drive.google.com/file/d/1dwbjlpQam2BZucnu01NLGK1K7eN19ZoP/view?usp=drive_link