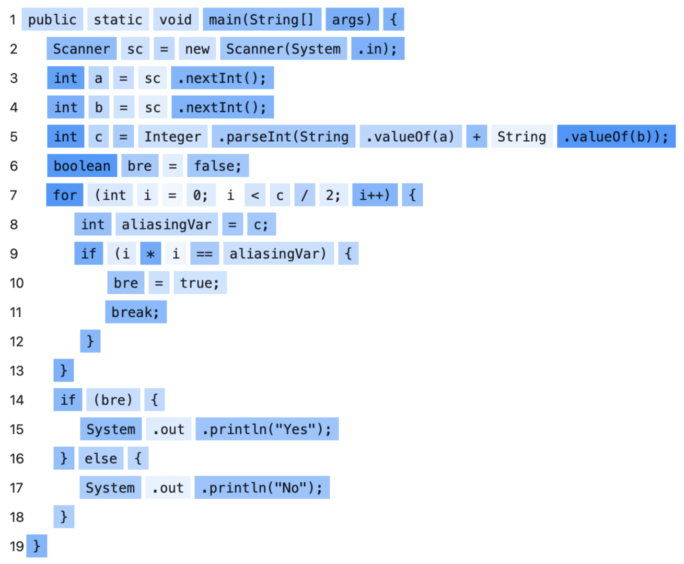
A

```
1     #include<iostream:h>
2     int main() {
3        int currentValue;
4        int currentMaximum;
5        int currentMinimum;
6        int nonNegativeCount;
7        int lcv;
8        int size;
9        int integerArray[100];
10       size = 0;
11       cout << "Input an integer or -999999 to stop\n";
12       cin >> currentValue;
13       while (currentValue = -999999) {
14          integerArray[size] = currentValue;
15          size++;
16          cout << "Input an integer or -999999 to stop\n";
17          cin >> currentValue;
18       }
19       currentMinimum = 0;
20       currentMaximum = 0;
21       nonNegativeCount = 0;
22       for (lcv = 0; lcv < size; lcv++) {
23          if (integerArray[lcv] < 0) {
24             if (integerArray[lcv] < currentMinimum)
25                currentMinimum = integerArray[lcv];
26             if (integerArray[lcv] > currentMaximum);
27                currentMaximum = integerArray[lcv];
28          }
29          else
30             nonNegativeCount++;
31       }
32       cout << end1 <<end1 <<end1;
33       cout <<"The number of non-negative numbers in the list was " << nonNegativeCount << end1;
34       cout << "The maximum negative number in the list was " << currentMaximum << end1;
35       cout << "The minimum negative number in the list was " << currentMinimum << end1;
36    }
```

Figure 1: A Buggy C++ Program (Fault Area: Line 20)

Listing 1 corresponds to the complete C++ program referenced in Section 1 (Footnote 1). Here, once the developer determines that the output value for `currentMaximum` on Line 35 is incorrect, the code fault area can be reduced to the corresponding slice: lines *1, 2, 3, 4, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 20, 22, 23, 26, 27, 28, 31, 32, 34, 36*. This decreases the size of the fault area from the original by 10 statements or 28%.

B



```
1  public static void main(String[] args) {
2    Scanner sc = new Scanner(System.in);
3    int a = sc.nextInt();
4    int b = sc.nextInt();
5    int c = Integer.parseInt(String.valueOf(a) + String.valueOf(b));
6    boolean bre = false;
7    for (int i = 0; i < c / 2; i++) {
8      int aliasingVar = c;
9      if (i * i == aliasingVar) {
10       bre = true;
11       break;
12     }
13   }
14   if (bre) {
15     System.out.println("Yes");
16   } else {
17     System.out.println("No");
18   }
19 }
```

Figure 2: Visualization of attention score heatmap from GraphCodeBERT for all words with respect to variable `c` on line 7.

Figure 2 illustrates a visualization of attention maps from a pre-trained GraphCodeBERT PLM off-the-shelf, for all the words in a Java program with respect to variable `c` on line 7 (referenced in Section 6.4.4, Footnote 2). Here, in contrast to Figure 5 (in the paper), we can see that the model does not pay attention to the data and control dependent variables. As a result, the program slice computed by fitting the pre-trained GraphCodeBERT model in NS-Slicer is not capable of predicting accurate backward and forward slices. Therefore, such PLMs cannot directly be used for the program slicing task, reiterating the need to formulate the program slicing task as in NS-Slicer.