**Programa A: findLast**

* **Defeito:** A lógica do laço for está incorreta em sua condição de parada. O laço for (int i = x.length-1; i > 0; i--) percorre o array do final para o começo, mas para quando i chega a 0, nunca executando o teste para o elemento na posição x[0]. Isso significa que, se o valor y que estamos procurando estiver localizado apenas no primeiro índice do array, a função falhará em encontrá-lo e retornará -1 incorretamente.
* **Caso que não executa o defeito:**
  + x = [2, 3, 5], y = 5
  + **Análise:** O defeito existe no código, mas não é ativado aqui. O programa procura pelo número 5, o encontra no índice 2 e retorna 2 imediatamente. A falha de não verificar o índice 0 nunca chega a impactar o resultado.
* **Caso com defeito, mas sem erro:**
  + x = [5, 7, 8], y = 7
  + **Análise:** Neste caso, o valor procurado (7) não está no índice 0. O laço encontra o 7 no índice 1 e retorna o valor correto. Embora o código tenha um defeito (ele pularia o índice 0), o resultado para *esta entrada específica* está correto por coincidência.
* **Caso com erro, mas sem falha:**
  + (Não aplicável para este exemplo)
* **Código Corrigido:**

Java

public int findLast (int[] x, int y) {

    for (int i = x.length-1; i >= 0; i--) {

        if (x[i] == y) {

            return i;

        }

    }

    return -1;

}

**Programa B: lastZero**

* **Defeito:** O programa deveria encontrar o índice do **último** zero no array. No entanto, a lógica original (não mostrada, mas inferida) provavelmente retornava o índice assim que encontrava o **primeiro** zero. Isso interrompe a busca prematuramente e impede que o resto do array seja verificado, levando a um resultado incorreto se houver zeros em posições posteriores.
* **Caso que não executa o defeito:**
  + x = [1, 2, 3]
  + **Análise:** Como não há nenhum zero no array, o laço termina sem nunca encontrar uma correspondência. Tanto a versão com defeito quanto a corrigida retornariam -1 corretamente.
* **Caso com defeito, mas sem erro:**
  + x = [0]
  + **Análise:** O código defeituoso encontraria o zero no índice 0 e retornaria 0. O código correto também identificaria o último zero no índice 0 e retornaria 0. O resultado é o mesmo, pois o primeiro e o último zero são o mesmo elemento.
* **Caso com erro, mas sem falha:**
  + (Não aplicável para este exemplo)
* **Código Corrigido:**

Java

public static int lastZero(int[] x) {

    int last = -1;   // guarda o último zero

    for (int i = 0; i < x.length; i++) {

        if (x[i] == 0) {

            last = i;

        }

    }

    return last;

}

**Programa C: countPositive**

* **Defeito:** A função visa contar números estritamente positivos (maiores que zero). O defeito na lógica original (provavelmente if (x[i] >= 0)) inclui o número 0 na contagem. Matematicamente, zero não é nem positivo nem negativo, então contá-lo como positivo leva a um resultado incorreto.
* **Caso que não executa o defeito:**
  + x = [1, 2, 3]
  + **Análise:** Este array contém apenas números estritamente positivos. Ambas as condições (>= 0 e > 0) produziriam o mesmo resultado correto (3), mascarando a presença do defeito.
* **Caso com defeito, mas sem erro:**
  + x = [-1, -2, 0]
  + **Análise:** O resultado esperado é 0, pois não há números positivos. A versão com defeito, no entanto, contaria o 0 e retornaria 1. Aqui, o defeito se manifesta como um erro claro no resultado (o valor obtido é 1, quando o esperado é 0).
* **Caso com erro, mas sem falha:**
  + (Não aplicável para este exemplo)
* **Código Corrigido:**

Java

public static int countPositive(int[] x) {

    int count = 0;

    for (int i = 0; i < x.length; i++) {

        if (x[i] > 0) {   // corrigido para > 0

            count++;

        }

    }

    return count;

}

**Programa D: oddOrPos**

* **Defeito:** O objetivo é contar números que são ímpares **ou** positivos. O problema está em como os números ímpares negativos são tratados. Em Java, o operador de resto (%) com um número negativo pode resultar em um valor negativo (ex: -3 % 2 é -1). Se a verificação original fosse x[i] % 2 == 1, ela falharia para todos os números ímpares negativos, que não seriam contados a menos que também fossem positivos (o que é impossível).
* **Caso que não executa o defeito:**
  + x = [2, 4, 6]
  + **Análise:** Todos os números são positivos. A segunda parte da condição (x[i] > 0) já garante que todos sejam contados. A lógica falha para identificar ímpares nunca é testada de forma decisiva.
* **Caso com defeito, mas sem erro:**
  + x = [1, 3, 5]
  + **Análise:** Os números são ímpares e positivos. Eles satisfazem ambas as partes da condição na versão corrigida. Na versão com defeito, eles satisfazem x[i] % 2 == 1 e também x[i] > 0, então o resultado final é o mesmo. O defeito relacionado a números negativos não é exposto.
* **Caso com erro, mas sem falha:**
  + (Não aplicável para este exemplo)
* **Código Corrigido:**

Java

public static int oddOrPos(int[] x) {

    int count = 0;

    for (int i = 0; i < x.length; i++) {

        if (x[i] % 2 != 0 || x[i] > 0) {   // corrigido para % 2 != 0

            count++;

        }

    }

    return count;

}